Cooperative Asynchronous Update of Shared Memory

Bogdan S. Chlebus [*]

Dariusz R. Kowalski [†]

ABSTRACT

The *Write-All* problem for an asynchronous shared-memory system has the objective for the processes to update the contents of a set of shared registers, while minimizing the total number of read and write operations. First abstracted by Kanellakis and Shvartsman [12], *Write-All* is among the standard problems in distributed computing. The model consists of n asynchronous processes and n registers, where every process can read and write to any register. Processes may fail by crashing. The most efficient previously known deterministic algorithm performs $\mathcal{O}(n^{1+\varepsilon})$ reads and writes, for an arbitrary fixed constant $\varepsilon > 0$, and is due to Anderson and Woll [4]. This paper presents a new deterministic algorithm that performs $\mathcal{O}(n \text{ polylog } n)$ read/write operations, thus improving the best previously known upper bound from polynomial to polylogarithmic in the average number of read/write operations per process. Using an approach to store and retrieve information about progress made in auxiliary registers, the novelty of the new algorithm is in using a family of multi-partite graphs with expansion properties to structure a set of registers as a graph and then have each asynchronous process explore a part of the graph according to its pattern of traversals. An explicit instantiation of our *Write-All* algorithm, based on best-known polynomial-time constructions of lossless expanders and a-expanding graphs, performs $n \cdot 2^{\mathcal{O}(\log^3 \log n)}$ reads and writes. In this *explicit* solution to *Write-All*, the processes perform asymptotically less read/write operations than the most efficient *non-explicit* solution known before.

[*]Department of Computer Science and Engineering, University of Colorado at Denver and Health Sciences Center, Campus Box 109, Denver, CO 80217, USA. The work of this author is supported by the NSF Grant 0310503.

[†]Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, Warszawa 02-097, Poland. The work of this author is supported by the KBN Grant 4T11C04425. This work has been partly done while the author was a postdoc at Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Categories and Subject Descriptors:
F.2.0 [Analysis of Algorithms and Problem Complexity]: General

General Terms:
Algorithms, Performance, Theory

Keywords:
distributed algorithm, read and write register, asynchrony, problem *Write-All*, work efficiency, expander, disperser.

1. INTRODUCTION

We consider an asynchronous distributed system with read and write registers. There are n processes, some of them may fail by crashing, but at least one process survives. The *Write-All* problem is about how to have these processes update cooperatively an array of n Multi-Reader Multi-Writer registers, while minimizing the total number of read and write operations.

Write-All is one of two dual distributed-system primitives, the other is *Collect*. *Write-All* is about writing to an array of registers, with the purpose to update all of them. *Collect* is about reading the contents of an array of registers, with the purpose to get each process know all values stored in the registers. In both problems processes may cooperate to improve efficiency.

Known solutions to *Collect* and *Write-All* have been developed with the goal to be efficient in terms of either asynchronous time [23], or the total number of read/write operations [10, 13], or competitiveness in various models [2, 6]. We use *work* as performance metric. It is defined as the total number of register accesses, for reading and writing.

The power of *Write-All* is captured by solutions of a special case when we need to write 1's into the registers, assuming we start with all of them containing 0's. There is given an array of n registers and an additional flag, which are all initialized to zero. The value 1 of the flag is interpreted as "done," and that is why the problem is often called *Certified Write-All*. The goal of n processes is to set all array registers to 1 and next set the flag to 1. To be correct, a solution has to provide that the flag is set to "done" only after all n registers have been set to 1.

An approach in which process i first updates the ith cell and next sets the flag to 1 is not correct for two reasons. First, even one crash results in some update not performed. Second, if some process is fast and some other slow, then the flag is set to "done" before the slow process performs its update. On the other hand, a direct solution, in which each process carries out all updates and then sets the flag, has a drawback that the total work is $\Omega(n^2)$ when few processes crash.

The flag improves efficiency if processes check regularly on its value. This happens when fast processes do most of the work and then set the flag to "done," while slow processes stop after learning that all tasks have been completed. The flag is not necessary, as long as no process terminates before the whole array has been updated, but it indicates how to use auxiliary memory to streamline solutions. A divide-and-conquer approach has the array recursively divided into parts and a flag assigned to each part. This results in a tree of auxiliary memory registers, each node representing a segment of the array of registers to update.

The first result by exploring this approach was obtained by Buss, Kanellakis, Ragde and Shvartsman [7] who developed a deterministic algorithm with a work per process that was sub-linear but still polynomial. Anderson and Woll [4] showed that the exponent $\varepsilon > 0$ in the bound $\mathcal{O}(n^\varepsilon)$ on the average work per process can be made arbitrarily small. The question if a polylogarithmic work per process can be achieved was left open. In this paper we settle this question in the affirmative.

A distributed algorithm $\mathcal{A}$ is *explicit* or *constructive* if there is a sequential algorithm that finds an instantiation of $\mathcal{A}$ for a given number n of processes in time that is polynomial in n. An algorithm that has not been shown to be explicit is called *existential.*

Our results. The summary of the contributions is as follows.

I. We show that *Write-All* can be solved with the amount of work that is $\mathcal{O}(n \text{ polylog } n)$ by a deterministic existential algorithm. More precisely, the work of this algorithm is $\mathcal{O}(n \log^{18} n/(\log\log n)^3)$. The most efficient previously known deterministic solution was given by Anderson and Woll [4]; it performed work $\mathcal{O}(n^{1+\varepsilon})$, for an arbitrary fixed constant $\varepsilon > 0$. This meant an exponential gap between the lower and upper bounds on the average work per process. Our solution narrows the gap to polynomial, by improving the average work per process to polylogarithmic.

