Performing Work with Asynchronous Processors: Message-Delay-Sensitive Bounds*

Dariusz R. Kowalski[1,2]
kowalski@cse.uconn.edu

Alex A. Shvartsman[1,3]
aas@cse.uconn.edu

[1]Computer Science & Engineering
University of Connecticut
Storrs, CT 06269

[2]Instytut Informatyki
Uniwersytet Warszawski
Warszawa, Poland

[3]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

This paper considers the problem of performing tasks in asynchronous distributed settings. This problem, called Do-All, has been substantially studied in synchronous models, but there is a dearth of efficient algorithms for asynchronous message-passing processors. Do-All can be trivially solved without any communication by an algorithm where each processor performs all tasks. Assuming p processors and t tasks, this requires work $\Theta(p \cdot t)$. Thus it is important to develop subquadratic solutions (when p and t are comparable) by trading computation for communication. Following the observation that it is not possible to obtain subquadratic work when the message delay d is substantial, e.g., $d = \Theta(t)$, this work pursues a *message-delay-sensitive* approach. Here the upper bounds on work and communication are given as functions of p, t, and d, the upper bound on message delays, however algorithms have no knowledge of d and they cannot rely on the existence of an upper bound on d. This paper presents two families of asynchronous algorithms achieving, for the first time, *subquadratic work* as long as $d = o(t)$. The first family uses as its basis a shared-memory algorithm without having to emulate atomic registers assumed by that algorithm. The second family uses specific permutations of tasks, with certain combinatorial properties, to sequence the work of the processors. Another important contribution in this work is the first *delay-sensitive lower bound* for this problem that helps explain the behavior of our algorithms.

1. INTRODUCTION

The effectiveness of distributed computing critically depends on the ability of multiple processors to cooperate on a common set of tasks. The need for such cooperation exists in several application domains, including grid computing, distributed simulation, multi-agent collaboration, and distributed search, such as SETI. In their seminal work [9], Dwork, Halpern and Waarts abstracted a distributed collaboration problem in terms of p message-passing processors that need to perform t idempotent and computationally similar tasks. We call this problem Do-All. The efficiency of Do-All algorithms is measured in terms of the *work* complexity W (one task consumes one work unit), and the cost of communication is measured in terms of the *message* complexity M (in some settings efficiency is also measured in terms of *effort*, defined as the sum $W + M$).

Common impediments to effective coordination in distributed settings include failures and asynchrony that manifests itself, e.g., in disparate processor speeds and varying message latency. Fortunately, the Do-All problem can always be solved as long as at least one processor continues to make progress. In particular, with the standard assumption that initially all tasks are known to all processors, the problem can be solved by a communication-oblivious algorithm where each processor performs all tasks. Such a solution has work $W = O(t \cdot p)$, and requires no communication. On the other hand, $\Omega(t)$ is the obvious lower bound on work and the best known lower bound is $W = \Omega(t + p \log p)$, e.g., [13, 15]. Therefore the trade-off expectation is that if we increase the number of messages we should be able to decrease the amount of work so that it is sub-quadratic in t and p.

In the message-passing settings, the Do-All problem has been substantially studied for synchronous failure-prone processors under a variety of assumptions, e.g., [5, 6, 7, 10, 11]. However there is a dearth of algorithms for asynchronous models. This is not that surprising. For an algorithm to be interesting, it must be better than the oblivious algorithm, in particular, it must have sub-quadratic work complexity. However, if messages can be delayed for a "long time", then the processors cannot coordinate their activities, leading to an immediate lower bound on work of $\Theta(p \cdot t)$, even if only one processor crashes. In particular, it is sufficient for messages to be delayed by $\Theta(t)$ time for this lower bound to hold. Thus it is interesting to develop algorithms that are correct for any pattern of asynchrony and failures (with at least one surviving processor), and whose work depends on the message latency upper bound (cf. [8]), such that work increases gracefully as the latency grows. The quality of the algorithms can be assessed by comparing their work to the corresponding delay-sensitive lower bounds.

*The work of the first author is supported in part by the NSF-NATO Award 0209588. The work of the second author is supported in part by the NSF CAREER Award 9984778 and by the NSF Grants 9988304 and 0121277.

One approach to obtaining message-passing algorithms for Do-All is based on emulating asynchronous shared-memory algorithms, such as the algorithms of Anderson and Woll [2] and Buss *et al.* [4]. For example, Momenzadeh [18] shows how to do this using replicated linearizable memory services that rely on quorum systems (or majorities), e.g., [3, 17]. This approach yields Do-All algorithms for asynchronous message-passing processors with sub-quadratic work complexity, however it is assumed that message delays are constant, and that quorum systems are not disabled by failures. When processor failures damage quorum systems, the work of such algorithms becomes quadratic, even if message latency is constant.

1.1 Contributions

Our goal is to obtain complexity bounds for work-efficient message-passing algorithms for the Do-All problem:
Given t similar and idempotent tasks, perform the tasks using p asynchronous message-passing processors.
We require that the algorithms tolerate any pattern of processor crashes with at least one surviving processor. Equally important, we are interested in algorithms whose work degrades gracefully as a function of the worst case message delay d. Here the requirement is that work must be sub-quadratic in t and p as long as $d = o(t)$. Thus for our algorithms we aim to develop *delay-sensitive* analysis of work and message complexity. Noting again that work must be $\Omega(p \cdot t)$ for $d \geq t$, we show that work need not exceed $O(p \cdot t)$. More interestingly, we give a comprehensive analysis for $d < t$.

In this paper we present the first delay-sensitive lowers bound for Do-All and the first asynchronous algorithms for Do-All that meet our criteria for fault-tolerance and efficiency. The summary of our results, stated here for $d < t$, is as follows:

1. We present delay-sensitive lower bounds for the Do-All problem for deterministic and randomized algorithms with asynchronous processors. Any deterministic (randomized) algorithm with p asynchronous processors and t tasks has worst-case work (expected work) of $\Omega(t + pd \log_{d+1} t)$, where d is the upper bound on message delay (unknown to the processors).

2. We present a family of deterministic algorithms DA, parameterized by a positive integer q and a list Ψ of q permutations on the set $[q] = \{1, \ldots, q\}$, where $2 \leq q < p \leq t$. We show that for any constant $\varepsilon > 0$ there is a constant q such that the corresponding algorithm has work $W = O(tp^{\varepsilon} + pd\lceil t/d \rceil^{\varepsilon})$ and message complexity $M = O(p \cdot W)$.

3. We present a family of algorithms PA, deterministic and randomized, parameterized by a list Ψ of p permutations on the set $[p] = \{1, \ldots, p\}$, where $p \leq t$.

   The randomized algorithms have expected work $W = O(t \log p + pd \log(2 + t/d))$ and expected message complexity $M = O(tp \log p + p^2 d \log(2 + t/d))$.

   We show that there exists a deterministic list of schedules Ψ such that the deterministic algorithm has work $W = O(t \log p + pd \log(2+t/d))$ and message complexity $M = O(tp \log p + p^2 d \log(2 + t/d))$.

The deterministic algorithm in PA has somewhat better work than the deterministic algorithm in DA, however it requires substantially larger permutations, the former requiring permutations of set $[q]$ and the latter of set $[p]$.

Algorithms in the family DA are modeled after the shared-memory algorithm of Anderson and Woll [2], and use a list of q permutations in the same way. The two main differences are: *(i)* instead of maintaining a global data structure representing a q-ary tree, in our algorithms each processor has a replica of the tree, and *(ii)* instead of using atomic shared memory to access the nodes of the tree, processors read values from the local tree, and instead of writing to the tree, processors multicast the tree; the local data structures are updated when multicast messages are received.

1.2 Related work

A number of algorithms solving the Do-All problem in the synchronous message-passing settings have been developed [5, 9, 10, 11, 12]. The algorithmic techniques in these papers rely on processor synchrony and assume constant-time message delay. It is not clear how such algorithms can be adapted to asynchrony. Anderson and Woll provided an asynchronous algorithm for a shared-memory model [2].

