Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Computation 203 (2005) 181–210
Information and Computation
www.elsevier.com/locate/ic

Performing work with asynchronous processors: Message-delay-sensitive bounds ☆

Dariusz R. Kowalski [a,b], Alex A. Shvartsman [c,d,*]

[a] *Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warsaw, Poland*
[b] *Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK*
[c] *Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Rd., Unit 2155 Storrs, CT 06269, USA*
[d] *MIT Computer Science and Artificial Intelligence Laboratory, The Stata Center, Building 32, Cambridge, MA 02139, USA*

Received 14 November 2003; revised 3 May 2005
Available online 3 October 2005

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

☆ Preliminary version of this paper appeared as: Performing Work with Asynchronous Processors: Message-Delay-Sensitive Bounds, Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (2003) 265–274. The work of the first author is supported in part by the NSF-NATO Award 0209588 and by the KBN Grant 4T11C04425. The work of the second author is supported in part by the NSF CAREER Award 9984778 and by the NSF Grants 9988304, 0121277, and 0311368.
* Corresponding author.
E-mail address: alex@theory.csail.mit.edu (A.A. Shvartsman).

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2005.08.002

algorithm without having to emulate atomic registers assumed by that algorithm. These deterministic algorithms have work $O(tp^{\varepsilon} + p\,d\lceil t/d\rceil^{\varepsilon})$ for any $\varepsilon > 0$. The second family uses specific permutations of tasks, with certain combinatorial properties, to sequence the work of the processors. These randomized (deterministic) algorithms have expected (worst-case) work $O(t \log p + p\,d \log(2 + t/d))$. Another important contribution in this work is the first *delay-sensitive lower bound* for this problem that helps explain the behavior of our algorithms: any randomized (deterministic) algorithm has expected (worst-case) work of $\Omega(t + p\,d \log_{d+1} t)$.

Keywords: Distributed algorithms; Message passing; Work; Asynchrony; Delay

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, Dwork et al. [9], 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 $W = \Theta(t \cdot p)$, 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 subquadratic in t and p.

Two work complexity measures have been considered in evaluating the efficiency of algorithms. The measure of [9] accounts only for the task-oriented work, thus allowing the processors to “idle” for free. The measure of [10,14] charges processors for each computation step, whether it is task oriented or not. In this work, we use the latter measure (charging for each step) for two reasons: (1) this measure is a natural generalization of the processor-time product, a standard complexity measure in parallel computing: if p processor compute for τ time units each, then the work efficiency is $p \cdot \tau$; (2) disallowing idling has the effect of speeding up computation (cf. [9], where idling leads to work-efficient but time-inefficient solutions).

1.1. Background and related work

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. [5–7,9–12]. 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 subquadratic 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 $\Omega(p \cdot t)$. In particular, it is sufficient for messages to be delayed by $\Theta(t)$ time for this lower bound to hold. The algorithmic techniques in the above-mentioned papers rely on processor synchrony and assume constant-time message delay. It is not clear how such algorithms can be adapted to deal with asynchrony. 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.

A similar problem, called Write-All, has been extensively study in the shared-memory models of computation. The problem is formulated as follows [14]: *using p processors write* 1*'s to all locations of an array of size t*. In this problem, writing 1 abstracts the notion of performing a constant-time task. Write-All has been substantially studied in the synchronous models of computation [14], however, the techniques used in the synchronous shared-memory setting are not easily ported to the asynchronous message-passing setting.

The most efficient known asynchronous algorithm is due to Anderson and Woll [2], and it has work $O(t \cdot p^{\varepsilon})$ for $t \leqslant n$. The algorithm uses a q-ary *progress tree* with t leaves that associates tasks (array elements) with the leaves of the tree. The progress tree directs processors to tasks, and helps balance the loads of the processors. This is done using the individual digits of the q-ary representation of processor identifiers, where each digit is used to choose between the q branches at the interior nodes within the progress tree. The algorithm uses a set of q permutations from S_q, the symmetric group of permutations of $[q]$. In more detail, the p processors use these permutations in conjunction with q-ary processor identifiers to determine the order of traversal of the q subtrees of each interior node of the progress tree. Thus, the algorithm is based on a recursive post-order traversal of the progress tree, in which processors traverse the tree in search of work until all tasks are performed. Whenever all tasks associated with a subtree are done, the root of the subtree is marked. After all tasks are done, the root of the whole tree is marked, and the algorithm terminates. It was shown [2] that for any $\varepsilon > 0$ it is possible to choose q and a set of q permutations with certain combinatorial properties such that the algorithm has work $O(t \cdot p^{\varepsilon})$. The required combinatorial property is measured in terms of *contention* introduced in [2]. Contention is used to assess the bounds on the number of tasks that are performed redundantly (we discuss this in detail in Section 4).

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 [2] show how to search for these permutations, taking exponential in q processing time. A different approach is given by Naor and Roth [20]. 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 $1/\varepsilon^3$.

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] (a precursor of [2] that uses binary progress trees). For example, Momenzadeh et al. [19,16] show how to do this using replicated linearizable memory services that rely on quorum systems (or majorities of processors), e.g. [3,18]. This approach yields Do-All algorithms for asynchronous message-passing processors with subquadratic work complexity, however, it is assumed that quorum systems are not disabled by failures, and that the delays are $O(K)$, where $K = o(p)$ is the size of quorum sets. When processor failures damage quorum systems, the work of such algorithms becomes quadratic, even if message latency is constant.

1.2. 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 subquadratic 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 $d \geqslant t$, 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 lower bounds 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 $\Omega(t + p\,d \log_{d+1} t)$, where d is the upper bound on message delay (unknown to the processors). This shows that work grows with d and becomes $\Omega(p\,t)$ as d approaches t.

(2) We present a family of deterministic algorithms DA, such that for any constant $\varepsilon > 0$ there is one with work $W = O(tp^{\varepsilon} + p\,d\lceil t/d\rceil^{\varepsilon})$ and message complexity $M = O(p \cdot W)$. More precisely, algorithms from the family DA are parameterized by a positive integer q and a list Ψ of q permutations on the set $[q] = \{1, \dots, q\}$, where $2 \leqslant q < p \leqslant t$. We show that for any constant $\varepsilon > 0$ there is a constant q and a corresponding set of permutation Ψ, such that the resulting algorithm has work $W = O(tp^{\varepsilon} + p\,d\lceil t/d\rceil^{\varepsilon})$ and message complexity $M = O(p \cdot W)$. These works of these algorithms are within a small polynomial factor of the corresponding lower bound (1).
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.

(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 $p \leqslant t$.
We show that when the required permutations are chosen randomly, the algorithms have expected work $W = \mathrm{O}(t \log p + p\,d \log(2 + t/d))$ and expected message complexity $M = \mathrm{O}(tp \log p + p^2 d \log(2 + t/d))$.
We show that there exists a deterministic list of schedules Ψ such that the deterministic algorithm has work $W = \mathrm{O}(t \log p + p\,d \log(2 + t/d))$ and message complexity $M = \mathrm{O}(tp \log p + p^2 d \log(2 + t/d))$.

The deterministic algorithms in PA have somewhat better work than the deterministic algorithms in DA, however, they requires substantially larger permutations, the former requiring permutations of set $[q]$ and the latter of set $[p]$. Hence, one can instantiate algorithms from the family PA that are more efficient than any in family DA, but to do so may require substantial preprocessing time to find good permutations, which are substantially larger in the case of algorithms in PA. Also note that efficient algorithms in PA are only a logarithmic factor worse work than the best possible Do-All algorithms in this model, which follows from the lower bound on work—the best algorithms from family DA have $\mathrm{O}(p^{\varepsilon})$ overhead on work for large $t = \Omega(p\,d)$ and $\mathrm{O}(\lceil t/d \rceil^{\varepsilon})$ overhead for small $t = \mathrm{O}(p\,d)$.

