Explicit Combinatorial Structures for Cooperative Distributed Algorithms

Dariusz Kowalski [*] Peter M. Musiał [†] Alexander A. Shvartsman [‡]

Abstract

Cooperation in distributed settings often involves activities that must be performed at least once by the participating processors. When processor failures or delays occur, it becomes unavoidable that some tasks are done redundantly. To make efficient use of the available processors, several distributed algorithms schedule the activities of the processors in terms of permutations of tasks that need to be performed at least once. This paper presents the first explicit *practical deterministic construction of sets of permutations with certain combinatorial properties that immediately make practical several deterministic distributed algorithms. These algorithms solve a variety of problems, for example, cooperation in shared-memory and message-passing settings, and the gossip problem. Prior to this work, the most efficient algorithms for some of these problems were primarily of theoretical interest — they relied on permutations that are known to exist, but very expensive to construct, with the cost of construction being at least exponential in the size of the permutations. In this paper, the explicitly constructed permutations are ultimately used directly to produce practical instances of several classes of efficient deterministic algorithms. Most importantly, for all of these algorithms, the schedule construction cost is reduced from* exponential *to* polynomial*, at the expense of slight detuning, at most* polylogarithmic*, of the efficiency of these algorithms.*

[*]Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, Warszawa 02-097, Poland. E-mail: darek@mimuw.edu.pl. The work of this author is supported in part by the KBN Grant 4T11C04425 and by the NSF-NATO Award 0209588.

[†]Department of Computer Science & Engineering, University of Connecticut, 371 Fairfield Rd., Unit 2155, Storrs CT 06269, USA. Email: piotr@cse.uconn.edu

[‡]Department of Computer Science & Engineering, University of Connecticut, 371 Fairfield Rd., Unit 2155, Storrs, CT 06269, USA, and Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA. Email: aas@cse.uconn.edu. The work of this author is supported in part by the NSF Grants 9984778, 9988304, 0121277, and 0311368.

1. Introduction

Enabling a collection of processors to cooperate in a decentralized system is at the basis of distributed computing. Many distributed algorithms are constructed using cooperation primitives, such as information aggregation, propagation of information among processors, collaboration of processors on a common set of tasks, parallel update of shared objects, leader election, and distributed consensus. For efficiency reasons, the processors ought to make computational progress while avoiding wasting resources (for example, time, computation, and communication) on activities that either have already been performed, or that will not contribute to the overall computational goal. When a distributed computation can be structured in terms of a collection of activities, one needs to balance computation vs. coordination. In particular, efficiency is enhanced when processors are able to sequence their activities locally, spending resources mostly on the required tasks, but only when needed, on coordination. Abstracting the activities as an unordered set of tasks, schedules need to be constructed that help avoid redundancy. Such schedules can be represented as permutations of tasks. There are several cooperation algorithms, whose efficiency depends on specific combinatorial properties of such permutations. Constructing permutations with the required properties is notoriously difficult, and in many cases, short of the exponential search of the space of permutations, one has to rely on showing the existence of needed permutations, for example, using the probabilistic method.

In this work, for the first time, we present an explicit construction of schedules that are needed by several classes of fault-tolerant cooperative distributed algorithms. These algorithms are used to update shared-memory, to perform a collection of tasks, to spread rumors in a message-passing setting, and to solve consensus. Using our deterministic and efficient construction, instances of such algorithms are readily produced in polynomial time (instead of exponential or worse), while preserving the efficiency of the original algorithms, or incurring a modest polylogarithmic overhead.

Background and previous work. 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. (As we will see, the maximum number of redundancies is the longest increasing subsequence in σ.)

The question we ask is: 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 [10] 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 $\pi_1 = \pi_2$. Although the expected number of left-to-right maxima in a random permutation of $[t]$ is close to $\ln t$ [10], finding sets of permutations that help minimizing redundant work of a large number of processors has proved to be very difficult.

The seminal work of Anderson and Woll [1] made a formal connection between the efficiency of cooperative algorithms, where processors order their activities using permutations, and the left-to-right maxima of these permutations. Here efficiency is measured in terms of *work* that accounts for all steps taken by the processors in the course of the computation. One class of such algorithms has work $O(t \cdot p^{\varepsilon})$ and uses q permutations of $[q]$, where q is such that $1 < q \leq p \leq t$ and $\varepsilon \approx \log_q \log_2 q^3$, thus q has to be quite large for ε to be small. Another class of algorithms has work $O(t \log p)$ and uses q permutations of $[q]$, where $q = p = \Theta(\sqrt{t})$. Work of these algorithms critically depends on identifying a set Ψ of q permutations of $[q]$ with a certain combinatorial property. Specifically, for any permutation σ of $[q]$, the *sum of the number of left-to-right maxima of permutations in* $\sigma \circ \Psi$ must be $O(q \log q)$, where $\circ$ is the permutation composition operator. This combinatorial measure yields a bound on the number of *primary* task executions, that is, the executions of tasks that have not been previously performed. (This measure is termed *contention* in [1].)

When the q permutations of q tasks are chosen randomly then the number of primary tasks is bounded by $O(q \log q)$ with high probability [1, 10]. The challenge is to find such a set of permutations deterministically and efficiently. If this cannot be done expediently, then algorithms that use such permutations is mainly of existential significance. Anderson and Woll [1] show how to search for these permutations, however it takes exponential in q processing time. A different approach is given by Naor and Roth [13]. They show that a set of q permutations where the number of primary tasks is $O(q^{1+\varepsilon})$ can be obtained in time $q \cdot \mathrm{polylog}(q)$ per permutation. The value of q for which the bound holds is however exponential in $1/\varepsilon^3$, and hence such approach is also impractical. Chlebus *et al.* [3] present analytical and experimental evidence that sets of q permutations of q tasks where the number of primary tasks is $O(q \log^2 q)$ can be constructed efficiently. However the proof guarantees this upper bound only for a subset of all asynchronous processor behaviors.

Performance of other distributed algorithms has been shown to depend on permutations with the properties stated in terms of specialized left-to-right maxima definitions. Such algorithms include solutions for the gossip problem [8], and the problem of performing tasks in message-passing settings [11]. All such algorithms, e.g., [1, 8, 11, 12], that are known to exist can be instantiated in exponential time by exhaustively searching all possible sets of schedules. Up to now no satisfactory deterministic constructions of sufficient quality have been given for asynchronous settings.