Our algorithm DA is a re-interpretation of the shared-memory algorithm [2] in the message-passing setting. Recall that in our algorithm a processor multicasts a message instead of writing a value to shared memory, and a processor consults local data structures instead of reading from shared memory. In this work we use (and extend) the notion of contention of permutations, introduced by Anderson and Woll. Contention helps to measure the number of tasks that are performed redundantly. We describe this in detail in Section 4, showing how reducing contention leads to lower work in certain algorithms.

Various approaches can be used to construct sets of permutations with low contention. When the q permutations on the set $[q]$ are chosen uniformly and independently at random, then contention is bounded by $O(q \log q)$ with high probability [2]. Anderson and Woll show how to search for these permutations, taking exponential in q processing time [2]. A different approach is given by Naor and Roth [19]. They show that a set of q permutations with contention $O(q^{1+\varepsilon})$ can be obtained such that each permutation can be computed in time $q \cdot \text{polylog}(q)$. The value of q for which the bound holds is exponential in $1/\varepsilon^3$.

1.3 Document structure

We define the model of computation and complexity measures in Section 2. In Section 3 we show the first delay-sensitive lower bounds for Do-All. In Section 4 we deal with permutations and contention. In Section 5 we present and analyze the first work-efficient asynchronous deterministic Do-All algorithm. In Section 6 we present and analyze two randomized and one deterministic algorithm that satisfy our efficiency criteria. We discuss our results and future work in Section 7.

2. MODEL AND DEFINITIONS

In our distributed setting we consider a system consisting of p processors with unique identifiers (pid) from the set $\{0, \ldots, p-1\}$. The processors must perform t tasks. A processor's activity is governed by its local clock. We model asynchrony as an adversary that introduces delays (possibly infinite) between local clock ticks. The processors are subject to crash failures, again determined by the adversary (we model crashes as infinite delays). The only restriction is that at least one processor is non-faulty. We assume that p and t are known to all processors. Processors communicate over a fully connected network by sending point-to-point messages via reliable asynchronous channels. When a processor sends a message to a group of processors, we call it a multicast message, however in the analysis we treat a multicast message as multiple point-to-point messages). Messages are subject to delays, but are not corrupted or lost.

For the purpose of algorithm analysis we assume the existence of a global real-timed clock that is unknown to the processors. For convenience we measure time in terms of units that represent the smallest possible time between consecutive clock-ticks of any processor. We assume that there exists an integer parameter d, that is not assumed to be a constant and that is unknown to the processors, such that messages are delayed by at most d time units. By the choice of the time units, a processor can take at most d local steps during any time period d.

The adversary that imposes message delays up to d time units is called the *d-adversary*. We also consider the *(d,σ)-adversary*, where σ is a permutation of t tasks, that is defined to be the d-adversary that schedules the asynchronous processors so that each of the t tasks is performed for the first time in the order given by σ. More precisely, if the execution of the task σ_i is completed for the first time by some processor at global time t_i (unknown to the processor), and the task σ_j, for any $1 \le i < j \le t$, is completed for the first time by some processor at time t_j, then $t_i \le t_j$. Note that any d and σ defines a family (d,σ)-adversaries, and any execution of an algorithm solving the Do-All problem corresponds to some (d,σ)-adversary for appropriate d and σ.

For this model we define the Do-All problem as follows: *Given t similar and idempotent tasks, perform the tasks using p asynchronous message-passing processors.*

We assess the work of algorithms by counting all processing steps performed by all processors [14]. We assume that it takes a unit of time for a processor to perform a unit of work according to its local clock. We also charge a unit of time per local step of any processor (whether the processor is idling or not, or halted voluntarily). We assume that it takes a unit of time to submit a broadcast request to the network (of course such messages are delivered after a delay is imposed by the adversary), and it takes a unit of work to process multiple received messages. For an execution $\mathcal{E}$ of an algorithm, we denote by $p_i(\mathcal{E})$ the number of processors completing a unit of work at time i of the computation (according to the global time that is not available to the processors). Let $\tau_{\mathcal{E}}$ be the time when all tasks have been performed and at least one processor is informed of this fact.

Definition 2.1. *For a p-processor algorithm that solves a given problem of size t the work complexity W is defined as* $W(p,t) = \max_{\forall \mathcal{E}} \left\{ \sum_{i \le \tau_{\mathcal{E}}} p_i(\mathcal{E}) \right\}$.

Message complexity is defined similarly. A single point-to-point message contributes 1 to the message complexity. When a processor broadcast a message to m destinations, this counts as m point-to-point messages in the message complexity. For an execution $\mathcal{E}$ of an algorithm, we denote by $m_i(\mathcal{E})$ the number of (point-to-point) messages sent at time i of the computation (according to some global time that is not available to the processors).

Definition 2.2. *For a p-processor algorithm that solves a given problem of size t the message complexity M is defined as* $M(p,t) = \max_{\forall \mathcal{E}} \left\{ \sum_{i \le \tau_{\mathcal{E}}} m_i(\mathcal{E}) \right\}$.

Now we formulate one observation that we use (without reference) in the rest of this paper. We say that a processor v *learns* that task z is performed by time τ if processor v has performed task z by itself by time τ or it received a message from processor w by time τ, and w learned about task z before it sent this message.

Proposition 2.1. *Let $\mathcal{A}$ be a Do-All algorithm such that there is some execution $\mathcal{E}$ of $\mathcal{A}$ in which there is a processor that (voluntarily) halts before until it learns that all tasks have been performed. Then there is an execution $\mathcal{E}_\infty$ of $\mathcal{A}$ with unbounded work in which some task is never performed.*

Proof. (Sketch) Assume that some processor v halts during execution $\mathcal{E}$ before it learns that some task z is performed. Then we construct execution $\mathcal{E}_\infty$ as follows: we delay all processors which are going to perform task z to infinity, effectively preventing the task z from being completed. The halted processor, however, contributes a unit to the work of the algorithm per each local clock tick. $\square$

Note that for large message delays the work of any Do-All algorithm is necessarily $\Omega(t \cdot p)$. The following proposition formalizes this lower bound and motivates our delay-sensitive approach.

Proposition 2.2. *Any algorithm that solves the Do-All problem in the presence of a $\Theta(t)$-adversary has work $W(p,t) = \Omega(t \cdot p)$.*

Since we are trading communication for work, we design algorithms with the focus on work, of course trying to keep communication as low as possible.

3. LOWER BOUND ON WORK FOR ASYNCHRONOUS ALGORITHMS

We now present delay-sensitive lower bounds for asynchronous algorithms for the Do-All problem.

3.1 Lower bound for deterministic algorithms

Theorem 3.1. *Any deterministic algorithm solving Do-All with t tasks using p asynchronous message-passing processors performs work $\Omega(t + p\min\{d,t\}\log_{d+1}(d+t))$ against the d-adversary.*

Proof. That the required work is at least t is obvious — each task must be performed. We may also assume that $t > 4$. We present the following adversarial strategy. The adversary partitions computation into stages, each containing $\min\{d, t/6\}$ steps. The adversary delivers all messages sent to a processor in stage i at the end of stage i (remind that the receiver can process such a message later, according to its own local clock); for stage i we are going to define the set of processors P_i such that the adversary delays all processors not in P_i.

Fix stage i. Let $u_i > 0$ be the number of tasks that remain unperformed at the beginning of stage i. We now show how to define the set P_i. Let $J_i(v)$, for every processor v, denote the set of at most $\min\{d, t/6\}$ tasks that are not performed by the end of stage $i-1$ and that would be performed by v in stage i, provided v is not delayed during stage i and does not receive any messages until the end of stage i.