II. We show that *Write-All* can be solved with the amount of work that is $n \cdot 2^{\mathcal{O}(\log^3 \log n)}$ by a deterministic *explicit* algorithm. This is asymptotically better than even *existential* instantiations of the algorithm in [4], since Malewicz [18] showed that, for any $\varepsilon > 0$, the work of an *arbitrary* instantiation of that algorithm has to be $\Omega(n^{1+(1-\varepsilon)\sqrt{2\ln\ln n/\ln n}})$.

Problem *Write-All* has also been defined with two parameters, when the number p of processes and the size t of an array of registers to update are independent. Our solution can be adapted to this case to obtain an algorithm with work $\mathcal{O}((p+t) \text{ polylog } \min\{p,t\})$. This may be achieved by a simulation using a solution for $n = p = t$ and then either grouping the tasks, when $t > p$, or the processes, when $p > t$.

Previously known efficient deterministic algorithms, like the one given in [4], relied on trees to store and retrieve information about progress made, while processes explored such progress trees following their individual patterns of traversal. We define *multi-expanders*, which are used in a way similar to that of progress trees. We first introduce the notion of a *connector* which is a bipartite graph that combines properties of expanders and dispersers. Connectors determine the pattern of connections between consecutive layers of multi-expanders. We show that connectors and multi-expanders with good expansion properties exist; this can be achieved either by counting, which yields graphs with stronger properties, or in a constructive way, which results in weaker properties. Multi-expanders are used to store and retrieve information about progress made. Each process explores its individual sparse subgraph of the multi-expander. The strength and novelty of the analysis we give is in showing how to use expansion properties to gauge the flow of information and progress made in this asynchronous setting.

Previous work. The problem *Write-All* was first considered by Kanellakis and Shvartsman [12]. Buss, Kanellakis, Ragde and Shvartsman [7] developed a deterministic algorithm with work $\mathcal{O}(n^{\log_2 3})$. A deterministic algorithm of Anderson and Woll [4] attains work $\mathcal{O}(n^{1+\epsilon})$, for any fixed constant $\epsilon > 0$. Known randomized solutions are optimal, but they have been analyzed in terms of performance against *oblivious* adversaries only. Such adversaries determine the order of events at processes prior to the start of an execution of a randomized algorithm. The randomized algorithm of Martel and Subramonian [20] performs work $\mathcal{O}(n)$ with a large probability on $n/\log n$ processes. Lower bounds $\Omega(n \log n)$ on work by deterministic algorithms and $\Omega(\log n)$ on time were given in [7, 14, 20].

The algorithm of Anderson and Woll [4] solving *Write-All* with work $\mathcal{O}(n^{1+\varepsilon})$ uses a progress tree of a certain degree. The algorithm uses a set of permutations, of a size equal to the degree. Suitable families of permutations have been shown to exist by the probabilistic method, but no explicit construction in time polynomial in n and ε^{-1} is known. A constructive implementation of the algorithm, by way of a specific set of permutations, was proposed by Kannelakis and Shvartsman [13] and studied by Chlebus *et al.* [9]. Malewicz [18] considered the impact of the degree of progress tree on the performance, and showed that, for infinitely many values of n, there are instances of the algorithm performing at most $n^{1+(1+\varepsilon)\sqrt{2\ln\ln n/\ln n}}$ work.

Algorithms that achieve optimal work $\mathcal{O}(t)$, for a number of processes p smaller than the size t of the array of registers to update, were studied by Malewicz [19] and Kowalski and Shvartsman [15]. Such algorithms are known to exist for $t = \Omega(p^{2+\varepsilon})$, for an arbitrary constant $\varepsilon > 0$, see [15]. It is still an open problem how to achieve work $\mathcal{O}(t)$ for $t = \Omega(p \text{ polylog } p)$.

Related work. The problem *Write-All* was defined as a primitive useful in simulating a PRAM on a PRAM subject to processor failures, see [12]. An overview of simulations of a fully operational PRAM on one prone to processor failures can be found in a book by Kanellakis and Shvartsman [13]. Kedem, Palem, Raghunathan and Spirakis [14] developed randomized simulations of PRAM on a faulty one.

The problem *Collect* is closely related to *Write-All. Collect* was abstracted by Saks, Shavit, and Woll [23] in the context of their work on randomized consensus algorithms. Ajtai, Aspnes, Dwork and Waarts [2] showed how to adapt the deterministic algorithm of Anderson and Woll [4] for *Write-All* to obtain a deterministic solution to *Collect* with $\mathcal{O}(n^{3/2}\log n)$ work performance. The first randomized solution to *Collect* with expected work exceeding the lower bound by only a polylogarithmic factor was given by Aspnes and Hurwood [6], but is has been analyzed only against oblivious adversaries.

A deterministic *Collect* algorithm developed by Chlebus, Kowalski and Shvartsman [10] attains work $\mathcal{O}(n \text{ polylog } n)$. This algorithm is parameterized by graphs embedded in the code. A constructive solution relies on the family of a-expanders described by Ta-Shma, Umans, and Zuckerman [26], yielding an *explicit* algorithm with the amount of work that is only $\mathcal{O}(n \text{ polylog } n)$, where the exponent is larger than in the existential solution. Iterative *Collect* can be used to implement atomic snapshot objects [1, 5]. Competitive-efficient solutions to iterative *Collect* were given in [2, 6, 10].

