A Method for Creating Near-Optimal Instances of a Certified Write-All Algorithm
(Extended Abstract)

Grzegorz Malewicz*

Laboratory for Computer Science, Massachusetts Institute of Technology
200 Technology Square, NE43-205, Cambridge, MA 02139
malewicz@theory.lcs.mit.edu

Abstract. This paper shows how to create near-optimal instances of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll [2]. This algorithm is the best known deterministic algorithm that can be used to simulate n synchronous parallel processors on n asynchronous processors. In this algorithm n processors update n memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations, where q can be chosen from a wide range of values. When implementing a simulation on a specific parallel system with n processors, one would like to use an instance of the algorithm with the best possible value of q, in order to maximize the efficiency of the simulation. This paper shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work. For any $\epsilon > 0$, and any large enough n, work of any instance of the algorithm must be at least $n^{1+(1-\epsilon)\sqrt{2 \ln \ln n / \ln n}}$. Under certain conditions, however, that q is about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ and for infinitely many large enough n, this lower bound can be nearly attained by instances of the algorithm with work at most $n^{1+(1+\epsilon)\sqrt{2 \ln \ln n / \ln n}}$. The paper also shows a penalty for not selecting q well. When q is significantly away from $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then work of any instance of the algorithm with this displaced q must be considerably higher than otherwise.

1 Introduction

This paper shows how to create near-optimal instances of the Certified Write-All algorithm called AWT that was introduced by Anderson and Woll [2]. In this algorithm n processors update n memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations, where q can

* The work of Grzegorz Malewicz was done during a visit to the Supercomputing Technologies Group ("the Cilk Group"), Massachusetts Institute of Technology, headed by Prof. Charles E. Leiserson. Grzegorz Malewicz was visiting this group during the 2002/2003 academic year while in his final year of the Ph.D. program at the University of Connecticut, where his advisor is Prof. Alex Shvartsman. Part of this work was supported by the Singapore/MIT Alliance.

G. Di Battista and U. Zwick (Eds.): ESA 2003, LNCS 2832, pp. 422–433, 2003.

be chosen from a wide range of values. This paper shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work.

Many existing parallel systems are asynchronous. However, writing correct parallel programs on an asynchronous shared memory system is often difficult, for example because of data races, which are difficult to detect in general. When the instructions of a parallel program are written with the intention of being executed on a system that is synchronous, then it is easier for a programmer to write correct programs, because it is easier to reason about synchronous parallel programs than asynchronous ones. Therefore, in order to improve productivity in parallel computing, one could offer programmers illusion that their programs run on a parallel system that is synchronous, while in fact the programs would be simulated on an asynchronous system.

Simulations of a parallel system that is synchronous on a system that is asynchronous have been studied for over a decade now (see e.g., [8,9]). Simplifying considerably, simulations assume that there is a system with p asynchronous processors, and the system is to simulate a program written for n synchronous processors. The simulations use three main ideas: idempotence, load balancing, and synchronization. Specifically, the execution of the program is divided into a sequence of phases. A phase executes an instruction of each of the n synchronous programs. The simulation executes a phase in two stages: first the n instructions are executed and the results are saved to a scratch memory, only then cells of the scratch memory are copied back to desired cells of the main memory. This ensures that the result of the phase is the same even if multiple processors execute the same instruction in a phase, which may happen due to asynchrony. The p processors run a load balancing algorithm to ensure that the n instructions of the phase are executed quickly despite possibly varying speeds of the p processors. In addition, the p processors should be synchronized at every stage, so as to ensure that the simulated program proceeds in lock-step.