Our algorithms in the family DA are obtained by a re-interpretation for the message-passing setting of the shared-memory algorithm of Anderson and Woll [2]. 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 [2]. We describe this in detail in Section 4, showing how reducing contention leads to lower work in certain algorithms.

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

2.1. The model of computation

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 arbitrary delays between local clock ticks. The processors are subject to crash failures, again determined by the adversary. (We give the detailed description of the adversarial model later in the section.) 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.

2.2. Modeling adversity

We assume an omniscient adversary that introduces delays. The adversary can introduce arbitrary delays between local processors steps and cause processor crash failures (crashes can be viewed as infinite delays). The only restriction is that at least one processor is non-faulty. The adversary can also impose message delays up to d time units. We call such an adversary the *d-adversary*. The adversary is *adaptive*, meaning that it makes its decisions during the execution of an algorithm. For deterministic algorithms, the adaptive adversary does not gain any advantage over the adversary that has to make all decisions off-line, however, adversarial adaptivity becomes important for randomized algorithms.

2.3. Timing assumptions

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. This time is assumed to be positive (non-zero). 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 global time period of duration d.

2.4. The Do-All problem

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.

When we say that the tasks are similar, this means that any task can be performed in constant time. When we say that the tasks are idempotent, this means that any task can be performed more than once, and that the results of multiple task executions are always the same. Any tasks may be performed concurrently.

2.5. Measures of efficiency

We assess the efficiency of algorithms in terms of work and message complexity.

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.

Let $\mathcal{E}$ be the set of all possible executions of an algorithm in a specific model of computation. For an execution $\alpha \in \mathcal{E}$ of an algorithm, we denote by $p_\tau(\alpha)$ the number of processors completing a unit of work at time τ of the computation (according to the global time that is not available to the processors). Let τ_α 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 solving a given problem of size t the work complexity W is defined as $W(p,t) = \max\left\{\sum_{\tau \leqslant \tau_\alpha} p_\tau(\alpha) : \alpha \in \mathcal{E}\right\}$. For a randomized algorithm the expected work complexity EW is defined as $EW(p,t) = \mathbf{E}\left\{\sum_{\tau \leqslant \tau_\alpha} p_i(\alpha) : \alpha \in \mathcal{E}\right\}$.

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 α of an algorithm, we denote by $m_\tau(\alpha)$ the number of (point-to-point) messages sent at time τ of the computation (according to some global time that is not available to the processors).

Definition 2.2. For a p-processor algorithm solving a given problem of size t the message complexity M is defined as $M(p,t) = \max\left\{\sum_{\tau \leqslant \tau_\alpha} m_\tau(\alpha) : \alpha \in \mathcal{E}\right\}$. For a randomized algorithm the expected message complexity EM is defined as $EM(p,t) = \mathbf{E}\left\{\sum_{\tau \leqslant \tau_\alpha} m_\tau(\alpha) : \alpha \in \mathcal{E}\right\}$.

In the cases where work or message complexity is a function of p, t, and d, we use the notation $W(p,t,d)$ to denote work and $M(p,t,d)$ to denote message complexity. Expected work and message complexity are denoted by $EW(p,t,d)$ and $EM(p,t,d)$, respectively. The above notation is convenient for expressing complexity results obtained for the d-adversary, where the set of executions $\mathcal{E}$ contains all executions in the presence of the d-adversary.

Next, we formulate a proposition leading us to not consider algorithms where a processor may halt voluntarily before learning that all tasks have been performed.

Proposition 2.1. *Let Alg be a* Do-All *algorithm such that there is some execution α of Alg in which there is a processor that (voluntarily) halts before it learns that all tasks have been performed. Then there is an execution α' of Alg with unbounded work in which some task is never performed.*

Proof. For the proof we assume a stronger model of computation where in one local step any processor can learn the complete state of another processor, including, in particular, the complete computation history of the other processor. Assume that, in some execution α, the Do-All problem is solved, but some processor i halts in α without learning that certain task z was performed. First, we observe that for any other processor j that i learns about in α, j does not perform task z by the time i learns j's state. (Otherwise i would know that z was performed.) We construct another execution α' from α as follows. Any processor j (except for i) proceeds as in α until it attempts to perform task z. Then j is delayed forever. We show that processor i can proceed exactly as in α. We claim that i is not able to distinguish between α and α'. Consider the histories of all processors that i learned about in α' (directly or indirectly). None of the histories contain information about task z being performed. Thus, the history of any processor j was recorded in advance of j's delay in α'. Then by the definition of α' these histories are identical to those in α. This means that in α'

processor i halts as in α. Since the problem remains unsolved, processor i continues to be charged for each local clock tick (recall that work is charged until the problem is solved). □

As the result of Proposition 2.1, in this paper we will only consider algorithms where a processor may voluntarily halt only after it knows that all tasks are complete, i.e., for each task the processor has local knowledge that either it performed the task or that some other processor did.

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* $(c \cdot t)$*-adversary, for a constant* $c > 0$, *has work* $W(p, t) = \Omega(t \cdot p)$.

Proof. We choose the adversary that delays each message by $c \cdot t$ time units, and does not delay any processor. If a processor halts voluntarily before learning that all tasks are complete, then by Proposition 2.1 work may be unbounded. Assume then that no processor halts voluntarily until it learns that all tasks are done. A processor may learn this either by performing all the tasks by itself and contributing t to the work of the system, or by receiving information from other processors by waiting for messages for $c \cdot t$ time steps. In either case, the contribution is $\Omega(t)$ to the work of the algorithm. Since there are p processors, the work is $\Omega(t \cdot p)$. □

Lastly, we note that since in our study we are trading communication for work, we design algorithms with the focus on work.

3. Delay-sensitive 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

First, we prove a lower bound on work that shows how the efficiency of work-performing algorithms depends on the number of processors, the number of tasks, and the message delay.

Theorem 3.1. *Any deterministic algorithm solving* Do-All *with t tasks using p asynchronous message-passing processors performs work*

$$W(p, t, d) = \Omega(t + p \min\{d, t\} \log_{d+1}(d + t))$$

against the d-adversary.

Proof. That the required work is at least t is obvious—each task must be performed. We present the analysis for $t > 5$ and t that is divisible by 6 (this is sufficient to prove the lower bound). We present the following adversarial strategy. The adversary partitions computation into stages, each containing $\min\{d, t/6\}$ steps. We assume that the adversary delivers all messages sent to a processor in stage s at the end of stage s (recall that the receiver can process any such message later, according to its own local clock)—this is allowed since the length of stage s is at most d. For stage s, we will define the set of processors P_s such that the adversary delays all processors not in P_s. More precisely, each processor in P_s is not delayed during stage s, but any processor not in P_s is delayed so it does not complete any step during stage s.

Consider stage s. Let $u_s > 0$ be the number of tasks that remain unperformed at the beginning of stage s, and let U_s be the set of such tasks. We now show how to define the set P_s. Suppose first that each processor is not delayed during stage s (with respect to the time unit). Let $J_s(i)$, for every processor i, denote the set of tasks from U_s (we do not consider tasks not in U_s in the analysis of stage s since they were performed before) which are performed by processor i during stage s (recall that inside stage s processor i does not receive any message from other processors, by the assumption on consider kind of the adversary). Note that $|J_s(i)|$ is at most $\min\{d, t/6\}$, which is the length of a stage.

Claim. *There are at least* $\frac{u_s}{3\min\{d,t/6\}}$ *tasks z such that each of them is contained in at most* $2p\min\{d, t/6\}/u_s$ *sets in the family* $\{J_s(i) \mid i = 0, \ldots, p-1\}$.