We claim that by the pigeonhole principle, there are at least $\frac{u_i}{3\min\{d,t/6\}}$ tasks z such that each of them is contained in at most $2p\min\{d, t/6\}/u_i$ sets in the family $\{J_i(v) \mid v = 0, \ldots, p-1\}$. If did not then there would be more than $u_i - \frac{u_i}{3\min\{d,t/6\}}$ tasks such that each of them would be contained in more than $2p\min\{d, t/6\}/u_i$ sets in family $\{J_i(v) \mid v = 0, \ldots, p-1\}$. This yields a contradiction because the following inequality holds

$$\begin{aligned} p\min\{d, t/6\} &= \sum_v |J_i(v)| \\ &\geq \left(u_i - \frac{u_i}{3\min\{d, t/6\}}\right) \cdot \frac{2p\min\{d, t/6\}}{u_i} \\ &= \left(2 - \frac{2}{3\min\{d, t/6\}}\right) \cdot p\min\{d, t/6\} \\ &> p\min\{d, t/6\} , \end{aligned}$$

since $d \geq 1$ and $t > 4$.

We denote a set of $\frac{u_i}{3\min\{d,t/6\}}$ such tasks by J_i. We define $P_i = \{v : J_i \cap J_i(v) = \emptyset\}$. By definition of tasks $z \in J_i$ we obtain that

$$|P_i| \geq p - \frac{u_i}{3\min\{d, t/6\}} \cdot \frac{2p\min\{d, t/6\}}{u_i} \geq p/3 .$$

Since all processors, other that those in P_i, are delayed during the whole stage i, work performed during stage i is at least $\frac{p}{3} \cdot \min\{d, t/6\}$, and all tasks from J_i remains unperformed. Hence the number u_{i+1} of undone tasks after stage i is still at least $\frac{u_i}{3\min\{d,t/6\}}$.

If $d < t/6$ then work during stage i is at least $pd/6$, and remains at least $\frac{u_i}{3d}$ unperformed tasks. Hence this process may be continued, starting with t tasks, for at least $\log_{3d} t = \Omega(\log_{d+1}(d+t))$ stages, until all tasks will be performed. Total work is then $\Omega(pd\log_{d+1}(d+t))$.

If $d \geq t/6$ then during the first stage work performed is at least $pt/18 = \Omega(pt\log_{d+1}(d+t)) = \Omega(pt)$, and at the end of stage 1 at least $\frac{t}{3t/6} = 2$ tasks remain unperformed. Notice that this asymptotic value does not depend on whether the minimum is selected among d and t, or among d and $t/6$. More precisely, the works is $\Omega(p\min\{d, t\}\log_{d+1}(d+t)) = \Omega(p\min\{d, t/6\}\log_{d+1}(d+t))$, which completes the proof. □

3.2 Lower bound for randomized algorithms

Let $G = (V, T, E)$ be an undirected bipartite random graph such that $|V| = p$, $|T| = t$ and every node $v \in V$ has a set of neighbors $N_v = U \subseteq T$, where $\Pr[N_v = U] = 0$ if $|U| > d$, and $\Pr[N_v = U] = p_v(U)$ if $|U| \leq d$, where $p_v(U)$ are reals from interval $[0, 1]$ such that $\sum_{U\subseteq T} p_v(U) = 1$. For any nonempty set $W \subseteq V$ we define $N_W = \bigcup_{v\in W} N_v$.

LEMMA 3.2. *For* $1 \leq d \leq \sqrt{t}$

$$\frac{1}{4} \leq \frac{\binom{t-d}{t/(d+1)}}{\binom{t}{t/(d+1)}} \leq \frac{1}{e} .$$

LEMMA 3.3. *There exists set* $T' \subseteq T$ *of size* $\frac{t}{d+1}$ *such that*

$$\Pr[\exists_{X\subseteq V} : |X| = p/64 \ \wedge \ N_X \cap T' = \emptyset] \geq 1 - e^{-p/512} .$$

PROOF. First observe that

$$\begin{aligned} \sum_{(T':T'\subseteq T, |T'|=\frac{t}{d+1})} \ \sum_{(v\in V)} \ \sum_{(Y:Y\subseteq T, Y\cap T'=\emptyset, |Y|\leq d)} p_v(Y) &= \\ = \sum_{(v\in V)} \ \sum_{(Y:Y\subseteq T, |Y|\leq d)} p_v(Y) \cdot \binom{t-|Y|}{t/(d+1)} \\ \geq p \cdot \binom{t-d}{t/(d+1)} . \end{aligned}$$

It follows that there exists set $T' \subseteq T$ of size $\frac{t}{d+1}$ such that

$$\sum_{(v\in V)} \ \sum_{(Y:Y\subseteq T, Y\cap T'=\emptyset, |Y|\leq d)} p_v(Y) \geq \frac{p \cdot \binom{t-d}{t/(d+1)}}{\binom{t}{t/(d+1)}} \geq \frac{p}{4} , \quad (1)$$

where the last inequality follows from Lemma 3.2. Fix such a set T'. For every node $v \in V$, let

$$S_v = \sum_{Y:Y\subseteq T, Y\cap T'=\emptyset, |Y|\leq d} p_v(Y) .$$

Notice that $S_v \leq 1$. Using the pigeonhole principle to inequality 1, there is a set $V' \subseteq V$ of size $p/8$ such that for every $v \in V'$

$$S_v \geq \frac{1}{8} .$$

(Otherwise more than $7p/8$ nodes $v \in V$ would have $S_v < 1/8$, and fewer than $p/8$ nodes $v \in V$ would have $S_v \leq 1$. Consequently $\sum_{v\in V} S_v < 7p/64 + p/8 < p/4$, which would contradict (1)). For every $v \in V'$, let X_v be the random variable equal 1 with probability S_v, and 0 with probability $1 - S_v$. These random variables constitute sequence of independent 0-1 trials. Let $\mu = \mathbf{E}[\sum_{v\in V'} X_v] = \sum_{v\in V'} S_v$. Applying Chernoff bound (see [1]) we obtain

$$\Pr\left[\sum_{v\in V'} X_v < \mu/2\right] < e^{-\mu/8} ,$$

and consequently, since $\mu \geq \frac{p}{8} \cdot \frac{1}{8} = \frac{p}{64}$, we have

$$\Pr\left[\sum_{v\in V'} X_v < p/64\right] \leq \Pr\left[\sum_{v\in V'} X_v < \mu/2\right] < \\ < e^{-\mu/8} \leq e^{-p/512} .$$

Finally observe that

$$\Pr\left[\exists_{X\subseteq V} : |X| = p/64 \ \wedge \ N_X \cap T' = \emptyset\right] \geq \\ \geq 1 - \Pr\left[\sum_{v\in V'} X_v < p/64\right] ,$$

which completes the proof of the lemma. □

We apply Lemma 3.3 to prove the final result.

THEOREM 3.4. *Any randomized algorithm solving Do-All with t tasks using p asynchronous message-passing processors performs expected work* $\Omega(t + p\min\{d,t\}\log_{d+1}(d+t))$ *against any d-adversary.*

PROOF. Lower bound $\Omega(t)$ with probability 1 is obvious.

Case 1. Inequalities $1 \le d \le \sqrt{t}$ and $1 - e^{-p/512}\cdot\log_{d+1} t < 1/2$ hold.

It follows that $\log_{d+1} t > e^{p/512}/2$, and next $\sqrt[3]{t} > p + d + \log_{d+1} t$ for sufficiently large p and t. More precisely:

$\sqrt[3]{t} > 3p$ for sufficiently large p, since $t > \log_{d+1} t > e^{p/512}$;

$\sqrt[3]{t} > 3d$ for sufficiently large p, since $d^{e^{p/512}/2} < t$;

$\sqrt[3]{t} > 3\log_{d+1} t$ for sufficiently large t, since $d \ge 1$ and properties of logarithm function.

