Bounding Work *and* Communication in Robust Cooperative Computation*

Bogdan S. Chlebus[1,2], Leszek Gąsieniec[3],
Dariusz R. Kowalski[2,4], and Alex A. Shvartsman[5,6]

[1] Department of Computer Science and Engineering, University of Colorado at Denver, Campus Box 109, Denver, CO 80217-3364, USA.
[2] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, Warszawa 02-097, Poland.
[3] Department of Computer Science, University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, UK.
[4] Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada.
[5] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 3155, Storrs, CT 06269, USA.
[6] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square NE43-316, Cambridge, MA 02139, USA.

Abstract. We consider the *Do-All* problem: p failure-prone processors perform t similar and independent tasks. We assume that processors are synchronous, communicate by message passing, and are subject to crashes determined by an adaptive adversary restricted only by the upper bound f on the number of crashes. The performance of algorithms in this setting is normally measured in terms of *work* (total available processor steps) and *communication* (total number of point-to-point messages) complexity. We consider work and communication as comparable resources and we develop algorithms that have efficient *effort* defined as *work* + *communication*. We present a p-processor, t-task algorithm that has effort $\mathcal{O}(t+p^{1.77})$, against the unbounded adversary ($f < p$). This is the *first* algorithm that achieves *subquadratic* in p effort efficiency for unbounded adversary, or even for linearly-bounded adversary that crashes up to a constant fraction of the processors. We present another algorithm that has work $\mathcal{O}(t + p\log^2 p)$ against f-bounded adversaries such that $p - f = \Omega(p^b)$ for a constant b, $0 < b < 1$. We show how to achieve effort $\mathcal{O}(t + p\log^2 p)$ against a linearly-bounded adversary; this result is close to lower bound $\Omega(t + p\log p/\log\log p)$.

1 Introduction

We study algorithms for a distributed system in which tasks need to be done by failure-prone processors. We consider the abstract problem, called *Do-All*, where the failure-prone processors communicate by message passing and where the tasks are similar, re-entrant, independent and idempotent. Such problems

* The work of the third author was in part supported by the NSF-NATO Grant 0209588. The work of the fourth author was supported by a NSF CAREER Award and NSF Grants 9988304 and 0121277.

are encountered when a set of tasks (in particular tasks admitting "at least once" execution semantics) needs to be performed in a distributed system.

The *Do-All* problem in the synchronous message-passing setting was introduced by Dwork, Halpern and Waarts [12], whose algorithms' efficiency is measured in terms of *effort*, defined as the sum of *task-oriented work* and *communication.* Task-oriented work counts only the processing steps expended on performing tasks and it discounts any idling or waiting steps. Communication is measured as the message complexity. A more conservative measure of work is *available processor steps* [16] that accounts for all processing steps, including task-oriented work, idling and waiting. It is used in most solutions [4, 10, 14].

We also consider the synchronous failure-prone setting where t tasks need to be performed by p message-passing processors subject to up to f crash failures imposed by an adversary. The adversary is adaptive, meaning that the decisions about the failures of specific processors are made on-line. A processor can send a message to any subset of processors in one step. If a processor crashes during a step then any messages it sent are delivered to some arbitrary subset of the destinations. Crashed processors do not restart.

We measure algorithm performance using the *effort* approach following [12], but we use the *work* component of effort measured in terms of *available processor steps* following De Prisco, Mayer and Yung [10]. Thus we define *effort* to be $\mathcal{W}+\mathcal{M}$, where *work* $\mathcal{W}$ accounts for each step taken by the operational processors, includes idling, and where *communication* $\mathcal{M}$ is measured as the number of point-to-point messages. Prior research (except for [12]) did not consider effort, and instead focused on developing *Do-All* algorithms that are efficient in terms of work, and then dealt with communication efficiency as a secondary goal

Our results. Let t be the number of tasks and p the number of processors. We present a new way of structuring algorithms for the p-processor t-task *Do-All* problem that allows for both work *and* communication to be controlled in the presence of adaptive adversaries. We give a generic algorithm for performing work in the networks of crash-prone processors. This algorithm is parameterized for task-assignment rules and virtual communication graphs superimposed on the underlying network.

We call a deterministic algorithm *constructive* if all implementation details are specified. This is in contrast to algorithms that refer to combinatorial objects that are known to exist. Starting with a fully constructive algorithm, we trade constructiveness for better effort bounds. In some cases, either the communication patterns or the patterns of performing tasks are determined by referring to permutations with certain properties that are known to exist. In all cases the algorithms also solve *Do-All* when exposed to the unbounded adversary.

I. We present a deterministic constructive algorithm, called algorithm BAL, that uses a balancing task allocation policy (Section 5.1). This algorithm has work $\mathcal{W} = \mathcal{O}(t + p\sqrt{p+t}\ \log^{3/2} p)$ against adaptive f-bounded adversaries, for any $f < p$. We also show that the effort of this algorithm against linearly-bounded adversaries is $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p\sqrt{t+p}\ \log^{3/2} p)$. In this case each processor sends $\mathcal{O}(1)$ messages in each step.

Table 1. Summary of previous results against the unbounded adversary.