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

$$\begin{aligned} p\min\{d,t/6\} &= \sum_{0\leqslant i<p} |J_s(i)| \\ &\geqslant \left(u_s - \frac{u_s}{3\min\{d,t/6\}}\right) \cdot \frac{2p\min\{d,t/6\}}{u_s} \\ &= \left(2 - \frac{2}{3\min\{d,t/6\}}\right) \cdot p\min\{d,t/6\} \\ &> p\min\{d,t/6\}\,, \end{aligned}$$

since $d \geqslant 1$ and $t > 4$. This proves the claim.

We denote the set of $\frac{u_s}{3\min\{d,t/6\}}$ tasks from the above claim by J_s. We define P_s to be the set $\{i : J_s \cap J_s(i) = \emptyset\}$. By the definition of tasks $z \in J_s$ we obtain that

$$|P_s| \geqslant p - \frac{u_s}{3\min\{d,t/6\}} \cdot \frac{2p\min\{d,t/6\}}{u_s} \geqslant p/3\,.$$

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

If $d < t/6$ then work during stage s is at least $p\,d/6$, and there remain at least $\frac{u_s}{3d}$ 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 are performed. The total work is then $\Omega(p\,d\log_{d+1}(d+t))$.

If $d \geqslant t/6$ 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 work 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. Delay-sensitive lower bound for randomized algorithms

In this section, we prove a delay-sensitive lower bound for randomized work-performing algorithms. We first state a technical lemma used in the lower bound proof (the proof of this lemma is found in Appendix A).

Lemma 3.2. *For* $1 \leqslant d \leqslant \sqrt{u}$

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

The idea behind the lower bound proof for randomized algorithms is similar to the one for deterministic algorithms in the previous section, except that sets $J_s(i)$ are random, hence we have to modify the construction of set P_s also. We partition the execution of the algorithms into stages, similarly to the lower bound for deterministic algorithms. Let V be the set of p processors. Let U_s denote the remaining tasks at the beginning of stage s. Suppose first that all processors are not delayed during stage s, and the adversary delivers all messages sent to processor i during stage s at the end of stage s. The set $J_s(i)$, for processor $i \in V$, denotes a certain set of tasks from U_s that i is going to perform during stage s. The size of $J_s(i)$ is at most d, because we consider at most d steps in advance (the adversary may delay all messages by d time steps, and so the choice of $J_s(i)$ does not change during next d steps, provided $|J_s(i)| \leqslant d$). The key point is that the set $J_s(i)$ is random, since we consider randomized algorithms, and so we deal with the probabilities that $J_s(i) = Y$ for the set of tasks $Y \subseteq U_s$ of size at most d. We denote these probabilities by $p_i(Y)$. For the given set of processors P, let $J_s(P)$ denote set $\bigcup_{i \in P} J_s(i)$.

The goal of the adversary is to prevent the processors from completing some sufficiently large set J_s of tasks during stage s. Here, we are interested in the events where there is a set of processors P_s that is "large enough" (linear size) so that the processors do not perform any tasks from J_s.

In the next lemma, we prove that, for some set J_s, such set of processors P_s exists with high probability. This is the main difference compared to the deterministic lower bound—instead of finding a suitably large set J_s *and* a linear-size set P_s, we prove that the set J_s exists, and we prove that the set P_s of processors not performing this set of tasks during stage *s exists with high probability*. However, in the final proof, the existence with high probability is sufficient—we can define the set on-line using the rule that if some processor wants to perform a task from the chosen set J_s, then we delay it, and do not put it in P_s. In the next lemma, we assume that s is known, so we skip lower index s from the notation for clarity of presentation.

Lemma 3.3. *There exists set* $J \subseteq U$ *of size* $\frac{u}{d+1}$ *such that*

$$\mathbf{Pr}[\exists_{P \subseteq V} : |P| = p/64 \ \wedge \ J(P) \cap J = \emptyset] \geqslant 1 - e^{-p/512} .$$

Proof. First, observe that

$$\sum_{\left(J:J\subseteq U,|J|=\frac{u}{d+1}\right)}\sum_{(v\in V)}\sum_{(Y:\,Y\subseteq U,Y\cap J=\emptyset,|Y|\leqslant d)} p_v(Y)$$
$$=\sum_{(v\in V)}\sum_{(Y:\,Y\subseteq U,|Y|\leqslant d)} p_v(Y)\cdot\binom{u-|Y|}{u/(d+1)}$$
$$\geqslant p\cdot\binom{u-d}{u/(d+1)}\,.$$

It follows that there exists set $J \subseteq U$ of size $\frac{u}{d+1}$ such that

$$\sum_{(v\in V)}\sum_{(Y:\,Y\subseteq U,Y\cap J=\emptyset,|Y|\leqslant d)} p_v(Y) \geqslant \frac{p\cdot\binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} \geqslant \frac{p}{4}\,, \tag{1}$$

where the last inequality follows from Lemma 3.2. Fix such a set J. For every node $v \in V$, let

$$S_v = \sum_{(Y:Y\subseteq U,Y\cap J=\emptyset,|Y|\leqslant d)} p_v(Y).$$

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

$$S_v \geqslant \frac{1}{8}\,.$$

(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 $S_v \leqslant 1$. 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

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

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

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

Finally, observe that

$$\mathbf{Pr}\left[\exists_{P\subseteq V} : |P| = p/64 \ \wedge \ J(P) \cap J = \emptyset\right]$$

$$\geqslant 1 - \mathbf{Pr}\left[\sum_{v\in V'} X_v < p/64\right],$$

which completes the proof of the lemma. □

We apply Lemma 3.3 in proving the following lower bound result.

Theorem 3.4. *Any randomized algorithm solving* Do-All *with t tasks using p asynchronous message-passing processors performs expected work*

$$EW(p,t,d) = \Omega(t + p\min\{d,t\}\log_{d+1}(d+t))$$

against any d-adversary.

Proof. That the lower bound of $\Omega(t)$ holds with probability 1 is obvious. We consider three cases, depending on how large is d comparing to t: in the first case d is very small comparing to t (in this case the thesis follows from the simple calculations), in the second case we assume that d is larger than in the first, but still no more than $\sqrt{t}$ (this is the main case), and in the third case d is large than $\sqrt{t}$ (here the proof is similar to the second case, but is restricted to one stage). We now give the details.

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

This case is a simple derivation. 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 $d \geqslant 1$ and by the properties of the logarithm function.

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

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

holds, with the probability 1, in this case.

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

Consider any Do-All algorithm. Similarly as in the 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 $s-1$. Consider stage s. We assume that the adversary delivers to a processor all messages sent in stage s at the end of stage s, provided the processor is not delayed at the end of stage s (any such message is processed by the receivers at a later time). Let $U_s \subseteq T$ denote set of tasks that remain unperformed by the end of stage $s-1$. Here, by the adversarial strategy (no message is received and processed during stage s),

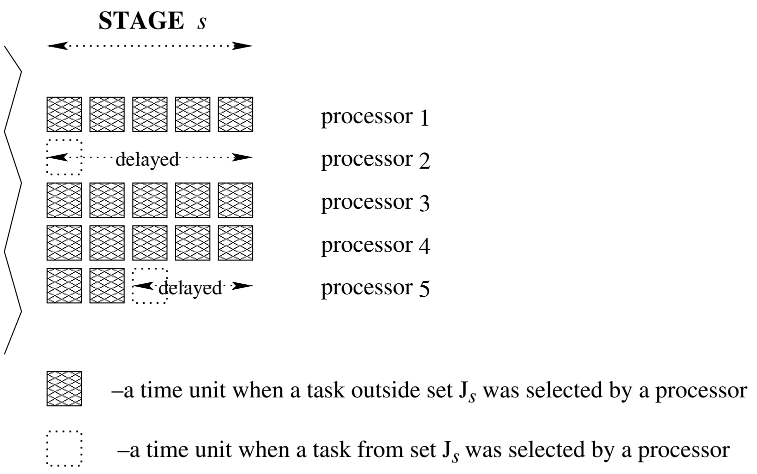
STAGE s
processor 1
delayed
processor 2
processor 3
processor 4
delayed
processor 5
–a time unit when a task outside set J_s was selected by a processor
–a time unit when a task from set J_s was selected by a processor