One challenge in realizing the simulations is the problem of "late writers" i.e., when a slow processor clobbers the memory of a simulation with a value from an old phase. This problem has been addressed in various ways (see e.g., [3,13]). Another challenge is the development of efficient load-balancing and synchronization algorithms. This challenge is abstracted as the Certified Write-All (CWA) problem. In this problem, introduced in a slightly different form by Kanellakis and Shvartsman [7], there are p processors, an array w with n cells and a flag f, all initially 0, and the processors must set the n cells of w to 1, and then set f to 1. A simulation uses an algorithm that solves the CWA problem, and the overhead of the simulation depends on efficiency of the algorithm. The efficiency of the algorithm is measured by *work* that is equal to the worst-case total number of instructions executed by the algorithm. Hence it is desirable to develop low-work algorithms that solve the CWA problem.

Deterministic algorithms that solve the CWA problem on an asynchronous system can be used to create simulations that have bounded worst-case overhead. Thus several deterministic algorithms have been studied [2,4,5,6,8,14]. The class of algorithms for the case when $p = n$ is especially interesting because they have

high parallelism. When such algorithm is used in a simulation, the simulation of a given synchronous program for $p = n$ processors may be faster, as compared to the simulation that uses an algorithm for $p \ll n$ processors, simply because in the former case more processors are available to simulate the program. However, the potential of producing a faster simulation can only be realized when the algorithm used has low work, so that not much computing resources are wasted during any simulation phase.

The best deterministic algorithm that solves the CWA problem on an asynchronous system for the case when $p = n$ was introduced by Anderson and Woll [2]. This algorithm is called AWT, and it generalizes the algorithm X of Buss et al. [4]. The AWT algorithm is instantiated with a list of q permutations on $\{1, \ldots, q\}$. Anderson and Woll showed that for any $\epsilon > 0$, there is q, a list of q permutations with desired *contention*, and a constant c_q, such that for any $h > 0$, the algorithm for $p = q^h$ processors and $n = p$ cells instantiated with the list, has work at most $c_q \cdot n^{1+\epsilon}$. Note that this upper bound includes a multiplicative constant factor that is a function of q. While the result that an $O(n^{1+\epsilon})$ work algorithm can be found is very interesting, a different search objective will occur when a simulation is developed for a specific parallel system.

A specific parallel system will have a fixed number p of processors. It is possible to create many instances of the AWT algorithm for these p processors and $n = p$ cells, that differ by the number q of permutations used to create an instance. It is possible that work of these different instances is different. If this is indeed the case, then it is interesting to find an instance with the lowest work, so as to create a relatively more efficient simulation on this parallel system.

Contributions. This paper shows how to create near-optimal instances of the AWT algorithm of Anderson and Woll. In this algorithm p processors update $n = p$ memory cells and then signal the completion of the updates. The algorithm is instantiated with q permutations on $\{1, \ldots, q\}$, where q can be chosen from a wide range of values. This paper shows that the choice of q is critical for obtaining an instance of the AWT algorithm with near-optimal work. Specifically, we show a tight (up to an absolute constant) lower bound on work of the AWT algorithm instantiated with a list of q permutations (appearing in Lemma 4). This lower bound generalizes the Lemma 5.20 of Anderson and Woll by exposing a constant that depends on q and on the contention of the list. We then combine our lower bound with a lower bound on contention of permutations given by Lovász [11] and Knuth [10], to show that for any $\epsilon > 0$, work of any instance must be at least $n^{1+(1-\epsilon)\sqrt{2 \ln \ln n / \ln n}}$, for any large enough n (appearing in Theorem 1). The resulting bound is nearly optimal, as demonstrated by our method for creating instances of the AWT algorithm. We show that for any $\epsilon > 0$ and for any m that is large enough, when $q = \lceil e^{\sqrt{1/2 \ln m \ln \ln m}} \rceil$, and $h = \lceil \sqrt{2 \ln m / \ln \ln m} \rceil$, then there exists an instance of the AWT algorithm for $p = q^h$ processors and $n = p$ cells that has work at most $n^{1+(1+\epsilon)\sqrt{2 \ln \ln n / \ln n}}$ (appearing in Theorem 2). We also prove that there is a penalty if one selects a q that is too far away from $e^{\sqrt{1/2 \ln n \ln \ln n}}$. For any fixed $r \geq 2$, and any large enough n, work is at