Contributions. The main contribution of this work is the explicit (which means polynomial in p) construction and combinatorial analysis of deterministic schedules that enable us to create practical instances of fault-tolerant distributed cooperative algorithms that previously were mainly of a theoretical significance. At the core of several such algorithms is a simple scheduling scheme, that we call OBLIDO, where the processors cooperate in an oblivious way by following a fixed set of schedules enumerating the activities that need to be performed to achieve a particular goal. We prove that our constructed schedules have certain combinatorial properties (Theorem 3.1) that are required by these algorithms in order to guarantee efficiency.

We formalize the notion of the worst-case *primary* task executions that is a "tighter" measure of quality of schedules than the previously used *contention* [1, 11] and *surfeit* [8], and we prove that our constructions produces good quality schedules according to this measure (Section 4).

Our construction uses series of expander graphs (Section 3). The effectiveness of this method depends on the

Algorithms	*Previous Results*		*Our Results*	
	Complexity (T: time, W: work, M: messages)	*Cost of Construction*	*Complexity (T: time, W: work, M: messages)*	*Cost of Construction*
BlockWriteI [1], $t \geq p^2$	$W = O(t \log p)$	$(p!)^p$	$W = O(t \text{ polylog } p)$	poly p
BlockWriteII [1], $t \geq p^{3/2}$	$W = O(t^{3/2} \log^2 p)$	$((\sqrt{p})!)^{\sqrt{p}}$	$W = O(t^{3/2} \text{ polylog } p)$	poly p
BlockWriteIII [1], $t = p$	$W = O(p^{1+\varepsilon})$, $\varepsilon > 0$	$\binom{q!}{q} q! q^2 \log q$	$W = O(p^{1+\varepsilon})$, $\varepsilon > 0$	poly q
Gossip [8] for p processors	$T = O(\log^2 p)$ $M = O(p^{1+\varepsilon})$	$(p!)^p$	$T = O(\text{ polylog } p)$ $M = O(p^{1+\varepsilon})$	poly p
PADET [11] t tasks, $t \geq p$	$W = O((t + pd) \log p)$ $M = O((tp + p^2 d) \log p)$	$(p!)^p$	$W = O((t + pd) \text{ polylog } p)$ $M = O((tp + p^2 d) \text{ polylog } p)$	poly p

Table 1. Application of the results: Comparison of algorithm efficiency and construction costs for selected algorithms: *BlockWriteI, II,* **and** *III* **are asynchronous Write-All algorithms,** PADET **is asynchronous Do-All algorithm.**

cost of the deterministic construction of such graphs (polynomial in p, see [16]) and the additional cost of schedule construction that is polynomial in the total size of schedules (linear for each schedule). We generalize this construction so that the length of the schedules is independent of the number of processors (Section 5).

We put our construction to good use by instantiating efficient deterministic versions of several important fault-tolerant cooperative algorithms (Section 6). Table 1 compares the original analysis of selected algorithms and the cost of the schedule construction, to the new analysis that uses our deterministic construction. The new analysis is given in Section 6. Most importantly, for all of these problems we reduce the schedule construction cost from *exponential* to *polynomial* at the expense of slight (polylogarithmic) detuning of the efficiency. Our method is general, and the degree of the polylogarithm depends on the quality of the explicit expanders used in the construction.)

Finally, of independent interest, our construction improves the result of Anderson and Woll [1] and Naor and Roth [13] by producing a set of n permutations from S_n with contention $O(n \text{ polylog } n)$ as compared to $O(n^{1+\varepsilon})$, for any $\varepsilon > 0$. Moreover, the processing cost of our construction is polynomial in n vs. the cost exponential in $1/\varepsilon^3$ (and consequently the cost $n^{\log^2 \log n}$ to obtain contention $O(n \text{ polylog } n)$) in [13]. (This follows with the help of Theorem 3.1.)

Document structure. Section 2 gives definitions and combinatorial landscape; this section also includes additional background and prior results. We show our construction in Section 3, and we prove its combinatorial properties in Section 4. We generalize the construction in Section 5. The utility of our construction is demonstrated in Section 6. Conclusions are in Section 7.

2. Models and definitions

In this section we define the objects considered in this paper: tasks, jobs, processors, mathematical operations, schedules. We present the computational properties of schedules, and relevant background.

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 $L - R$ to stand for L with all elements in the set (or list) R removed. S_n is the symmetric group, the group of all permutations of $[n] = \{1, \ldots, n\}$ the symbol $\circ$ denotes the composition operator, and $\mathbf{u}_n$ denotes the identity permutation.

We use the parameter p that alludes to the number of processors — in some parts of the presentation p is simply an abstract parameter. When p is connected to the number of processors, we assume that the processors have unique identifiers from the set $P = \{0, 1, \ldots, p-1\}$.

Processor model. We assume asynchronous processors whose processing is governed by local clocks. In each clock-tick a processor is able to perform some constant amount of work. For the purpose of the analysis we introduce the notion of a *step* defined in terms of $O(1)$ contiguous local clock-ticks. The computation performed by the processors does not depend on the knowledge of what a step is.

Tasks and jobs. A *task* is a computation that can be done by any processor in $O(1)$ time (for simplicity we can calibrate the notion of local time so that the longest task takes no more than one time unit, unknown to the processor). The tasks are independent and idempotent, that is, an execution of a task does not depend on any other task, and a task may be performed more than once with the same results. A *job* is a collection of one or more tasks. We make the distinction between tasks and jobs for convenience to simplify the analysis. Tasks (and jobs) have unique identifiers — we assume that there are t tasks with identifiers from the set $[t]$ (a job will consist of a contiguous segment of tasks from which the job identifier is readily derived.)

Oblivious scheduling and primary steps. We now interpret the list $\Psi = \langle \pi_0, \pi_1, \ldots, \pi_{p-1} \rangle$ of permutations from S_t as schedules for p asynchronous processors and t jobs. Processor i performs the t jobs in the order given by π_i in Ψ. We call this oblivious algorithm OBLIDO and give the code[1] in Figure 1. This simple algorithm abstracts an important component of several fault-tolerant and asynchronous distributed cooperation algorithms; the analysis of these algorithms also depends on the analysis of an oblivious algorithm, such as OBLIDO (cf. [1, 4, 8, 11]). In Section 6 we put this observation to use and produce practical instances of several cooperative algorithms. (In general a "processor" may be modelling a group of processors following the same sequence of actions.)

```
const Ψ = {π_r | 0 ≤ r < p ∧ π_r ∈ S_t}
                                   % Fixed set of p permutations of [t]
forall processors pid = 0 to p − 1 do
  for r = 1 to t do          % Work according to permutations
    perform Job(π_pid(r))
```

Figure 1. Algorithm OBLIDO.