Fig. 1. Strategy of the adversary during stage s, where $p = d = 5$. Using the set J_s, which exists by Lemma 3.3, the adversary delays a processor from the moment where it wants to perform a task from J_s. Lemma 3.3 guarantees that at least a fraction of processors will not be delayed during stage s, with high probability.

given that the execution is fixed at the end of stage $s - 1$, one can fix a distribution of processor i performing the set of tasks Y during stage s—this distribution is given by the probabilities $p_i(Y)$. The adversary derives the set $J_s \subseteq U_s$, using Lemma 3.3 according to the set of all processors, the set of the unperformed tasks U_s, and the distributions $p_i(Y)$ fixed at the beginning of stage s according to the action of processors i in stage s. (In applying Lemma 3.3 we use the same notation, except that the quantities are subscripted according to the stage number s.)

The adversary additionally delays any processor i, not belonging to some set P_s, that attempts to perform a task from J_s before the end of stage s. The set P_s is defined on-line (this is one of the difference between the adversarial constructions in the proofs of the lower bounds for deterministic and randomized Do-All algorithms): at the beginning of stage s set P_s contains all processors; every processor i that is going to perform some task $z \in J_s$ at time τ in stage s is delayed till the end of stage s and removed from set P_s. We illustrate the adversarial strategy for five processors and $d = 5$ in Fig. 1.

We now give additional details of the adversarial strategy. Suppose $u_s = |U_s| > 0$ tasks remain unperformed at the beginning of stage s. As described above, we apply Lemma 3.3 to the set U_s and probabilities $p_i(Y)$ to find, at the very beginning of stage s, the set $J_s \subseteq U_s$ such that the probability that there exists a subset of processors P_s of cardinality $p/64$ such that none of them would perform any tasks from J_s during stage s is at least $1 - e^{-p/512}$. Next, during stage s the adversary delays (to the end of stage s) all processors that (according to the random choices during stage s) are going to perform some task from J_s. By Lemma 3.3, the set P_s of not-delayed processors contains at least $p - 63p/64 \geqslant p/64$ processors, and the set of the remaining tasks $U_{s+1} \supseteq J_s$ contains at least $\frac{u_s}{d+1}$ tasks, all with probability at least $1 - e^{-p/512}$. If this happens, we call stage s *successful.*

It follows that the probability, that every stage $s < \log_{d+1} t$ is successful is at least $1 - e^{-p/512} \cdot \log_{d+1} t$. Hence, using the assumption for this case, with the probability at least $1 - e^{-p/512} \cdot \log_{d+1} t \geqslant 1/2$, at the beginning of stage s 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 the work in one successful stage is at least $p/64$ (the number of undelayed processors) times d (the duration of one stage). It follows that the expected work of this algorithm in the presence of our adversary is $\Omega(p\, d \log_{d+1} t) = \Omega(p\, d \log_{d+1}(d + t))$, because $1 \leqslant d \leqslant \sqrt{t}$. This completes the proof of Case 2.

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

Here, we follow similar reasoning as in the Case 2, except that we consider a single stage. Consider first $\min\{d, t/6\}$ steps. Let T be the set of all tasks, and $p_i(Y)$ denote the probability that processor $i \in V$ performs tasks in $Y \subseteq T$ of cardinality $\min\{d, t/6\}$ during the considered steps. Applying Lemma 3.3 we obtain that at least $p/64$ processors are undelayed during the considered steps, and after these steps at least $\frac{\min\{d,t/6\}}{d+1} \geqslant 1$ tasks remain unperformed, all with the probability at least $1 - e^{-p/512}$. Since $1 \leqslant \log_{d+1}(d + t) < 2$, work is $\Omega(p \min\{d, t/6\}) = \Omega(p \min\{d, t\} \log_{d+1}(d + t))$. This completes the proof of the third case and of the theorem. □

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 $\langle \cdots \rangle$ 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.

We next provide a motivation for the material in this section. Consider the situation where two asynchronous processors, $\mathfrak{p}_1$ and $\mathfrak{p}_2$, need to perform t independent tasks with known unique identifiers from the set $[t] = \{1, \ldots, t\}$. Assume that before starting a task, a processor can check whether the task is complete; however, if both processors work on the task concurrently, then the task is done twice. We are interested in the number of tasks done redundantly. Let $\pi_1 = \langle a_1, \ldots, a_t \rangle$ be the sequence of tasks giving the order in which $\mathfrak{p}_1$ intends to perform the tasks. Similarly, let $\pi_2 = \langle a_{s_1}, \ldots, a_{s_t} \rangle$ be the sequence of tasks of $\mathfrak{p}_2$. We can view π_2 as π_1 permuted according to $\sigma = \langle s_1, \ldots, s_t \rangle$ (π_1 and π_2 are permutations). With this, it is possible to construct an asynchronous execution for $\mathfrak{p}_1$ and $\mathfrak{p}_2$, where $\mathfrak{p}_1$ performs all t tasks by itself, and any tasks that $\mathfrak{p}_2$ finds to be unperformed are performed redundantly by both processors.

In the current context, it is important to understand how does the structure of π_2 affect the number of redundant tasks. Clearly, $\mathfrak{p}_2$ may have to perform task a_{s_1} redundantly. What about a_{s_2}? If $s_1 > s_2$ then by the time $\mathfrak{p}_2$ gets to task a_{s_2}, it is already done by $\mathfrak{p}_1$ according to π_1. Thus, in order for a_{s_2} to be done redundantly, it must be the case that $s_2 > s_1$. It is easy to see, in general, that for task a_{s_j} to be done redundantly, it must be the case that $s_j > \max\{s_1, \ldots, s_{j-1}\}$. Knuth [17] refers to such s_j as a *left-to-right maximum* of σ. The total number of tasks done redundantly by $\mathfrak{p}_2$ is thus the number of left-to-right maxima of σ. Not surprisingly, this number is minimized when $\sigma = \langle t, \ldots, 1 \rangle$, i.e, when π_2 is the reverse order of π_1, and it is maximized when $\sigma = \langle 1, \ldots, t \rangle$, i.e., when $\pi_1 = \pi_2$. In this section, we will define the notion *contention* of permutations that captures the relevant left-to-right maxima properties of permutations that are to be used as processor schedules.

Now we proceed with formal development. 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

```
00 const $\Psi = \{\pi_r \mid 0 \leq r < n \wedge \pi_r \in S_n\}$     % Fixed set of n permutations of [n]
01 forall processors $pid = 0$ to $n-1$ parbegin
02     for $r = 1$ to $n$ do
03         perform $Job(\pi_{pid}(r))$
04     od
05 parend.
```

Fig. 2. Algorithm OBLIDO.

