

Writing-All Deterministically and Optimally Using a Non-Trivial Number of Asynchronous Processors

Dariusz R. Kowalski*
Max-Planck-Institut für Informatik,
Stuhlsatzenhausweg 85,
Saarbrücken, 66123, Germany
and
Instytut Informatyki, Uniwersytet Warszawski,
Banacha 2, 02-097 Warszawa, Poland

Alexander A. Shvartsman†
Department of Computer Science and
Engineering, University of Connecticut,
Storrs, CT 06269, USA
and
MIT CSAIL, The Stata Center,
Cambridge, MA 02139, USA

ABSTRACT

The problem of performing n tasks on p asynchronous or undependable processors is a basic problem in distributed computing. This paper considers an abstraction of this problem called *Write-All: using p processors write 1's into all locations of an array of size n* . In this problem writing 1 abstracts the notion of performing a simple task. Despite substantial research, there is a dearth of efficient deterministic asynchronous algorithms for *Write-All*. Efficiency of algorithms is measured in terms of *work* that accounts for all local steps performed by the processors in solving the problem. Thus an optimal algorithm would have work $\Theta(n)$, however it is known that optimality cannot be achieved when $p = \Omega(n)$. The quest then is to obtain work-optimal solutions for this problem using a non-trivial, compared to n , number of processors p . Recently it was shown that optimality can be achieved using a non-trivial number M of processors, where $M = \sqrt[4]{n/\log n}$. The new result in this paper *significantly* extends the range of processors for which optimality is achieved. The result shows that optimality can be achieved using close to M^2 processors; more precisely, using $(M \log M)^{2-\varepsilon}$ processors, for any $\varepsilon > 0$. Additionally, the new result uses *only* the atomic read/write memory, without resorting to using the test-and-set primitive that was necessary in the previous solution. This paper presents the algorithm and gives its analysis showing that the work complexity of the algorithm is $O(n + p^{2+\varepsilon})$, which is optimal when $p = O(n^{1/(2+\varepsilon)})$, while all prior deterministic algorithms require super-linear work when $p = \Omega(n^{1/4})$.

*E-mail: darek@mpi-sb.mpg.de. The work of this author is supported in part by the KBN Grant 4T11C04425 and by the NSF-NATO Award 0209588.

†This author dedicates his work on this paper to I.A.T. Email: aas@cse.uconn.edu. Work supported in part by the NSF CAREER Award 9984778 and by the NSF Grants 9988304, 0121277, and 0311368.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'04, June 27–30, 2004, Barcelona, Spain.

Copyright 2004 ACM 1-58113-840-7/04/0006 ...\$5.00.

Categories and Subject Descriptors

F.2.m [Analysis of Algorithms and Problem Complexity]: Miscellaneous; F.1.2 [Modes of Computation]: Parallelism and concurrency

General Terms

Algorithms, performance, reliability.

Keywords

Distributed algorithms, shared memory, Write-All, asynchrony, work.

1. INTRODUCTION

The problem of performing n tasks on p asynchronous or undependable processors is a basic problem in distributed computing. This problem has been studied in a variety of models, including shared-memory [15, 18, 24] and message-passing [10, 11, 12]. Efficiency of algorithms in most settings is measured in terms of *work* that accounts for all steps taken by the processors in solving the problem. Developing efficient algorithms solving such task-performing problems has proven to be difficult. Looking beyond the immediacy of the intellectual challenge, the ability to cooperate on a common set of tasks in distributed settings is key to solving a broad range of computation problems ranging from simulations of synchronous parallel algorithms on asynchronous multiprocessors, to distributed search such as SETI@home, and distributed multi-agent collaboration.

Background and motivation. This paper considers the p -processor, n -size problem formulated for shared-memory models of computation and called the *Write-All* problem [15]: *using p processors write 1's to all locations of an array of size n* . In this problem writing 1 abstracts the notion of performing a constant-time task. Despite its simplicity, solutions for *Write-All* can be used in constructing other interesting robust algorithms (e.g., [1, 14]) and developing simulations of synchronous parallel algorithms on asynchronous or undependable parallel processors (e.g., [19, 24, 27]), ultimately leading to system implementations (e.g., [9]).

In the design of practical parallel programs one needs to ensure good performance and dependability on multiprocessors with unpredictable load patterns. Here a common challenge is to efficiently perform n independent tasks on

p processors. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix. In such cases a *Write-All* algorithm can be used with the only change being that the assignment to a particular array element is preceded by the execution of a distinct task.

The *Write-All* problem is trivially solved when the processors are synchronous and failure-free. However, when failures or delays are introduced, the problem becomes very challenging. *Write-All* has been substantially studied for synchronous failure-prone processors [16], and a number of related randomized algorithms have been developed for the asynchronous settings, e.g., [4, 5, 8, 13, 17, 22, 26]. Yet there is still a dearth of efficient deterministic asynchronous algorithms.

We measure the efficiency of algorithms in terms of *work* that accounts for all machine instructions executed by the p processors during the computation (this generalizes the processor-time product measure of synchronous parallel algorithms). The first deterministic asynchronous algorithm for *Write-All* is due to Buss *et al.* [6] and its worst case work complexity is $O(n \cdot p^{\log 3/2})$. Note that a work-optimal algorithm would have performed work $\Theta(n)$, however it was shown [15, 18] that optimality cannot be achieved when $p = \Omega(n)$. Moreover, when $p = n$, both deterministic and randomized asynchronous solutions are subject to the lower bound of $\Omega(n \log n)$ [6, 18, 23]. In fact, the best of the known deterministic asynchronous algorithms ([3, 6, 7, 14, 21]) for *Write-All* has work $\omega(n)$, whenever $p = \Omega(n^{1/4})$.

The quest then is to obtain work-optimal solutions for this problem using the largest possible, and non-trivial compared to n , number of processors p .

Contribution. For a long time, the most efficient known deterministic asynchronous algorithms for the *Write-All* problem were the elegant algorithms of Anderson and Woll [3]. One class of these algorithms has work $O(n \cdot p^\varepsilon)$ for $p \leq n$ and another has work $O(n \log n)$ for $p = \sqrt{n}$.

These algorithms, and the novel algorithms of Groote *et al.* [14] that are quite efficient for $p \ll n$, have work complexity $\omega(n)$ for all but a trivial number p of processors. Recently Malewicz [21] showed that work-optimality can be achieved using a non-trivial number M of processors, where $M = \sqrt[4]{n/\log n}$.

Our new result in this paper substantially improves the range of processors for which optimality can be achieved. Specifically, we show that optimality can be achieved using close to M^2 processors; more precisely, using $(M \log M)^{2-\varepsilon}$ processors, for any $\varepsilon > 0$. Our solution is constructed as a two-level nested algorithm, where the pivotal decision leading to optimality is whether and when to allow the processors to work in isolation, and when to force them to cooperate. When cooperating, the processors use a careful parameterization of one of the algorithms of Anderson and Woll [3].

We present an original analysis of the new solution showing that the work complexity of our solution is $O(n + p^{2+\varepsilon})$, and that the algorithm is optimal when $p = O(n^{1/(2+\varepsilon)})$. Thus our result significantly extends the range of optimality of deterministic algorithms for *Write-All*. By contrast, the algorithms of Anderson and Woll have super-linear work when $p = \omega(1)$, and all other prior algorithms have super-linear work when $p = n^a$, for any $a > 1/4$. Additionally,

our algorithm does not assume the test-and-set primitive used by Malewicz [21], and relies *only* on atomic read/write memory.

Algorithmic background. The algorithm of Buss *et al.* [6] solving the *Write-All* problem uses a binary *progress tree* that associates tasks (array elements) with the leaves of the tree. The progress tree directs processors to tasks, and helps balance the loads of the processors. This is done using the individual bits of processor identifiers, where each bit is used to choose between the left and the right branches within the progress tree.