least $n^{1+r/3\cdot\sqrt{2\ln\ln n/\ln n}}$, whenever the AWT algorithm is instantiated with q permutations, such that $16 \leq q \leq e^{\sqrt{1/2\ln n\ln\ln n}/(r\cdot\ln\ln n)}$ or $e^{r\cdot\sqrt{1/2\ln n\ln\ln n}} \leq q \leq n$ (appearing in Preposition 1).

Paper organization. The reminder of the paper is organized as follows. In Section 2, we report on some existing results on contention of permutations and present the AWT algorithm of Anderson and Woll. In Section 3, we show our optimization argument that leads to the development of a method for creating near-optimal instances of the AWT algorithm. Finally, in Section 4, we conclude with future work. Due to lack of space some proofs were omitted, and they will appear the upcoming doctoral dissertation of the author.

2 Preliminaries

For a permutation ρ on $[q] = \{1, \ldots, q\}$, $\rho(v)$ is a *left-to-right maximum* [10] if it is larger than all of its predecessors i.e., $\rho(v) > \rho(1), \rho(v) > \rho(2), \ldots, \rho(v) > \rho(v-1)$. The *contention* [2] of ρ with respect to a permutation α on $[q]$, denoted as $Cont(\rho, \alpha)$, is defined as the number of left-to-right maxima in the permutation $\alpha^{-1}\rho$ that is a composition of α^{-1} with ρ. For a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$ and a permutation α on $[q]$, the contention of R_q with respect to α is defined as $Cont(R_q, \alpha) = \sum_{v=1}^{q} Cont(\rho_v, \alpha)$. The contention of the list of permutations R_q is defined as $Cont(R_q) = max_{\alpha\ on\ [q]} Cont(R_q, \alpha)$.

Lovász [11] and Knuth [10] showed that the expectation of the number of left-to-right maxima in a random permutation on $[q]$ is H_q (H_q is the qth harmonic number). This immediately implies the following lower bound on contention of a list of q permutations on $[q]$.

Lemma 1. *[11,10] For any list R_q of q permutations on $[q]$, $Cont(R_q) \geq qH_q > q\ln q$.*

Anderson and Woll [2] showed that for any q there is a list of q permutations with contention $3qH_q$. Since $H_q/\ln q$ tends to 1, as q tends to infinity, the following lemma holds.

Lemma 2. *[2] For any q that is large enough, there exists a list of q permutations on $[q]$ with contention at most $4 \cdot q\ln q$.*

We describe the algorithm AWT of Anderson and Woll [2] that solves the CWA problem when $p = n$. There are $p = q^h$ processors, $h \geq 1$, and the array w has $n = p$ cells. The identifier of a processor is represented by a distinct string of length h over the alphabet $[q]$. The algorithm is instantiated with a list of q permutations $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ on $[q]$, and we write AWT(R_q) when we refer to the instance of algorithm AWT for a given list of permutations R_q. This list is available to every processor (in its local memory). Processors have access to a shared q-ary tree called *progress tree*. Each node of the tree is labeled with a string over alphabet $[q]$. Specifically, a string $s \in [q]^*$ that labels a node identifies the path from the root to the node (e.g., the root is labeled with the

```
AWT(R_q)
01    Traverse(h, λ)
02    set f to 1 and Halt

Traverse(i, s)
01    if i = 0 then
02        w[val(s)] := 1
03    else
04        j := q_i
05        for v := 1 to q
06            a := ρ_j(v)
07            if b_{s·a} = 0 then
08                Traverse(i − 1, s · a)
09                b_{s·a} := 1
```

Fig. 1. The instance AWT(R_q) of an algorithm of Anderson and Woll, as executed by a processor with identifier $\langle q_1 \dots q_h \rangle$. The algorithm uses a list of q permutations $R_q = \langle \rho_1, \dots, \rho_q \rangle$.

empty string λ, the leftmost child of the root is labeled with the string 1). For convenience, we say node s, when we mean the node labeled with a string s. Each node s of the tree, apart from the root, contains a *completion bit*, denoted by b_s, initially set to 0. Any leaf node s is canonically assigned a distinct number $val(s) \in \{0, \dots, n-1\}$.