Since OBLIDO does not involve any coordination among the processors the total of pt jobs are performed (counting multiplicities). However, it was shown [1] that if we count only the job executions such that each job has not been previously finished by any processor, then there *exists* a set of schedules Ψ such that the total number of such job executions is bounded by $O(t \log p)$, again counting multiplicities, which is substantially less than $\Theta(pt)$. 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 t, since each job is performed at least once for the first time. In general this number is going to be between t and pt, since several processors may be executing the same job concurrently for the first time.

We now extend the notion of primary steps. A local step of a processor is *d-primary*, if the job it completes in this step has not been completed, or it has been completed at most d local steps ago. More precisely, consider ith step of processor $\mathfrak{p}$, we call this step d-primary if the job which processor $\mathfrak{p}$ completes in this step has not been completed, or it has been completed in the execution after the event corresponding to the $(i-d)$th step of processor $\mathfrak{p}$. We define (d,q)-Prim(Ψ), for integers $1 \leq d \leq t$ and $1 \leq q \leq p$, and the set of schedules Ψ, as the maximum number of d-primary steps over all executions of algorithm OBLIDO, in which the adversary chooses a subset Q of q processors, and crashes all processors in $P - Q$ at the beginning of the computation. To reiterate, we are interested in counting steps taken by a processor on a job that has either not been performed, or that was performed at most d local steps in the past (that is, a processor can take up to d steps until it learns that the job was already performed).

[1]We borrow the parallel parbegin/parend notation, but this does not imply that processors have access to shared memory.

The main challenge is to *explicitly* (and efficiently, in time polynomial in p) construct an instantiation Ψ of algorithm OBLIDO such that (d,q)-Prim$(\Psi) = O((t + p + dq) \operatorname{polylog} p)$. Note that the trivial lower bound for (d,q)-Prim(Ψ) is $\Omega(t + dq)$, hence the construction we seek has the upper bound that differs from the lower bound by only a *polylogarithmic* factor, while all previous explicit constructions of schedules Υ have additional *polynomial* factor overhead in (d,q)-Prim(Υ).

3. Construction of schedules

In this section we give the construction of schedules for $p = t$, which we denote by n (this is extended in Section 5 to the general case where $p \neq t$). For simplicity we assume that n is a power of 2 (we comment on this later).

We use a-expanding graphs (see Pippenger [15]), for $a = 2^0, 2^1, \ldots, 2^{\log n}$. An undirected regular graph $G = (V, E)$ of n-nodes is called *a-expanding*, where $a \leq n$ is a positive integer, if every subset $A \subseteq V$ of size at least a has more than $n - a$ neighbors in G. We call such graphs a-expanders (there are alternative definitions of expanders, see [16]). The idea of "good" a-expanders is to have the smallest possible degree. For graph $G = (V, E)$ and a subset $A \subseteq V$ of nodes, we denote by $N_G(A)$ the set of all neighbors of A, that is, $v \in N_G(A)$ iff there is an edge between v and some $w \in A$.

The probabilistic argument shows that for every n and a there is an a-expander of degree $\Delta = O(\frac{n}{a} \log \frac{n}{a})$ (see [16]). Recently Ta-Shma, Umans, and Zuckerman [16] gave an explicit construction of a-expander with degree $\Delta = O(\frac{n}{a} \operatorname{polylog} n)$ in time polynomial in n; the polynomial is fixed for given n and a (note also that every node can list its own neighbors in polynomial time).

We take n to be a power of 2 because a-expanders are usually constructed for n nodes where n is a power of 2; in our case this is not a problem, since if n is not a power of 2 we do a construction and analysis for $n' = 2^{\lceil \log n \rceil}$ and after the construction we drop the additional rows and skip task identifiers bigger than the original number of tasks — this does not impact the asymptotic results.

Let X_k denote a 2^k-expander graph and let $\{X_k\}_{k=0}^{\log n}$ represent a family of 2^k-expanders. This family is defined on the set of nodes $\{0, \ldots, n-1\}$, were each node has degree $\Delta_k \le \frac{n}{2^k} \cdot \delta_n$ and $\delta_n = \sup_j \frac{\Delta_j \cdot 2^j}{n}$, such that $2^j \le n$. Using standard probabilistic arguments, e.g., Pinsker [14], one can show that there exists a family of expander graphs $\{X_k\}$ such that $\delta_n = O(\log n)$. For the (polynomial in n) construction in [16] of expander graphs $\{X_k\}$ it is shown that $\delta_n = O(\text{ polylog } n)$.

To provide more intuition we view the expanding graph as a bipartite graph — we take two copies of set V, denote them as P and T, and denote the copy of node $v \in V$ in set P as v_P and in set T as v_T; then we map each edge (v, w) from E to two edges in the bipartite graph: (v_P, w_T) and (w_P, v_T). (Here P corresponds to the set of processors, and T corresponds to the set of tasks, or jobs. This approach is abstract, and the correspondence is mentioned to provide an intuitive link between the expanding graphs and the schedules. The first coordinate in the edge corresponds always to a processor from set P, while the second one corresponds to a task from set T.)

We define schedule π_i, for $i = 0, \ldots, n-1$, as follows:

1. For each graph X_k, where $0 \le k \le \log n$, we order all neighbors of node $i \in P$ in a sequence, denoted by $\sigma_{i,\log n - k}$ (hence there are $\log n + 1$ such sequences); note that the sequence $\sigma_{i,0} \ldots \sigma_{i,\log n}$ enumerates the neighbors of node i in the expander graphs considered in the following order $X_{\log n}, \ldots, X_0$.

2. Then we concatenate sequences $i\, \sigma_{i,0}\, \sigma_{i,1} \ldots \sigma_{i,\log n}$. For each identifier appearing in this sequence we remove repeated references to that identifier. Specifically, if l is the first occurrence of (location of) some identifier x in the concatenated sequence, then we remove all references to x in locations following l. Let the newly obtained sequence $i\, \hat{\sigma}_{i,0}\, \hat{\sigma}_{i,1} \ldots \hat{\sigma}_{i,\log n}$ be the schedule π_i for processor i (where each $\hat{\sigma}_{i,j}$ is the sequence derived from $\sigma_{i,j}$ by removing duplicate references found in $i, \sigma_{i,0}, \ldots, \sigma_{i,j-1}$).

We denote the set of schedules constructed in this way as Ψ. Note that each constructed sequence contains neighbors of nodes in P, and hence the elements from T, i.e., the elements in sequences correspond to tasks (or jobs). We claim that these sequences are permutations. In any sequence, an element appears in the sequence at most once by the second point of the construction. Note that X_0 must be a complete graph, hence all values except i appear in the sequence $\sigma_{i,\log n}$, and consequently in π_i. Thus every value from T appears once and only once.