The algorithm of Anderson and Woll [3] uses q -ary progress trees with n leaves ($q \leq n$) and a set of q permutations from S_q , the symmetric group of permutations of $[q]$. The p processors use these permutations in conjunction with processor identifiers to determine the order of traversal of the q subtrees of each interior node of the progress tree. This is done by using the q -ary representation of processor identifiers. It was shown [3] that for any $\varepsilon > 0$ it is possible to choose q and a set Ψ of q permutations such that the algorithm has work $O(n \cdot p^\varepsilon)$. (Note that this work is non-optimal whenever $p = \omega(1)$).

The analysis in [3] uses the notion of *left-to-right maxima* of permutations (see Knuth vol. 3, p. 13 [20]). The key in the analysis is to show that one can select a set Ψ of q permutations from S_q such that if one chooses an arbitrary permutation from S_q , and composes it with each permutation in Ψ , then the sum of the number of *left-to-right maxima* of the resulting q permutations is $O(q \log q)$. This sum is called the *contention* of the set of permutations. In the settings where q can be assumed to be a constant, it is reasonable to find the requisite permutations using an exhaustive search. Recently, Chlebus *et al.* [7] presented analytical and experimental evidence that suitable permutations can be constructed efficiently.

In this work we present an algorithm that is based on a two-level nested application of progress tree algorithms. At the lowest level the processor are forced to make a decision on whether to work in isolation using a sequential algorithm and a local schedule, or to cooperate. A processor cooperates when it believes that a certain collection of tasks is *saturated* with processors. By carefully choosing the saturation point, we produce an algorithm that can be made optimal for a non-trivial range of processors. The algorithm is additionally parameterized by a set of permutations. It is correct for any set of permutations, but its efficiency depends on selecting a set of permutations with certain combinatorial (contention) properties that are known to exist. We present an original analysis showing that our algorithm is not only efficient, but surprisingly, that it has optimal work $O(n)$ for a substantial range of processors. Whereas the algorithm of Anderson and Woll achieves optimality only when $p = O(1)$, our algorithm is optimal for $p = O(n^{1/(2+\varepsilon)})$, which nearly *squares* the best previous range of processors given by Malewicz [21].

Document structure. We define the computation model and the needed combinatorial structures in Section 2. In Section 3 we present our algorithm, and in Section 4 we give its correctness and analysis. We discuss our results and future work in Section 5.

2. MODEL, DEFINITIONS, AND COMBINATORIAL STRUCTURES

We now define the computation model, the measure of efficiency, and contention of permutations.

Model of computation. In our parallel setting we consider a system consisting of p processors with unique identifiers (pid) from $P = \{0, \dots, p-1\}$. The processors have access to (atomic) shared memory, including the shared array $wa[1..n]$. We assume that each memory cell is able to store $O(\log(p+n))$ bits. The *Write-All* problem is for the p processors to perform n tasks, where each task consists of setting a distinct location of the array $wa[\cdot]$ to 1.

A processor's activity is governed by its local clock. We model asynchrony as an adversary that introduces delays between local clock ticks. The delays can be arbitrary, but the adversary is constrained not to delay forever at least one processor. (Note that this model can be used to model a processor crash as an infinite delay following some step of the processor. The only restriction is that at least one processor is non-faulty.)

Measuring efficiency. We study the complexity of *Write-All* measured as *work* that accounts for all steps taken by the processors. For the purpose of algorithm analysis we assume the existence of a global real-timed clock that is unknown to the processors. We assume that it takes a unit of time for a processor to perform a unit of work, such as writing to a shared memory cell, according to its local clock. Idling processors consume a unit of work per idling step. For an execution ξ of an algorithm, we denote by $p_i(\xi)$ the number of processors completing a unit of work at time i of the computation (according to the global time that is not available to the processors). Let τ_ξ be the time when all tasks have been performed and at least one processor is informed of this fact. Then we define work as follows.

DEFINITION 2.1. *Given a p -processor algorithm that solves a given problem of size n , its work complexity W is defined as:* $W_{n,p} = \max_{\xi} \left\{ \sum_{i \leq \tau_\xi} p_i(\xi) \right\}$.

Schedules and contention. We now introduce and analyze the combinatorial structures used in formulating algorithms and evaluating their efficiency.

Consider a list of some idempotent computational *jobs* with identifiers from the set $[q] = \{1, \dots, q\}$. (We make the distinction between *tasks* and *jobs* for convenience to structure algorithm presentation and analysis; a job may be composed of one or more tasks, e.g., a job may be defined as writing to one or more locations of the shared array $wa[\cdot]$ and performing some other computation.) We refer to a list of job identifiers as a *schedule*. When a schedule for q jobs is a permutation of job identifiers π in S_q , we call it q -*schedule*. Here S_q is the symmetric group, the group of all permutations on the set $[q]$; we use \circ as the composition operator, and e_q to denote the identity permutation. For a q -schedule $\pi = \langle \pi(1), \dots, \pi(q) \rangle$ a *left-to-right maximum*, *lrm* for short (see Knuth vol. 3, p. 13 [20]), is an element $\pi(i)$ of π that is larger than all of its predecessors, i.e., $\pi(i) > \max_{j < i} \{\pi(j)\}$.

Given a q -schedule π , we define $\text{LRM}(\pi)$, to be the number of left-to-right maxima in the q -schedule π (see [3]). For a

```

00 const  $\Psi = \{\pi_v \mid 0 \leq v < q \wedge \pi_v \in S_q\}$     %-  $q$  permutations
01 forall processors  $pid = 0$  to  $q-1$  parbegin
02     for  $i = 1$  to  $q$  do                               %- Do work according
03         perform  $Job(\pi_{pid}(i))$                    %- to permutations
04 parend.
```

Figure 1: Algorithm OBLIDO.

list $\Psi = \langle \pi_0, \dots, \pi_{q-1} \rangle$ of q permutations from S_q and a permutation δ in S_q , the contention of Ψ with respect to δ is defined as $\text{Cont}(\Psi, \delta) = \sum_{u=0}^{q-1} \text{LRM}(\delta^{-1} \circ \pi_u)$. The contention of the list of schedules Ψ with respect to the set of permutations Υ is defined as $\text{Cont}(\Psi, \Upsilon) = \max_{\delta \in \Upsilon} \{\text{Cont}(\Psi, \delta)\}$. The *contention of the list of schedules* Ψ is defined as

$$\text{Cont}(\Psi) = \text{Cont}(\Psi, S_q) = \max_{\delta \in S_q} \{\text{Cont}(\Psi, \delta)\}.$$

Note that for any Ψ , $q \leq \text{Cont}(\Psi) \leq q^2$. A family of permutations with low contention was introduced in [3], where the following is shown ($H_q = \sum_{j=1}^q \frac{1}{j}$ is the q th harmonic number).

LEMMA 2.1. [3] *For any integer $q > 0$ there exists a list $\Psi = \langle \pi_0, \dots, \pi_{q-1} \rangle$ of q permutations from S_q , such that $\text{Cont}(\Psi) \leq 3qH_q = \Theta(q \log q)$.*

Lemma 2.1 holds also for random set of permutations Ψ , with the probability at least $1 - 2^{-q}$.

Contention and oblivious tasks scheduling. Here we show how to use contention to analyze asynchronous job scheduling. We use this approach as a building block of our *Write-All* algorithm presented later. Assume that q distinct asynchronous processors perform t jobs such that processor pid performs the jobs in the order given by the permutation π_{pid} in Ψ . We call this oblivious algorithm OBLIDO and give the code¹ in Figure 1.