The algorithm, shown in Figure 1, starts by each processor calling procedure AWT(R_q). Each processor traverses the q-ary progress tree by calling a recursive procedure $Traverse(h, \lambda)$. When a processor visits a node that is the root of a subtree of height i (the root of the progress tree has height h) the processor takes the ith letter j of its identifier (line 04) and attempts to visit the children in the order established by the permutation ρ_j. The visit to a child $a \in [q]$ *succeeds* only if the completion bit $b_{s \cdot a}$ for this child is still 0 at the time of the attempt (line 07). In such case, the processor recursively traverses the child subtree (line 08), and later sets to one the completion bit of the child node (line 09). When a processor visits a leaf s, the processor performs an assignment of 1 to the cell $val(s)$ of the array w. After a processor has finished the recursive traversal of the progress tree, the processor sets f to 1 and halts.

We give a technical lemma that will be used to solve a recursive equation in the following section.

Lemma 3. *Let h and q be integers, $h \geq 1$, $q \geq 2$, and $k_1 + \ldots + k_q = c > 0$. Consider a recursive equation $W(0, r) = r$, and $W(i, r) = r \cdot q + \sum_{v=1}^{q} W(i - 1, k_v \cdot r/q)$, when $i > 0$. Then for any r,*

$$W(h, r) = r \left(q \cdot \frac{(c/q)^h - 1}{c/q - 1} + (c/q)^h \right) .$$

3 Near-Optimal Instances of AWT

This section presents a method for creating near-optimal instances of the AWT algorithm of Anderson and Woll. The main idea of this section is that for fixed number p of processors and $n = p$ cells of the array w, work of an instance

of the AWT algorithm depends on the number of permutations used by the instance, along with their contention. This observation has several consequences. It turns out (not surprisingly) that work increases when contention increases, and conversely it becomes the lowest when contention is the lowest. Here a lower bound on contention of permutations given by Lovász [11] and Knuth [10] is very useful, because we can bound work of any instance from below, by an expression in which the value of contention of the list used in the instance is replaced with the value of the lower bound on contention. Then we study how the resulting lower bound on work depends on the number q of permutations on $[q]$ used by the instance. It turns out that there is a single value for q, where the bound attains the global minimum. Consequently, we obtain a lower bound on work that, for fixed n, is independent of both the number of permutations used and their contention. Our bound is near-optimal. We show that if we instantiate the AWT algorithm with about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ permutations that have small enough contention, then work of the instance nearly matches the lower bound. Such permutations exist as shown by Anderson and Woll [2]. We also show that when we instantiate the AWT algorithm with much fewer or much more permutations, then work of the instance must be significantly greater than the work that can be achieved. Details of the overview follow.

We will present a tight bound on work of any instance of the AWT algorithm. Our lower bound generalizes the Lemma 5.20 of Anderson and Woll [1]. The bound has an explicit constant which was hidden in the analysis given in the Lemma 5.20. The constant will play a paramount role in the analysis presented in the reminder of the section.

Lemma 4. *Work W of the AWT algorithm for $p = q^h$ processors, $h \geq 1$, $q \geq 2$, and $n = p$ cells, instantiated with a list $R_q = \langle \rho_1, \ldots, \rho_q \rangle$ of q permutations on $[q]$, is bounded by* $\frac{c}{84} \cdot n^{1+\log_q \frac{Cont(R_q)}{q}} \leq W \leq c \cdot n^{1+\log_q \frac{Cont(R_q)}{q}}$, *where* $c = \frac{28q^2}{Cont(R_q)}$.