Consequently, $t = (\sqrt[3]{t})^3 > pd\log_{d+1} t$ for sufficiently large p and t, and the lower bound

$$\Omega(t) = \Omega(pd\log_{d+1} t) = \Omega(pd\log_{d+1}(d+t))$$

holds, with the probability 1, in this case.

Case 2. Inequalities $1 \le d \le \sqrt{t}$ and $1 - e^{-p/512}\cdot\log_{d+1} t \ge 1/2$ hold.

Consider any Do-All algorithm. Similarly as in proof of Theorem 3.1, the adversary partitions computation into stages, each containing d steps.

Let us fix an execution of the algorithm through the end of stage $i-1$. Consider stage i. The adversary delivers to a processor all messages sent in stage i by the end of stage i, provided the processor is not delayed at the end of stage i. Let $U_i \subseteq T$ denote set of tasks unperformed by the end of stage $i-1$. Here, by the adversarial strategy (no message is received during stage i), given that the execution is fixed at the end of stage $i-1$, one can fix a distribution that processor v performs the set of tasks Y during stage i — this distribution is given by the probabilities $p_v(Y)$. The adversary derives set $J_i \subseteq U_i$, using Lemma 3.3 according to the set of all processors, the set of unperformed tasks U_i and the distributions $p_v(Y)$ fixed at the beginning of stage i according to the action of processors v in stage i.

The adversary also delays every processor v not belonging to some set P_i from the first step it is going to perform a task from J_i by the end of stage i. Set P_i is defined on-line (this is one of the difference between the construction of strategy of adversary in the proof of lower bounds for deterministic and randomized Do-All algorithms): at the beginning of stage i set P_i contains all processors; every processor v which is going to perform some task $z \in J_i$ during step j in stage i, is delayed till the end of stage i and removed from set P_i.

Here we present more details of the strategy of adversary. Suppose $u_i = |U_i| > 0$ tasks are unperformed at the beginning of stage i. As it was announced before, we apply Lemma 3.3 to the set U_i and probabilities $p_v(Y)$ to find, at the very beginning of stage i, set $J_i \subseteq U_i$ such that probability that exists subset of processor X of size $p/64$ such that none of them would perform some of task from J_i during stage i is at least $1 - e^{-p/512}$. Next, during stage i we delay all processors that are going (according to random choices during stage i) to perform some task from J_i, and delay is by the end of stage i. By Lemma 3.3, the set P_i of not-delayed processors is of size at least $p - 63p/64 \ge p/64$ and set of remaining tasks $U_{i+1} \supseteq J_i$ is of size at least $\frac{u_i}{d+1}$, all with probability at least $1 - e^{-p/512}$. If this happens, we call stage i *successful.*

It follows that the probability, that every stage $i < \log_{d+1} t$ is successful is at least $1 - e^{-p/512}\cdot\log_{d+1} t$. Hence, using an assumption for this case, with the probability at least $1 - e^{-p/512}\cdot\log_{d+1} t \ge 1/2$ at the beginning of stage i there will be at least $t\cdot\left(\frac{1}{d+1}\right)^{\log_{d+1} t - 1} > 1$ unperformed tasks and work will be at least $(\log_{d+1} t - 1)\cdot dp/64$, since work in one successful stage is at least $p/64$ undelayed processors times the length d of one stage. It follows that the expected work of this algorithm in the presence of our adversary is $\Omega(pd\log_{d+1} t) = \Omega(pd\log_{d+1}(d+t))$, because $1 \le d \le \sqrt{t}$. This completes the proof of Case 2.

Case 3. Inequality $d > \sqrt{t}$ holds.

Consider first $\min\{d, t/6\}$ steps. Let T be the set of all tasks, and $p_v(Y)$ denotes probability, that processor $v \in V$ performs tasks in $Y \subseteq T$ of size $\min\{d, t/6\}$ during considered step. Applying Lemma 3.3 we obtain, that at least $p/64$ processors are undelayed during considered steps and after these steps at least $\frac{\min\{d,t/6\}}{d+1} \ge 1$ task is unperformed, all with the probability at least $1 - e^{-p/512}$

Since $1 \le \log_{d+1}(d+t) < 2$, work is $\Omega(p\min\{d, t/6\}) = \Omega(p\min\{d,t\}\log_{d+1}(d+t))$. □

4. CONTENTION OF PERMUTATIONS

In this section we extend and generalize the notion of *contention* of permutations [2], and study its properties. We use braces $\langle \ldots \rangle$ to denote an ordered list. For a list L and an element a, we use the expression $a \in L$ to denote the element's membership in the list, and the expression $L - S$ to stand for L with all elements in S removed.

Consider a list of some idempotent computational *jobs* with identifiers from the set $[n] = \{1, \ldots, n\}$. (We make the distinction between *tasks* and *jobs* for convenience to simplify algorithm analysis; a job may be composed of one or more tasks.) We refer to a list of job identifiers as a *schedule.* When a schedule for n jobs is a permutation of job identifiers π in S_n, we call it *n-schedule*. Here S_n is the symmetric group, the group of all permutations on the set $[n]$; we use $\circ$ as the composition operator, and $\mathbf{u}_n$ to denote the identity permutation. For a n-schedule $\pi = \langle \pi(1), \ldots, \pi(n)\rangle$ a *left-to-right maximum* (see Knuth vol. 3, p. 13 [16]) is an element $\pi(j)$ of π that is larger than all of its predecessors, i.e., $\pi(j) > \max_{i<j}\{\pi(j-i)\}$.

Given a n-schedule π, we define $\mathrm{LRM}(\pi)$, to be the number of left-to-right maxima in the n-schedule π (see [2]). For a list $\Psi = \langle \pi_0, \ldots, \pi_{n-1}\rangle$ of permutations from S_n and a permutation δ in S_n, the contention of Ψ with respect to δ is defined as $\mathrm{Cont}(\Psi, \delta) = \sum_{u=0}^{n-1}\mathrm{LRM}(\delta^{-1}\circ\pi_u)$. The *contention of the list of schedules* Ψ is defined as $\mathrm{Cont}(\Psi) = \max_{\delta\in S_n}\{\mathrm{Cont}(\Psi,\delta)\}$. Note that for any Ψ, $n \le \mathrm{Cont}(\Psi) \le n^2$. A family of permutations with low contention was introduced in [2], where the following is shown ($H_n = \sum_{j=1}^{n}\frac{1}{j}$ is the nth harmonic number).

LEMMA 4.1. *[2] For any $n > 0$ there exists a list of permutations $\Psi = \langle \pi_0, \ldots, \pi_{n-1}\rangle$ with* Cont$(\Psi) \le 3nH_n = \Theta(n\log n)$.

For a constant n, a list Ψ with $\mathrm{Cont}(\Psi) \le 3nH_n$ can be found by exhaustive search. This costs only a constant number of operations on integers (however, this cost might be of order $(n!)^n$).

Assume now that n distinct asynchronous processors perform the n jobs such that processor i performs the jobs in the order given by π_i in Ψ. We call this oblivious algorithm ObliDo and give the code* in Figure 1.

```
const Ψ = {π_s | 0 ≤ s < n ∧ π_s ∈ S_n}
  % Fixed set of n permutations of [n]
forall processors pid = 0 to n − 1 parbegin
     for i = 1 to n do
          perform Job(π_pid(i))
     od
parend.
```

Figure 1: Algorithm ObliDo.

Since ObliDo does not involve any coordination among the processors the total of n^2 jobs are performed (counting multiplicities). However, it was shown [2] that if we count only the job executions such that each job has not been previously performed by any processor, then the total number of such job executions is bounded by $\mathrm{Cont}(\Psi)$, again counting multiplicities. We call such job executions *primary*; we also call all other job executions *secondary*. Note that the number of primary executions cannot be smaller than n, since each job is performed at least once for the first time. In general this number is going to be between n and n^2, because several processors may be executing the same job concurrently for the first time.