Observe that if $\{X_k\}_{k=0}^{\log n}$ are constructed in polynomial time in n (see [16]) then so is π_i, for every i, is constructed in polynomial time in n, since given the family of expanding graphs we need only to enumerate neighbors and remove multiplicities, which can be done in time $O(n)$.

We use this construction to obtain the main result of this work: an upper bound on the primary task executions of algorithm OBLIDO. We state the result below, then prove it in Section 4.

Theorem 3.1 *For any positive integers $q, d \le n$, (d,q)-Prim$(\Psi) = O((dq+n)\delta_n \log^2 n)$.*

We now provide additional intuition regarding the number of d-primary steps in Theorem 3.1. First note that the factor $dq+n$ is asymptotically obvious—each of the n tasks must be performed at least once, and since the knowledge about performed tasks can be delayed by d local steps, each of the non-faulty q processors can perform up to d steps ingeminating the same work done in parallel by other processors. The factor δ_n follows directly from the construction of schedules—to guarantee progress when a tasks remain undone we use the property of a-expansion in the analysis, but then we incur the additional length factor δ_n from the overhead in the a-expander degree. The factor $\log^2 n$ follows from the impact of asynchronous environment on the analysis (see the definition of stages in Section 4).

Using Theorem 3.1 together with the construction of the desired expander graphs from [16] we get the following.

Theorem 3.2 *For every n, d, q we can construct, in time polynomial in n, the family Ψ of n schedules from S_n such that (d,q)-Prim$(\Psi) = O((dq+n) \text{ polylog}(n))$.*

Proof: Directly from the proof of Theorem 3.1 and [16]. $\delta_n = O(\text{ polylog } n)$ follows from the degree-overhead of a-expanding graphs constructed in [16]. The polylogarithmic factor subsumes the additional $O(\log^2 n)$ factor that comes from Theorem 3.1.

Note that construction of $\pi \in \Psi$ takes time $O(n)$ if we have access to the lists of neighbors in the expander graphs X_k, for $k = 0, 1, \ldots, \log n$. So the total cost of construction of $\pi \in \Psi$ depends directly on the construction of the list of neighbors in expander graphs X_k, for $k = 0, 1, \ldots, \log n$. If we use the construction of expanders from [16], we get the polynomial in n construction of such lists. $\square$

4. Upper bound on primary executions

We now prove Theorem 3.1. We consider the executions as in the definition of (d,q)-Prim(Ψ); let Q be the set of $q \le n$ processors that do not fail during the computation (Q is selected by the adversary).

The execution of the algorithm is divided into *stages*, each of length $4m\delta_n \log n$, where $m = d(q+1)+n$. For a given stage ℓ, let U_ℓ stands for the set of unperformed tasks by the end of stage ℓ. Recall that by the definition of *primary* tasks, any processor i performing any task not from U_ℓ after the first d local steps in stage $\ell+1$ will not contribute to the number of d-primary steps. (Note that it is this fact that implies the additional summant d in the definition of m above.) The proof proceeds by induction of the stages of the algorithm.

Stage 1. Consider the first $4m\delta_n \log n$ steps in the execution taken by all active processors — we refer to this execution segment as *Stage 1*. In the following claim we estimate the minimum number of tasks preformed by any active processor participating in the first stage.

Claim 4.1 *There exists set $Q_1 \subseteq Q$, such that $|Q_1|$ is a power of 2, and every processor $i \in Q_1$ performs at least $2(m/|Q_1|)\delta_n$ local steps during Stage 1.*

Proof: Let A_k, where $0 \leq k \leq \log n$, stand for the set of processors such that each of them performs at least $2(m/2^k)\delta_n$ tasks in *Stage 1*, and let A_* contain each processor which performs less than $2(m/n)\delta_n$ tasks in *Stage 1*. Note that $\{A_*, A_0, A_1, \ldots, A_{\log n}\}$ is a partition of the set of all processors. We show that there is k such that Q_1 defined as some 2^k processors from A_k satisfies the claim. Hence it is sufficient to show that there exists k such that $|A_k| \geq 2^k$. Suppose, to the contrary, that for every $0 \leq k \leq \log n$, $|A_k| < 2^k$. Hence the total number of steps (or a total number of primary tasks performed) in *Stage 1* is upper-bounded by $|A_*| \cdot 2(m/n)\delta_n + \sum_{k=0}^{\log n} |A_k| \cdot 2(m/2^k)\delta_n < 2m\delta_n + \sum_{k=0}^{\log n} 2^k \cdot 2(m/2^k)\delta_n = 2m\delta_n(\log n + 2) \leq 4m\delta_n \log n$, which contradicts the assumption about the length of the first stage. This completes the proof. ■

Let Q_1 be as stated in Claim 4.1. It follows that every processor $i \in Q_1$ performs at least $2(m/|Q_1|)\delta_n$ tasks in its schedule π_i. Therefore, each $i \in Q_1$ performs all tasks from $\sigma_{i,\log(n/|Q_1|)}$ (this follows from the fact that there is at most $\sum_{k=0}^{\log(n/|Q_1|)} |\hat{\sigma}_{i,k}| \leq 1 + \sum_{k=0}^{\log(n/|Q_1|)} (n/2^{\log n - k})\delta_n \leq 2(n/|Q_1|)\delta_n \leq 2(m/|Q_1|)\delta_n$ positions in permutation π_i by the end of its part $\sigma_{i,\log(n/|Q_1|)}$). By the property of $2^{\log|Q_1|}$-expander $X_{\log|Q_1|}$ (the sequence $\sigma_{i,\log(n/|Q_1|)}$), and hence also the sequence $i\,\hat{\sigma}_{i,0}\,\hat{\sigma}_{i,1}\ldots\hat{\sigma}_{i,\log(n/|Q_1|)}$, contains all neighbors of node i in $2^{\log|Q_1|}$-expander $X_{\log|Q_1|}$), we get that $|N_{X_{\log|Q_1|}}(Q_1)| > n - |Q_1|$, which means that less than $|Q_1|$ tasks remain unperformed after *Stage 1* in any execution. Thus the set of undone tasks after *Stage 1*, denoted as U_1, has cardinality $|U_1| < |Q_1|$.

There are two cases to consider. First, Q_1 contains all processors. Each of these processors performs, by definition, at least $2(m/n)\delta_n \geq 2\delta_n$ local steps in *Stage 1*, in particular each processor performs its first task, which is the task with identifier i. Therefore, $|U_1| = 0$. Second, if $|Q_1| < n$ then by Claim 4.1 we have $|Q_1| \leq n/2$, and the number of undone tasks is $|U_1| < |Q_1| \leq n/2$. In both cases, by the definition of the length of *Stage 1*, the number of d-primary steps is $O(m\delta_n \log n)$.