Since OBLIDO does not involve any coordination among the processors the total of q^2 jobs are performed (counting multiplicities). However, it was shown [3] that if we count only the job executions where a job has not been previously performed by any processor, then the total number of such job executions is bounded by $\text{Cont}(\Psi)$, again counting multiplicities. We call such job executions *primary*, we call all other job executions *secondary*. Note that the number of primary executions cannot be smaller than q , since each job is performed at least once for the first time. In general the number of primary executions is going to be between q and q^2 , because several processors may be executing the same job concurrently for the first time.

LEMMA 2.2. [3] *In algorithm OBLIDO with q processors, q jobs, and using the list Ψ of q permutations, the number of primary job executions is at most $\text{Cont}(\Psi)$.*

Clearly our oblivious algorithm performs all tasks, however when this algorithm is used as a building block in algorithms based on the progress tree paradigm, the overall efficiency depends on $\text{Cont}(\Psi)$, which may be much larger than q , leading to unexciting work complexity.

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

```

00  const q                                     %- Arity of the progress tree
01  const  $\Psi = \langle \pi_r \mid 0 \leq r < q \wedge \pi_r \in S_q \rangle$   %- Fixed list of q permutations of [q]
02  const  $h = \log_q p$                                      %- The height of the progress tree
03  const  $g = (q^{h+1} - 1) / (q - 1)$                        %- The size of the progress tree
04  shared array  $wa[1 .. n]$  of  $\{0, 1\}$                    %- The Write-All array
05  shared array  $T[0 .. g-1]$  of  $\{0, 1\}$  init  $\{0\}^g$       %- The progress tree
06  shared array  $Chunk[1 .. p][1 .. t]$  of  $[p] \times [tp]$   %- Array of pairs (job, chunk) for processors
07  const array  $JobChunks[1 .. p]$  of set of  $[tp]$         %- Fixed sets of chunks in jobs
08  private set  $Local_{pid}$  of  $P$                           %- Local set of performed chunks in current job
09  private array  $Pos_{pid}[0 .. p-1]$  of  $[t]$            %- Local array of indices of right-most chunks
10  private set  $Coop_{pid}$  of  $P$                           %- Local set of processors performing same job

```

Figure 2: Constants definition and the state of algorithm $WA(q, t)$ for each processor pid

3. ALGORITHM $WA(Q, T)$

We now present a deterministic algorithm, called $WA(q, t)$, for p processors and n array elements (tasks), parameterized by q and t (Figures 2, 3, and 4). In Section 4 we show that its work is $O(n + p^{2+\varepsilon})$, for any $\varepsilon > 0$. The algorithm uses a list Ψ of q schedules (permutations) from S_q , $\Psi = \langle \pi_0, \dots, \pi_{q-1} \rangle$. We assume that $\log_q p$ is a positive integer. If it is not, we pad the processors with at most qp “infinitely delayed” processors so this assumption is satisfied (given that q is a constant, we will see that this padding does not affect the asymptotic complexity of the algorithm). Thus we define a family of algorithms, where each algorithm is parameterized by q and t ; additionally we assume that $n \geq p^2$.

Our algorithm uses a complete q -ary *progress tree* with p leaves. The n tasks are represented by the n locations of the array $wa[1..n]$. We divide the n tasks into p *jobs*, each of size at most $\lceil n/p \rceil$. Each job is associated with a leaf of the progress tree. The nodes of the progress tree are initially 0. When all jobs corresponding to the leaves of a given subtree are complete, the root of the subtree is set to 1. When the root of the tree is set 1, all jobs are complete. The processors traverse the progress tree in search of unfinished jobs as in [3, 6]. Each processor chooses the order of traversal according to its *pid* and the permutations from Ψ . When a processor reaches a leaf, it checks whether the job at the leaf is complete. If not, it starts a procedure for performing the job. As we pointed out, the idea of progress tree traversal was used before [3, 6], however such algorithms have super-linear work for all but a trivial number of processors ($p = O(1)$). The new idea in our algorithm is to augment the tree traversal with a subtle procedure for performing jobs. We divide each job into t *chunks*, where each chunk consists of at most $\lceil (n/p)/t \rceil$ tasks. When a processor works on a job, it chooses a different algorithm depending on the knowledge of how many processors are performing this job. Within a job, the tasks are processed at the granularity of chunks. We now present the algorithm in detail. The code is given in Figure 2 (state components), Figure 3 (progress tree traversal), and Figure 4 (doing jobs). The line numbers in the presentation below refer to these figures.

3.1 Progress tree traversal

The main data structure of the algorithm is a boolean *progress tree* defined to be a complete q -ary tree of height h with p leaves, where $h = \log_q p$. Each of the n *tasks* consists of writing to one shared memory locations of the *Write-All* array $wa[1 .. n]$. The tasks are partitioned into p *jobs*, each consisting of $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ tasks. To perform a job means setting all elements of $wa[\cdot]$ associated with the job to 1.

(The assignment of tasks to jobs can be done statically and we do not discuss this further.) The p jobs are associated with the leaves of the progress tree.

The progress tree is stored in a boolean array $T[0 .. g-1]$ (line 5), where $g = \sum_{i=0}^{h-1} q^i = (q^{h+1} - 1) / (q - 1) = (pq - 1) / (q - 1)$. Here $T[0]$ is the root², and the q children of the interior node $T[d]$ are the nodes $T[qd+1], T[qd+2], \dots, T[qd+q]$. The space occupied by the tree is $O(p)$. Initially all nodes of the progress tree are set to 0 (false) indicating that not all jobs have been performed. The node of tree is set to 1 (true) whenever all jobs in a subtree rooted at that node are complete. Note that such updates of the progress tree T are always monotone: initially each node contains 0, then once a node changes its value to 1 it remains 1 forever. Thus no race conditions arise, which in turn allows the algorithm to use read/write memory without any additional synchronization primitives.

The progress tree is traversed using the recursive procedure `WRITEALL` (lines 30-40). Each processor independently searches for work in the smallest immediate subtree that has remaining work. It then performs any jobs it finds, and moves out of that subtree when all jobs within it are complete. When exploring the subtrees rooted at an interior node at height l , a processor visits the subtrees in the order given by one of the permutations in $\Psi = \langle \pi_0, \pi_1, \dots, \pi_{q-1} \rangle$. We represent the *pid* of each of the p processors in terms of its q -ary expansion stored in the local array $\alpha[0 .. h-1]$. The processor then uses the permutation $\pi_{\alpha(l)} \in \Psi$, where $\alpha(l)$ is the value of the l -th digit in the q -ary expansion of its *pid* (lines 37-38). The traverses within a subtree only if the corresponding bit in the progress tree is not set (line 33). In other words, each processor *pid* traverses its progress tree in a post-order fashion using its *pid* and the permutations in Ψ to establish the order of the traversals, except that the traversal can be pruned based on the progress of other processors recorded in the tree. When a processor reaches a leaf corresponding to an unfinished job, it performs the job by calling the procedure `DOJOB` (line 35).

3.2 Performing jobs

Jobs are structured as follows. We partition each job into t *chunks*, where each chunk consists of $\lfloor \frac{n}{pt} \rfloor$ or $\lceil \frac{n}{pt} \rceil$ tasks, where t is a parameter of algorithm. (As with jobs, the assignment of tasks to chunks can be done statically and we do not detail this.) Two different procedures are used to perform tasks within a job, depending on how many pro-

²The root doubles as the “certification bit” if one is interested in solving *Certified Write-All* problem, where the certification bit is to be set to one when the problem is solved.

```

20 forall processors  $pid = 0$  to  $p - 1$  parbegin                               %- Traverse progress tree in search of work
21     const array  $\alpha[0 .. h - 1]$  of integer =  $pid_{(base\ q)}$                 %-  $h$  digits of  $q$ -ary expansion of  $pid$ 
22     integer  $d$  init = 0                                                    %- Current node index is  $d$ , begin at the root  $T[0]$ 
23     integer  $l$  init = 0                                                    %- Current depth in the tree
24     WRITEALL( $d, l$ )
25 parent.