Lemma 4.2. *[2] In algorithm* ObliDo *with n processors, n tasks, and using the list Ψ of n permutations, the number of primary job executions is at most Cont(Ψ).*

Now we generalize the notion of contention and define *d-contention*. For a schedule $\pi = \langle \pi(1), \ldots, \pi(n) \rangle$, an element $\pi(j)$ of π is a *d-left-to-right maximum* (d-lrm for short) if the number the elements in π preceding and greater than $\pi(j)$ is less than d, i.e., $|\{i : i < j \ \wedge \ \pi(i) > \pi(j)\}| < d$.

Given a n-schedule π, we define (d)-$\mathrm{LRM}(\pi)$ as the number of d-lrm's in the schedule π. For a list $\Psi = \langle \pi_0, \ldots, \pi_{p-1} \rangle$ of permutations from S_n and a permutation δ in S_n, the d-contention of Ψ with respect to δ is defined as

$$(d)\text{-Cont}(\Psi, \delta) = \sum_{u=0}^{p-1} (d)\text{-LRM}(\delta^{-1} \circ \pi_u) \ .$$

The *d-contention of the list of schedules* Ψ is defined as $(d)\text{-Cont}(\Psi) = \max_{\delta \in S_n}\{(d)\text{-Cont}(\Psi, \delta)\}$.

Lemma 4.3. *Let Ψ be a list of p random permutations from S_n. For every fixed positive integer d, the probability that (d)-Cont$(\Psi, \mathbf{u}_n) > n \ln n + 8pd \ln(e + n/d)$ is at most $e^{-[n \ln n + 7pd \ln(e+n/(3d))] \ln(7/e)}$.*

The proof of the lemma will appear in the full paper.

Theorem 4.4. *For a random list of schedules Ψ containing p permutations from S_n, and for sufficiently large p and n, the event: "for every positive integer d, (d)-Cont$(\Psi) > n \ln n + 8pd \ln(e + n/d)$", holds with probability at most $e^{-n \ln n \cdot \ln(7/e^2) - p}$.*

*We borrow the parallel parbegin/parend notation, but of course the processors have no shared memory.

Proof. For $d \geq n/5$ the result is straightforward, moreover the event holds with probability 0. In the following we assume that $d < n/5$.

Note that since Ψ is a random list of schedules, then so is $\sigma^{-1} \circ \Psi$, where $\sigma \in S_n$ is an arbitrary permutation. Consequently, by Lemma 4.3, $(d)\text{-Cont}(\Psi, \sigma) > n \ln n + 8pd \ln(e + n/d)$ holds with probability at most $e^{-[n \ln n + 7pd \ln(e+\frac{n}{3d})] \ln \frac{7}{e}}$.

Hence the probability that a random list of schedules Ψ has d-contention greater than $n \ln n + 8pd \ln(e + n/d)$ is at most

$$n! \cdot e^{-[n \ln n + 7pd \ln(e+\frac{n}{3d})] \ln \frac{7}{e}} \leq$$

$$\leq e^{n \ln n - [n \ln n + 7pd \ln(e+\frac{n}{3d})] \ln \frac{7}{e}} \leq$$

$$\leq e^{-n \ln n \cdot \ln \frac{7}{e^2} - 7pd \ln(e+\frac{n}{d})} \ .$$

Then the probability that, for every d, $(d)\text{-Cont}(\Psi) > n \ln n + 8pd \ln(e + n/d)$ is at most

$$\sum_{d=1}^{\infty} \Pr[(d)\text{-Cont}(\Psi) > n \ln n + 8pd \ln(e + n/d)] \ \leq$$

$$\leq \ \sum_{d=1}^{n/5-1} e^{-n \ln n \cdot \ln(7/e^2) - 7pd \ln(e+n/d)} + \sum_{d=n/5}^{\infty} 0$$

$$\leq \ e^{-n \ln n \cdot \ln(7/e^2) - p}$$

for sufficiently large p and n. $\square$

Using the probabilistic method we obtain the following.

Corollary 4.5. *For sufficiently large p and n, there is a list of p schedules Ψ from S_n such that (d)-Cont$(\Psi) \leq n \log n + 8pd \ln(e + n/d)$, for every positive integer d.*

5. DETERMINISTIC ALGORITHMS DA

We now present a deterministic algorithm for Do-All with p processors and t tasks. We show that its work W is $O(tp^{\varepsilon} + p \min\{t, d\} \lceil t/d \rceil^{\varepsilon})$ for any constant $\varepsilon > 0$, when $p \leq t$, and its message complexity M is $O(p \cdot W)$.

5.1 Definition of algorithm DA(q)

Let q be some constant such that $2 \leq q \leq p$. We assume that the number of tasks t is an integer power of q, specifically let $t = q^h$ for some $h \in \mathbb{N}$. When the number of tasks is not a power of q we can use a standard padding technique by adding just enough "dummy" tasks so that the new number of tasks becomes a power of q; the final results show that this padding does not affect the asymptotic complexity of the algorithm. We also assume that $\log_q p$ is a positive integer. If it is not, we pad the processors with at most qp "infinitely delayed" processors so this assumption is satisfied; in this case the upper bound is increased by a (constant) factor of at most q. The algorithm uses a list of q permutations $\Psi = \langle \pi_0, \ldots \pi_{q-1} \rangle$ from S_q such that Ψ has the minimum contention among all such lists. We define a family of algorithms, where each algorithm is parameterized by q, and a list Ψ with the above contention property. We call this algorithm DA(q).

Algorithm DA(q), utilizes a q-ary boolean *progress tree* with t leaves, where the tasks are associated with the leaves.

```
00 const q  % Arity of the progress tree
01 const Ψ = ⟨π_s | 0 ≤ s < q ∧ π_s ∈ S_q⟩
02  % Fixed list of q permutations of [q]
03 const l = (qt−1)/(q−1)  % The size of the progress tree
04 const h = log_q t  % The height of the progress tree
05 type ProgressTree: array [0..l − 1] of boolean  % Progress tree
06 forall processors PID = 0 to p − 1 parbegin
07      ProgressTree T_pid  % The progress tree at processor PID
10      thread  % Traverse progress tree in search of work
11           integer n init = 0  % Current node, begin at the root
12           integer i init = 0  % Current depth in the tree
13           DOWORK(n, i)
14      end
20      thread  % Receive broadcast messages
21           set of ProgressTree B  % Incoming messages
22           while T_pid[0] ≠ 1 do  % While not all tasks certified
23                receive B  % Deliver the set of received messages
24                T_pid := T_pid ∨ (⋁_{b∈B} b)  % Learn progress
25           od
26      end
27 parend.
```

```
40 procedure DOWORK(n,i)  % Recursive progress tree traversal
41                          % n : current node index ; i : node depth
42      const array x[0..h − 1] of integer = PID_(base q)
43       % h digits of q-ary expansion of PID
44      if T_pid[n] = 0 then  % Node not done – still work left
45           if i = h then  % Node n is a leaf
46                perform Task(t − l + n + 1)  % Do the task
47           else  % Node n is not a leaf
48                for j = 1 to q do
49                 % Visit subtrees in the order of π_x[i]
50                     DOWORK(qn + π_x[i](j), i + 1)
51                od
52           fi
53           T[n] := 1  % Record completion of the subtree
54           broadcast T  % Share the good news
55      fi
56 end.
```

Figure 2: The deterministic algorithm DA ($t \le p$).

Initially all nodes of the tree are 0 (false) indicating that no tasks have been performed. Whenever a processor learns that all tasks in a subtree rooted at a certain node have been performed, it sets the node to 1 (true) and shares the news with all other processors. Each processor, acting independently, searches for work in the smallest immediate subtree that has remaining. It then performs any tasks it finds, and moves out of that subtree when all work within it is completed. When exploring the subtrees rooted at an interior node at height i, a processor visits the subtrees in the order given by one of the permutations in Ψ. Specifically, the processor uses the permutation π_s such that s is the value of the i-th digit in the q-ary expansion of the processor's identifier.