Remaining Stages. We define the remaining stages by induction, then prove the following: if U_ℓ is the set of unperformed tasks at the end of *Stage ℓ*, then $|U_{\ell+1}| \leq |U_\ell|/2$. Notice first that for *Stage 1* ($\ell = 1$) we already proved that $|U_1| \leq |U_0|/2$, where $|U_0|$ is the initial set of tasks. Now we proceed to the inductive step. Assume that we defined and analyzed all stages up to *Stage ℓ*. We define *Stage $\ell+1$*. Consider the first $4m\delta_n \log n$ steps in the execution after the end of *Stage ℓ*. Using the same argument as in the proof of Claim 4.1 for *Stage 1*, we prove the claim that follows below.

First notice that, by assumption on the adversary, every processor $i \in Q$ after its first d steps has only a subset of U_ℓ tasks to perform. Thus, by the end of the analysis for *Stage $\ell + 1$*, for every processor $i \in Q$ we consider only the remaining steps in *Stage $\ell + 1$* (when i is aware that all tasks not in U_ℓ have already been done). There are at least $4n\delta_n \log n$ of such steps in total.

Claim 4.2 *There exists set $Q_{\ell+1} \subseteq Q$, such that $|Q_{\ell+1}|$ is a power of 2, $|Q_{\ell+1}| \leq |U_\ell|/2$, and every processor $i \in Q_{\ell+1}$ completed its $\hat{\sigma}_{i,\log(n/|Q_{\ell+1}|)}$ sequence during Stage $\ell + 1$.*

Proof: Recall that we analyze only the steps of processor $i \in Q$ in *Stage $\ell + 1$* when i is aware of the fact that all tasks not in U_ℓ are done. Let A_k, where $\log(2n/|U_\ell|) \leq k \leq \log n$, stand for the set of processors such that $i \in A_k$ iff i completed its $\hat{\sigma}_{i,k}$ sequence, but not $\hat{\sigma}_{i,k+1}$, during *Stage $\ell + 1$*, and let A_* stand for the set of processors i such that i did not complete its $\hat{\sigma}_{i,\log(2n/|U_\ell|)}$. Note that $\{A_*, A_{\log(2n/|U_\ell|)}, \ldots, A_{\log n}\}$ is a partition of the set of all processors. We show that there is k such that $Q_{\ell+1}$, defined as some $n/2^k$ processors from A_k, satisfies the claim. Hence it is sufficient to show that there exists k such that $|A_k| \geq n/2^k$. Suppose, to the contrary, that for every $\log(2n/|U_\ell|) \leq k \leq \log n$, $|A_k| < n/2^k$. Hence the total number of considered steps which are taken by some processor from A_k is upper-bounded by

$$\sum_{k=\log(2n/|U_\ell|)}^{\log n} |A_k| \cdot 2^{k+2}\delta_n < \sum_{k=\log(2n/|U_\ell|)}^{\log n} n/2^k \cdot 2^{k+2}\delta_n =$$

$$4n\delta_n(\log(|U_\ell|/2) + 1) = 4n\delta_n \log|U_\ell|.$$

It follows that at least $4n\delta_n \log(n/|U_\ell|)$ steps are performed by the processors in A_*. Consequently, by the counting argument, there is a task z in U_ℓ that has at least $\frac{4n\delta_n \log(n/|U_\ell|)}{|U_\ell|} \geq 4(n/|U_\ell|)\delta_n$ neighbors in A_*, since $|U_\ell| < n/2$ for $\ell \geq 1$. However, this contradicts the definition of the set A_* — we consider only the neighbors in A_* of task z, which, in terms of the expander graphs $X_{\log n}, X_{\log n-1}, \ldots, X_{\log n - \log(2n/|U_\ell|)}$ corresponding to sequences $\hat{\sigma}_{\cdot,0}, \hat{\sigma}_{\cdot,1}, \ldots, \hat{\sigma}_{\cdot,\log(2n/|U_\ell|)}$, means that the total number of neighbors of z must not exceed the sum of degrees

$$\delta_n + 2\delta_n + \ldots + (2n/|U_\ell|)\delta_n < 4(n/|U_\ell|)\delta_n\ .$$

This completes the proof of the claim. ■

It follows from Claim 4.2 that each processor $i \in Q_{\ell+1}$ completes all tasks that are its neighbors in the expander graph $X_{\log n - \log(n/|Q_{\ell+1})} = X_{\log|Q_{\ell+1}|}$ by the end of *Stage* $\ell+1$. By definition of expansion, the total number of tasks that are neighbors of some processor in $Q_{\ell+1}$ is greater than $n - |Q_{\ell+1}| \geq n - |U_\ell|/2$, so $|U_{\ell+1}| \leq |U_\ell|/2$. This completes the analysis of *Stage* $\ell+1$.

We conclude the proof of the Theorem 3.1 as follows. Observe that we need only $\log n$ stages to have $|U_{\log n}| = 0$, since $|U_{\log n}| \leq |U_1|/2^{\log n - 1} < n/2^{\log n} = 1$. Each stage lasts $4m\delta_n \log n$ steps in the execution, so by the end of *Stage* $\log n$ we perform $O(\log n \cdot m\delta_n \log n)$ steps. For each processor $i \in Q$ that has not completed its tasks by the end of stage $\log n$, we consider its d next local steps. By definition of the adversary, processor i must learn by this time that $U_{\log n} = \emptyset$ (i learns that all tasks are completed). Hence

$$\begin{aligned}(d,q)\text{-Prim}(\Psi) &= O(\log n \cdot m\delta_n \log n + qd) \\ &= O((dq+n)\delta_n \log^2 n)\ .\end{aligned}$$

This completes the proof of Theorem 3.1.

5. Generalization for schedules with $p \neq t$

As advertised, we now relax the assumption that $n = p = t$. If $p \neq t$ then we choose n such that $n = p+t$. Effectively we add t virtual processors and p virtual tasks. Now we use our construction and analysis for such n. An operation involving a virtual object (be it a task or a processor) is disregarded. For the sake of the analysis we assume that all virtual processors are processors that are delayed infinitely at the beginning of the execution. A processor scheduled to perform a virtual task simply proceeds to the next task in its schedule (performing a virtual task has no effects on the execution).