Paper	Metrics	Complexity	Remarks
[12]	*Work (tasks)* *Messages*	$\mathcal{O}(t+p)$ $\mathcal{O}(p\sqrt{p})$	Task-oriented work (allows idling and waiting)
[10]	*Work* *Messages*	$\mathcal{O}(t+p^2)$ $\mathcal{O}(p^2)$	
[14]	*Work* *Messages*	$\mathcal{O}(t+p^2)$ $\mathcal{O}(fp^\epsilon + p\log p)$	
[4]	*Work* *Messages*	$\mathcal{O}(t\log f + p\frac{\log p \log f}{\log\log p})$ $\mathcal{O}(t + p\frac{\log p}{\log\log p} + fp)$	Using atomic broadcast for unbounded adversary, $f < p$
[4]	*Work* *Messages*	$\mathcal{O}((t\log f + p\log p \log f)$ $\mathcal{O}(t + p\log p + fp)$	Using atomic broadcast; results are restated here for less than p restarts

II. We also develop a deterministic algorithm which is more efficient than algorithm BAL, but it is not constructive (Section 5.2). This algorithm, called algorithm DPAL, has work $\mathcal{W} = \mathcal{O}(t + p\log^2 p)$ against the adaptive f-bounded adversary, if $p - f = \Omega(p^b)$, for a constant b, $0 < b < 1$. When subjected to linearly-bounded adversaries, this algorithm has the best known effort of $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p\log^2 p)$. This effort of algorithm DPAL is close to the lower bound $\Omega(p\log p/\log\log p)$ on work (see [6]).

III. We ultimately develop a deterministic algorithm, called algorithm UAL, that is efficient against the unbounded adversary (Section 5.3). This algorithm is obtained by combining our algorithm DPAL with the algorithm of De Prisco, Mayer and Yung [10]. The effort of our algorithm UAL is $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p^{1.77})$. This is the only known algorithm with both work and communication subquadratic in p in the worst case.

Previous work. Dwork, Halpern and Waarts [12] were the first to study the *Do-All* problem; further research includes [4, 6, 7, 10, 14, 15]. Paper [12] considers a variation of effort, defined as the sum of *task-oriented work* and *communication*. A single-coordinator algorithmic paradigm is common in prior work.

De Prisco, Mayer and Yung [10] were the first to use *available processor steps* as the work measure. They gave an algorithm with work $\mathcal{O}(t + (f+1)p)$ and communication $\mathcal{O}((f+1)p)$. Galil, Mayer and Yung [14] gave an algorithm with better communication cost of $\mathcal{O}(fp^\epsilon + \min\{f+1, \log p\}p)$, for any $\epsilon > 0$. Chlebus, De Prisco and Shvartsman [4] developed algorithms based on aggressive coordination, where the number of coordinators grows exponentially following crashes. Their algorithms rely on *atomic* multicasts. One of their algorithms has work $\mathcal{O}((t + p\log p/\log\log p)\log f)$ and communication $\mathcal{O}(t + p\log p/\log\log p + fp)$; these bounds were improved in [15] for $f \le p/\log\log p$. Another algorithm in [4] incorporates restarted processors, and is the only known algorithm to do so efficiently, and it uses $\mathcal{O}((t + p\log p + f)\cdot\min\{\log p, \log f\})$ work and its message complexity is $\mathcal{O}(t + p\log p + fp)$. A summary of these results is given in Table 1.

Table 1 does not include work dealing with specialized adversaries and models of communication. Chlebus and Kowalski [6] studied *Do-All* with fail-stop failures in the presence of the weakly-adaptive linearly-bounded adversary. They developed a randomized algorithm with the expected effort $\mathcal{O}\big(t+p\big(1+\log^*\big(\frac{t\log p}{p}\big)\big)\big)$. This performance is provably better than that of any deterministic algorithm in such a setting, where work $\Omega(p\log p/\log\log p)$ has to be performed ([6, 15]). Chlebus, Kowalski and Lingas [7] and Clementi, Monti and Silvestri [8] studied *Do-All* in the model where communication is performed over a multiple-access channel.

2 Model: computation, tasks, failures and performance

We assume a system of p synchronous, message-passing processors with unique identifiers in the interval $[1..p]$. We refer to the processor with identifier v as "processor v". The computation is structured in terms of synchronous constant-time steps. There are t tasks that need to be performed; these tasks have unique identifiers in the interval $[1..t]$. Each processor knows the values p and t. We place no restrictions on the relationship between p and t.

Communication. Processors communicate by message passing. We assume a complete network. A processor may send a message to any subset of processors during a step. A message sent in one step is delivered to its recipients by the start of the next step. We assume that no messages are lost or corrupted in transit, and we assume that the size of the messages is at most linear in $p + t$.

Failures. Processors fail by crashing. Non-faulty processors are called *operational*. We denote by f the maximum number of failures that may occur in any execution. We do not consider processor restarts (cf. [4]), and so f does not exceed p. Processors may crash during a multicast; in this case some arbitrary subset of the destinations may receive the message.

Tasks. Every processor knows all the tasks, and it can perform any task given its identifier. We assume that it takes exactly one step to perform a task. Tasks are *idempotent*, that is, they can be performed many times and concurrently. They are also *independent*, in the sense that any order of performance is allowed.

Adversaries. The occurrence of failures is governed by adversarial models. These models are characterized in quantitative and qualitative terms that respectively determine the magnitude of processor failures and the flexibility to select faulty processors by the adversary.

We call an adversary *size-bounded* if there is an upper bound on the number of faults; when an explicit bound f is given we call the adversary f-*bounded*. An adversary is called *linearly-bounded* if it is (cp)-bounded for some constant $0 < c < 1$. An adversary is called *unbounded* if at least one processor is infallible. An adversary is called *adaptive* if it decides on-line what processors to crash and when (possibly subject to the quantitative restrictions). All the adversaries we consider are adaptive, with a possible restriction of being size-bounded. Our goal is to design algorithms that are correct against the unbounded adversary and as efficient as possible against specific size-bounded adversaries.

Correctness. An algorithm solving the *Do-All* problem is considered to be correct if the following two conditions are satisfied:

1. Each task is performed by some processor eventually.
2. Each processor stops to work on the current instance of the problem eventually.

There are two ways in which a processor can satisfy requirement 2. A processor may voluntarily stop to work, we say that it *halts*, or it may be forced to do so by the adversary who crashes the processor. Processors may halt at different steps. Halted processors are considered operational. Halting is a property of algorithms and does not limit the adversary in choosing which f processors are prone to failures. An algorithm *terminates* at a given step if this is the first step by which each processor either halted or crashed.