Proof. The idea of the lemma is to carefully account for work spent on traversing the progress tree, and spent on writing to the array w. The lower bound will be shown by designing an execution during which the processors will traverse the progress tree in a specific, regular manner. This regularity will allow us to conveniently bound from below work inside a subtree, by work done at the root of the subtree and work done by quite large number of processors that traverse the child subtrees in a regular manner. A similar recursive argument will be used to derive the upper bound.

Consider any execution of the algorithm. We say that the execution is *regular at a node s* (recall that s is a string from $[q]^*$) iff the following three conditions hold:

(i) the r processors that ever visit the node during the execution, visit the node at the same time,

(ii) at that time, the completion bit of any node of the subtree of height i rooted at the node s is equal to 0,

(iii) if a processor visits the node s, and x is the suffix of length $h - i$ of the identifier of the processor, then the q^i processors that have x as a suffix of their identifiers, also visit the node during the execution.

We define $W(i, r)$ to be the largest number of basic actions that r processors perform inside a subtree of height i, from the moment when they visit a node s that is the root of the subtree until the moment when each of the visitors finishes traversing the subtree, maximized across the executions that are regular at s and during which exactly r processors visit s (if there is no such execution, we put $-\infty$). Note that the value of $W(i, r)$ is well-defined, as it is independent of the choice of a subtree of height i (any pattern of traversals that maximizes the number of basic actions performed inside a subtree, can be applied to any other subtree of the same height), and of the choice of the r visitors (suffixes of length $h - i$ do not affect traversal within the subtree). There exists an execution that is regular at the root of the progress tree, and so the value of $W(h, n)$ bounds work of $\mathrm{AWT}(R_q)$ from below.

We will show a recursive formula that bounds $W(i, r)$ from below. We do it by designing an execution recursively. The execution will be regular at every node of the progress tree. We start by letting the q^h processors visit the root at the same time. For the recursive step, assume that the execution is regular at a node s that is the root of a subtree of height i, and that exactly r processors visit the node. We first consider the case when s is an internal node i.e., when $i > 0$. Based on the i-th letter of its identifier, each processor picks a permutation that gives the order in which completion bits of the child nodes will be read by the processor. Due to regularity, the r processors can be partitioned into q collections of equal cardinality, such that for any collection j, each processor in the collection checks the completion bits in the order given by ρ_j. Let for any collection, the processors in the collection check the bits of the children of the node in lock step (the collection behaves as a single "virtual" processor). Then, by Lemma 2.1 of Anderson and Woll [2], there is a pattern of delays so that every processor in some $k_v \geq 1$ collections succeeds in visiting the child $s \cdot v$ of the node at the same time. Thus the execution is regular at any child node. The lemma also guarantees that $k_1 + \ldots + k_q = Cont(R_q)$, and that these $k_1, \ldots, k_q$ do not depend on the choice of the node s. Since each processor checks q completion bits of the q children of the node, the processor executes at least q basic actions while traversing the node. Therefore, $W(i, r) \geq rq + \sum_{v=1}^{q} W(i - 1, k_v \cdot r/q)$, for $i > 0$. Finally, suppose that s is a leaf i.e., that $i = 0$. Then we let the r processors work in lock step, and so $W(0, r) \geq r$.