Note that if t is large compared to p, we can use a different approach than the one suggested above. We simply partition t tasks into p chunks, each of size $\lfloor t/p \rfloor$ or $\lceil t/p \rceil$. Now we use our construction for $n = p$ processors and n chunks. Note that the current information delay measured in terms of chunks is $d' \leq 1 + dp/t$ (not in terms of tasks, for which the delay is d). The analysis for given t, p, q and d' gives the following results: $(d,q)\text{-Prim}(\Psi) = O((t/p) \cdot (d'q+n)\delta_n \log^2 n) = O((t+dq)\delta_p \log^2 p)$, by Theorem 3.1 (note that now logs are of p not t). Summarizing, we get the following result.

Corollary 5.1 *For every p, t, d, q such that $q \leq p$ and $d \leq t/q$, we have $(d,q)\text{-Prim}(\Psi) = O((t+p+dq)\delta_p \log^2 p)$, where Ψ is our constructed set of p schedules from S_p.*

Again, using the results about a-expanders from [16], we get that $(d,q)\text{-Prim}(\Psi) = O((t+p+dq)\ \text{polylog}\ p)$, and Ψ is constructed in time polynomial in p (precisely, in time $O(p^2)$ plus the cost, polynomial in p, of construction of neighborhoods in a-expanders, for $a = 2^0, 2^1, \ldots, 2^{\log n}$).

6. Constructing practical algorithms

We now demonstrate the utility of our constructions by creating practical instances of algorithms for several distributed problems: asynchronous writing to shared memory ([1]), performing tasks in

asynchronous message-passing system ([11]), fault-tolerant gossip in message-passing environment with applications to performing tasks and consensus ([4, 8]). All those algorithms are the most efficient ones, however in the original setting they use procedure OBLIDO with schedules which are only proved to exist.

To analyze complexity of considered problems, the authors in [1, 4, 8, 11], in essence, measure the number of d-primary steps performed when an algorithm is executing code equivalent to the procedure OBLIDO. They provide upper-bounds for the number of d-primary steps using results for algebraic measures of set of permutation, such as contention [1] or surfeit [8], and proving that the number of d-primary steps is upper bounded by those measures. Unlike the earlier cited works, we directly use the upper-bound for d-primary job executions (a tighter measure).

Table 1 (in the introduction) compares the original analysis of several algorithms and the cost of the respective schedule construction, to the new analysis that uses our deterministic construction. The new analysis is given in the remainder of the section. Most importantly, for all of these problems we reduce the schedule construction cost from *exponential* to *polynomial* (by Theorem 3.2) at the expense of slight (polylogarithmic) detuning of the efficiency.

In the later part, Ψ^{dt} denotes the set of new deterministic schedules obtained in this paper, while Ψ^{pr} denote the set of schedules from [1, 4, 8, 11] whose existence is shown using the probabilistic method and that can be constructed by exhaustive search (which is exponential).

6.1. Applications to the Write-All problem

Among the standard problems in distributed computing is the Write-All problem ([1, 2, 3, 9]), defined in terms of p processors cooperatively updating all locations of a shared-memory array of size t. Here we consider the general asynchronous version of this problem. The efficiency is measured in terms of *work*, that is the total number of processor steps taken until each memory location is written. The algorithms of Anderson and Woll in [1] (also see [12]) are the most efficient asynchronous solutions using the full range of processors ($1 \leq p \leq t$). Those algorithms internally use the approach abstracted by algorithm OBLIDO (Figure 1).

The first algorithm *BlockWriteI* uses p processors to write to $t \geq p^2$ memory locations. The memory is divided into p *blocks* (i.e., jobs) each containing at least $t/p \geq p$ tasks. The blocks are associated with completion bits (one per block, initially set to 0), that indicates whether all tasks in a block are complete. Once a processor starts a block, having found the completion bit to be 0, it sets the bit to 1 after successfully writing the block (i.e., this job is primary).

The precise analysis of *BlockWriteI* gives the bound on work of $O(p^2 + \frac{t}{p}(1,p)\text{-Prim}(\Psi))$ for the set Ψ of $q = p$ schedules from S_p. Since $(1,p)\text{-Prim}(\Psi^{pr})$ is $O(p \log p)$ and the cost of performing one job by one processor is $O(t/p)$, the work of the algorithm is $O(t/p) \cdot (1,p)\text{-Prim}(\Psi^{pr}) = O(t \log p)$. Applying our set of schedules Ψ^{dt} we obtain that the work of the algorithm *BlockWriteI* is at most $O(t/p) \cdot (1,p)\text{-Prim}(\Psi^{dt}) = O(t \operatorname{polylog} p)$. Observe that the new analysis yields work that is increased by the factor of $\operatorname{polylog} p$. However, the construction of Ψ^{pr} requires time exponential in p, whereas the cost of constructing Ψ^{dt} is polynomial in p (by Theorem 3.2). Summarizing, we obtain the following.

Theorem 6.1 *Algorithm BlockWriteI with schedules Ψ^{dt} solves Write-All with work $O(t \operatorname{polylog} p)$ for p processors and $t \geq p^2$ memory cells. The cost of constructing Ψ^{dt} is polynomial in p.*

The second algorithm, *BlockWriteII*, arranges its blocks in two levels. (For simplicity we are going to assume that p is a square of some integer — there is at least one square in the interval $[p, 2p+1]$ and a standard padding approach can be used without affecting the asymptotic results.) The first level has $\sqrt{p}$ *big* blocks, the second level has $\sqrt{p}$ *small* blocks, each containing $\sqrt{p}$ tasks. The completion bits are used similarly to *BlockWriteI*. Hence, $t \geq p\sqrt{p}$. The analysis of *BlockWriteII* gives the bound on work of $O(\sqrt{p}\ (1,p)\text{-Prim}(\Psi)^2 + p\ (1,p)\text{-Prim}(\Psi))$. Using Ψ^{pr} consisting of $q = \sqrt{p}$ schedules from $S_{\sqrt{p}}$, the work is $O(t \log^2 p)$. Using our schedules Ψ^{dt}, the work is upper bound by $O(t \operatorname{polylog} p)$. Our solution incurs a polylogarithmic penalty, however the cost constructing the schedules is significantly reduced (cf. Table 1). The result follows.

Theorem 6.2 *Algorithm BlockWriteII with schedules Ψ^{dt} solves Write-All with work $O(t \operatorname{polylog} p)$ for p processors and $t \geq p\sqrt{p}$ memory cells. The cost of constructing Ψ^{dt} is polynomial in p.*

Now we consider algorithm *BlockWriteIII* that solves the Write-All problem for p processors and p tasks. Here processors independently traverse a q-ary tree, where each processor visits the children of the internal nodes according to the set Ψ of q permutations from S_q. For a chosen $\varepsilon > 0$, one can fix q to be a sufficiently large constant, such that if $(1,p)\text{-Prim}(\Psi) = O(q \log q)$, then work becomes $O(p^{1+\varepsilon})$ [1].