```

```

30 procedure WRITEALL( $d, l$ )                                                %- Recursive progress tree traversal
31                                                                                                                    %-  $d$  : current node index;  $l$  : depth of the current node
32                                                                                                                    %- Current node is not done – still work left
33     if  $T[d] = 0$  then                                                       %- Node  $T[d]$  is a leaf
34         if  $l = h$  then                                                    %- Do the job at the leaf  $T[d]$ 
35             DOJOB( $d + p - q$ )
36         else                                                                 %- Node  $d$  is not a leaf
37             for  $j = 1$  to  $q$  do                                           %- Visit subtrees in the order of  $\pi_{\alpha[l]}$ 
38                 WRITEALL( $q \cdot d + \pi_{\alpha[l]}(j), l + 1$ )             %- Visit subtree in the  $j$ -th position of  $\pi_{\alpha[l]}$ 
39                  $T[d] := 1$                                                %- Record completion of the subtree rooted at node  $d$ 
40     end.

```

Figure 3: Algorithm WA(q, t): progress tree traversal at processor pid

```

50 procedure DOJOB( $j$ )
51      $Local_{pid} := \emptyset$                                                %- No chunks done for the job
52      $Coop_{pid} := \{pid\}$                                                %- No other cooperating processors
53      $Pos_{pid}[1..p] := \{0\}^p$                                          %- Clear all processors' positions
54      $Chunk[pid][1..t] := (\perp, \perp)^t$                                   %- No chunks are yet done by processor  $pid$ 
55     while  $JobChunks[j] \neq Local_{pid}$  do                                %- While not all chunks in job  $j$  done
56         CHECK( $j$ )                                                       %- Check for done chunks and update the set of cooperating processors
57         if  $|Coop_{pid}| > s$  then                                         %- Has the job been saturated with processors?
58             ALLTASKS( $j$ )                                               %- The job is saturated, run cooperative algorithm
59         else
60             DOCHUNK( $j$ )                                                 %- The job is not saturated, do another chunk
61     end.

```

```

70 procedure CHECK( $j$ )
71     for all  $w \in P - \{pid\}$  while  $|Coop_{pid}| \leq s$  do                %- Check other processors, while job is not saturated
72         if  $Pos_{pid}[w] < t$  then                                         %- Is there more chunks to check?
73             ( $job, chunk$ )  $\leftarrow Chunk[w][Pos_{pid}[w] + 1]$           %- Get next chunk
74             while  $job = j \wedge Pos_{pid}[w] + 1 \leq t$  do              %- While  $w$  working on the same job
75                  $Coop_{pid} := Coop_{pid} \cup \{w\}$                     %- Record  $w$ 's  $pid$ 
76                  $Local_{pid} := Local_{pid} \cup \{chunk\}$                 %- Add chunk
77                  $Pos_{pid}[pid] := Pos_{pid}[pid] + 1$                   %- Advance own position
78                  $Chunk[pid][Pos_{pid}[pid]] \leftarrow (job, chunk)$     %- Record progress for others
79                  $Pos_{pid}[w] := Pos_{pid}[w] + 1$                       %- This chunk of  $w$  is now recorded
80                 if  $Pos_{pid}[w] + 1 \leq t$  then                          %- More chunks to check?
81                     ( $job, chunk$ )  $\leftarrow Chunk[w][Pos_{pid}[w] + 1]$   %- Get next chunk of  $w$ 
82                 if  $w \in Coop_{pid} \wedge job \neq \perp$  then                %- Processor  $w$  moved to another job
83                      $Local_{pid} := JobChunks[j]$                     %- This means all chunks are done
84     end.

```

```

90 procedure DOCHUNK( $j$ )
91      $c := BALANCE(pid, j, Coop_{pid}, Local_{pid})$ 
92     perform chunk  $c$                                                      %- Do all  $t$  tasks in chunk  $c$ 
93      $Local_{pid} := Local_{pid} \cup \{c\}$                                   %- Record the local progress
94      $Pos_{pid}[pid] := Pos_{pid}[pid] + 1$                                 %- Advance own position
95      $Chunk[pid][Pos_{pid}[pid]] \leftarrow (j, c)$                        %- Record own progress for others
96     end.

```

Figure 4: Algorithm WA(q, t): doing a job at processor pid

processors participate in working on the same job. The code is given in Figure 4.

Let s be a fixed threshold (that depends on t ; we define the value of s explicitly in Section 4). We say that a job is *saturated*, if more than s processors are working on the job, otherwise, the job is *unsaturated*. If a processor thinks that a job is unsaturated, it performs the tasks in the job, one chunk at a time. Else, when it learns that a job is saturated, it runs an instance of the algorithm of Anderson and Woll [3] for p processors and n/p tasks corresponding to the job (procedure ALLTASKS, which we discuss later).

The top level procedure DOJOB implements this process (lines 50-61). The threshold s is used to decide whether to perform procedure DOCHUNK(j) or procedure ALLTASKS(j).

Shared array $Chunk[1 .. p][1 .. t]$ has size $p \times t$ and, for every processor pid , $Chunk[pid][i]$ is a pair $(job, chunk)$, where job is the number of a job and $chunk$ is the number of a chunk. Row $Chunk[pid]$ is written only by processor pid , while it can be read by any processor (this is a property of the algorithm and not a model assumption).

While executing procedure DOJOB(j) we use the following local structures. Each processor pid maintains a local list

$Local_{pid}$ of performed chunks in its current job j . It also has local array Pos_{pid} containing job numbers. The intuition behind $Pos_{pid}[w]$ is that if processor w performs the same job as processor pid , then pid has already read all chunks stored in $Chunk[w][1 \dots Pos_{pid}[w]].chunk$. Additionally processor pid stores the subset $Coop_{pid}$ of processors, about which it has learnt that performs the same job as pid does.

When a processor pid starts on a new job j , it clears its data structures (lines 51-54), then checks for the progress of other processors on the same job by examining the list of chunks performed by other processors. This is done using the call to procedure $CHECK(j)$ in line 56. If the job is not saturated, then pid does another chunk from the job using the call to procedure $DOCHUNK(j)$ in line 60. If the job is saturated, the processor starts executing the chunks of job j cooperatively with other processors. This is done using a call to procedure $ALLTASKS(j)$ in line 58.

Procedure $ALLTASKS(j)$ is the same as the top-level algorithm of $WRITEALL(\cdot, \cdot)$. (In fact, it can be implemented as a parameterized recursive call to $WRITEALL$, but the parameterization would unnecessarily obscure the code in Figures 3 and 4). $ALLTASKS$ is a procedure specified for p processors and n/p tasks (array locations) that correspond to all tasks in job j . We use the same parameter q as the arity of the progress tree with n/p leaves. The main difference between $ALLTASKS$ and $WRITEALL$, is that in $ALLTASKS$ a processor performs one task at a leaf of the progress tree, instead of doing one job in $WRITEALL$ at line 35.

3.3 Checking progress of other processors

We now describe the procedure $CHECK(j)$ (lines 70-84). Although the code is detailed, the idea is very simple: while the job j is not saturated, the processor pid executing the procedure checks the array $Chunk[w][\cdot]$ for all other processors w and records any progress of other processors on job j . If w is also working on the same job, then pid adds it to its local variable $Coop_{pid}$ (lines 74-75). In more detail, processor pid takes the following actions for every other processor $w \neq pid$ (lines 72-81): it checks which job is being done by w by reading from $Chunk[w][Pos_{pid}[w] + 1]$ (line 73) and examining the job field (line 74). The loop in lines 74-81 records the progress on the chunks in $Local_{pid}$ (line 76) and in $Chunk[pid][Pos_{pid}[pid]]$ (line 78), and advances to the next chunk. Finally, if processor w previously worked on job j , but is found to be working on some other job, this means that all chunks in job j are complete (lines 82-83).