We can bound the value of $W(h, n)$ using Lemma 3, the fact that $h = \log_q n$, and that for any positive real a, $a^{\log_q n} = n^{\log_q a}$, as follows

$$
\begin{aligned}
W(h,n) &\geq n \cdot (Cont(R_q)/q)^h \left(q \cdot \frac{1 - (q/Cont(R_q))^h}{Cont(R_q)/q - 1} + 1 \right) \\
&= n^{1+\log_q(Cont(R_q)/q)} \left(q^2/Cont(R_q) \cdot \frac{1 - (q/Cont(R_q))^h}{1 - q/Cont(R_q)} + 1 \right)
\end{aligned}
$$

$$> q^2/Cont(R_q) \cdot n^{1+\log_q(Cont(R_q)/q)} \left(1 - (q/Cont(R_q))^h\right)$$
$$\geq 1/3 \cdot q^2/Cont(R_q) \cdot n^{1+\log_q(Cont(R_q)/q)} ,$$

where the last inequality holds because for all $q \geq 2$, $q/Cont(R_q) \leq 2/3$, and $h \geq 1$.

The argument for proving an upper bound is similar to the above argument for proving the lower bound. The main conceptual difference is that processors may write completion bits in different order for different internal nodes of the progress tree. Therefore, while the coefficients $k_1, \ldots, k_q$ were the same for each node during the analysis above, in the analysis of the upper bound, each internal node s has its own coefficients $k_1^s, \ldots, k_q^s$ that may be different for different nodes. The proof of the upper bound is omitted.

How does the bound from the lemma above depend on contention of the list R_q? We should answer this question so that when we instantiate the AWT algorithm, we know whether to choose permutations with low contention or perhaps with high contention. The answer to the question may be not so clear at first, because for any given q, when we take a list R_q with lower contention, then although the exponent of n is lower, but the constant c is higher. In the lemma below we study this tradeoff, and demonstrate that it is indeed of advantage to choose lists of permutations with as small contention as possible.

Lemma 5. *The function $c \mapsto q^2/c \cdot n^{\log_q c}$, where $c > 0$ and $n \geq q \geq 2$, is a non-decreasing function of c.*

The above lemma, simple as it is, is actually quite useful. In several parts of the paper we use a list of permutations, for which we only know an upper bound or a lower bound on contention. This lemma allows us to bound work respectively from above or from below, even though we do not actually know the exact value of contention of the list.

We would like to find out how the lower bound on work depends on the choice of q. The subsequent argument shows that careful choice of the value of q is essential, in order to guarantee low work. We begin with two technical lemmas, the second of which bounds from below the value of a function occurring in Lemma 4.

The lemma below shows that an expression that is a function of x must vanish inside a "slim" interval. The key idea of the proof of the lemma is that x^2 creates in the expression a highest order summand with factor either $1/2$ or $(1+\epsilon)/2$ depending on which of the two values of x we take, while $\ln x$ creates a summand of the same order with factor $1/2$ independent of the value of x. As a result, for the first value of x, the former "is less positive" than the later "is negative", while when x has the other value, then the former "is more positive" than the later "is negative". The proof is omitted.

Lemma 6. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough n, the expression $x^2 - x + (1 - \ln x) \cdot \ln n$ is negative when $x = x_1 = \sqrt{1/2 \ln n \ln\ln n}$, and positive when $x = x_2 = \sqrt{(1+\epsilon)/2 \ln n \ln\ln n}$.*

Lemma 7. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough n, the value of the function $f : [\ln 3, \ln n] \to \mathbb{R}$, defined as $f(x) = e^x/x \cdot n^{\ln x/x}$, is bounded from below by $f(x) \geq n^{(1-\epsilon)\sqrt{2 \cdot \ln \ln n / \ln n}}$.*

Proof. We shall show the lemma by reasoning about the derivative of f. We will see that it contains two parts: one that is strictly convex, and the other that is strictly concave. This will allow us to conveniently reason about the sign of the derivative, and where the derivative vanishes. As a result, we will ensure that there is only one local minimum of f in the interior of the domain. An additional argument will ascertain that the values of f at the boundary are larger than the minimum value attained in the interior.