Theorem 6.3 *For every constant $\varepsilon > 0$ there is integer $q > 1$ such that algorithm BlockWriteIII with set Ψ^{dt} of q schedules from S_q completes Write-All with work $O(p^{1+\varepsilon})$ for p processors and $t = p$ memory cells. The cost of constructing Ψ^{dt} is polynomial in $q \leq p$.*

6.2. Application to performing tasks in asynchronous message-passing systems

Kowalski and Shvartsman in [11] considered the asynchronous Do-All problem ([6, 5, 7]): p message-passing asynchronous processors must perform t tasks, subject to the d-adversary that delays messages by up to d time units (d is unknown to the processors). They present delay-sensitive upper and lower bounds on work and message complexity for the Do-All problem.

The most important result of this work is the efficient deterministic algorithm PADET. The algorithm assumes a set of permutations Ψ used by processors to perform the tasks. PADET proceeds in rounds: a processor receives notifications from other processors (if any) about tasks being completed, updates the list of undone tasks, then choses the next task according to a schedule, lastly notification messages are sent. Using our set of schedules Ψ^{dt} we obtain the following.

Theorem 6.4 *Using a set Ψ^{dt} of p schedules from S_p, constructed in time polynomial in p, algorithm* PADET *performs work $O((t+pd) \operatorname{polylog} p)$ and has message complexity $O((tp + p^2 d) \operatorname{polylog} p)$, for $d < t$.*

Note that this result, besides it is constructive, is also close to the lower bound proved in [11] within polylogarithmic factor.

6.3. Application to the gossip problem

Georgiou *et al.* [8] considered a gossip problem for synchronous, crash-prone, message-passing processors. More precisely, the *Gossip*(p, f) problem for p processors is to share the rumors among the correct processors in the presence of f crashes, and the additional condition that correct processor knows either rumor or failures of failed processor. (Given that synchrony with crashes is a special case of asynchrony, our results can be used here as well.)

Georgiou *et al.* [8] give an efficient deterministic gossip algorithm in synchronous, crash-prone, message-passing model. This algorithm uses expanders to provide reliable communication inside large components, while schedules are used to ensure that processors in a large component collect other rumors outside their component. The expanders used for communication are constructed in polynomial time, however the required schedules are only shown to exist using the probabilistic method. Hence, using our new schedules we obtain fully polynomially-constructed algorithms for gossip, with only additional polylogarithm overhead for time and message complexity.

The gossip algorithm in [8] works in $\ell = 1, \ldots, 1/\varepsilon - 2$ iterations, where for each iteration ℓ the number of messages sent is upper-bounded by (d,q)-Prim(Ψ), for $q = p^{(\ell+1)\varepsilon}$ ($\varepsilon > 0$ is any constant), and $d = (31 \log p + 1)p^{(\ell+1)\varepsilon}$, representing message delay. The time bound is $O((d,q)\text{-Prim}(\Psi^{pr})/(qd/\log p))$. Using probabilistic arguments, the existence of set Ψ^{pr} of p schedules from S_p with (d,q)-Prim$(\Psi^{pr}) = O(dq \log p)$ is shown in [8]. This yields time complexity $O(\log^2 p)$ and message complexity (the total number of point-to-point messages) is $O(p^{1+\varepsilon})$, for any constant $\varepsilon > 0$.

The above analysis uses a combinatorial measure called *surfeit*. Recall that for the given set of schedules Ψ, contention is a combinatorial measure that yields an upper bound on the number of *primary* task executions (i.e. $(1,p)$-Prim(Ψ)). *Surfeit* is a slight generalization of d-contention. Let Ψ be the list of permutations from S_t, and let $\Upsilon \subseteq \Psi$ be a set of q permutations from S_t. For a given Υ and $\sigma \in S_t$, let $(d,|\Upsilon|)$-Surf(Υ,σ) be equal to $\sum_{\pi\in\Upsilon}(d)\text{-LRM}(\sigma^{-1}\circ\pi)$. Let $q \le p$ and $d \le t$ be positive integer parameters. The (d,q)-surfeit of a set Ψ is then defined as:

$$(d,q)\text{-Surf}(\Psi) =$$

$$= \max\{(d,|\Upsilon|)\text{-Surf}(\Upsilon,\sigma) : \Upsilon \subseteq \Psi, |\Upsilon| = q, \sigma \in S_t\}.$$

It immediately follows that Cont$(\Psi) = (1,p)$-Surf(Ψ) and (d)-Cont$(\Psi) = (d,p)$-Surf(Ψ).

Using our schedules Ψ^{dt}, we obtain a bound on time of $O(\operatorname{polylog} p)$, since $O((d,q)\text{-Prim}(\Psi^{dt})/(qd/\log p)) = O(\operatorname{polylog} p)$. Observe that the time complexity is increased by a small factor of $\operatorname{polylog} p$. The message complexity does not change since the factor of $\operatorname{polylog} p$ is subsumed in the asymptotic result (since it is polynomial, $O(p^{1+\varepsilon})$, for every constant $\varepsilon > 0$; the additional factor $\operatorname{polylog} p$ is elided by using a slightly smaller $\varepsilon > 0$ and performing the analysis for the new ε.) Hence we get the following result.

Theorem 6.5 *For every constant $\varepsilon > 0$, for p processors and f failures, where $f < p$, deterministic gossip algorithm based on schedules Ψ^{dt} is constructed in polynomial time in p, and solves the Gossip(p,f) problem with time complexity $O(\operatorname{polylog} p)$ and message complexity $O(p^{1+\varepsilon})$.*

Efficient deterministic gossip algorithm can be effectively used to solve other problems. For example, it is used in [8] to produce a deterministic algorithm for the Do-All problem of [6], defined as p processors cooperatively performing t tasks. Using our construction for the gossip also yields an efficient and practical solution for Do-All. Another example is the consensus problem. Chlebus and Kowalski in [4] show how to reach consensus using gossiping where processors may fail before consensus is reached. Their algorithm is efficient both in time and number of messages sent. Here the schedules are also employed and again a probabilistic construction method is used. Again our new deterministic construction can be used, instead of relying on the probabilistic method. We do not go into details here, but both the time and message complexity remain efficient when using our deterministic construction.

6.4. Constructing low-contention permutations

We define contention of set of schedules Ψ, which has been used to provide upper bound for $(1,p)$-Prim(Ψ) in context of Write-All problem, see Subsection 6.1 and [1, 12].