Performance measures. We consider the following measures of performance: work complexity and communication complexity. *Work* $\mathcal{W}$ counts the total number of steps of processors, including idling, which is accrued till termination. That is, we count the *available processor steps* [16]. Halted processors continue contributing to work until the algorithm terminates. *Communication* $\mathcal{M}$ counts the total number of point-to-point messages sent. We define *effort* as $\mathcal{W} + \mathcal{M}$.

3 Communication and graphs

Let $G = (V, E)$ be a graph, with V the set of nodes and E the set of edges. For an induced subgraph H ($H \subseteq V$) we define $N_G(H)$ to be the subset of V consisting of all the nodes in H and their neighbors in G. We also define G^k to be the k-th power of graph G, that is, $G^k = (V, E')$, where the edge $(u, v) \in E'$ iff there is a path between u and v in G of length at most k. The number of nodes in subgraph H is denoted by $|H|$. The specific families of graphs $G(p)$ that we use have a uniform upper bound on the degrees of nodes, and we denote by Δ the maximum node degree of each graph $G(p)$.

We interpret each processor as a node and each sender/receiver pair as an edge of a *communication graph* $G = G(p)$. The notation $G(p)$ denotes the initial topology of a communication graph on p processors. Each faulty or halted processor is removed causing the communication graph to evolve through a sequence of subgraphs until it vanishes altogether when the algorithm terminates.

Suppose we take snapshots of the communication graphs at certain time-steps s_i, let G_i be the subgraph of $G(p)$ induced by the sets V_i of processors still operational at step s_i. Sets V_i have these properties: $V_{i+1} \subseteq V_i$ and $|V_i| \geq p - f$. Note that progress is achieved if there is a least one "good" connected component H_i of G_i, which evolves suitably with time: (1) it is "large" to be able to perform many tasks concurrently, (2) "of small diameter" to have a small delay in interprocessor coordination, and finaly (3) $H_{i+1} \subseteq H_i$ to guarantee consistency of computation. We quantify the notion of being "good" as follows.

Let f be an upper bound on the number of failures. Let α, β and γ be positive constants. We say that a subgraph $H \subseteq G = G(p)$ is *compact* if $|H| \geq \alpha(p - f)$ and the diameter of H is not larger than $\beta \log p + \gamma$.

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

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

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

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

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

Suppose $\langle G_i \rangle$ is a sequence of subgraphs of $G(p)$ such that $G_{i+1} \subseteq G_i$ and $|G_i| \geq p-f$, for $i \geq 1$. We may use PROPERTY $\mathcal{R}$ to obtain a sequence of compact subgraphs $\langle H_i \rangle$ in $G(p)$, such that $H_{i+1} \subseteq H_i$. To this end it is sufficient to define $H_i = P(G_i) \subseteq G_i$.

We use the family $L(n)$ of regular expander graphs introduced by Lubotzky, Phillips and Sarnak [17]. These graphs are defined and can be constructed for each number n of the form $q(q^2-1)/2$, where q is a prime integer congruent to 1 modulo 4. The node degree Δ_0 can be any number such that $\Delta_0 - 1$ is a prime congruent to 1 modulo 4 and a quadratic nonresidue modulo q. It follows, from the properties of the distribution of prime numbers (see e.g. [9]), that Δ_0 can be selected to be a constant independent n and q such that $n = q(q^2-1)/2 = \Theta(p)$. Since for each p there is a number n with such properties which is $\Theta(p)$, we let each processor simulate $\mathcal{O}(1)$ nodes, and we henceforth assume that p is as required so that $L(p)$ can be constructed.

Upfal [20] showed that there is a function P' such that if R is a subgraph of $L(p)$ of size at least $\frac{71}{72} \cdot p$ then subgraph $P'(R)$ of R has size at least $|R|/6$ and diameter at most $30 \log_2 p$. (These constants in the case of a linear-size subgraphs can be improved, see [2, 13].) We show that a similar fact holds for sublinear-size subgraphs as well if a better expansion of the underlying communication graph is available. To this end we use power $L(p)^k$, for sufficiently large k. Such graphs are also constructive and of bounded degree. Magnifying expansion by taking a power of an expander was already suggested by Margulis [18] and used for instance by Diks and Pelc [11] in the context of broadcasting and fault diagnosis.

Let $\lambda > 1$ be a constant witht he property that the inequality $N_{L(p)}(R) \geq \lambda|R|$ holds if R is a subgraph of $L(p)$ satisfying $|R| < 71p/72$. Such λ exists because there is a constant $c > 0$ for which the inequality $|N(R)| \geq (1 + c(1 - |R|/p))|R|$ holds for an arbitrary set R of nodes (see [1]).

Lemma 1. *For every $f < p$ there exists a positive integer ℓ such that graph $L(p)^\ell$ has* PROPERTY $\mathcal{R}$ *with $\alpha = 1/7$, $\beta = 30$ and $\gamma = 2$.*

Proof. If $p - f \geq 71p/72$ then we may take $\ell = 1$ and rely on the construction in [20]. Suppose that $f > p/72$. Take as k the smallest positive integer satisfying $(p-f)\lambda^k \geq 71p/72$, and let $\ell = 2k+1$. Let R be a subgraph of graph $L(p)^\ell$ of size at least $p-f$. Since $|N_{L(p)^k}(R)| \geq 71p/72$ holds, we have $|P'(N_{L(p)^k}(R))| \geq |N_{L(p)^k}(R)|/6$. Let A denote the subgraph $P'(N_{L(p)^k}(R)) \cap R$ and B the subgraph $P'(N_{L(p)^k}(R)) \setminus R$. Define $P(R)$ to be the subgraph $P(R) = N_{L(p)^k}(A \cup B) \cap R$. Note that $P(R)$ is defined as a set of vertices in graph $L(p)^\ell$ and is treated as a subgraph of $L(p)^\ell$ induced by these vertices.