Data structures: Given the t tasks, the progress tree is a q-ary ordered tree of height h, where $t = q^h$. The number of nodes in the progress tree is $l = \sum_{i=0}^{h-1} q^i = (q^{h+1}-1)/(q-1) = (qt-1)/(q-1)$. Each node of the tree is a boolean, indicating whether the subtree rooted at the node is done (value 1) or not (value 0).

The progress tree is stored in a boolean array $T[0..l-1]$, where $T[0]$ is the root, and the q children of the interior node $T[n]$ being the nodes $T[qn+1], T[qn+2], \ldots, T[qn+q]$. The space occupied by the tree is $O(t)$. The t tasks are associated with the leaves of the progress tree, such that the leaf $T[n]$ corresponds to the task $Task(n+t+1-l)$.

We represent the PID of each of the p processors in terms of its q-ary expansion. We care only about the h least significant q-ary digits of each PID (thus when $p > t$ several processors may be indistinguishable in the algorithm). The q-ary expansions of each PID is stored in the array $x[0..h-1]$.

Control flow: The code is given in Figure 2. Each of the p processors executes two concurrent threads. One thread (lines 10-14) traverses the local progress tree in search work, performs the tasks, and broadcasts the updated progress tree. The second thread (lines 20-26) receives messages from other processors and updates the local progress tree. (Each processor is asynchronous, but we assume that its two threads run at approximately the same speed. This is assumed for simplicity only, as it is trivial to explicitly schedule the threads on a single processor.) Note that the updates of the local progress tree T are always monotone: initially each node contain 0, then once a node changes its value to 1 it remains 1 forever. Thus no issues of consistency arise.

The progress tree is traversed using the recursive procedure DOWORK (lines 40-56). The order of traversals within the progress tree is determined by the list of permutations $\Psi = \langle \pi_0, \pi_1, \ldots, \pi_{q-1} \rangle$. Each processor uses, at the node of depth i, the i^{th} q-ary digit $x[i]$ of its PID to select the permutation $\pi_{x[i]}$ from Ψ. The processor traverses the q subtrees in the order determined by $\pi_{x[i]}$ (lines 48-51), but it traverses within a subtree only if the corresponding bit in the progress tree is not set (line 44). In other words, each processor PID traverses its progress tree in a post-order fashion using the q-ary digits of its PID and the permutations in Ψ to establish the order of the subtree traversals, except that when the messages from other processors are received, the progress tree of processor PID is potentially pruned based on the progress of other processors.

Correctness: We claim that algorithm DA(q) correctly solves the Do-All problem. This follows from the observation that a processor leaves a subtree by returning from a recursive call to DOWORK if and only if the subtree contains no unfinished work and its root is marked accordingly.

5.2 Complexity analysis of algorithm DA(q)

We start by showing a lemma that relates the work of the algorithm to its recursive structure.

We consider the case $t \le p$. Let $W(p,t)$ denote work of algorithm DA(q) through the first global step in which some processor completes the last remaining task and broadcasts the message containing the progress tree where $T[0] = 1$. We note that $W(p,1) = O(p)$. This is because the progress tree has only one leaf. Each processor makes a single call to DOWORK, performs the sole task and broadcasts the completed progress tree.

LEMMA 5.1. *For p-processor, t-task algorithm DA with $t \le p$ and t and p divisible by q:*

$$W(p,t) = O(Cont(\Psi) \cdot W(p/q, t/q) + p \cdot q \cdot \min\{d, t/q\}) .$$

PROOF. Since the root of the progress tree has q children, each processor makes the initial call to DOWORK$(0,0)$ (line 13) and then (in the worst case) it makes q calls to DOWORK (line 50) corresponding to the children of the root. We consider the performance of all tasks in the specific subtree rooted at a child of the progress tree as a job, thus such a job consists of all invocations of DoWork on that subtree. We now account separately for the primary and secondary job executions (recall the definitions in Section 4).

Observe that the code in lines 48-51 of DA is essentially algorithm ObliDo (lines 03-05 in Figure 1) and we intend to use Lemma 4.2. The only difference is that instead of q processors we have q groups of p/q processors where in each group the PIDs differ in their q-ary digit corresponding to the depth 0 of the progress tree. From the recursive structure of algorithm DA it follows that the work of each such group in performing a single job is $W(p/q, t/q)$, since each group has p/q processors and the job includes t/q tasks. Using Lemma 4.2 the primary task executions contribute $O(\text{Cont}(\Psi) \cdot W(p/q, t/q))$ work.

If messages were delivered without delay, there would be no need to account for secondary job executions because the processors would instantly learn about all primary job completions. Since messages can be delayed by up to d time units, each processor may spend up to d time steps, but no more than $O(t/q)$ steps performing a secondary job (this is because it takes a single processor $O(t/q)$ steps to perform a post-order traversal of a progress tree with t/q leaves). There are q jobs to consider, so for p processors this amounts to $O(p \cdot q \cdot \min\{d, t/q\})$ work.

For each processor there is also a constant overhead due to the fixed-size code executed per each call to DoWork. The work contribution is $O(p \cdot q)$. Finally, given the assumption about thread scheduling, the work of message processing thread does not exceed asymptotically the work of the DoWork thread. Putting all these work contributions together yields the desired result. $\square$

We now prove the following theorem about work.

Theorem 5.2. *Consider algorithm DA(q) with p processors and t tasks where $t \le p$. Let d be the maximum message delay. For any constant $\varepsilon > 0$ there is a constant q such that the algorithm has work $O(p \min\{t, d\}\lceil t/d\rceil^{\varepsilon})$.*

Proof. First suppose that $\log_q t$ and $\log_q p$ are positive integers. We prove by induction on p and t that

$$W(p,t) \le q \cdot t^{\log_q(4a \log q)} \cdot p \cdot d^{1-\log_q(4a\log q)},$$

where some positive constant a is sufficiently large and follows from Lemma 5.1. For the base case of $t = 1$ the statement is correct since $W(p, 1) = O(p)$. For $t > 1$ we choose the list of permutations Ψ with $\text{Cont}(\Psi) \le 3q \log q$ per Lemma 4.1. Due to our choice of parameters, $\log_q t$ is an integer and $t \le p$. Let ξ stand for $\log_q(4a \log q)$. Using Lemma 5.1 and inductive hypothesis we obtain

$$\begin{aligned} W(p,t) &\le a \cdot \Big(3q \log q \cdot q \cdot \left(\frac{t}{q}\right)^{\xi} \cdot \frac{p}{q} \cdot d^{1-\xi} \\ &\qquad + p \cdot q \cdot \min\{d, t/q\}\Big) \\ &\le a \cdot \Big(\left(q \cdot t^{\xi} \cdot p \cdot d^{1-\xi}\right) \cdot 3 \log q \cdot q^{-\xi} \\ &\qquad + p \cdot q \cdot \min\{d, t/q\}\Big) . \end{aligned}$$

We now consider two cases:

Case 1: $d \le t/q$. It follows that

$$p \cdot q \cdot \min\{d, t/q\} = pqd \le pqd^{1-\xi} \cdot \left(\frac{t}{q}\right)^{\xi} .$$

Case 2: $d > t/q$. It follows that

$$p \cdot q \cdot \min\{d, t/q\} = pt \le pqd^{1-\xi} \cdot \left(\frac{t}{q}\right)^{\xi} .$$

Putting everything together we obtain the desired inequality

$$W(p,t) \le a\left(\left(q \cdot t^{\xi} p \cdot d^{1-\xi}\right) 4 \log q \cdot q^{-\xi}\right) \le q \cdot t^{\xi} p \cdot d^{1-\xi}.$$