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 a *n-schedule*. Here, S_n is the symmetric group, the group of all permutations on the set $[n]$; we use the symbol $\circ$ to denote 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 [17, vol. 3, p. 13]) 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 $\text{LRM}(\pi)$ 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 $\text{Cont}(\Psi, \delta) = \sum_{u=0}^{n-1} \text{LRM}(\delta^{-1} \circ \pi_u)$. The *contention of the list of schedules* Ψ is defined as $\text{Cont}(\Psi) = \max_{\delta \in S_n}\{\text{Cont}(\Psi, \delta)\}$. Note that for any Ψ, we have $n \leqslant \text{Cont}(\Psi) \leqslant n^2$. A family of permutations with low contention was introduced in [2], where the following is shown $\left(H_n \text{ is the } n\text{th harmonic number, } H_n = \sum_{j=1}^{n} \frac{1}{j}\right)$.

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

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

4.1. Contention and oblivious tasks scheduling

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[1] in Fig. 2. (Here, each "processor" may be modeling a group of processors following the same sequence of activities.)

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 $\text{Cont}(\Psi)$, 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.

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

Note that while an algorithm solving the Do-All problem may attempt to reduce the number of secondary job executions by sharing information about complete jobs among the processors, it is not possible to eliminate (redundant) primary job executions in the asynchronous model we consider. The following lemma [2] formalizes the relationship between the primary job executions and the contention of permutations used as schedules.

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* $\mathrm{Cont}(\Psi)$.

4.2. Generalized contention

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* (or d-lrm for short) if the number of 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

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

The *d-contention of the list of schedules* Ψ is defined as

$$(d)\text{-}\mathrm{Cont}(\Psi) = \max_{\delta \in S_n} \{(d)\text{-}\mathrm{Cont}(\Psi, \delta)\} \ .$$

We first show a lemma about the d-contention of a set of permutations with respect to $\mathbf{u}_n$, the identity permutation.

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)-$\mathrm{Cont}(\Psi, \mathbf{u}_n) > n \ln n + 8p\,d \ln(e + n/d)$ *is at most* $e^{-(n \ln n + 7p\,d \ln(e + \frac{n}{3d})) \ln(7/e)}$.

Proof. For $d \geqslant n/5$ the thesis is obvious. In the remainder of the proof we assume $d < n/5$.

First, we describe a well-known method for generating a random schedule by induction on the number of elements $n' \leqslant n$ to be permuted. For $n' = 1$ the schedule consists of a single element chosen uniformly at random. Suppose we can generate a random schedule of $n' - 1$ different elements. Now, we show how to schedule n' elements uniformly and independently at random. First, we choose uniformly and independently at random one element among n' and put it as the last element in the schedule. By induction we generate random schedule from remaining $n' - 1$ elements and put them as the first $n' - 1$ elements. Simple induction proof shows that every obtained schedule of n' elements has equal probability (since the above method is a concatenation of two independent and random events).

A random list of schedules Ψ can be selected by using the above method p times, independently.

For a schedule $\pi \in \Psi$, let $X(\pi, i)$, for $i = 1, \ldots, n$, be a random value such that $X(\pi, i) = 1$ if $\pi(i)$ is a d-lrm, and $X(\pi, i) = 0$ otherwise.

Claim. *For any $\pi \in \Psi, X(\pi, i) = 1$ with probability $\min\{d/i, 1\}$, independently from other values $X(\pi, j)$, for $j > i$. Restated precisely, we claim that $\mathbf{Pr}[X(\pi, i) = 1 | \bigwedge_{j>i} X(\pi, j) = a_j] = \min\{d/i, 1\}$, for any 0–1 sequence $a_{i+1}, \ldots, a_n$.*

This is so because $\pi(i)$ might be a d-lrm if during the $(n - i - 1)$th step of generating π, we select uniformly and independently at random one among the d greatest remaining elements (there are i remaining elements in this step). This proves the claim.

Note that

(1) for every $\pi \in \Psi$ and every $i = 1, \ldots, d$, $\pi(i)$ is d-lrm, and
(2) $\mathbf{E}\left[\sum_{\pi\in\Psi} \sum_{i=d+1}^{n} X(\pi, i)\right] = p\,d \cdot \sum_{i=d+1}^{n} \frac{1}{i} = p\,d\,(H_n - H_d)$.

Applying the well-known Chernoff bound of the following form: for 0–1 independent random variables Y_j and any constant $b > 0$,

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

and using the fact that $2 + \dfrac{n \ln n}{p\,d(H_n - H_d)} > 0$, we obtain

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi, i) > n\ln n + 3p\,d(H_n - H_d)\right] \\
&= \mathbf{Pr}\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi, i) > p\,d(H_n - H_d)\left(1 + \left(2 + \frac{n\ln n}{p\,d(H_n - H_d)}\right)\right)\right] \\
&\leqslant e^{-(n\ln n + 3p\,d(H_n - H_d))\ln\frac{n\ln n + 3p\,d(H_n-H_d)}{e\cdot p\,d(H_n-H_d)}} \\
&\leqslant e^{-[n\ln n + 3p\,d(H_n - H_d)]\ln(3/e)}.
\end{aligned}$$

Since $\ln i \leqslant H_i \leqslant \ln i + 1$ and $n > 5d$, we obtain that

$$\begin{aligned}
&\mathbf{Pr}\left[\sum_{\pi\in\Psi}\sum_{i=1}^{n} X(\pi, i) > n\ln n + 5p\,d\ln\left(e + \frac{n}{d}\right)\right] \\
&\qquad\leqslant \mathbf{Pr}\left[\sum_{\pi\in\Psi}\sum_{i=d+1}^{n} X(\pi, i) > n\ln n + 3p\,d(H_n - H_d) + p\,d\right] \\
&\qquad\leqslant e^{-[n\ln n + 3p\,d(H_n - H_d)]\ln(3/e)}. \qquad \square
\end{aligned}$$

Now we generalize the result of Lemma 4.3.

Theorem 4.4. *For a random list of schedules Ψ containing p permutations from S_n, the event:*

"for every positive integer d, (d)-Cont$(\Psi) > n\ln n + 8p\,d\ln(e + n/d)$,"

holds with probability at most $e^{-n\ln n\cdot\ln(7/e^2) - p}$.

Proof. For $d \geqslant n/5$ 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)-Cont$(\Psi, \sigma) > n \ln n + 8p\,d \ln(e + n/d)$ holds with probability at most $e^{-[n \ln n + 7p\,d \ln(e+\frac{n}{3d})] \ln \frac{7}{e}}$.

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

$$n! \cdot e^{-[n \ln n + 7p\,d \ln(e+\frac{n}{3d})] \ln \frac{7}{e}} \leqslant e^{n \ln n - [n \ln n + 7p\,d \ln(e+\frac{n}{3d})] \ln \frac{7}{e}}$$
$$\leqslant e^{-n \ln n \cdot \ln \frac{7}{e^2} - 7p\,d \ln(e+\frac{n}{d})} .$$

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

$$\sum_{d=1}^{\infty} \mathbf{Pr}\big[(d)\text{-Cont}(\Psi) > n \ln n + 8p\,d \ln(e + n/d)\big]$$
$$\leqslant \sum_{d=1}^{n/5-1} e^{-n \ln n \cdot \ln(7/e^2) - 7p\,d \ln(e+n/d)} + \sum_{d=n/5}^{\infty} 0$$
$$\leqslant e^{-n \ln n \cdot \ln(7/e^2)} \cdot \sum_{d=1}^{n/5-1} (e^{-7p})^d$$
$$\leqslant e^{-n \ln n \cdot \ln(7/e^2)} \cdot \frac{e^{-7p}}{1 - e^{-7p}}$$
$$\leqslant e^{-n \ln n \cdot \ln(7/e^2) - p} . \quad \square$$

Using the probabilistic method we obtain the following.

Corollary 4.5. *There is a list of p schedules Ψ from S_n such that* (d)-Cont$(\Psi) \leqslant n \log n + 8p\,d \ln(e + n/d)$, *for every positive integer d.*

We put our generalized notion of contention to use in the delay-sensitive analysis of work-performing algorithms in Section 6.