3.4 Doing one chunk and function $BALANCE$

When processor pid decides to perform the next chunk of the unsaturated job j (lines 57 and 60), it calls procedure $DOCHUNK$ (lines 90-96). The process chooses the next chunk c according to the load-balancing implemented by the function $BALANCE$ (line 91); we next describe this function.

Function $BALANCE(pid, j, Coop_{pid}, Local_{pid})$ returns the chunk number, say c , within the current job j . Recall that the chunks for job j are stored in $JobChunks[j]$. Processor pid chooses chunk c as follows.

- Processor pid lists the sorted chunks in $JobChunks[j] - Local_{pid}$, yielding an ordered list c_1, \dots, c_x of chunks, where $x = |JobChunks[j] - Local_{pid}|$;
- Processor pid lists the sorted processors in $Coop_{pid}$, yielding an ordered list w_1, \dots, w_y processors, where $y = |Coop_{pid}|$;

- Let z be such that $pid = w_z$, then the chosen chunk is $c := c_{\lceil zx/y \rceil}$.

We let processors keep local sets $JobChunks[j] - Local_{pid}$ and $Coop_{pid}$ sorted, which means that additions to, and deletions from, these sets can take $O(\log t)$ and $O(\log p)$ time respectively. With this, function $BALANCE$ can be done in (local) time $O(t)$, contributing $O(t)$ to the work complexity.

Having determined the next chunk c , the processor performs it (line 92), i.e., it performs all $O(n/(pt))$ tasks in the chunk. The processor then updates $Local_{pid}$, advances its position $Pos_{pid}[pid]$, and records its progress in the location $Chunk[pid][Pos_{pid}[pid]]$ (lines 93-95).

3.5 Correctness

The correctness of the algorithm $WA(q, t)$ follows from the following observations.

LEMMA 3.1. *In algorithm $WA(q, t)$, if any processor returns from procedure call to $WRITEALL$ in line 24, then each leaf of the progress tree $T[\cdot]$ is set to 1.*

PROOF. (Sketch; follows from the correctness of [3].) A processor exits a subtree by returning from a recursive call to $WRITEALL$ iff all roots of the subtrees are set to 1 (at the lowest level of the tree, the leaves are set to 1). When a processor returns from the top-level call to $WRITEALL$, all leaves are marked. \square

In procedure $WRITEALL$, a leaf of $T[\cdot]$ can be set to 1 only at line 39. This is done after a processor determines that it reached a leaf (line 34), and after the processor completes the call to $DOJOB$ for that leaf (line 35). Thus we need to show that upon the return from that $DOJOB$, all tasks in the corresponding job are done and the corresponding leaf is set to 1.

We start with preliminary lemmas describing certain properties of the algorithm.

LEMMA 3.2. *In algorithm $WA(q, t)$, if a processor pid adds a chunk to $Local_{pid}$ then this chunk is done.*

PROOF. By induction—the processor either performs the chunk by itself, or it learns about it by reading information in $Chunk[v][\cdot]$ recorded by some other processor v . The first time any processor v writes the pair (j, c) in $Chunk[v][\cdot]$ (line 95) is after it performs chunk c in job j by itself. \square

LEMMA 3.3. *In algorithm $WA(q, t)$, if any processor returns from procedure call to $DOJOB(j)$ in line 35, then all tasks in job j are done.*

PROOF. (Sketch.) A processor pid return from $DOJOB$ after it finds that its variable $Local_{pid}$ contains all chunks (line 55). If the job ever becomes saturated, then the processor calls $ALLTASKS$. When any processor returns from $ALLTASKS$, then all tasks are indeed performed (the proof of this is the same as the proof above for $WRITEALL$). (Recall that in $ALLTASKS$ writing 1 in the leaf is equivalent to performing a single task associated with this leaf.) Else the job never becomes saturated. In this case, any processor either does chunks by itself ($DOCHUNK$), or it learns that the chunks are done from other processors ($CHECK$), and it records its progress in $Local_{pid}$. By Lemma 3.2 this means that all tasks in the current job are done. In all cases, all tasks for each chunk of the job are complete. \square

LEMMA 3.4. *In algorithm $WA(q, t)$, if processor pid writes 1 into leaf corresponding to job j then all tasks in job j are complete.*

PROOF. Any leaf of $T[\cdot]$ is set to 1 only at line 39. This happens after the return from $DOJOB$ corresponding to the leaf. By Lemma 3.3, all tasks in the job are complete. \square

Next we argue termination.

LEMMA 3.5. *Any execution of algorithm $WA(q, t)$ takes finite time for at least processor.*

PROOF. (Sketch.) By the model assumptions, there is at least one processor that is not delayed forever. The progress tree used by the algorithm has finite number of nodes. By code inspection, each processor executing the algorithm makes at most one recursive $WRITEALL$ call per each node of the tree. By the same reasoning, there is a finite number of calls to $ALLTASKS$. Procedures $CHECK$ and $DOCHUNK$ clearly take finite time. Thus any call to $DOJOB$ takes finite time. Putting all of this together yields that at least one processor completes the algorithm in finite time. \square

Putting all of this together leads to the following.

THEOREM 3.6. *Algorithm $WA(q, t)$ performs all tasks in finite time.*

PROOF. By Lemma 3.5, at least one processor completes the algorithm in finite time. By Lemma 3.3, each return from $DOJOB(j)$ means that all tasks in job j are done. By Lemma 3.1, each leaf of the main progress tree is set to 1. By Lemma 3.4, setting any leaf to 1 means that the job at the leaf is done. Therefore all tasks are done. \square

4. ANALYSIS OF ALGORITHM $WA(Q, T)$

The main and the more challenging result is the analysis of work complexity. We give it next.

Let $W(p, n)$ denote the work performed by the p processors using algorithm $WA(q, t)$ to write the n locations of the *Write-All* array. We analyze complexity for threshold value $s = \sqrt[3]{t}$. We say that processor v starts job j if it makes a call to $DOJOB(j)$. We say that a job is completed if 1 is written into the corresponding leaf. Consider the leaf of the progress tree corresponding to job j . For an execution of the algorithm, let p_j denote the total number of processors that ever call $DOJOB(j)$. We start with preliminary lemmas,

LEMMA 4.1. *If processor v adds processor w to set $Coop_v$ during some step of the execution of $DOJOB(j)$ then w started job j before that step.*

PROOF. Processor v adds another processor w to $Coop_v$ only upon reading a pair (j, c) , for some c , from $Chunks[w][\cdot]$. Only processor w can write to $Chunks[w][\cdot]$. This happens when w performs chunk c of job j by itself, and only during the execution of $DOJOB(j)$ by processor w . Thus w started job j earlier. \square

Now we focus on complexity. We first state a result that follows from [3].

LEMMA 4.2. *For every constant $\xi > 0$ there exist constants $p_\infty, q \geq 0$ such that for every integer parameters*

$t > 0, n > 0$ and $p > p_\infty$:

- (a) *the total number of calls to procedure $DOJOB(\cdot)$ in algorithm $WA(q, t)$ is less than $p^{1+\xi}$,*
- (b) *if procedure $ALLTASKS(j)$, for any job j , is executed by all processors, the total work in it is $O(\frac{n}{p} \cdot p^\xi + p^{1+\xi})$.*

PROOF. The top-level structure of algorithm $WA(q, t)$ is the same as that of the q -ary progress tree algorithm of [3]. The existence of p_∞ and q such that clause (a) is satisfied follows directly from the work analysis in [3] for p processors and the progress tree with p leaves. Here the constant $q > 0$ can be chosen depending on ξ . More precisely, we can choose q such that algorithm $WA(q, t)$ makes $O(p^{1+\xi/2})$ calls to procedure $DOJOB(\cdot)$. It follows that there are positive constants p', c such that for every $p > p'$ the number of calls is at most $c \cdot p^{1+\xi/2}$. We choose the constant p_∞ such that $p_\infty^{\xi/2} > \max\{c, p'\}$. Thus for every $p > p'$ the number of calls is at most $p_\infty^{\xi/2} \cdot p^{1+\xi/2} \leq p^{1+\xi}$.