To complete the proof, consider any $t < p$. We add $t' - t$, where $t' - t < qt - 1$, new "dummy" tasks and $p' - p$, where $p' - p < qp - 1$, new "virtual" processors, such that $\log_q t'$ and $\log_q p'$ are positive integers. We assume that all "virtual" processors are delayed to infinity. It follows that

$$W(p,t) \le W(p', t') \le q \cdot (t')^{\xi} p' \cdot d^{1-\xi} \le q^{2+\xi} t^{\xi} p \cdot d^{1-\xi}.$$

Observe that $\lim_{q\to\infty} \xi = \lim_{q\to\infty} \log_q(4a \log q) = 0$, hence one can asymptotically choose q so that $\log_q(4a \log q) < \varepsilon$ for any positive constant ε (however, q is of order $2^{\frac{\log(1/\varepsilon)}{\varepsilon}}$). Consequently we obtain that work of algorithm DA(q) is $O(\min\{t^{\varepsilon} p \ d^{1-\varepsilon}, tp\}) = O(p \min\{t, d\}\lceil t/d\rceil^{\varepsilon})$, which completes the proof of the theorem. $\square$

Now we consider the case $t \ge p$. This is accomplished by a trivial preparation. We partition the t tasks into p jobs of size t/p and let the algorithm work with these jobs. It takes a processor $O(t/p)$ work (instead of a constant) to process such a job.

Theorem 5.3. *Consider algorithm DA(q) with p processors and t tasks where $p \le t$. Let d be the maximum message delay. For any constant $\varepsilon > 0$ there is a constant q such that DA(q) has work $O(tp^{\varepsilon} + p \min\{t, d\}\lceil t/d\rceil^{\varepsilon})$.*

Proof. We use Theorem 5.2 with p jobs (instead of t tasks) and given that a single job takes $O(t/p)$ unit of work, and we upper-bound the maximal delay for receiving messages that some job was already performed by $d' = \lceil pd/t\rceil = O(1 + pd/t)$ "job" units (one "job" unit lasts $\Theta(t/p)$ work units). We obtain the following bound on work:

$$\begin{aligned} &O\left(p \min\{p, d'\}\lceil p/d'\rceil^{\varepsilon} \cdot \tfrac{t}{p}\right) = O\left(\min\left\{p^{\varepsilon} p (d')^{1-\varepsilon}, p^2\right\} \cdot \tfrac{t}{p}\right) \\ &\quad = \ O\left(\min\left\{tp^{\varepsilon} + pt^{\varepsilon} d^{1-\varepsilon}, tp\right\}\right) \\ &\quad = \ O\left(tp^{\varepsilon} + p \min\{t, d\} \left\lceil \frac{t}{d} \right\rceil^{\varepsilon}\right) . \quad \square \end{aligned}$$

Finally we consider message complexity.

Theorem 5.4. *Algorithm DA(q) has message complexity $O(p \cdot W(p,t))$.*

Proof. In each step, a processor broadcasts at most one message to $p - 1$ other processors. $\square$

6. PERMUTATION ALGORITHMS PA

In this section we present and analyze a family of algorithms that are simpler than algorithms DA and that directly rely on permutation schedules. Two algorithms are randomized (algorithms PaRan1 and PaRan2), and one is deterministic (algorithm PaDet).

The common pattern in the three algorithms is that each processor, while it has not ascertained that all tasks have been performed, performs a specific task from its local list and broadcasts this fact to other processors. The known performed tasks are removed from the list. The code is given in Figure 3. The common code for the three algorithms is in lines 00-29. The three algorithms differ in two ways: (1) the initial ordering of the tasks by each processor (call to procedure ORDER on line 20), and (2) the selection of the next task to perform (call to function SELECT on line 24). We now describe the specialization of the code made by each algorithm (the code for ORDER+SELECT).

```
00 use package ORDER+SELECT  % Algorithm-specific procedures
01 type TaskId : [t]
02 type TaskList : list of TaskId
03 type MsgBuff : set of TaskList
10 forall processors pid = 0 to p − 1 parbegin
11     TaskList Tasks_pid init [t]
12     MsgBuf B  % Incoming messages
13     TaskId tid  % Task id; next to done
20     ORDER(Tasks_pid)
21     while Tasks_pid ≠ ∅ do
22         receive B  % Deliver the set of received messages
23         Tasks_pid := Tasks_pid − (⋃_{b∈B} b)  % Remove tasks
24         tid := SELECT(Tasks_pid)  % Select next task
25         perform Task(tid)
26         Tasks_pid := Tasks_pid − {tid}  % Remove done task
27         broadcast Tasks_pid  % Share the news
28     od
29 parend
```

```
40 package ORDER+SELECT  % Used in algorithm PARAN1
41     list Ψ = ⟨TaskList π_s | 0≤s<p ∧ π_s = random list of[t]⟩
42      % Random list of p permutations
43     procedure ORDER(T) begin  T := π_pid  end
44     TaskId function SELECT(T) begin  return(T(1))  end
```

```
50 package ORDER+SELECT  % Used in algorithm PARAN2
51     procedure ORDER(T) begin  no-op  end
52     TaskId function SELECT(T) begin  return(random(T))  end
```

```
60 package ORDER+SELECT  % Used in algorithm PADET
61     const Ψ = {TaskList π_s | 0 ≤ s < p ∧ π_s ∈ S_t}
62      % Fixed list of p permutations
63     procedure ORDER(T) begin  T := π_pid  end
64     TaskId function SELECT(T) begin  return(T(1))  end
```

Figure 3: Permutation algorithm and its specializations for PARAN1, PARAN2, and PADET.

We initially assume that $p \geq t$. The case of $p \leq t$ is obtained by dividing the t tasks into p jobs, each of size at most $\lceil t/p \rceil$. In this case we deal with jobs instead of tasks in the code of permutation algorithms.

Randomized algorithm PARAN1**.** The specialized code is in Figure 3, lines 40-44. Each processor *pid* performs tasks according to a local permutation π_{pid}. These permutations are selected uniformly at random at the beginning of computation (line 41), independently by each processor. We refer to the collection of these permutation as Ψ. The drawback of this approach is that the number of random selections is $p \cdot t$, each of $O(\log t)$ random bits.

Randomized algorithm PARAN2**.** The specialized code is in Figure 3, lines 50-52. Initially the tasks are left unordered. Each processor selects tasks uniformly and independently at random, one at a time (line 52). Clearly the expected work W is the same for algorithms PARAN1 and PARAN2, however the number of random bits needed by PARAN2 becomes $W \cdot \log t$ and, as we will see, this is an improvement.

Deterministic algorithm PADET**.** The specialized code is in Figure 3, lines 60-64. We assume the existence of the list of permutations Ψ chosen per Corollary 4.5. Each processor *pid* permutes its list of tasks according to the local permutation $\pi_{pid} \in \Psi$.

6.1 Complexity analysis

Let $n = \min\{t, p\}$ be the number of tasks, or jobs if $t > p$, to perform by algorithms PA.

LEMMA 6.1. *For algorithms* PADET *and* PARAN1*, the respective worst case work and expected work is at most* (d)*-Cont*(Ψ) *against any* d*-adversary.*

PROOF. Suppose processor v starts performing task z at (real) time k. By the definition of d-adversary, no other processor successfully performed task z and broadcast its message by time $(k - d)$. Consider (d, σ)-adversary, for any permutation $\sigma \in S_n$.

For each processor v, let J_v contain all pairs (v, i) such that v performs task $\pi_v(i)$ during the computation. We construct function L from the set $\bigcup_v J_v$ to the set of all d-lrm's of the list $\sigma^{-1} \circ \Psi$ and show that L is a bijection. We do the construction independently for each processor v. It is obvious that $(v, 1) \in J_v$, and we let $L(v, 1) = 1$. Suppose that $(v, i) \in J_v$ and we defined function L for all elements from J_v less than (v, i) in lexicographic order. We define $L(v, i)$ as the first $j \leq i$ such that $(\sigma^{-1} \circ \pi_v)(j)$ is a d-lrm not assigned by L to any element in J_v.