One can see that $(\mathcal{R}.1)$ holds by the definitions of $P(R)$.

To prove ($\mathcal{R}$.2) we estimate $|P(R)|$. Suppose, to the contrary, that $|R \setminus P(R)| \geq 6|R|/7$. Then $|N_{L(p)^k}(R \setminus P(R))| \geq 6(p-f)\lambda^k/7 \geq 6 \cdot 71p/(7 \cdot 72) > 5p/6$, which implies that $N_{L(p)^k}(R \setminus P(R)) \cap N_{L(p)^k}(P'(N_{L(p)^k}(R))) \neq \emptyset$. This contradicts the fact that $R \setminus P(R)$ and $P(R)$ are disjoint.

Next we prove ($\mathcal{R}$.3). Let v_1 and v_2 be two vertices in subgraph $P(R)$ of $L(p)^\ell$. Consider two vertices v_1 and v_2 from $P(R)$. Suppose that they are from $P'(N_{L(p)^k}(R))$. There is a path between node v_1 and node v_2 in subgraph $P'(N_{L(p)^k}(R))$ of $L(p)$ of length at most $30 \log p$. Let $v_1 = w_1, w_2, \ldots, w_m = v_2$ be a sequence of consecutive nodes on this path. We show that there is a path between node v_1 and node v_2 of the same length in subgraph $P(R)$ of $L(p)^\ell$, details are omitted. The case when either v_1 or v_2 are not in $P'(N_{L(p)^k}(R))$ is straightforward, as both of them have neighbors in $P'(N_{L(p)^k}(R))$ in subgraph $L(n)^k$, by the inclusion $P(R) \subseteq N_{L(p)^k}(P'(N_{L(p)^k}(R)))$. It follows that the distance between them in the subgraph $P(R)$ of $L(p)^\ell$ is at most $2 + 30 \log p$.

To prove ($\mathcal{R}$.4) let us consider $R_1 \subseteq R_2$, both subgraphs of sizes at least $p - f$. For any subsets of nodes A and B, let A_1 and B_1 denote the subgraphs induced by A and B, respectively, in subgraph R_1. Similarly, let A_2 and B_2 denote the subgraphs induced by A and B, respectively, in subgraph R_2. It is clear that $N_{L(p)^k}(R_1) \subseteq N_{L(p)^k}(R_2)$. If $X \subseteq Y$ are sets of nodes such that $|X| \geq 71p/72$ then $P'(X) \subseteq P'(Y)$, see [20]. It follows that inclusions $A_1 \subseteq A_2$ and $B_1 \subseteq B_2 \cup (A_2 \setminus A_1)$ hold. We obtain the final inclusion $P(R_1) \subseteq P(R_2)$, which completes the proof. $\square$

Lemma 1 has been used recently by Chlebus and Kowalski [5] in their work on fault-tolerant gossiping and consensus algorithms.

4 Generic algorithm

A processor that has not halted nor crashed is called *active*. We say a processor is *busy* at a certain step if its local knowledge does not imply that all tasks have been done. Our algorithms have the property that if some tasks are still outstanding then all the operational processors are both active and busy: they all iterate the main phase of the algorithm, with the goal of having all the tasks performed.

Recall that a subgraph of the communication graph is compact if its size is at least $(n-t)/7$ and its diameter is at most $30 \log p + 2$, see Lemma 1. Let the *neighborhood* of a processor v be the subgraph containing each operational processor whose distance from v in the communication graph is at most $30 \log p + 2$. A processor is said to be *compact* if its neighborhood is such. These notions are dynamic, since the status operational/faulty of a processor my change over time, so the status of being compact may be lost at some point of computation.

Local state. Each processor v maintains a local state consisting of three ordered lists and additional variables. The list $\texttt{Tasks}_v$ contains the tasks assumed to be outstanding, the list $\texttt{Processors}_v$ contains the identifiers of the processors assumed to be active, and list $\texttt{Busy}_v$ contains the identifiers of the processors assumed to be busy performing tasks. These lists are initially ordered by the identifiers of the respective items. The position of an item in a list is its *rank* in the list. Items may be removed from lists, which affects the ranks of the remaining items. Each item u on list $\texttt{Processors}_v$ has an associated *distance variable* that stores the distance from processor v to u in the local view of the communication graph.

Each processor v maintains a counter (of phases) in its variable $\texttt{Counter}_v$. Additionally, processor v maintains variable $\texttt{Done}_v$, which remains false until v learns that all tasks have been performed. Variable $\texttt{Selected\_Processor}_v$ is used to remember a processor. Variable $\texttt{Phase}_v$ contains the name of the current phase performed by the processor; this name is either `Main` or `Closing`.

Messages. A message sent by processor v includes its identifier and the following components of its state: $\texttt{Tasks}_v$, $\texttt{Processors}_v$, and $\texttt{Busy}_v$. Additionally, the signal `stop` may be included if needed.

Handling lists. When in the course of a computation an item needs to be selected from a list, then this is done using some `Selection_Rule`. In Section 5 we define selection rules to instantiate specific algorithms.

Each active processor v updates the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$ after receiving messages from its neighbors in the current communication graph. If an item is not on the copy of a list received from some neighbor then it is removed from the corresponding local list. If no message is received from a neighboring processor u then u is removed from the list $\texttt{Processors}_v$. If signal `stop` is received from processor u or if processor u was chosen as the $\texttt{Selected\_Processor}_v$ then u is removed from the list $\texttt{Busy}_v$.

Distances among processors. Processors estimate distances among them in the communication graph using their local views of the graph. The distances may change during the computation due to failures. In each phase the distance variables attached to the items in the lists `Processors` are updated: for each processor v, the minimum of the distance variables attached to v on all the received processors lists is taken and incremented by one.