Let us investigate where the derivative

$$\frac{\partial f}{\partial x} = e^x n^{\ln x/x}/x^3 \cdot \left(x^2 - x + (1 - \ln x) \ln n\right)$$

vanishes. It happens only for such x, for which the parabola $x \mapsto x^2 - x$ "overlaps" the logarithmic plot $x \mapsto \ln n \ln x - \ln n$. We notice that the parabola is strictly convex, while the logarithmic plot is strictly concave. Therefore, we conclude that one of the three cases must happen: plots do not overlap, plots overlap at a single point, or plots overlap at exactly two distinct points. We shall see that the later must occur for any large enough n.

We will see that the plots overlap at exactly two points. Note that when $x = \ln 3$, then the value of the logarithmic plot is negative, while the value of the parabola is positive. Hence the parabola is "above" the logarithmic plot at the point $x = \ln 3$ of the domain. Similarly, it is "above" the logarithmic plot at the point $x = \ln n$, because for this x the highest order summand for the parabola is $\ln^2 n$, while it is only $\ln n \ln \ln n$ for the logarithmic plot. Finally, we observe that when $x = \sqrt{\ln n}$, then the plots are "swapped": the logarithmic plot is "above" the parabola, because for this x the highest order summand for the parabola is $\ln n$, while the highest order summand for the logarithmic plot is as much as $1/2 \ln n \ln \ln n$. Therefore, for any large enough n, the plots must cross at exactly two points in the interior of the domain.

Now we are ready to evaluate the monotonicity of f. By inspecting the sign of the derivative, we conclude that f increases from $x = \ln 3$ until the first point, then it decreases until the second point, and then it increases again until $x = \ln n$. This holds for any large enough n.

This pattern of monotonicity allows us to bound from below the value of f in the interior of the domain. The function f attains a local minimum at the second point, and Lemma 6 teaches us that this point is in the range between $x_1 = \sqrt{1/2 \ln n \ln \ln n}$ and $x_2 = \sqrt{(1+\epsilon)/2 \ln n \ln \ln n}$. For large enough n, we can bound the value of the local minimum from below by $f_1 = e^{x_1}/x_2 \cdot n^{\ln x_1/x_2}$. We can further weaken this bound as

$$\begin{aligned} f_1 &= n^{-\ln x_2/\ln n + \ln x_1/x_2 + x_1/\ln n} \geq n^{-\ln x_2/\ln n + 1/2 \ln \ln n/x_2 + \sqrt{1/2 \ln \ln n/\ln n}} \\ &\geq n^{(1-\epsilon)\sqrt{2 \cdot \ln \ln n/\ln n}} , \end{aligned}$$

where the first inequality holds because for large enough n, $\ln(1/2 \ln \ln n)$ is positive, while the second inequality holds because for large enough n, $\ln x_2 \leq \ln \ln n$, and $1/\sqrt{1+\epsilon} \geq 1-\epsilon$, and for large enough n, $\sqrt{1/2 \ln \ln n / \ln n} - \ln \ln n / \ln n$ is larger than $\sqrt{1/(2+2\epsilon) \ln \ln n / \ln n}$.

Finally, we note that the values attained by f at the boundary are strictly larger then the value attained at the second point. Indeed, $f(\ln n)$ is strictly grater, because the function strictly increases from the second point towards $\ln n$. In addition, $f(\ln 3)$ is strictly grater because it is at least $n^{1.08}$, while the value attained at the second point is bounded from above by n raised to a power that tends to 0 as n tends to ∞ (in fact it suffices to see that the exponent of n in the bound on f_1 above, tends to 0 as n tends to ∞).

This completes the argument showing a lower bound on f.

The following two theorems show that we can construct an instance of AWT that has the exponent for n arbitrarily close to the exponent that is required, provided that we choose the value of q carefully enough.