Claim. For every $(v, i) \in J_v$, $L(v, i)$ is well defined.

For $i = 1$ we have $L(v, 1) = 1$. For the first d elements in J_v this is also easy to show. Suppose L is well defined for all elements in J_v less than (v, i), and (v, i) is at least the $(d + 1)$st element in J_v. We show that $L(v, i)$ is also well defined. Suppose, to the contrary, that there is no position $j \leq i$ such that $(\sigma^{-1} \circ \pi_v)(j)$ is a d-lrm and j is not assigned by L before step of construction for $(v, i) \in J_v$. Let $(v, j_1) < \ldots < (v, j_d)$ be the elements of J_v less than (v, i) such that $(\sigma^{-1} \circ \pi_v)(L(v, j_1)), \ldots, (\sigma^{-1} \circ \pi_v)(L(v, j_d))$ are greater than $(\sigma^{-1} \circ \pi_v)(i)$. They exist from the fact, that $(\sigma^{-1} \circ \pi_v)(i)$ is not a d-lrm and all "previous" d-lrm's are assigned by L. Obviously task $\pi_v(L(v, j_1))$ has been performed at least $d + 1$ local steps (and hence also global steps) before the global step when task $\pi_v(i)$ is performed by v. It follows from this and the definition of (d, σ)-adversary, that task $\pi_v(i)$ has been performed by some other processor in a local step, which ended also at least $(d + 1)$ global steps before the global step when task $\pi_v(i)$ is performed by v. This contradicts the observation made at the beginning of the proof of lemma. This proves the claim.

That L is a bijection follows directly from the definition of L. It follows that the number of performances of tasks, which is equal to the total number of local steps until completion of all tasks, is at most (d)-Cont(Ψ, σ), against any (d, σ)-adversary. Hence work is at most (d)-Cont(Ψ) against any d-adversary. $\square$

THEOREM 6.2. *Algorithms* PARAN1 *and* PARAN2 *have expected work* $O(t \log n + p \min\{t, d\} \log(2 + t/d))$ *and expected communication* $O(tp \log n + p^2 \min\{t, d\} \log(2 + t/d))$.

PROOF. We prove the work bound for algorithm PARAN1 using the random list of schedules Ψ and Theorem 4.4, together with Lemma 6.1. If $t \leq p$ we obtain formula $O(t \log t+$

$p \min\{t, d\} \log(2+t/d))$, in view of Theorem 4.4 and the obvious upper bound for work is tp. If $t > p$ then we argue that $d' = \lceil pd/t \rceil$ is the upper bound on the number of "job" units taking to deliver a message to recipients, and consequently we obtain formula $O(p \log p + pd' \log(2 + p/d')) \cdot O(t/p) = O(t \log p + pd \log(2+t/d))$, which together with upper bound tp yields the formula $O(t \log p + p \min\{t, d\} \log(2 + t/d))$. Since the only difference in the above two cases is the factor $\log t$ which becomes $\log p$ in the case where $p < t$, we conclude the final formula for work. All these derivations hold with the probability at least $1 - e^{-n \ln n \cdot \ln(7/e^2) - p}$. Since the work can be in the worst case tp with probability at most $e^{-n \ln n \cdot \ln(7/e^2) - p}$, this contributes at most the summant t to the expected work.

Message complexity follows from the fact that in every local step each processor sends p messages. The same result applies to PaRan2 as observation in the description of the the algorithm. □

Theorem 6.3. *There exists a deterministic list of schedules Ψ such that algorithm* PaDet *performs work $O(t \log n + p \min\{t, d\} \log(2+t/d))$ and has communication $O(tp \log n + p^2 \min\{t, d\} \log(2 + t/d))$.*

Proof. The result follows from using the set Ψ from Corollary 4.5 together with Lemma 6.1, using the same derivation for work formula as in the proof of Theorem 6.2. Message complexity follows from the fact, that in every local step each processor sends p messages. □

Corollary 6.4. *For any $d < t$, algorithms* PaRan1 *and* PaRan2 *perform expected work $O(t \log p + pd \log(2 + t/d))$ and have expected communication $O(tp \log p + p^2 d \log(2 + t/d))$ when $p \leq t$.*

Corollary 6.5. *For every p, t such that $p \leq t$ there exists a deterministic list of schedules Ψ algorithm* PaDet *performs work $O(t \log p + pd \log(2 + t/d))$ and has communication $O(tp \log p + p^2 d \log(2 + t/d))$ when $d \leq t$.*

7. DISCUSSION AND FUTURE WORK

In this paper we presented the first message-delay-sensitive analysis of the Do-All problem for asynchronous processors. We gave a delay-sensitive bounds for the problem and presented deterministic and randomized algorithms that have subquadratic in p and t work complexity for any message delay d as long as $d = o(t)$. One of the two deterministic algorithms relies on large permutations of tasks with certain combinatorial properties. One open problem is how to construct such permutations efficiently. There also exists a gap between the upper and the lower bounds in this paper. It will be very interesting to narrow the gap. Finally, while the focus of this paper is on the work complexity, we intend to investigate algorithms that simultaneously control work and message complexity.

8. REFERENCES

[1] Alon, N., Spencer, J.H.: The Probabilistic Method. J. Wiley and Sons, Inc., second edition (2000)

[2] Anderson, R.J., Woll, H.: Algorithms for the certified Write-All problem. SIAM J. on Comp. **26** (5) (1997) 1277–1283

[3] Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message passing systems. J. of the ACM, **42** (1) (1996) 124–142

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

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

[6] Chlebus, B., Gąsieniec, L., Kowalski, D., Shvartsman, A.A.: Bounding work *and* communication in robust cooperative computation. Proc. of 16th Int. Symposium on Distributed Computing, (2002) Springer LNCS 2508, 295–310

[7] Chlebus, B., Kowalski, D., Lingas, A.: The Do-All problem in broadcast networks. Proc. of 20th ACM Symp. on Principles of Distributed Computing (2001) 117–126

[8] C. Dwork, N. Lynch and L. Stockmeyer.: Consensus in the presence of partial synchrony. Journal of the ACM, **35** (2) (1988) 288-323

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

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

[11] Galil, Z., Mayer, A., Yung, M.: Resolving message complexity of byzantine agreement and beyond. Proc. of 36th IEEE Symp. on Foundations of Computer Science, (1995) 724–733

[12] Georgiou, Ch., Kowalski, D., Shvartsman, A.A.: Robust distributed cooperation using inexpensive gossip. Manuscript (submitted for publication), 2003.

[13] Georgiou, Ch., Russell, A., Shvartsman, A.A.: Complexity of distributed cooperation in the presence of failures. Proc. of the 4th International Conference on Principles of Distributed Systems (OPODIS 2000) 245–264

[14] Kanellakis, P.C., Shvartsman, A.A.: Fault-tolerant parallel computation. Kluwer Academic Publishers (1997)

[15] Kedem, Z., Palem, K., Raghunathan, A., Spirakis, P.: Combining tentative and definite executions for very fast dependable parallel computing. Proc. of the 23rd ACM Symposium on Theory of Computing, (1991) 381–389

[16] Knuth, D.E.: The art of computer programming Vol. 3 (third edition). Addison-Wesley Pub Co. (1998)

[17] Lynch, N., Shvartsman, A.: RAMBO: A Reconfigurable Atomic Memory Service. Proc. of 16th Int-l Symposium on Distributed Computing, (DISC'2002) 173–190

[18] Momenzadeh, M.: Emulating shared-memory Do-All in asynchronous message passing systems. Masters Thesis, Computer Sci. & Eng., Univ. of Conn, (2003)

[19] Naor, J., Roth, R.M.: Constructions of permutation arrays for certain scheduling cost measures. Random Structures and Algorithms **6** (1) (1995) 39–50