Expander-based communication schemes can be designed to guarantee fast proliferation of information in an asynchronous shared-memory setting, as was shown by Chlebus, Kowalski and Shvartsman [10]. Expander-like graphs have been used to extract randomness from weak sources of randomness using a few additional random bits, as first shown by Santha [24] and Sipser [25], see [8, 22, 26, 27] for recent work. Efficient constructions of graphs with good expansion properties were given in [16, 22, 26, 27].

Structure of this document. Section 2 discusses the model and compares solutions of *Write-All* and *Collect*. Section 3 defines connectors, which are later used in a construction of multi-expanders given in Section 4. The algorithm is described in Section 5 and analyzed in Section 6. We conclude with a discussion in Section 7.

2. MODELS AND PROBLEMS

Models. We consider n asynchronous processes and a set of shared memory registers such that each process can read from any register and write to any one. This model is often called the *Asynchronous PRAM*. The PRAM model used in parallel computing, see [11], is understood as a synchronous machine with a constant-time access of any processor to any shared register. If the assumption of global synchrony is abandoned, leaving only the capability of any process to access any shared-memory register for either reading or writing, then the result is the Asynchronous PRAM, see [4, 5, 20], which is used in this paper.

The size of shared registers, in terms of the number of bits that a register can store, significantly affects design of algorithms. The logarithmic size of registers captures an architecture where memory registers are implemented in hardware. A logarithmic size of registers is sufficient to implement all known most efficient solutions to *Write-All*, see [4, 9, 14, 15, 18]. In this paper we follow this approach and work with an Asynchronous PRAM in which every register is of a logarithmic size. In contract to *Write-All*, solutions to *Collect* require the size of registers to be polynomial. Such size can be achieved when shared memory is implemented in message passing, see [17].

A process of an asynchronous PRAM performs two actions during an event in an execution: a constant-time local computation, followed by either a read from an arbitrary register or a write to one. Such writes and reads are the only externally-visible operations. An execution is represented as an externally-visible *trace*, specified as a sequence $F = \langle k_0, k_1, k_2, \ldots \rangle$, where each k_i is a process name. The j-th event in the trace is the currently pending read or write operation of the process with name k_j. An occurrence of such a j-th event in an execution means that it has been actually performed by the process k_j. We consider executions determined by infinite traces. A name of a specific process may or may not occur infinitely many times in the trace, but there is at least one name that does occur in this way. If a name occurs finitely many times, then this means that the process has crashed. When each process name occurs infinitely many times, like in executions for solutions to *Collect*, then such traces/executions are *fair*, see [17].

Problems. A deterministic solution of *Collect* with an average polylogarithmic work per process was given by Chlebus, Kowalski and Shvartsman [10]. In this paper, we present a solution to *Write-All* with a similar work efficiency. First we compare the problems *Write-All* and *Collect*, and discuss properties of the solutions given in this paper and in [10].

Let us define the two problems in terms of the same terminology and notations. Suppose that there are n processes and and array $A[1..n]$ of n shared-memory registers $A[i]$, for $1 \leq i \leq n$. There could be more registers available, in the case of *Write-All*. Properties of these registers are *different* for the two problems, even though they are referred to by the same name.

Problem *Write-All*:

Initial configuration: All the values are $A[i] = 0$, for $1 \leq i \leq n$.

To achieve: Set each $A[i]$ to 1, for $1 \leq i \leq n$.

Problem *Collect*:

Initial configuration: Each $A[i]$ stores some initial value, for $1 \leq i \leq n$.

To achieve: Every $A[i]$ stores all the original initial values in $A[1..n]$.

From a low-level technical point of view, there is a similarity that registers are of the Multi-Reader type, but except for that the features are opposing. Among them is the fact that the environment for *Collect* is without failures, while crashes may happen in executions of solutions of *Write-All*. However, because of asynchrony, a wait-free solution of either *Collect* or *Write-All* that is efficient in an environment without crashes has the same asymptotic performance with crashes. This is because the processes that crashed might be considered very slow in an environment without failures: suppose that each of them does not really crash but stops performing any action until the remaining processes have learned all values, and only then completes its computation.

Since the problems *Write-All* and *Collect* are related, it is natural to ask if having a work-efficient solution to one of them helps in obtaining a solution of a comparable work performance for the other one as well. A solution to *Write-All* was shown to be helpful to solve *Collect* by Ajtai, Aspnes, Dwork and Waarts [2]. They used an approach of Anderson and Woll [4], that allows to achieve work $\mathcal{O}(n^{1+\varepsilon})$ while solving *Write-All*, to obtain a solution for *Collect* with work $\mathcal{O}(n^{3/2} \text{ polylog } n)$.

A question more relevant in this context is if one could achieve a transformation in the opposite direction: to obtain a *Write-All* solution from an algorithm for *Collect*. One could argue that this is the case by pointing that repeating a pattern of memory accesses for reading, in an execution of an algorithm solving *Collect*, as a pattern of memory accesses for writing, in an execution of an algorithm solving *Write-All*, would result in a pattern that also handles *Write-All*. This is true in the sense that any solution of *Collect* can be mapped into one for updating an array of *large* registers. If we use a solution of *Collect* given in [10], then the resulting algorithm has $\mathcal{O}(n \text{ polylog } n)$ work performance. This solution of updating shared memory *relies significantly on a polynomial size of registers*, and on the ability to manipulate an arbitrary subset of the set $\{1, 2, \ldots, n\}$ in a single-step read/write operation, which is then counted as contributing only a unit to the amount of work. A solution like this one cannot be implemented on an Asynchronous PRAM with a logarithmic size of registers.