Clause (b) follows similarly. In procedure $ALLTASKS$ we have a q -ary progress tree with n/p leaves and p processors. Note that the above analysis does not depend on parameter t . \square

Fix $\varepsilon > 0$. It is sufficient to show the result for $\varepsilon \leq 1/2$. Let $\xi = \varepsilon/9$ and let $t = p^{6\xi}$. We set the threshold s to be $s = \sqrt[3]{t} = p^{2\xi}$. Let q and p_∞ be the constants satisfying Lemma 4.2 for ξ . We partition the leaves of the progress tree into two sets: A and B . Set A contains all leaves corresponding to the unsaturated jobs j such that $p_j \leq s$ ($= \sqrt[3]{t} = p^{2\xi}$). Set B contains all remaining leaves; these correspond to the saturated jobs. By Lemma 4.2 (a) and the counting argument we get that $|B| \leq p^{1-\xi}$.

We now give some helpful intuition. If a job is saturated then processors switch to procedure $ALLTASKS$ causing p^ξ work overhead, which is not too high by the property that $|B| \leq p^{1-\xi}$. If a job is unsaturated then we perform an increased number reads to gain the knowledge needed for balancing (this is amortized by a polynomial in p since additional structures have small size depending on p only). What we gain is optimal task balancing (up to a constant factor and a small additive summand)—consequently we get optimal work plus some a small polynomial in p .

The technical difficulty is to guarantee that the above properties hold in any asynchronous execution, e.g., some processors may think that job j is saturated, while others may think that it is unsaturated. We show how to overcome this in Lemma 4.4. Another difficulty is how to balance (nearly optimally) the chunks in the job which is unsaturated—we show the analysis in Lemma 4.3.

Now we resume the detailed analysis. First we establish the upper bounds on time of the procedures of $WA(q, t)$.

FUNCTION $BALANCE$ involves no shared memory operations, and takes $O(t)$ steps to select a chunk; note that the size of $Local_v$ is at most t .

PROCEDURE $DOCHUNK(j)$ takes $O(t)$ steps to execute $BALANCE$, $O(n/(pt))$ shared-memory operations to perform tasks from the chunk chosen in $BALANCE$, and $O(\log p)$ steps for update in lines 93-95. In total this is $O(n/(pt) + t)$ steps.

PROCEDURE $ALLTASKS(j)$ for the job j requires additional $O(n/p)$ shared memory for expressing tasks in job j , and to form its progress tree. This costs work $O(\frac{n}{p} \cdot p^\xi + p^{1+\xi})$ in total (as follows from Lemma 4.2(b)).

PROCEDURE CHECK requirements are as follows. The total work done by some processor v executing CHECK for a specific job j is $O(pt \log p)$: the inner while loop in CHECK may be iterated $O(t)$ times, once for each chunk in job j , and the outer for/while is executed $p - 1$ times, once for each other processor; one iteration of loop while (lines 75-81) takes $O(\log p)$ steps.

PROCEDURE DOJOB(j) takes work as follows. Initialization of Pos_v takes $O(p)$ steps per processor v , and initialization of $Chunk[v]$ takes $O(t) \subseteq O(p)$ steps per processor v . Checking the while guard in line 55 takes $O(\log t)$ steps per each call to DOCHUNK inside the loop (line 60). Inside the while loop, either ALLTASKS(j) is called — at most once per job j for each processor — with total work $O(\frac{n}{p} \cdot p^\xi + p^{1+\xi})$, or CHECK(j) and DOCHUNK(j) are executed in the loop until all chunks are done. (Note that each processor but one might run one more iteration of the loop, until it checks by itself, during its call to CHECK(j), that all chunks in job j are done.)

The next lemmas gives work in all executions of DOJOB(j) when j is unsaturated and when j is saturated.

LEMMA 4.3. *For an unsaturated job, $j \in A$, the total number of steps spent by processors executing DOJOB(j) is $O(\frac{n}{p} + pts \log p) = O(\frac{n}{p} + p^{1+9\xi})$.*

PROOF. We consider only such p that $p > p_\infty$, where p_∞ is taken from Lemma 4.2. Additionally, assume that $s > 9$ (if not, then $p = s^{1/(2\xi)}$ is a constant and asymptotic optimality is obvious; in fact we can alternatively assume that p_∞ satisfies the additional condition that $p_\infty^{2\xi} > 9$). Recall that job j has $O(n/p)$ individual tasks, and on the other hand the number of chunks is $t = p^{6\xi}$, each of $O(n/(pt)) = O(n/p^{1+6\xi})$ tasks.

The idea of the proof is as follows. We partition the corresponding the execution of job j into stages. Each stage consists of steps of some processor in performing a single iteration of the while loop in lines 56-60. Then we select intervals of stages such that each interval satisfies the following conditions: (i) there is no “big gap” between two consecutive stages of one processor performing job j (we later denote the set of such stages by D_2), and (ii) there are no new processors joining the work on job j (we later denote the set of such stages by D_3 , where $D_3 \subseteq D_2$). Then will we argue that the total number of stages during such intervals is large enough, and this allows us to analyze the performance of chunks only for these stages (see Claim 1 and the comments following the claim). Finally, we argue that during such stages processors choose different chunks to perform (see Claim 2). Thus the work is split optimally during sufficiently many stages.

Now we start the detailed analysis. We call a *stage of processor v* the sequence of consecutive steps performed by v from the end of line 60 of the code to the next end of line 60 of the code (one iteration of the while loop in lines 56-60 of Figure 4). For each processors v , consider its stages starting from the stage in which it starts performing job j , and ending with the stage in which it completes job j . Denote the set of all such stages, over all processors, by D . We consider the linear order of the stages in D , according the ends of the stages in the execution order.

Let T denote the upper bound on the number of local steps taken by a processor in one stage, but without counting the steps taken in executing lines 74-81 (iterations of the

while loop). Bound T holds for every job j . We claim that $T = O(n/(pt) + t + p \log p) = O(n/p^{1+6\xi} + p \log p)$. This is obtained by adding the number of steps needed to run one instance of procedure DOCHUNK(j) and one instance of procedure CHECK(j) (but subtracting the number of steps performed during the execution the the inner loop in lines 74-81). Broken down, these are $O(n/(pt) + t)$ plus $O(p \log p)$ steps. To ensure that we do not underestimate any calculation, we add the total cost of performing lines 74-81 during executions of procedure CHECK(j) in the final calculation of work: this is $O(pt \log p)$ steps per each processor performing job j , which gives total work $O(p_j \cdot pt \log p) \subseteq O(pts \log p)$.

If $|D| \leq 9t = 9p^{6\xi}$ then we are done. In this case the total number of steps during the execution of procedure DOJOB is at most $9t \cdot T + O(pts \log p) = O(\frac{n}{p} + pts \log p)$ (by the definition of T , the assumption on the size of D , and the total cost of performing lines 74-81).

Suppose to the contrary that $|D| > 9t = 9p^{6\xi}$. Consider the first $2t = 2 \cdot p^{6\xi}$ stages in D , denote this set of stages (linearly ordered by their ends) by D_1 . Let P'_j be the set of processors with stages in D_1 . Let $p'_j = |P'_j|$. Obviously $p'_j \leq p_j \leq s = \sqrt[3]{t} = p^{2\xi}$.