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 $t \geqslant p$, and its message complexity M is $O(p \cdot W)$.

5.1. Construction of algorithm DA(q)

Let q be some constant such that $2 \leqslant q \leqslant p$. 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). In this section, we first present the algorithm for $p \geqslant t$, then state the parameterization for $p < t$.

Algorithm DA(q), utilizes a q-ary boolean *progress tree* with t leaves, where the tasks are associated with the leaves. 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 unperformed tasks. 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 m, 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 m-th digit in the q-ary expansion of the processor's identifier.

5.1.1. 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 $Tree[0..l-1]$, where $Tree[0]$ is the root, and the q children of the interior node $Tree[n]$ being the nodes $Tree[qn+1], Tree[qn+2], \ldots, Tree[qn+q]$. 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 $Tree[n]$ 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 $x[0..h-1]$.

5.1.2. Control flow

The code is given in Fig. 3. 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 *Tree* 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.

```
00 const $q$                                          % Arity of the progress tree
01 const $\Psi = \langle \pi_r \mid 0 \le r < q \wedge \pi_r \in S_q \rangle$   % Fixed list of q permutations of [q]
02 const $l = (qt-1)/(q-1)$                           % The size of the progress tree
03 const $h = \log_q t$                               % The height of the progress tree
04 type ProgressTree: array $[0 .. l-1]$ of boolean   % Progress tree
05 forall processors $pid = 0$ to $p-1$ parbegin
06     ProgressTree $Tree_{pid}$                      % 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 $m$ init $= 0$                     % Current depth in the tree
13         DOWORK$(n, m)$
14     end
20     thread                                         % Receive broadcast messages
21         set of ProgressTree $B$                    % Incoming messages
22         while $Tree_{pid}[0] \neq 1$ do            % While not all tasks certified
23             receive $B$                            % Deliver the set of received messages
24             $Tree_{pid} := Tree_{pid} \vee (\bigvee_{b \in B} b)$   % Learn progress
25         od
26     end
27 parend.
40 procedure DOWORK$(n, m)$                           % Recursive progress tree traversal
41                                                    % n : current node index ; m : node depth
42 const array $x[0 .. h-1] = pid_{(base\ q)}$        % h digits of q-ary expansion of pid
43     if $Tree_{pid}[n] = 0$ then                    % Node not done – still work left
44         if $m = h$ then                            % Node n is a leaf
45             perform $Task(t - l + n + 1)$          % Do the task
46         else                                       % Node n is not a leaf
47             for $r = 1$ to $q$ do                  % Visit subtrees in the order of $\pi_{x[m]}$
48                 DOWORK$(qn + \pi_{x[m]}(r),\ m+1)$
49             od
50         fi
51         $Tree_{pid}[n] := 1$                       % Record completion of the subtree
52         broadcast $Tree_{pid}$                     % Share the good news
53     fi
54 end.
```

Fig. 3. The deterministic algorithm DA ($p \geqslant t$).

The progress tree is traversed using the recursive procedure DOWORK (lines 40–54). 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 m, the mth q-ary digit $x[m]$ of its pid to select the permutation $\pi_{x[m]}$ from Ψ. The processor traverses the q subtrees in the order determined by $\pi_{x[m]}$ (lines 47–49), but it traverses within a subtree only if the corresponding bit in the progress tree is not set (line 43). 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.

5.1.3. Parameterization for large number of tasks

When the number of tasks t' exceeds the number of processors p, we divide the tasks into *jobs*, where each job consists of at most $\lceil t'/p \rceil$ tasks. The algorithm in Fig. 3 is then used with the resulting p jobs ($p = t$), where $Task(j)$ now refers to the job number j ($1 \leqslant j \leqslant t$).

5.1.4. 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. We formalize this as follows.

Lemma 5.1. *In any execution of algorithm* DA(q), *whenever a processor returns from a call to* DoWork(n, m), *all tasks associated with the leaves that are the descendants of node n have been performed.*

Proof. First, by code inspection (Fig. 3, lines 45, 51, and 52), we note that processor *pid* reaching a leaf n at depth $m = h$ broadcasts its $Tree_{pid}$ with the value $Tree_{pid}[n]$ set to 1 if and only if it performs the task corresponding to the leaf.
We now proceed by induction on m.
Base case, $m = h$:
In this case, processor *pid* makes the call to DoWork(n, m). If $Tree_{pid}[n] = 0$, as we have already observed, the processor performs the task at the leaf (line 45), broadcasts its $Tree_{pid}$ with the leaf value set to 1 (lines 51–52), and returns from the call. If $Tree_{pid}[n] \neq 0$ then the processor must have received a message from some other processor indicating that the task at the leaf is done. This can be so if the sender itself performed the task (as observed above), or the sender learned from some other processor the fact that the task is done.

Inductive step, $0 \leqslant m < h$:
In this case, processor *pid* making the call to DoWork(n, m) executes q calls to DoWork($n', m + 1$), one for each child n' of node n (lines 47–49). By inductive hypothesis, each return from DoWork($n', m + 1$) indicates that all tasks associated with the leaves that are the descendants of node n' have been performed. The processor then broadcasts its $Tree_{pid}$ with the the value $Tree_{pid}[n]$ set to 1 (lines 51–52), indicating that all tasks associated with the leaves that are the descendants of node n have been performed, and returns from the call. □

Theorem 5.2. *Any execution of algorithm* DA(q) *terminates in finite time having performed all tasks.*

Proof. The progress tree used by the algorithm has finite number of nodes. By code inspection, each processor executing the algorithm makes at most one recursive call per each node of the tree. Thus, the algorithm terminates in finite time. By Lemma 5.1, whenever a processor returns from the call to DoWork($n(= 0), m(= 0)$), all tasks associated with the leaves that are the descendants of the node $n = 0$ are done, and the value of node is set to 1. Since this node is the root of the tree, all tasks are done. □

5.2. Complexity analysis of algorithm DA(q)

We start by showing a lemma that relates the work of the algorithm, against the d-adversary, to its recursive structure.

We consider the case $p \geqslant t$. Let $W(p, t, d)$ 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 $W(p, 1, d) = \mathrm{O}(p)$. 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 5.3. *For p-processor, t-task algorithm* DA(q) *with* $p \geqslant t$ *and t and p divisible by q:*

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

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 (lines 47–49) 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 47–49 of DA is essentially algorithm ObliDo (lines 02–04 in Fig. 2) 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 *pid*s 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 $W(p/q, t/q, d)$, since each group has p/q processors and the job includes t/q tasks. Using Lemma 4.2 the primary task executions contribute $\mathrm{O}(\mathrm{Cont}(\Psi) \cdot W(p/q, t/q, d))$ 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 $\mathrm{O}(t/q)$ steps performing a secondary job (this is because it takes a single processor $\mathrm{O}(t/q)$ 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 $\mathrm{O}(p \cdot q \cdot \min\{d, t/q\})$ 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 $\mathrm{O}(p \cdot q)$. 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. □

We now prove the following theorem about work.

Theorem 5.4. *Consider algorithm* DA(q) *with p processors and t tasks where* $p \geqslant t$. *Let d be the maximum message delay. For any constant* $\varepsilon > 0$ *there is a constant q such that the algorithm has work* $W(p, t, d) = \mathrm{O}(p \min\{t, d\} \lceil t/d \rceil^{\varepsilon})$.