We use abbreviation $g(p)$ for function $30\log_2 p + 2$. During a phase each processor v estimates the size of its neighborhood. This is done by comparing the distance variables on the updated $\texttt{Processors}_v$ list with $g(p)$: if the value of the distance variable is not greater than $g(p)$ then the corresponding processor is included in the neighborhood of v. If the estimated size of the neighborhood of processor v is at least $(p-f)/7$ then v is said *to believe to be compact.*

Structure of the generic algorithm. The algorithm consists of the initialization and a loop that iterates through one of the two phases.

The phases are of two kinds: Main Phase and Closing Phase. Each phase consists of three stages: (1) receiving messages, (2) local computation, and (3) multicasting messages. The goal of the Main Phase is to have all the tasks completed. The Main Phase is given in Figure 1.

Once a processor detects an empty list `Tasks`, it switches to the Closing Phase. Now the goal is to have all the active processors informed that the tasks have already been completed. This is accomplished in a similar manner: informing a processor is treated as a "task." To perform a "task" of this kind, an item

STAGE 1: Receive messages.
STAGE 2: Perform local computations:

Update the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$.

Update the distance variables, attached to the processors in the list $\texttt{Processors}_v$, and estimate the size of the neighborhood.

If some processor is in list $\texttt{Processors}_v$ and not in list $\texttt{Busy}_v$ then set variable $\texttt{Done}_v$ to true.

If list $\texttt{Tasks}_v$ is nonempty then select a task from it by applying `Selection_Rule`, remove the task from list $\texttt{Tasks}_v$, perform the task, and increment $\texttt{Counter}_v$.

If list $\texttt{Tasks}_v$ is empty then set $\texttt{Done}_v$ to true and switch $\texttt{Phase}_v$ to `Closing`.

If either ($\texttt{Done}_v$ is true and v does not believe to be compact) or ($\texttt{Counter}_v > t$) then halt.

STAGE 3: Multicast messages to these neighbors in the communication graph that are still in the list $\texttt{Processors}_v$.

Fig. 1. Generic algorithm: Main Phase. Code for processor v.

from the list `Busy` is selected, then removed from the list, and signal `stop` is sent to it. In the Closing Phase the list `Tasks` is no longer needed.

Lemma 2. *The generic algorithm is correct against the unbounded adversary.*

Lemma 3. *If an algorithm uses only the graph $L(p)^\ell$ for communication, where ℓ is minimal such that Lemma 1 holds, and if its work complexity is $\mathcal{W}$, then its communication cost is equal to $\mathcal{M} = \mathcal{O}(\mathcal{W} \cdot (p/(p-f))^{2\log_\lambda \Delta_0})$.*

5 Specific algorithms

In this section we instantiate several algorithms obtained from the generic algorithm of Section 4 by specifying the following: (1) the patterns used to select tasks to be performed at a step, and (2) the messaging patterns using communication graphs. The performance of each algorithm is analyzed against specific adaptive size-bounded adversaries. We bound the maximum number of failures using parameter f. This number in turn determines the exponent ℓ of the power of the graph $L(p)$ we need for communication in order to establish the specific performance results.

For the purpose of analysis of the performance of specific instantiations of the generic algorithm, we partition the computation into disjoint epochs: each *epoch* $\mathcal{E}_i$ is defined to consist of $g(p) = 30\log_2 p + 2$ consecutive phases. An epoch has sufficient duration for all processors in a compact subgraph to communicate.

Throughout the proofs, the notation G_i means the subgraph of communication graph $G(p)$ induced by the vertices that are active during the first phase of epoch $\mathcal{E}_i$. Note that $G_{i+1} \subseteq G_i$ for $i \geq 1$. We let H_i stand for a compact subgraph of G_i in G, such that $H_{i+1} \subseteq H_i$ for $i \geq 1$. We resort to PROPERTY $\mathcal{R}$ to define H_i.

In every epoch $\mathcal{E}_i$ there is a compact node in G_i. This fact, in terms of communication, means that in every step there are $\Omega(p-f)$ operational processors which can communicate with each other during the next $g(p)$ phases, provided all of them are still active. If a processor halts then its list `Busy` is empty and this information needs $g(p)$ phases to proliferate. It follows that there are $\Omega(p-f)$ processors, which stay active till the end of the computation, possibly with the exception of the last $g(p)$ steps, and can communicate with each other with the delay of at most $g(p)$ phases.

A specific deterministic algorithm and a given adversary determines graphs G_i induced by active nodes during the first phase of epoch $\mathcal{E}_i$. If the adversary crashes processor nodes depending on the random decisions to select items from their lists then graphs G_i depend on the execution. In this case graphs H_i also depend on the execution.

Let $T_{v,i}$ be the set of elements in $\texttt{Tasks}_v$ during the first phase of $\mathcal{E}_i$. We use the following notations: U_i stands for $\bigcup_{v\in H_i} T_{v,i}$, S_i for $\bigcap_{v\in H_i} T_{v,i}$, and $u_i = |U_i|$, $s_i = |S_i|$.

5.1 Balancing algorithm BAL

Algorithm BAL is instantiated from the generic algorithm by defining the rule `Load_Balance` used to select items by load balancing. Let q be the number of elements in the list at hand, then processor v selects the item of rank $r(v)$, where the number $r(v)$ is uniquely determined by the inequalities

$$p(r(v)-1)/q < v < p(r(v)+1)/q \ . \qquad (1)$$

We now analyse the performance of the algorithm.