Both solutions of problems *Collect* and *Write-All*, given in [10] and in this paper, respectively, use graphs with expansion properties in their codes. The two approaches differ significantly, which results in different types of performance bounds for best explicit solutions obtained. Whereas an *explicit* solution of *Collect* with work performance that is only $\mathcal{O}(n \text{ polylog } n)$ was given in [10], the most-efficient *explicit* algorithm for *Write-All* given in this paper performs work $n \cdot 2^{\mathcal{O}(\log^3 \log n)}$.

3. CONNECTORS

Processes executing our algorithm communicate by reading and writing to shared registers according to patterns determined by specific graphs with expansion properties. Such general graphs can be defined in many ways, see for instance [8, 16, 21, 22, 26, 27]. The graphs we use combine properties of expanders and dispersers.

First we fix some graph terminology.

For a set X of nodes in a graph G, the notation $N_G(X)$ means the set of all neighbors of nodes in X; the subscript G is dropped when the underlying graph G is clear from context. The set of all nodes w such that there is a path of length at most i in graph G from the node w to some node in X is denoted $N^i_G(X)$, for a positive integer i.

Next we recall definitions of expansion properties. Let $G = (V, W, E)$ be a bipartite graph, with left-hand side nodes in V and right-hand side nodes in W; the elements of set V are called *inputs* and those in set W are called *outputs*. Let $K > 0$, $\alpha > 1$ and $\varepsilon > 0$ be real numbers.

Graph G is said to be a (K, ε)-disperser, if every subset $X \subseteq V$ of at least K nodes is connected to at least $(1-\varepsilon)|W|$ neighbors.

Graph G is said to be a (K, α)-expander, if every subset $X \subseteq V$ of at most K nodes is connected to at least $\alpha|X|$ neighbors; number α is called the *expansion* of G.

An expander is *lossless* if its inputs are of the same degree and its expansion is close to the input degree.

A simple graph is *a-expanding* if every two disjoint sets of nodes of size at least a are joined by an edge.

Now we introduce new types of graphs:

Let $H' = (V, W, F)$ be a bipartite graph, where $|V| = |W| = n$. Graph H' is defined to be a *weak* $(n, \delta', \varepsilon')$-*connector* when it has the following four properties:

Definition of a WEAK $(n, \delta', \varepsilon')$-CONNECTOR:

(degree bounds) The degrees of inputs in V are all equal to δ' and the degrees of outputs in W are all between $(1-\varepsilon')\delta'$ and $(1+\varepsilon')\delta'$.

(left expansion) The inequality
$$|N_{H'}(A)| \geq (1-\varepsilon')\delta' \cdot |A|$$
holds for every set $A \subseteq V$ of a size that is at most $n\varepsilon'/(2\delta')$.

(right expansion) The inequality
$$|N_{H'}(B)| \geq (1-\varepsilon')^2\delta' \cdot |B|$$
holds for every set $B \subseteq W$ of a size at most n/δ'.

(left dispersion) The inequality
$$|N_{H'}(A)| > n - n/(\delta'\varepsilon')$$
holds for every set $A \subseteq V$ of a size at least $n/(\delta'\varepsilon')$.

LEMMA 3.1. *If $n \geq 8b'$, then there is a weak $(n, \delta', \varepsilon')$-connector H' for $\delta' = b' \log^3 n$ and $\varepsilon' = 1/(3 \log n)$, for a sufficiently large constant $b' > 0$.*

Let $H = (V, W, F)$ be a bipartite graph with the inputs in V, and the outputs in W, where $|V| = |W| = n$. We say that H is a (n, δ, ε)-*connector* when it has the following properties:

Definition of (n, δ, ε)-CONNECTOR:

(regularity) Every node is of degree δ.

(expansion) The inequality
$$|N_H(A)| \geq (1-\varepsilon)\delta \cdot |A|$$
holds for every set $A \subseteq V$ or $A \subseteq W$ of a size at most $n\varepsilon/(6\delta)$.

(dispersion) The inequality
$$|N_H(A)| > n - 4n/(\delta\varepsilon)$$
holds for every set $A \subseteq V$ or $A \subseteq W$ of a size at least $4n/(\delta\varepsilon)$.

Notice that connectors have inputs and outputs treated symmetrically, unlike weak connectors. We can construct connector H using a weak connector H' proved to exist in Lemma 3.1.

THEOREM 3.2. *There exists a (n, δ, ε)-connector for the parameters $\delta = b \log^3 n$ and $\varepsilon = 1/\log n$, for a sufficiently large constant $b > 0$ and when n is a sufficiently large power of 2.*

Alternative constructions of connectors. We show a construction relying on the existence of graphs with good expansion properties. Additionally, this construction is explicit if the used expanders are explicit.

Let $\varepsilon = 1/\log n$. Suppose Δ is such that there are two (n, n)-bipartite regular graphs $\mathcal{H}_1$ and $\mathcal{H}_2$ such that $\mathcal{H}_1$ is an $(n\varepsilon/(6\Delta), (1-\varepsilon/2)\Delta)$-lossless expander of degree $\Delta \geq b \log^3 n$ and $\mathcal{H}_2$ is a $\frac{4n}{(\Delta+\delta')\varepsilon}$-expanding graph of degree δ', where $\Delta > 2\delta'/\varepsilon$. Coalesce the pairs of corresponding nodes and take the union of the sets of edges, to obtain an (n, n)-bipartite regular graph $\mathcal{G}$ of degree $\delta = \delta' + \Delta$.