Proof. Fix a constant $\varepsilon > 0$. Without loss of generality we can assume that $\varepsilon \leqslant 1$. Let a be the sufficiently large positive constant "hidden" in the big-oh upper bound for $W(p,t,d)$ in Lemma 5.3. We consider a constant $q > 0$ such that $\log_q(4a\log q) \leqslant \varepsilon$. Such q exists since $\lim_{q\to\infty}\xi = \lim_{q\to\infty}\log_q(4a\log q) = 0$ (however, q is a constant of order $2^{\frac{\log(1/\varepsilon)}{\varepsilon}}$).

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

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

For the base case of $t = 1$ the statement is correct since $W(p,1,d) = O(p)$. For $t > 1$ we choose the list of permutations Ψ with $\mathrm{Cont}(\Psi) \leqslant 3q\log q$ per Lemma 4.1. Due to our choice of parameters, $\log_q t$ is an integer and $t \leqslant p$. Let ξ stand for $\log_q(4a\log q)$. Using Lemma 5.3 and inductive hypothesis we obtain

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

We now consider two cases:

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

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

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

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

Putting everything together we obtain the desired inequality

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

To complete the proof, consider any $t \leqslant p$. 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

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

Since $\xi \leqslant \varepsilon$, 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. □

Now we consider the case $p < t$. Recall that in this case we divide the t tasks into p jobs of size at most $\lceil t/p\rceil$, and we let the algorithm work with these jobs. It takes a processor $O(t/p)$ work (instead of a constant) to process a single job.

Theorem 5.5. *Consider algorithm* DA(q) *with p processors and t tasks where $p < t$. Let d be the maximum message delay. For any constant $\varepsilon > 0$ there is a constant q such that* DA(q) *has work* $W(p,t,d) = \mathrm{O}(tp^{\varepsilon} + p\min\{t,d\}\lceil t/d\rceil^{\varepsilon})$.

Proof. We use Theorem 5.4 with p jobs (instead of t tasks), where a single job takes $\mathrm{O}(t/p)$ units of work. The upper bound on the maximal delay for receiving messages about the completion of some job is $d' = \lceil p\,d/t\rceil = \mathrm{O}(1 + p\,d/t)$ "job units," where a single job unit takes $\Theta(t/p)$ time. We obtain the following bound on work:

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

Finally, we consider message complexity.

Theorem 5.6. *Algorithm* DA(q) *with p processors and t tasks has message complexity* $M(p,t,d) = \mathrm{O}(p\cdot W(p,t))$.

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

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 are complete, performs a specific task from its local list and broadcasts this fact to other processors. The known complete tasks are removed from the list. The code is given in Fig. 4. 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, implemented by the call to procedure Order on line 20.
(2) The selection of the next task to perform, implemented by the call to function Select on line 24.

We now describe the specialization of the code made by each algorithm (the code for Order+Select).

As with algorithm DA, we initially consider the case of $p \geqslant t$. The case of $p < t$ 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.

```
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 $Tasks_{pid} \neq \emptyset$ do
22         receive $B$                             % Deliver the set of received messages
23         $Tasks_{pid} := Tasks_{pid} - (\bigcup_{b \in B} b)$                   % 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 $\Psi = \langle \text{TaskList } \pi_r \mid 0 \leq r < p \ \wedge \ \pi_r =$ random list of$[t]\rangle$
42  % $\Psi$ is a list of $p$ random permutations
43     procedure ORDER($T$) begin  $T := \pi_{pid}$  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 list $\Psi = \langle \text{TaskList } \pi_r \mid 0 \leq r < p \ \wedge \ \pi_r \in S_t \rangle$
62  % $\Psi$ is a fixed list of $p$ permutations
63     procedure ORDER($T$) begin  $T := \pi_{pid}$  end
64     TaskId function SELECT($T$) begin  return($T(1)$)  end
```

Fig. 4. Permutation algorithm and its specializations for PaRan1, PaRan2, and PaDet ($p \geqslant t$).

Randomized algorithm PaRan1. The specialized code is in Fig. 4, 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 $p \cdot \min\{t, p\}$, each of $O(\log \min\{t, p\})$ random bits (we have $\min\{t, p\}$ above because when $p < t$, we use p jobs, each of size $\lceil t/p \rceil$, instead of t tasks).

Randomized algorithm PaRan2. The specialized code is in Fig. 4, 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 EW is the same for algorithms PaRan1 and PaRan2, however, the (expected) number of random bits needed by PaRan2 becomes at most $EW \cdot \log t$ and, as we will see, this is an improvement.

Deterministic algorithm PaDet. The specialized code is in Fig. 4, 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

In the analysis, we use the quantity n defined as $n = \min\{t, p\}$. When $t < p$, n represents the number of tasks to be performed. When $t \geqslant p$, n represents the number of jobs (of size at most $\lceil t/p \rceil$) to be performed; in this case, each task in Fig. 4 represents a single job. In the sequel we continue referring to "tasks" only—from the combinatorial perspective there is no distinction between a task and a job, and the only accounting difference is that a task costs $\Theta(1)$ work, while a job costs $\Theta(\lceil t/p \rceil)$ work.

Recall that we measure global time units according to the time steps defined to be the smallest time between any two clock-ticks of any processor (Section 2). Thus, during any d global time steps no processor can take more than d local steps.

For the purpose of the next lemma, we introduce the notion of a (d, σ)*-adversary*, where σ is a permutation of t tasks. This is a specialization of 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 the global time τ_i (unknown to the processor), and the task σ_j, for any $1 \leqslant i < j \leqslant t$, is completed for the first time by some processor at time τ_j, then $\tau_i \leqslant \tau_j$. Note that any execution of an algorithm solving the Do-All problem against the d-adversary corresponds to the execution against some (d, σ)-adversary for the specific σ.

Lemma 6.1. *For algorithms* PaDet *and* PaRan1, *the respective worst-case work and expected work is at most* (d)-$\mathrm{Cont}(\Psi)$ *against any* d*-adversary.*

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

For each processor i, let J_i contain all pairs (i, r) such that i performs task $\pi_i(r)$ during the computation. We construct function L from the pairs in the set $\bigcup_i J_i$ 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 i. It is obvious that $(i, 1) \in J_i$, and we let $L(i, 1) = 1$. Suppose that $(i, r) \in J_i$ and we defined function L for all elements from J_i less than (i, r) in lexicographic order. We define $L(i, r)$ as the first $s \leqslant r$ such that $(\sigma^{-1} \circ \pi_i)(s)$ is a d-lrm not assigned by L to any element in J_i.

Claim. *For every* $(i, r) \in J_i$, $L(i, r)$ *is well defined.*

For $r = 1$ we have $L(i, 1) = 1$. For the (lexicographically) first d elements in J_i this is also easy to show. Suppose L is well defined for all elements in J_i less than (i, r), and (i, r) is at least the $(d + 1)$st element in J_i. We show that $L(i, r)$ is also well defined. Suppose, to the contrary, that there is no position $s \leqslant r$ such that $(\sigma^{-1} \circ \pi_i)(s)$ is a d-lrm and s is not assigned by L before the step of the construction for $(i,r) \in J_i$. Let $(i,s_1) < \ldots < (i,s_d)$ be the elements of J_i less than (i,r) such that $(\sigma^{-1} \circ \pi_i)(L(i,s_1)), \ldots, (\sigma^{-1} \circ \pi_i)(L(i,s_d))$ are greater than $(\sigma^{-1} \circ \pi_i)(r)$. They exist from the fact, that $(\sigma^{-1} \circ \pi_i)(r)$ is not a d-lrm and all "previous" d-lrm's are assigned by L. Let τ_r be the global time when task $\pi_i(r)$ is performed by i. Obviously task $\pi_i(L(i,s_1))$ has been performed at time that is at least $d+1$ local steps (and hence also global time units) before τ_r. It follows from this and the definition of (d,σ)-adversary, that task $\pi_i(r)$ has been performed by some other processor in a local step, which ended also at least $(d+1)$ time units before τ_r. 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 the d-adversary. □

Theorem 6.2. *Algorithms* PaRan1 *and* PaRan2 *perform expected work* $EW(p,t,d) = \mathrm{O}(t\log n + p\min\{t,d\}\log(2+t/d))$ *and have expected communication* $EM(p,t,d) = \mathrm{O}(tp\log n + p^2\min\{t,d\}\log(2+t/d))$.