Lemma 4. *If $u_i \geq 11p^2 g(p)$ then $u_i - u_{i+1} \geq |H_{i+1}| g(p)$.*

Lemma 5. *Let $u = u_{i_0}$. For $i > i_0$, if during epoch $\mathcal{E}_i$ all the processors in H_{i+1} are in the Main Phase and $u_i > \sqrt{\frac{u}{\log u}}$, then $u_i - u_{i+1}$ is either at least $\min\left\{\frac{1}{14p}\sqrt{\frac{u}{\log u}}\,, 1\right\}\cdot|H_{i+1}|$ or at least $u_i \cdot \sqrt{\frac{\log u}{u}}$.*

Theorem 1. *The Do-All problem can be solved by a constructive deterministic algorithm with work $\mathcal{W} = \mathcal{O}\big(t + p\sqrt{p+t}\ \log^{3/2} p\big)$ against adaptive f-bounded adversaries, for any number of faults $f < p$.*

Proof. If the number of active processors in H_i decreases by more than twice in $\mathcal{E}_i$ then $\mathcal{E}_i$ is called *stormy*, otherwise it is *peaceful*. Since the total work accrued during stormy epochs is $\mathcal{O}(p\log p)$, we may assume in the rest of the proof that all the epochs are peaceful. Let H be a set of processors such that all its elements remain active during at least one full (peaceful) epoch, starting from some $\mathcal{E}_m$, where $H \subseteq H_m$, and $|H| \geq |H_m|/2$.

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

We estimate the sizes of intervals I_j. By Lemma 4, during epochs in I_1 all processors in H perform different tasks, hence $|I_1| = \mathcal{O}(t/(|H|g(p)))$. During I_2 all processors from H are in the Main Phase. Let i_0 be the minimal element in I_2. Let $x = \sqrt{u/\log u}$, where $u = u_{i_0}$. We further partition I_2 into $I_2 = I_2^1 \cup I_2^2 \cup I_2^3$ as follows. Number $i \in I_2^1$ iff $u_i \leq x$. Let the remaining numbers of epochs in I_2 satisfy the following: if $i \in I_2^2$ then $u_i - u_{i+1} \geq u_i/x$, and if $i \in I_2^3$ then

$$u_i - u_{i+1} \geq \min\left\{\frac{x|H|}{14p}, |H|\right\} .$$

Lemma 5 guarantees that such a partition of I_2 is possible. We estimate the sizes of components of I_2: $|I_2^1| = \mathcal{O}(x/g(p)) = \mathcal{O}(\sqrt{t/\log t}/g(p))$. If $i \in I_2^2$ then $u_{i+1} \leq u_i(1-1/x)$ so the size of I_2^2 is $\mathcal{O}(\sqrt{u\log u}) = \mathcal{O}(\sqrt{t\log t})$, because $u(1-1/x)^{x\log u} = \mathcal{O}(1)$. The size of I_2^3 is

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

because $u < 11p^2 g(p)$. It follows that $|I_2| = \mathcal{O}((\sqrt{t(g(p)+\log t)})\cdot p/|H|)$.

The cases of I_4 and I_5 are omitted. The whole work contributed by H is $\mathcal{O}\big(|H|g(p)\sum_{j=1}^{5}|I_j|\big)$, which yields the bound

$$\mathcal{O}\left(t + p\sqrt{(p+t)(\log p + \log t)}\log p\right) \tag{2}$$

by our estimates. This bound was proved under the assumption that the contributing processors are in the same set H. In reality, set H may evolve through sets that get smaller at certain epochs. Then, as the next set H takes over, we are still within the bound (2) because it does not depend on the size of H. □

Corollary 1. *The Do-All problem can be solved by a constructive deterministic algorithm with effort* $\mathcal{W}+\mathcal{M} = \mathcal{O}(t + p\sqrt{t+p}\log^{3/2} p)$ *against adaptive linearly-bounded adversaries.*

Proof. The algorithm in Theorem 1 uses a constant-degree network for communication, and each processor sends $\mathcal{O}(1)$ messages in each step, by Lemma 3. □

5.2 Permutation algorithms RPAL and DPAL

We now explore another selection paradigm where processors choose items from lists in the order given by their private permutations. We consider randomized algorithm RPAL and a scheme of deterministic algorithms DPAL parametrized by families of permutations.

The new selection rule `Select_According_To_Permutations` operates as follows. Suppose each processor v has two private permutations, π_1 over $[1..p]$ and π_2 over $[1..11p^2g(p)]$. Processor v starts by permuting the list $\mathtt{Busy}_v$ according to π_1. If it needs to select an item from $\mathtt{Busy}_v$ then it takes the first one in the list. Tasks are selected the same way as in rule `Load_Balance` as long as the size of $\mathtt{Tasks}_v$ is greater than $11p^2g(p)$. As soon as exactly $11p^2g(p)$ items remain in $\mathtt{Tasks}_v$, processor v permutes the list $\mathtt{Tasks}_v$ according to π_2. Then, at any time when processor v needs to select an item from $\mathtt{Tasks}_v$, it simply takes the first one in the list.

To specify the permutations required by the selection rule, we consider two alternatives. The randomized algorithm RPAL starts by each processor selecting two permutations at random, independently among all the processors. The deterministic algorithm DPAL equips each processor with a fixed pair of permutations. Algorithm DPAL is more efficient than the deterministic algorithm BAL, however the implementation of BAL is immediate, whereas algorithm DPAL is not constructive. Once the permutation is selected in an execution of algorithm RPAL, the future behavior of the adversary is determined by a *failure pattern* F consisting of a subset R of processors of size f, and for each processor $v \in R$ the time step during which it fails. Let H denote a compact subgraph $H \subseteq G$ of size $\Omega(p-f)$. We let $\mathcal{F}_H$ denote the set of all the failure patterns such that processors in H do not crash.