For a n-schedule $\pi = \langle \pi(1), \ldots, \pi(n)\rangle$ a *left-to-right maximum* (see Knuth vol. 3, p. 13 [10]) is an element $\pi(j)$ of π that is larger than all of its predecessors, i.e., $\pi(j) > \max_{i<j}\{\pi(i)\}$.

Given a n-schedule π, we define LRM(π), to be the number of left-to-right maxima in the n-schedule π (see [1]). For a list $\Psi = \langle \pi_0, \ldots, \pi_{n-1}\rangle$ of permutations from S_n and a permutation τ in S_n, the *contention* of Ψ with respect to τ is defined as $\text{Cont}(\Psi,\tau) = \sum_{u=0}^{n-1} \text{LRM}(\tau^{-1}\circ\pi_u)$. The *contention of the list of schedules* Ψ is defined as $\text{Cont}(\Psi) = \max_{\tau\in S_n}\{\text{Cont}(\Psi,\tau)\}$. Note that for any Ψ, we have $n \le \text{Cont}(\Psi) \le n^2$.

A family of permutations with low contention was introduced in [1], where the following is shown (here H_n is the nth harmonic number, $H_n = \sum_{j=1}^{n}\frac{1}{j} = \Theta(\log n)$).

Lemma 6.6 [1] *For any $n > 0$ there exists a list of permutations $\Psi = \langle \pi_0, \ldots, \pi_{n-1}\rangle$ with* Cont$(\Psi) \le 3nH_n$.

A list Ψ with Cont$(\Psi) = \Theta(n \log n)$ can be found by exhaustive search (at the cost of order $(n!)^n$). A lower bound for contention, following directly from [10], is $\Omega(n \log n)$.

Anderson and Woll [1] posted a question how to construct a set of n permutations from S_n with contention close to the best possible $\Theta(n \log n)$ (which was only proved to exist). They present solution, constructed in polynomial time, with contention $O(n^{1+\varepsilon})$, for any constant $\varepsilon > 0$. Recently Malewicz [12] proved that their construction is substantially higher than $\Omega(n \operatorname{polylog} n)$. Another alternative, although similar, approach was presented by Naor and Roth [13], still being far from optimal, more precisely they could achieve only contention $O(n^{1+\varepsilon})$ in polynomial time. Our construction solves this open problem: using expanders from [16] we obtain, in polynomial time, set Ψ^{dt} of n schedules from S_n having $(1,n)$-Prim(Ψ^{dt}) at most $O(n \operatorname{polylog} n)$. Additionally, the following fact shows when Cont(Ψ) is equal to $(1,n)$-Prim(Ψ).

Theorem 6.7 *For every set Ψ of n schedules from S_n such that the first column is a permutation in S_n,* $\text{Cont}(\Psi) = (1, n)\text{-Prim}(\Psi)$.

In view of Theorem 6.7 we obtain the sought result, since our construction satisfies the assumption of the Theorem.

7 Discussion

In this paper we presented the first deterministic and explicit construction of permutation schedules that can be used by processors solving distributed cooperation problems in the presence of failures and delays. Our construction has the cost polynomial in the size of the schedules, which *substantially* reduces the exponential cost of exhaustive search that was previously required. The construction can be used directly in solving several important cooperation problems by producing practical instances of efficient algorithms that previously were only known to exist. The price we pay for these deterministic constructions is a very slight detuning—at most polylogarithmic—of the efficiency of these algorithms. Our future work includes reducing the remaining inefficiency associated with our construction and applying our technique to producing efficient practical fault-tolerant algorithms for a broad variety of distributed cooperation problems. Finally, we note that any improvements in the time complexity of construction and the expansion properties of expanders, automatically improves our construction and its applications—the construction of a-expanders in [16] has a $O((n/a)\text{ polylog } n)$ degree (which in fact is at most $O((n/a)\log^{24} n)$); when compared to the lower bound of $\Omega((n/a)\log n)$, this leaves polylogarithmic room for improvement.

Another interesting issue is studying relations between our (d, q)-Prim(Ψ) measure, which is motivated by computational model, and other previously used measures (defined by algebraic operations on Ψ), as generalized contention or surfeit. We conjecture that if each task z appears at least once as a left-to-right maximum in some permutation in the given schedule S_t, then (d, q)-Prim(Ψ) is equal to (d, q)-Surf(Ψ).

Acknowledgement. We thank the anonymous referees for many insightful comments that substantially contributed to the quality of this paper.

References

[1] R. Anderson and H. Woll. Algorithms for the certified write-all problem. *SIAM J. on Computing*, 26(5):1277–1283, 1997.

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

[3] B. Chlebus, S. Dobrev, D. Kowalski, G. Malewicz, A. Shvartsman, and I. Vrto. Towards practical deterministic Write-All algorithms. In *Proc. of 13th ACM Symp. on Par. Alg. and Arch. (SPAA)*, pages 271–280, 2001.

[4] B. Chlebus and D. Kowalski. Gossiping to reach consensus. In *Proc. of 14th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 220–229, 2002.

[5] R. De Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proc. of 13th ACM Symp. on Principles of Distributed Comp. (PODC)*, pages 161–172, 1994.

[6] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM J. on Computing*, 27:457–1491, 1998.

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

[8] C. Georgiou, D. Kowalski, and A. Shvartsman. Efficient gossip and robust distributed computation. In *Proc. of 17th International Symp. on Distributed Computing (DISC)*, pages 224–238, 2003.

[9] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. of 22nd ACM Symp. on Theory of Computing (STOC)*, pages 138–148, 1990.

[10] D. Knuth. *The art of computer programming*, volume 3. Addison-Wesley Pub Co., third edition, 1998.

[11] D. Kowalski and A. Shvartsman. Performing work with asynchronous processors: Message-delay-sensitive bounds. In *Proc. of 22nd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 265–274, 2003.

[12] G. Malewicz. A method for creating near-optimal instances of a certified write-all algorithm. In *Proc. of 11th Annual European Symp. on Algorithms (ESA)*, pages 422–433, 2003.

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

[14] M. Pinsker. On the complexity of a concentrator. In *Proc. of 7th Annual Teletraffic Conference*, pages 318/1–318/4, 1973.

[15] N. Pippenger. Sorting and selecting in rounds. *SIAM J. on Computing*, 16:1032–1038, 1987.

[16] A. Ta-Shma, C. Umans, and D. Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *Proc. of 33rd Annual ACM Symp. on Theory of Computing (STOC)*, pages 143–152, 2001.