The dispersion condition holds for $\mathcal{G}$, since $\delta = \delta' + \Delta$ is the degree of $\mathcal{G}$ and by the expansion for $\mathcal{H}_2$, which is a subgraph of $\mathcal{G}$. To show the expansion property, note that it follows from the definition of a lossless expander $\mathcal{H}_1$ that for every set $A \subseteq V$ of a size at most $n\varepsilon/(6\Delta)$ the inequalities

$$|N_{\mathcal{G}}(A)| \geq (1-\varepsilon/2)\cdot\Delta|A| \geq (1-\varepsilon)\delta|A|$$

hold, since $\Delta > \delta(1-\varepsilon)$ for sufficiently large n.

It is known that such graphs $\mathcal{H}_1$ with $\Delta = \Theta(\log^3 n)$ exist, as do graphs $\mathcal{H}_2$ with the degree $\delta' = \Theta(\log n)$. This results in the same quality of connectors as in graphs showed to exist by direct counting.

Explicit graphs $\mathcal{H}_1$ with $\Delta = 2^{\mathcal{O}(\log^3 \log n)}$ were constructed in [8]. Explicit construction of $\mathcal{H}_2$ with $\delta' = \mathcal{O}(\text{polylog } n)$ was presented in [26]. For the explicit version of construction, the degree δ of graph $\mathcal{G}$ is $2^{\mathcal{O}(\log^3 \log n)}$, which is asymptotically larger than polylog n.

4. MULTI-EXPANDERS

Let $G = (V, E)$, with $|V| = (n+1)D$, be a balanced $(D+1)$-partite graph, that is, there is a partition of V into disjoint subsets $V_0, V_1, \ldots, V_D$ of size n each so that every edge in E links two nodes in consecutive sets V_j and V_{j+1}. Graph G is a $(n, D, \delta, \varepsilon)$-*multi-expander* when the following four *multi-expansion conditions* hold:

Definition of $(n, D, \delta, \varepsilon)$-MULTI-EXPANDER:

Me0: The i-th node in layer V_j is joined to the i-th node in layer V_{j+1}, for every $1 \leq i \leq n$ and $0 \leq j < D$.

Me1: The inequalities

$$(1-\varepsilon)^{D-j}\cdot\delta^{D-j} \leq |N^{D-j}(v)\cap V_D| \leq \delta^{D-j}$$

hold for every $1 \leq j \leq D$ and every $v \in V_j$.

Me2: The equality $|N^D(v) \cap V_D| = n$ holds for every $v \in V_0$.

Me3: If $0 \leq j+i \leq D$, then

$$|N^i(A)\cap V_{j+i}| > \min\{|A|\delta^i\cdot(1-\varepsilon)^i, n-4n/(\delta\varepsilon)\},$$

and if $0 \leq j-i \leq D$, then

$$|N^i(A)\cap V_{j-i}| > \min\{|A|\delta^i(1-\varepsilon)^i, n-4n/(\delta\varepsilon)\},$$

for every $0 \leq j \leq D$ and every $0 < i \leq D$ and for every set $A \subseteq V_j$.

The sets V_j are called *layers*. The property Me1 is about how many nodes in the last layer can be reached, while Me2 is about reachability through all the layers to the last one. The property Me3 expresses iterated expansion and dispersion. In this section we show that multi-expanders exist. From now on we fix $\delta = b\log^3 n$, for a constant b as in Theorem 3.2, and $\varepsilon = 1/\log n$. We also assume that $n \geq 8b$.

THEOREM 4.1. *There exists a family of graphs $\langle G_n\rangle$, where G_n is a $(n, D, \delta, \varepsilon)$-multi-expander, for $\delta = b\log^3 n$, $\varepsilon = 1/\log n$, and $D = \lceil\log_\delta n\rceil$, for sufficiently large n and a sufficiently large constant $b > 0$. The degrees of nodes in graph G_n are at most $5\delta\log n = \mathcal{O}(\log^4 n)$.*

To build such graphs G_n, we may use (n, δ, ε)-connector H, that exist by Theorem 3.2. Connect two consecutive layers V_j and V_{j+1} by mapping the pattern of connections of H, for $j = 1, \ldots, D-1$, where V_j and V_{j+1} play the role of inputs and outputs of H. The obtained graph has properties Me0, Me1 and Me3. For instance, properties Me1 and Me3 for G' follow directly from the properties of a (n, δ, ε)-connector H: iterate regularity and expansion of a connector while moving through consecutive layers. Property Me2 does not need to hold, and the construction is augmented by adding more edges in a certain constructive way, details are omitted.

Explicit multi-expanders. If we use the explicit connectors from Section 3, then the obtained graph is an explicit $(n, D, \delta, \varepsilon)$-multi-expander, where $\delta = 2^{\mathcal{O}(\log^3 \log n)}$, $\varepsilon = 1/\log n$, and $D = \lceil\log_\delta n\rceil$, for sufficiently large n. This is because our construction of a multi-expander from a connector is explicit. The maximum degree of G_n is $5\delta\log n = 2^{\mathcal{O}(\log^3 \log n)}$.

5. ALGORITHM

We describe an algorithm that we call LAYERED-WRITE. Suppose there are some n processes and an array $A[1..n]$ of n registers whose entries are all zeroes.

We use a multi-expander $G = G_n$ as given in Theorem 4.1. Every node of this graph is represented by a unique shared-memory register. The topology of G_n is a part of code of the algorithm. The array $A[1..n]$ makes the layer V_D. We assign a unique node in the layer V_0 to each process.