Lemma 6. *Suppose the adversary can select only the failure patterns from $\mathcal{F}_H$. If $u_i \geq a$ then the inequality $u_i - u_{i+1} \geq (1/4)\min\{a, |H|g(p)\}$ holds with the probability at least $1-\exp(-\Omega(|H|\log p))$.*

Lemma 7. *If the adversary can select only the failure patterns from $\mathcal{F}_H$ then the number of epochs in an execution of* RPAL *is $\mathcal{O}((t+p)/(|H|g(p)) + \log p)$ with the probability at least $1-\exp(-\Omega(|H|\log p))$.*

Theorem 2. *There is a deterministic algorithm that solves the Do-All problem with work $\mathcal{W} = \mathcal{O}\big(t + p\log^2 p\big)$ against the adaptive f-bounded adversary, if $p - f = \Omega(p^b)$, for a constant $b > 0$.*

Proof. Consider an execution of algorithm RPAL. Let $H_0 = G, H_1, \ldots, H_m$, for $m = \lceil \log \frac{p}{p-f} \rceil$, be a sequence of subgraphs such that $H_{j+1} \subseteq H_j$ and $|H_{j+1}| = |H_j|/2$ hold, and additionally, at every step of computation of algorithm RPAL the set H of operational processors satisfies the condition $H_j \subseteq H \subseteq H_{j-1}$, for some j where $1 \leq j \leq m$. Such a subsequence H_i depends on the random permutations chosen randomly at the start of RPAL.

Consider a set H, such that $H_j \subseteq H \subseteq H_{j-1}$ for some $j = 1, \ldots, m$, and H remains operational after phase l, while after phase $l-1$ the set of operational processors included set H_{j-1}. The sets $H_{j+1}, \ldots, H_m$ cannot be determined by step l. For a fixed set H we can estimate the number of possible components of size $|H_j| \geq |H|/2$ as follows:

$$\binom{|H|}{|H_j|} \leq \binom{|H|}{|H|/2} \leq 2^{|H|\cdot h(1/2)} \leq 2^{|H_{j-1}|\cdot h(1/2)} ,$$

where $h(x)$ is the binary entropy function. The fraction of families of permutations that result in only $\mathcal{O}((t+p)/(|H|g(p))+\log p) = \mathcal{O}((t+p)/(|H_j|g(p))+\log p)$ epochs of algorithm RPAL, while it is is executed, is at least

$$1 - 2^{|H_{j-1}|h(1/2)} \exp(-\Omega(|H|g(p))) = 1 - \exp(-\Omega(|H|g(p)))$$

by Lemma 7. Hence the part of computation during which the set of operational processors includes H_j and is included in H_{j-1} takes at most $\mathcal{O}((t+p)/(|H_j|g(p)) + \log p)$ epochs, for a fraction of at least $1 - \exp(-\Omega(|H|g(p)))$ of the families of initial permutations. It follows that a fraction of the number of families of permutations that satisfy the claim of the theorem is at least

$$1 - \sum_{j=1}^{m} \exp(-b|H_j|g(p)) \geq 1 - \sum_{j=1}^{m} \exp(-b(p-f)g(p)) \geq 1 - \exp(-c(p-f)g(p))$$

for some constants $b, c > 0$. Algorithm DPAL using one of such families performs work at most $\max_{j \leq m} \{d((t+p)/(|H_j|g(p)) + \log p) \cdot |H_{j-1}|g(p)\} = \mathcal{O}(t + p\log^2 p)$ for some constant $d > 0$, by an argument similar to that used in the proof of Theorem 1. □

Corollary 2. *There is a deterministic algorithm that solves Do-All with effort* $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p\log^2 p)$ *against adaptive linearly-bounded adversaries.*

5.3 Hybrid algorithm UAL

We now present algorithm UAL that is efficient against the unbounded adversary. The algorithms consists of two parts: *Part 1* and *Part 2*. *Part 1* is based on the permutation algorithm DPAL that completes its job if the number of faults f is at most some f_0. It uses the graph $L(p)^\ell$ as its communication graph, where ℓ is set to some ℓ_0. The original t tasks are partitioned into chunks of size Δ, where Δ is the node degree of graph $L(p)^\ell$. Then each of the chunks is treated as a single task by DPAL, in the sense that a communication is performed only after the chunk is completed. This guarantees that the communication of *Part 1* is the same as its work. The *Part 1* algorithm is performed for some T_0 steps. After that all the processors that have not halted yet, if any, run the *Part 2* algorithm. *Part 2* starts by all the active processors sending messages to each other to learn who is alive and what tasks are still outstanding. The processors reset the lists of processors and tasks accordingly, containing, say, at most p_1 processors and t_1 tasks. Then algorithm DMY of De Prisco, Mayer and Yung [10], is run. The work of algorithm DMY is $\mathcal{O}(t_1+(f_1+1)p_1)$ and its communication is $\mathcal{O}((f_1+1)p_1)$, where f_1 is the number of faults that occur during this phase.

The parameters f_0, ℓ_0 and T_0 are selected so as to make our estimates of work of *Part 1* and *Part 2* approximately equal. Let us fix $a>0$ such that also the inequality

$$a < 1 - \frac{2\log_\lambda \Delta_0}{1+2\log_\lambda \Delta_0}$$

holds. We set the parameters of algorithm UAL as follows:

$$f_0 = p - p^{1-a} \ , \qquad \ell_0 = 2\log_\lambda \frac{p}{p-f_0} + 1 \qquad \text{and} \qquad T_0 = \frac{t+\Delta_0^{\ell_0} p \log^2 p}{p-f_0} \ . \tag{3}$$

The work and communication of the resulting algorithm UAL is $\mathcal{W}+\mathcal{M} = \mathcal{O}(t+p^{2-a})$, as stated in Theorem 3.

Theorem 3. *There is a deterministic algorithm that solves the Do-All problem with effort* $\mathcal{W}+\mathcal{M} = \mathcal{O}(t+p^{2-a})$ *against the unbounded adversary, where* $a>0.23$ *is a constant.*

Proof. The effort of *Part 1* is $\mathcal{O}(t+\Delta p\log^2 p)$ and that of *Part 2* is $\mathcal{O}(t+p(p-f))$, where $f<p$ is a variable. Equating these two bounds we obtain

$$\Delta \log^2 p = p - f \ . \tag{4}$$

As in the proofs of Lemmas 1 and 3, the node degree of the power of graph $L(p)$ is $\Delta = \Delta_0^k$, where $k = 2\log_\lambda \frac{p}{p-f}+1$, provided that $k+1$ is not bigger than the girth of $L(p)$, that is, the length of the shortest cycle. It was shown in [17] that this girth is at least $\frac{1}{2}\log_{\Delta_0} p$. The inequality

$$2\log_\lambda \frac{p}{p-f_0} + 1 < \frac{1}{2}\log_{\Delta_0} p$$

can be checked by inspection, so we can proceed.