Let D_2 contain all stages k in D_1 such that:

If k is performed by processor v from P'_j then between the end of the previous stage performed by v in D_1 and the end of stage k there are at most $2\sqrt[3]{t} = 2p^{2\xi}$ other stages from D_1 according to the linear order.

Note that $D_2 \subseteq D_1 \subseteq D$. First we prove that D_2 is sufficiently large.

CLAIM 1. $|D_2| \geq t = p^{6\xi}$.

PROOF. Partition the linearly ordered set D_1 into consecutive segments, each of size $2\sqrt[3]{t} = 2p^{2\xi}$. Consider such a segment. Suppose we do not consider any stage k in the segment such that k is the first stage performed by some processor v in the segment — this would correspond to stages started before the segment, and we would have no control on the length of this stage. Still there remain at least $\sqrt[3]{t} = p^{2\xi}$ stages considered in the segment. All of them are in D_2 . Thus the total number of stages in D_2 is at least

$$\frac{|D_1|}{2\sqrt[3]{t}} \cdot \sqrt[3]{t} = |D_1|/2 = t = p^{6\xi},$$

which completes the proof of the claim. \square

Now let D_3 contain every stage $k \in D_2$ such that:

No new processor starts performing job j during stage k .

By the definition of set D_2 , if a processor starts performing job j inside some stage $k \in D_2$, this event is in at most $2\sqrt[3]{t} = 2p^{2\xi}$ stages from D_1 . Since the total number of processors performing job j is at most $s = \sqrt[3]{t} = p^{2\xi}$ (by the definition of set A), the total number of stages in D_2 but not in D_3 is at most

$$\sqrt[3]{t} \cdot 2\sqrt[3]{t} = 2\sqrt[3]{t^2} = 2p^{4\xi}.$$

Hence, in view of Claim 1, $|D_3| \geq t - 2\sqrt[3]{t^2} = p^{6\xi} - 2p^{4\xi}$.

Consider two stages k_v, k_w in D_3 . Suppose that they are performed by processors v and w respectively, and that stage k_v precedes stage k_w . In the next claim we prove that, under some conditions, chunks performed during stages k_v, k_w are

different, and to show this we use the crucial properties of function BALANCE that locally computes the chunks for the executing processor to do (using only the local knowledge of this processor).

CLAIM 2. *If there are more than $5\sqrt[3]{t^2} = 5p^{4\xi}$ unperformed chunks at the end of stage k_w , then the chunk performed during stage k_w by processor v is different from the chunk performed in stage k_w by processor w .*

(Proof of Claim 2 is omitted.)

CLAIM 3. *There exists a stage k^* in D_3 such that at the end of stage k^* there are at most $5\sqrt[3]{t^2} = 5p^{4\xi}$ unperformed chunks in job i .*

PROOF. Suppose to the contrary that there is no such stage. It follows that in each stage of D_3 the number of unperformed chunks is greater than $5\sqrt[3]{t^2} = 5p^{4\xi}$. Then, by Claim 2, every chunk performed during stage D_3 is different. Size of D_3 is at least $t - 2\sqrt[3]{t^2} = p^{6\xi} - 2p^{4\xi}$, the number of performed chunks during stages in D_3 is at least $|D_3|$, that is at least $t - 2\sqrt[3]{t^2} = p^{6\xi} - 2p^{4\xi}$. It follows that there are at most

$$t - (t - 2\sqrt[3]{t^2}) = 2\sqrt[3]{t^2} = 2p^{4\xi}$$

unperformed chunks at the end of the last stage in D_3 , which is a contradiction. Hence the claim is proved. \square

Now we conclude the proof. Let k^* be taken from Claim 3. Recall that D contains more than $9t$ stages. Since D_1 contains the first $2t$ stages from D and k^* is in $D_3 \subseteq D_1$, there are more than $7t = 7p^{6\xi}$ stages in D that end after the end of stage k^* . Consider all stages in D that start after the end of stage k^* . Call this set D' . There are more than $7t - s = 6t = 6p^{6\xi}$ of such stages, since for every processor performing job j (there are at most s of them) we count all stages in $D \setminus D_1$ except possibly the first (by definition, it ends after stage k^* ends, but may start earlier).

Every processor performing job j executes at most $5\sqrt[3]{t^2} = 5p^{4\xi}$ stages in D' , since, by definition of D' , in each stage k during CHECK processor learns that at most $5\sqrt[3]{t^2} = 5p^{4\xi}$ chunks are unperformed, and in each stage it never performs chunks that it knows to be complete. Thus the number of stages in D' must be at most $s \cdot 5\sqrt[3]{t^2} = 5p^{6\xi} = 5t$, which is a contradiction. This concludes the proof of the lemma.

LEMMA 4.4. *For a saturated job, $j \in B$, the total number of steps spent by processors executing DOJOB(j) is $O(\frac{n}{p} \cdot p^\xi + p^{1+9\xi} + p_j \cdot p \log p)$.*

PROOF. We split steps during performing of job j into three parts.

Part 1. From the first time when some processor starts job j to the first time when at least $s + 1$ processors v has written pair (j, c) , for some chunk c of job j , in $Chunk[v][\cdot]$. It means that none processor v realizes that $p_j > s = \sqrt[3]{t} = p^{2\xi}$ in considered period. Thus the total work performed before this event by all processors performing job j is, by Lemma 4.3, $O(\frac{n}{p} + pts \log p) = O(\frac{n}{p} + p^{1+9\xi})$.

Part 2. Work perform only during executions of CHECK(j) and DOCHUNK(j), after the first time when at least $s + 1$ processors v has written pair (j, c) , for some chunk c of job

j , in $Chunk[v][\cdot]$. If processor v calls a procedure CHECK(j) after the events analyzed in (1), it finds the job j to be saturated and switches to cooperative work using procedure ALLTASKS(j) following this execution of CHECK(j). Thus it is sufficient to estimate only the cost of such execution CHECK(j). More precisely, there are at most p_j processors v which may enter the inner while loop in considered execution of CHECK(j), and the work done is $O(st)$ by each of such processors v , since the number of entrances is bounded by s by condition in line 71. This gives total work $O(p_j \cdot st)$. On the other hand, each of processors performing job j additionally may perform $O(\log p)$ steps in lines 71-73, and there are at most $p - 1$ of such iterations inside loop for (line 71). This gives additional work $O(p_j \cdot p \log p)$. Hence the total work considered in Part 2 for job j is $O(p_j \cdot st + p_j \cdot p \log p) = O(p_j \cdot p \log p)$.

Part 3. Now we estimate the total work spent by all processors doing job j according to procedure ALLTASKS(j). There are at most $p_j \leq p$ processors participating in this, so the work is $O(\frac{n}{p} \cdot p^\xi + p^{1+9\xi})$, by Lemma 4.2 (b).

Summing the three part considered above yields the thesis of the lemma. \square

THEOREM 4.5. *For every constant $\varepsilon > 0$, there are constants q, t such that p -processor algorithm WA(q, t) performs work $O(n + p^{2+\varepsilon})$, where n is the input size.*

PROOF. By Lemma 4.3 we get that work spent on each job in A is $O(\frac{n}{p} + p^{1+9\xi})$, where $|A| \leq p$. By Lemma 4.4 we get that work spent on each job in B is $O(\frac{n}{p} \cdot p^\xi + p^{1+9\xi} + p_j \cdot p \log p)$. Recall that $|B| \leq p^{1-\xi}$ from the conclusion of Lemma 4.2 and the counting argument. Using Lemma 4.2 (a), we also obtain that $\sum_{j \in B} p_j \leq p^{1+\xi}$. Putting all of the above together with $\varepsilon = 9\xi$, we obtain that the total work $W(p, n)$ is at most

$$\begin{aligned} & O\left(\frac{n}{p} + p^{1+9\xi}\right) \cdot p + \sum_{j \in B} O\left(\frac{n}{p} \cdot p^\xi + p^{1+9\xi} + p_j \cdot p \log p\right) \\ &= O(n + p^{2+9\xi}) + |B| \cdot O\left(\frac{n}{p} \cdot p^\xi + p^{1+9\xi}\right) + \sum_{j \in B} O(p_j \cdot p \log p) \\ &= O(n + p^{2+\varepsilon}). \end{aligned}$$

\square

As an immediate corollary, we have that our algorithm is optimal when $p^{2+\varepsilon} \leq n$. This is by far the strongest optimality result to date.