Every register representing a node stores just a single-bit value, which is initialized to zero. The bit value stored at the node v in V_0 assigned to process $\mathfrak{p}$ is called the *completion bit* of the process. There is also a global flag initialized to zero. A graph induced by the nodes $\mathcal{P}(\mathfrak{p}) = \bigcup_{j=0}^{D}(N^j(v)\cap V_j)$ is called the *progress graph* for process $\mathfrak{p}$, where $v \in V_0$ is the node assigned to $\mathfrak{p}$. This graph plays a role similar to that of a progress trees used in the algorithms in [4, 9, 18, 19]. Every node of G is in some progress graph. We refer to $\mathcal{P}(\mathfrak{p})$ as both a set of nodes and a graph induced by these nodes, the specific meaning is clear from context.

Every process $\mathfrak{p}$ traverses its progress graph $\mathcal{P}(\mathfrak{p})$. The traversal starts at node $v \in V_0$ assigned to $\mathfrak{p}$. In an odd-numbered step, process $\mathfrak{p}$ performs the consecutive step of a DFS-like traversal of graph $\mathcal{P}(\mathfrak{p})$. In an even-numbered step, it reads the value stored at the global flag. A process terminates as soon as it detects that this flag is set to 1.

Now we define the traversal rules. If process $\mathfrak{p}$ arrives at node $w \in \mathcal{P}(\mathfrak{p}) \cap V_j$ for the first time and $j < D$, then $\mathfrak{p}$ systematically checks the values of all neighbors z of w in the next layer V_{j+1} in some order. If the bit-value of such a z is equal to zero, then $\mathfrak{p}$ processes the node z recursively. When finally all bits of the neighbors z are set to value 1, then $\mathfrak{p}$ returns back to the neighbor of w in the previous layer from which $\mathfrak{p}$ arrived to w originally.

This traversal is similar to the ordinary DFS one, with an additional caveat regarding order: if w is the i-th node of layer V_j, then $\mathfrak{p}$ always checks first on the i-th node in layer V_{j+1}, which is consistent with property Me0 of graph G. This implies that if a process $\mathfrak{p}$ is fast, then it traverses all the layers and updates the register $A[\mathfrak{p}]$ as a first thing to do regarding the array A, which is a natural priority. Process $\mathfrak{p}$ completes its traversal when its completion bit in layer V_0 is set to 1. Then it sets the global flag to 1 and halts.

THEOREM 5.1. *The algorithm* LAYERED-WRITE *performs work $\mathcal{O}(n\log^{18} n/(\log\log n)^3)$.*

A proof of Theorem 5.1 is given in Section 6.

Explicit version of the algorithm. An instantiation of the algorithm is explicit when we use an explicit multi-expander. To this end take the one described in Section 4. The complexity of this algorithm will increase by the factor $(2^{\mathcal{O}(\log^3 \log n)})^5 = 2^{\mathcal{O}(\log^3 \log n)}$, if the formula $\mathcal{O}(nD^3\delta^5)$ is applied to estimate the work. This is because the degree of the explicit multi-expander is by this factor bigger than the one in the existential multi-expander, while D is $\mathcal{O}(\log n / \log \log n)$.

Corollary 5.2. *There is an explicit instantiation of algorithm* Layered-Write *for n processes which performs work $n \cdot 2^{\mathcal{O}(\log^3 \log n)}$.*

6. ANALYSIS OF THE ALGORITHM

Nodes in every layer V_j of multi-expander G are ordered, where this ordering is the one used to state the property $\mathsf{Me0}$. Let $v(i,j)$ denote the i-th node in layer V_j. We say that a node w in G is *checked* if there is value 1 written in it, otherwise it is *void*. Initially all nodes are void. For a node $v \in V_j$, for some layer number $0 \le j < D$, the *progress graph rooted at v* is a graph induced by the nodes in $\mathcal{P}(\mathfrak{p}, v) = \bigcup_{j'=j+1}^{D} (N^{j'-j}(v) \cap V_{j'})$.

Lemma 6.1. *The algorithm is correct and each process performs $\mathcal{O}(n)$ steps by its termination.*

We partition the trace into consecutive *segments*. The position when the i-th segment ends is denoted by τ_i. The sequence of positions $\langle \tau_i \rangle_{i \ge 0}$ is defined inductively as follows. The position τ_0 is set to 0. Suppose that we have the i-th segment already defined, by some position τ_i. For each position $\mu > \tau_i$, and every k such that $0 \le k \le D-1$, consider a set $A(k,\mu)$ consisting of these processes that perform at least $12\frac{dn}{\delta^{k-3}}$ and less than $12\frac{dn}{\delta^{k-4}}$ steps each after position τ_i and up to the position μ. Let $A(0,\mu)$ be a set consisting of these processes that perform at least $12dn\delta^3$ steps each by position μ. Define τ_{i+1} to be the minimum position μ after τ_i such that $|A(k,\mu)| \ge \delta^k$, for some $0 \le k \le D-1$. Accordingly, for every such a segment $i+1$, we define a set P_{i+1} to be the set $A(k, \tau_{i+1})$, where k is the biggest index such that $0 \le k \le D-1$. Note that it always exists provided position τ_{i+1} exists, by the definition of τ_{i+1}. If the former case applies, then P_{i+1} is of the size δ^k, for $0 \le k \le D-1$, and the $(i+1)$-st segment is said to be *k-dense*.