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 $p \geqslant t$ we obtain the formula $\mathrm{O}(t\log t + p\min\{t,d\}\log(2+t/d))$ with high probability, in view of Theorem 4.4, and the obvious upper bound for work is tp. If $p < t$ then we argue that $d' = \lceil pd/t \rceil$ is the upper bound, in terms of the number of "job units," that it takes to deliver a message to recipients, and consequently we obtain the formula

$$\mathrm{O}(p\log p + pd'\log(2+p/d')) \cdot \mathrm{O}(t/p) = \mathrm{O}(t\log p + pd\log(2+t/d)),$$

which, together with the upper bound tp, yields the formula

$$\mathrm{O}(t\log p + p\min\{t,d\}\log(2+t/d)).$$

Since the only difference in the above two cases is the factor $\log t$ that 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 summand t to the expected work.

Message complexity follows from the fact that in every local step each processor sends $p-1$ 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* $W(p,t,d) = \mathrm{O}(t\log n + p\min\{t,d\}\log(2+t/d))$ *and has communication* $M(p,t,d) = \mathrm{O}(tp\log n + p^2\min\{t,d\}\log(2+t/d))$.

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 $p-1$ messages. □

We now specialize Theorem 6.2 for $p \leqslant t$ and $d \leqslant t$ and obtain our main result for algorithms PaRan1 and PaRan2.

Corollary 6.4. *Algorithms* PaRan1 *and* PaRan2 *perform expected work* $EW(p,t,d) = \mathrm{O}(t \log p + p\, d \log(2 + t/d))$ *and have expected communication* $EM(p,t,d) = \mathrm{O}(tp \log p + p^2 d \log(2 + t/d))$ *for any* $d < t$, *when* $p \leqslant t$.

Finally, we specialize Theorem 6.3 for $p \leqslant t$ and $d \leqslant t$ and obtain our main result for algorithm PaDet.

Corollary 6.5. *There exists a list of schedules* Ψ *such that algorithm* PaDet *performs work* $W(p,t,d) = \mathrm{O}(t \log p + p\, d \log(2 + t/d))$ *and has communication* $M(p,t,d) = \mathrm{O}(tp \log p + p^2 d \log(2 + t/d))$, *for any* $d \leqslant t$, *when* $p \leqslant t$.

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. This leads to the open problem of how to construct such permutations efficiently. There also exists a gap between the upper and the lower bounds shown in this paper. It will be very interesting to narrow the gap. Finally, while the focus of this paper is on the work complexity, it is also important to investigate algorithms that simultaneously control work and message complexity.

Appendix A. Proof of Lemma 3.2

Lemma 3.2 For $1 \leqslant d \leqslant \sqrt{u}$

$$\frac{1}{4} \leqslant \frac{\binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} \leqslant \frac{1}{e}\,.$$

Proof. We have

$$\frac{\binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} = \frac{(u-d)\cdot(u-d-1)\cdots(u-d-\frac{u}{d+1}+1)}{u\cdot(u-1)\cdots\left(u-\frac{u}{d+1}+1\right)}$$

$$= \left(1-\frac{d}{u}\right)\cdot\left(1-\frac{d}{u-1}\right)\cdots\left(1-\frac{d}{u-\frac{u}{d+1}+1}\right).$$

For every $i = 0, \dots, \frac{u}{d+1} - 1$ we can bound

$$\left(1 - \frac{d}{u - \frac{u}{d+1} + 1}\right) \leqslant \left(1 - \frac{d}{u - i}\right) \leqslant \left(1 - \frac{d}{u}\right) .$$

It follows, that

$$\left(1 - \frac{d}{u - \frac{u}{d+1} + 1}\right)^{\frac{u}{d+1}} \leqslant \frac{\binom{u-d}{u/(d+1)}}{\binom{u}{u/(d+1)}} \leqslant \left(1 - \frac{d}{u}\right)^{\frac{u}{d+1}} ,$$

and consequently, since

$$\begin{aligned}\left(1 - \frac{d}{u - \frac{u}{d+1} + 1}\right)^{\frac{u}{d+1}} &= \left(1 - \frac{d}{u - \frac{u}{d+1} + 1}\right)^{\frac{u - \frac{u}{d+1} + 1}{d} \frac{d}{u - \frac{u}{d+1} + 1} \frac{u}{d+1}} \\ &\geqslant \left(\frac{1}{4}\right)^{\frac{d}{u - \frac{u}{d+1} + 1} \frac{u}{d+1}} = \left(\frac{1}{4}\right)^{\frac{du}{ud+d+1}} \geqslant \frac{1}{4}\end{aligned}$$

and

$$\left(1 - \frac{d}{u}\right)^{\frac{u}{d+1}} \leqslant e^{-\frac{d}{d+1}} \leqslant \frac{1}{e} ,$$

we obtain the thesis of the lemma. □

References

[1] N. Alon, J.H. Spencer, The Probabilistic Method, second ed., Wiley, New York, 2000.

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

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

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

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

[6] B. Chlebus, L. Gąsieniec, D. Kowalski, A.A. Shvartsman, Bounding work and communication in robust cooperative computation, in: Proceedings of the 16th International Symposium on Distributed Computing, Lecture Notes in Computer Science, vol. 2508, Springer, Berlin, 2002, pp. 295–310.

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

[8] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323.

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

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

[11] Z. Galil, A. Mayer, M. Yung, Resolving message complexity of byzantine agreement and beyond, in: Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, 1995, pp. 724–733.
[12] Ch. Georgiou, D. Kowalski, A.A. Shvartsman, Robust distributed cooperation using inexpensive gossip, in: Proceedings of the 17th International Symposium on Distributed Computing, 2003, pp. 224–238.
[13] Ch. Georgiou, A. Russell, A.A. Shvartsman, Complexity of distributed cooperation in the presence of failures, in: Proceedings of the 4th International Conference on Principles of Distributed Systems, 2000, pp. 245–264.
[14] P.C. Kanellakis, A.A. Shvartsman, Fault-tolerant Parallel Computation, Kluwer Academic Publishers, Dordrecht, 1997.
[15] Z. Kedem, K. Palem, A. Raghunathan, P. Spirakis, Combining tentative and definite executions for very fast dependable parallel computing, in: Proceedings of the 23rd ACM Symposium on Theory of Computing, 1991, pp. 381–389.
[16] D. Kowalski, M. Momenzadeh, A. Shvartsman, Emulating shared-memory Do-All algorithms in asynchronous message-passing systems. Principles of distributed systems, in: 7th International Conference, Revised Selected Papers, Lectures Notes in Computer Science, vol. 3144, 2004, pp. 210–222.
[17] D.E. Knuth, in: The Art of Computer Programming, vol. 3, Addison-Wesley Pub, Reading, MA, 1998.
[18] N. Lynch, A. Shvartsman, RAMBO: A reconfigurable atomic memory service, in: Proceedings of the 16th International Symposium on Distributed Computing, 2002, pp. 173–190..
[19] M. Momenzadeh, Emulating shared-memory Do-All in asynchronous message passing systems, Masters Thesis, Comput. Sci. Eng., University of Conn, 2003..
[20] J. Naor, R.M. Roth, Constructions of permutation arrays for certain scheduling cost measures, Random Struct. Algor. 6 (1) (1995) 39–50.