COROLLARY 4.6. *For every constant $\varepsilon > 0$, there are constants q, t such that p -processor algorithm WA(q, t) performs optimal work $O(n)$ when $p^{2+\varepsilon} \leq n$, where n is the input size.*

To conclude the analysis, we assess the memory requirements of our algorithm for the case when optimal work is achieved. It is sufficient to assume that that $p^2 \leq n$. Array $Chunk[1..p][1..t]$ takes $O(pt) = O(n)$ of shared space. Shared (constant) array $JobChunks[1..p]$ is used for convenience only — it represents the static partitioning of jobs into chunks. Private $Local_{pid}$ for p processors takes $O(p \cdot t) = O(n)$. Private $Pos_{pid}[0..p - 1]$ for p processors takes $O(p \cdot p) = O(n)$. Private sets $Coop_{pid}$ for p processors take $O(p \cdot p) = O(n)$. Finally, for each job, procedure ALLTASKS uses a progress tree of n/p leaves. Since there are p jobs, this takes $O(n)$ space. Thus the overall space used by the algorithm is $O(n)$.

5. DISCUSSION

In this paper we considered the problem of performing n tasks on p asynchronous processors in a shared-memory model of computation. Here a task can be any local computation that can be performed by a processor in constant time. We abstract each task in terms of writing to a shared memory location. We showed that an optimal upper bound on work of $O(n)$ can be obtained by a *deterministic* algorithm for a non-trivial number of processors. Our result almost *squares* the previously known number of processors for which optimality can be achieved.

There are two main challenges that we intend to address in future work. The first is increasing the range of optimality beyond $p = O(n^{1/(2+\epsilon)})$. The second challenge is to be able to efficiently construct permutations with the contention properties required by our algorithms. Such permutations can be identified using exhaustive search, but even though q is constant, this expense can be large. (Our algorithm can be used with random permutations, in which case the work upper bound becomes expected.) Various approaches can be used to construct sets of permutations with low contention (cf. [3, 25]). We are continuing to explore ways of constructing required permutations based on multiplicative groups [7].

6. REFERENCES

- [1] Ajtai, M., Aspnes, J., Dwork, C., Waarts, O.: A theory of competitive analysis of distributed algorithms. Proc. of the 35th IEEE Symp. of Found. of Computer Science, FOCS'1994, 401–411
- [2] Alon, N., Spencer, J.: The probabilistic method. John Wiley, Second Edition (2000)
- [3] Anderson, R.J., Woll, H.: Algorithms for the certified Write-All problem. SIAM Journal on Computing **26** (5) (1997) 1277–1283
- [4] Aumann, Y., Rabin, M.O.: Clock construction in fully asynchronous parallel systems and PRAM simulation. Proc. of the 33rd IEEE Symp. on Foundations of Computer Science, FOCS'1992, 147–156
- [5] Aumann, Y., Kedem, Z.M., Palem, K.V., Rabin, M.O.: Highly efficient asynchronous execution of large-grained parallel programs. Proc. of the 34th IEEE Symp. on Foundations of Computer Science, FOCS'1993, 271–280
- [6] Buss, J., Kanellakis, P.C., Ragde, P.L., Shvartsman, A.A.: Parallel algorithms with processor failures and delays. J. of Algorithms **20** (1996) 45–86
- [7] Chlebus, B., Dobrev, S., Kowalski, D., Malewicz, G., Shvartsman, A., Vrto, I.: Towards practical deterministic Write-All algorithms. Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures, SPAA'2001, 271–280
- [8] R. Cole and O. Zajicek: The expected advantage of asynchrony. Proc. of the 2nd ACM Symp. on Parallel Algorithms and Architectures, SPAA'1990, 85–94
- [9] Dasgupta, P., Kedem, Z., Rabin, M.: Parallel processing on networks of workstation: A fault-tolerant, high performance approach. Proc. of the 15th International Conference on Distributed Computing Systems, ICDCS'1995, 467–474
- [10] De Prisco, R., Mayer, A., Yung, M.: Time-optimal message-efficient work performance in the presence of faults. Proc. of the 13th ACM Symp. on Principles of Distributed Computing, PODC'1994, 161–172
- [11] Dwork, C., Halpern, J., Waarts, O.: Performing work efficiently in the presence of faults. SIAM J. on Computing **27** (1998) 1457–1491
- [12] Galil, Z., Mayer, A., Yung, M.: Resolving message complexity of byzantine agreement and beyond. Proc. of the 36th IEEE Symp. on Foundations of Computer Science, FOCS'1995, 724–733
- [13] Gibbons, P.: A more practical PRAM model. Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures, SPAA'1989, 158–168
- [14] Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. Distributed Computing (2001) 14:75–81
- [15] Kanellakis, P.C., Shvartsman, A.A.: Efficient parallel algorithms can be made robust. Distributed Computing **5** (1992) 201–217
- [16] Kanellakis, P.C., Shvartsman, A.A.: Fault-tolerant parallel computation. Kluwer Academic Publishers (1997)
- [17] Kedem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient program transformations for resilient parallel computation via randomization. Proc. of the 24th ACM Symp. on Theory of Computing, STOC'1992, 306–318
- [18] Kedem, Z.M., Palem, K.V., Raghunathan, A., Spirakis, P.: Combining tentative and definite executions for dependable parallel computing. Proc. of the 23rd ACM Symposium on Theory of Computing, STOC'1991, 381–390
- [19] Kedem, Z.M., Palem, K.V., Spirakis, P.: Efficient robust parallel computations. Proc. of the 22nd ACM Symp. on Theory of Computing, STOC'1990, 138–148
- [20] Knuth, D.E.: The art of computer programming Vol. 3 (third edition). Addison-Wesley Pub Co. (1998)
- [21] Malewicz, G.: A work-optimal deterministic algorithm for the asynchronous Certified Write-All problem. Proc. of the 22nd ACM Symposium on Principles of Distributed Computing, PODC'2003, 255–264
- [22] Martel, C., Park, A., Subramonian, R.: Work-optimal asynchronous algorithms for shared memory parallel computers. SIAM J. on Comp. **21** (1992) 1070–1099
- [23] Martel, C., Subramonian, R.: On the complexity of Certified Write-All algorithms. J. Algorithms **16** (3) (1994) 361–387
- [24] Martel, C., Subramonian, R., Park, A.: Asynchronous PRAMs are (almost) as good as synchronous PRAMs. Proc. of the 32d IEEE Symp. on Foundations of Computer Science, FOCS'1990, 590–599
- [25] Naor, J., Roth, R.M.: Constructions of permutation arrays for certain scheduling cost measures. Random Structures and Algorithms **6** (1) (1995) 39–50
- [26] Nishimura, N.: A model for asynchronous shared memory parallel computation, SIAM Journal on Computing **23** (6) (1994) 1231–1252
- [27] Shvartsman, A.A.: Achieving optimal CRCW PRAM fault-tolerance. Information Processing Letters, **39** (2) (1991) 59–66