Lemma 6.2. *The segments are well defined, and at most $\mathcal{O}(nD\delta^5)$ steps are performed during one segment, for all sufficiently large n.*

Consider the first D^2 segments. There must be a number ℓ, for $0 \le \ell \le D-1$, such that there are at least D ℓ-dense segments among the considered ones. If $\ell = 0$ has this property, then by definition of 0-dense segment some process performs $12dn\delta^3$ steps, and during them it also performs all tasks in the worst case, unless it learns earlier that some of them are completed, and sets the global flag to 1. Hence the algorithm performs work $\mathcal{O}(nD\delta^5)$ by Lemma 6.2.

Next suppose that the number ℓ with the property is at least 1. Consider only the first D among the ℓ-dense segments, and, for the sake of simplicity of notation, assume that they are numbered from 1 to D, although such segments do not need to cover a contiguous region. Let $U(i,j)$ denote a set of nodes in layer V_j which are void at the end of the ith k-dense segment. Denote by c the number $c = 12n/\delta^{\ell-3}$, which is the minimum number of traversing steps performed by a process associated with the considered segment, that is, a process in P_r, where r is the position of the segment. The following *i-invariant* holds for $1 \le i < \ell$:

$$|U(i,j)| \le 4n/(\delta^{j-D+i}\varepsilon), \text{ for every } D-i+1 \le j \le D \quad (1)$$

It can be shown by induction on i using the following fact:

Lemma 6.3. *Let $S \subseteq V_{\ell-i}$ and $S' \subseteq S$ be of a size at least $|S|/2$, while $U \subseteq V_j$ is of a size at most $4n/(\delta^{j-D+i-1}\varepsilon)$, for $D-(i-1)+1 \le j \le D$. Then there is a set $S'' \subseteq S'$ of a size at least $|S'|(1-1/\delta)$ such that*

$$\sum_{v \in S''} |N^{j-(\ell-i)}(v) \cap U| \le 28n\delta^2/\varepsilon \ .$$

Concluding the analysis. We apply the invariant (1) for $i = \ell - 1$. Note that $\ell - 1 < D$, hence the ℓ-th ℓ-dense segment is among the first D ℓ-dense segments and among the first D^2 segments.

Consider the ℓ-th ℓ-dense segment and a set R containing δ^ℓ processes $\mathfrak{p}$ such that $\mathfrak{p}$ performs at least c steps during the considered segment, which exists by the definition of a ℓ-dense segment. Let S be the set $\bigcup_{\mathfrak{p} \in R} \{v(\mathfrak{p}, 0)\}$, and let R^* be a set of these processes $\mathfrak{p} \in R$ that traverse the whole progress graph rooted in $v(\mathfrak{p}, 0)$ during the first c steps of the considered segment. Let $S^* = \bigcup_{\mathfrak{p} \in R} \{v(\mathfrak{p}, 0)\}$ be the set of nodes corresponding to the processes in R^*.

Let Δ be the maximum degree of a node in graph G. From the proof of Theorem 4.1, we have that $\Delta \le 5\delta \log n$. Moreover, each node v in G has exactly δ neighbors in the next layer, and exactly δ neighbors in the previous layer, except for the nodes in layer V_0 that may have at most Δ neighbors in layer V_1 and the nodes in layer V_1 that may have at most Δ neighbors in layer V_0.

Apply Lemma 6.3 to the layers from $D-(\ell-1)+1$ to D and obtain that there is a set $\hat{S} \subseteq S$ of a size at least $|S|(1 - 1/\delta)^{\ell+1} > |S|/2$ such that the number of void nodes in $\hat{S}$ at the beginning of the considered segment is at most

$$\sum_{j=D-(\ell-1)+1}^{D} \sum_{v \in \hat{S}} |N^j(v) \cap U| \le (\ell+1) \cdot 28n\Delta\delta/\varepsilon \ ,$$

where U here denotes the set of void nodes at the beginning of the considered segment in the layers from $V_{D-(\ell-1)+1}$ to V_D. Consequently, the set $\hat{S}^* \subseteq \hat{S}$ of nodes v such that the progress graph rooted at v contains at most $c/4 = 3n/\delta^{\ell-3}$ void nodes in the layers from $D-(i-1)+1$ to D at the beginning of the considered segment is nonempty. Indeed, otherwise the number of void nodes would be at least

$$|S \setminus (\hat{S} \cap S^*)| \cdot 3n/\delta^{\ell-3} \ge \delta^\ell \cdot 3n/\delta^{\ell-3} > (\ell+1) \cdot 28n\Delta\delta/\varepsilon,$$

for a sufficiently large constant $b = \delta/\log^3 n$, say for $b > 10$. Similarly $\hat{S}^* \subseteq S^*$, which implies that S^* is nonempty.

Each process $\mathfrak{p} \in S^*$ has traversed all nodes in the progress graph rooted at $v(\mathfrak{p}, 0)$. This is the progress graph $\mathcal{P}(\mathfrak{p})$, which by property $\mathsf{Me2}$ contains all nodes in layer V_D. During the first D^2 segments in execution the global flag is set to 1 and all processes halt. By Lemma 6.2, the total work during these D^2 segments is $\mathcal{O}(nD^3\delta^5)$, for $\delta = \mathcal{O}(\log^3 n)$ and $D = \mathcal{O}(\log_\delta n) = \mathcal{O}(\log n / \log \log n)$.

This completes the proof of Theorem 5.1.

7. DISCUSSION

It remains an open problem to find an *explicit* solution to *Write-All* with bound $\mathcal{O}(n \text{ polylog } n)$ on its work.