Since $\log_\lambda \frac{p}{p-f} = \log_{\Delta_0} \frac{p}{p-f} \cdot \log_\lambda \Delta_0$, we can convert equation (4) into an equivalent form

$$\Delta_0 \left(\frac{p}{p-f}\right)^{2\log_\lambda \Delta_0} = \frac{p-f}{\log^2 p} \ ,$$

which is then equivalent to $\Delta_0 p^{2\log_\lambda \Delta_0}\log^2 p = (p-f)^{1+2\log_\lambda \Delta_0}$. Solving for f we obtain $f_0 = p - p^{1-a}$, where $a = \frac{1}{1+2\log_\lambda \Delta_0} + o(1)$. The effort of algorithm UAL is

$$\mathcal{O}(t+\Delta p \log^2 p) = \mathcal{O}(t+p(p-f_0)) = \mathcal{O}(t+p^{2-a}) \ .$$

The estimates in [3, 19, 20] imply that with $\Delta_0 = 70$ the constant λ is at least as large as $7/2$. If the constant a and parameters f_0, ℓ_0 and T_0 in equations (3) are set with $7/2$ as the value of λ, then one can verify that $a > 0.23$. □

6 Discussion

We presented the first solution for *Do-All* that has both work *and* communication *subquadratic* in p in the worst case. There remains a substantial gap between our upper bound and the best known lower bound on work of $\Omega(t+p\log p/\log\log p)$. Narrowing this gap, in particular, improving the lower bound, is challenging.

References

1. N. Alon, Eigenvalues and expanders, *Combinatorica*, 6 (1986) 83 - 96.
2. N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, Scalable secure storage when half the system is faulty, in *Proc., 27th Int. Colloquium on Automata, Languages and Programming*, 2000, Springer LNCS 1853, pp. 577 - 587.
3. N. Alon, and V.D. Milman, λ_1, isoperimetric inequalities for graphs, and superconcentrators, *J. Combin. Theory Ser. B*, 38 (1985) 73 - 88.
4. B.S. Chlebus, R. De Prisco, and A.A. Shvartsman, Performing tasks on synchronous restartable message-passing processors, *Distributed Computing*, 14 (2001) 49 - 64.
5. B.S. Chlebus, and D.R. Kowalski, Gossiping to reach consensus, in *Proc., 14th ACM Symp. on Parallel Algorithms and Architectures,* 2002, Winnipeg, Manitoba, Canada, to appear.
6. B.S. Chlebus, and D.R. Kowalski, Randomization helps to perform independent tasks reliably, submitted. A preliminary version appeared as: Randomization helps to perform tasks on processors prone to failures, in *Proc. 13th Int. Symposium on Distributed Computing*, 1999, Springer LNCS 1693, pp. 284 - 296.
7. B.S. Chlebus, D.R. Kowalski, and A. Lingas, The do-all problem in broadcast networks, in *Proc., 20th ACM Symp. on Principles of Distributed Computing,* 2001, Newport, Rhode Island, pp. 117 - 126.
8. A. Clementi, A. Monti, and R. Silvestri, Optimal f-reliable protocols for the do-all problem on single-hop wireless networks, in *Proc., 13th Int. Symposium on Algorithms and Computation,* 2002, to appear.
9. H. Davenport, " ," 2nd ed. revised by H.L. Montgomery, Springer Verlag, Berlin, 1980.
10. R. De Prisco, A. Mayer, and M. Yung, Time-optimal message-efficient work performance in the presence of faults, in *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994, pp. 161 - 172.
11. K. Diks, and A. Pelc, Optimal adaptive broadcasting with a bounded fraction of faulty nodes, *Algorithmica*, 28 (2000) 37 - 50.
12. C. Dwork, J. Halpern, and O. Waarts, Performing work efficiently in the presence of faults, *SIAM J. on Computing*, 27 (1998) 1457 - 1491.
13. A. Fiat and J. Saia, Censorship resistant peer-to-peer content addressable networks, in *Proc., 13th ACM-SIAM Symp. on Discrete Algorithms*, 2002.
14. Z. Galil, A. Mayer, and M. Yung, Resolving message complexity of byzantine agreement and beyond, in *Proc. 36th IEEE Symp. on Foundations of Computer Science*, 1995, pp. 724 - 733.
15. C. Georgiou, A. Russell, and A.A. Shvartsman, The complexity of synchronous iterative do-all with crashes, in *Proc., 15th Int. Symposium on Distributed Computing,* 2001, Springer LNCS 2180, pp. 151 - 165.
16. P.C. Kanellakis, and A.A. Shvartsman, Efficient parallel algorithms can be made robust, *Distributed Computing*, 5 (1992) 201 - 217.
17. A. Lubotzky, R. Phillips, and P. Sarnak, Ramanujan graphs, *Combinatorica*, 8 (1988) 261 - 277.
18. G.A. Margulis, Explicit constructions of concentrators, *Prob. Per. Infor.* 9 (1973) 71 - 80; English transl. in *Problems of Information Transmission*, (1975) 325 - 332.
19. R.M. Tanner, Explicit concentrators from generalized N-gons, *SIAM J. Alg. Disc. Methods*, 5 (1984) 287 - 293.
20. E. Upfal, Tolerating a linear number of faults in networks of bounded degree, *Information and Computation*, 115 (1994) 312 - 320.