Algorithm LAYERED-WRITE can be implemented as a randomized one, by selecting a connector at random in the beginning of an execution. This algorithm performs work $\mathcal{O}(n \text{ polylog } n)$ with a large probability against the strongest adaptive adversary who knows all random bits prior to the start of an execution. Previously given randomized solutions were analyzed only against a weaker oblivious adversary.

Write-All has been abstracted and studied because this primitive can be applied in simulations of a PRAM on weaker variants, say, those allowing process crashes. Then *Write-All* is applied iteratively, and the next iteration imposes the reversed roles for the auxiliary memory and the registers to be updated. For this not to incur an additional overhead, the capacities of dedicated and auxiliary memory need to be comparable. The solution we describe uses $\Theta(nD) = \Theta(n \log n / \log\log n)$ auxiliary memory cells. To adapt this to $\mathcal{O}(n)$ registers of a size $\mathcal{O}(\log n)$ each, partition the dedicated registers into groups of size D each, and treat them as individual registers for the purpose to build a multi-expander graph. Each process updates the whole group of registers as a single task. This results in work performance increased by a factor of only $D = \mathcal{O}(\log n / \log\log n)$.

8. REFERENCES

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic snapshots of shared memory, *Journal of the ACM,* 40 (1993) 873 - 890.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts, A theory of competitive analysis of distributed algorithms, in *Proc., 33rd IEEE Symposium on Foundations of Computer Science (FOCS),* 1994, pp. 401 - 411.

[3] N. Alon, and J.H. Spencer, *"The Probabilistic Method,"* 2nd ed., J. Wiley, New York, 2000.

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

[5] J. Aspnes, and M. Herlihy, Wait-free data structures in the asynchronous PRAM model, in *Proc., 2nd ACM Symposium on Parallel Algorithms and Architectures (SPAA),* 1990, pp. 340 - 349.

[6] J. Aspnes, and W. Hurwood, Spreading rumors rapidly despite an adversary, *Journal of Algorithms*, 26 (1998) 386 - 411.

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

[8] M. Capalbo, O. Reingold, S. Vadhan, and A. Wigderson, Randomness conductors and constant-degree lossless expanders, in *Proc., 34th ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 659 - 668.

[9] B.S. Chlebus, S. Dobrev, D.R. Kowalski, G. Malewicz, A.A. Shvartsman, and I. Vřto, Towards practical deterministic write-all algorithms, in *Proc., 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA),* 2001, pp. 271 - 280.

[10] B.S. Chlebus, D.R. Kowalski, and A. A. Shvartsman, Collective asynchronous reading with polylogarithmic worst-case overhead, in *Proc., 36th ACM Symposium on Theory of Computing (STOC)*, 2004, pp. 321 - 330.

[11] F. Fich, The complexity of computing on a parallel random access machine, in J. Reif, editor, *"Synthesis of Parallel Algorithms,"* Morgan Kaufmann, San Mateo, CA, 1993.

[12] P.C. Kanellakis, and A.A. Shvartsman, Efficient parallel algorithms can be made robust, *Distributed Computing*, 5 (1992) 201 - 217.

[13] P.C. Kanellakis, and A.A. Shvartsman, *"Fault-Tolerant Parallel Computation,"* Kluwer Academic, New York, 1997.

[14] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis, Combining tentative and definite executions for dependable parallel computing, in *Proc., 23rd ACM Symposium on Theory of Computing (STOC),* 1991, pp. 381 - 390.

[15] D.R. Kowalski, and A.A. Shvartsman, Writing-all deterministically and optimally using a non-trivial number of asynchronous processors, in *Proc., 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA),* 2004, pp. 311 - 320.

[16] A. Lubotzky, R. Phillips, and P. Sarnak, Ramanujan graphs, *Combinatorica*, 8 (1988) 261 - 277.

[17] N.A. Lynch, *"Distributed Algorithms,"* Morgan Kaufmann, San Mateo, CA, 1996.

[18] G. Malewicz, A method for creating near-optimal instances of a certified write-all algorithm, in *Proc., 11th European Symposium on Algorithms (ESA)*, 2003, pp. 422 - 433.

[19] G. Malewicz, A work-optimal deterministic algorithm for the asynchronous certified write-all problem, in *Proc., 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, 2003, pp. 255 - 264.

[20] C. Martel, and R. Subramonian, On the complexity of certified write-all algorithms, *Journal of Algorithms*, 16 (1994) 361 - 387.

[21] N. Pippenger, Sorting and selecting in rounds. *SIAM Journal on Computing*, 16 (1987) 1032–1038.

[22] O. Reingold, S.P. Vadhan, and A. Wigderson, Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors, *Annals of Mathematics,* 155 (2002) 157 - 187.

[23] M. Saks, N. Shavit, and H. Woll, Optimal time randomized consensus - making resilient algorithms fast in practice, in *Proc., 2nd SIAM–ACM Symposium on Discrete Algorithms, (SODA)*, 1991, pp. 351 - 362.

[24] M. Santha, On using deterministic functions in probabilistic algorithms, *Information and Computation,* 74 (1987) 241 - 249.

[25] M. Sipser, Expanders, randomness, or time vs. space, *Journal of Computer and System Sciences,* 36 (1988) 379 - 383.

[26] A. Ta-Shma, C. Umans, and D. Zuckerman, Loss-less condensers, unbalanced expanders, and extractors, in *Proc., 33rd ACM Symposium on Theory of Computing (STOC),* 2001, pp. 143–152.

[27] A. Wigderson, and D. Zuckerman, Expanders that beat the eigenvalue bound: explicit construction and applications, *Combinatorica*, 19 (1999) 125 - 138.