Theorem 1. *Let $\epsilon > 0$ be any fixed constant. Then for any n that is large enough, any instance of the AWT algorithm for $p = n$ processors and n cells has work at least $n^{1+(1-\epsilon)\sqrt{2 \ln \ln n / \ln n}}$.*

Proof. This theorem is proven by combining the results shown in the preceding lemmas. Take any AWT algorithm for n cells and $p = n$ processors instantiated with a list R_q of q permutations on $[q]$. By Lemma 4, work of the instance is bounded from below by the expression $q^2/(3Cont(R_q)) \cdot n^{1+\log_q(Cont(R_q)/q)}$. By Lemma 5, we know that this expression does not increase when we replace $Cont(R_q)$ with a number that is smaller or equal to $Cont(R_q)$. Indeed, this is what we will do. By Lemma 1, we know that the value of $Cont(R_q)$ is bounded from below by $q \ln q$. Hence work of the AWT is at least $n/3 \cdot q / \ln q \cdot n^{\ln \ln q / \ln q}$.

Now we would like have a bound on this expression that does not depend on q. This bound should be fairly tight so that we can later find an instance of the AWT algorithm that has work close to the bound. Let us make a substitution $q = e^x$. We can use Lemma 7 with $\epsilon/2$ to bound the expression from below as desired, for large enough n, when q is in the range from 3 to n. What remains to be checked is how large work must be when the AWT algorithm is instantiated with just two permutations (i.e., when $q = 2$). In this case we know what contention of any list of two permutations is at least 3, and so work is bounded from below by n raised to a fixed power strictly greater than 1. Thus the lower bound holds for large enough n.

The following theorem explains that the lower bound can be nearly attained. The proof uses permutations described in Lemma 2. The proof is omitted.

Theorem 2. *Let $\epsilon > 0$ be any fixed constant. Then for any large enough m, when $q = \lceil e^{\sqrt{1/2 \ln m \ln \ln m}} \rceil$, and $h = \lceil \sqrt{2 \ln m / \ln \ln m} \rceil$, there exists an instance of the AWT algorithm for $p = n = q^h$ processors and n cells that has work at most $n^{1+(1+\epsilon)\sqrt{2 \ln \ln n / \ln n}}$.*

The above two theorems teach us that when q is selected carefully, we can create an instance of the AWT algorithm that is nearly optimal. A natural question that one immediately asks is: what if q is *not* selected well enough? Lemma 4 and Lemma 5 teach us that lower bound on work of an instance of the AWT algorithm depends on the number q of permutations on $[q]$ used by the instance. On one extreme, if q is a constant that is at least 2, then work must be at least n to some exponent that is greater than 1 and that is bounded away from 1. On the other extreme, if $q = n$, then work must be at least n^2. In the "middle", when q is about $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then the lower bound is the weakest, and we can almost attain it as shown in the two theorems above. Suppose that we chose the value of q slightly away from the value $e^{\sqrt{1/2 \ln n \ln \ln n}}$. By how much must work be increased as compared to the lowest possible value of work? Although one can carry out a more precise analysis of the growth of a lower bound as a function of q, we will be contented with the following result, which already establishes a gap between the work possible to attain when q is chosen well, and the work required when q is not chosen well. The proof is omitted.

Proposition 1. *Let $r \geq 2$ be any fixed constant. For any large enough n, if the AWT algorithm is instantiated with q permutations on $[q]$, such that $16 \leq q \leq e^{\sqrt{1/2 \ln n \ln \ln n}/(r \cdot \ln \ln n)}$ or $e^{r \cdot \sqrt{1/2 \ln n \ln \ln n}} \leq q \leq n$, then its work is at least $n^{1+r/3 \cdot \sqrt{2 \ln \ln n / \ln n}}$.*

4 Conclusions and Future Work

This paper shows how to create near-optimal instances of the Certified Write-All algorithm called AWT for n processors and n cells. We have seen that the choice of the number of permutation is critical for obtaining an instance of the AWT algorithm with near-optimal work. Specifically, when the algorithm is instantiated with about $e^{\sqrt{1/2 \ln n \ln \ln n}}$ permutations, then work of the instance can be near optimal, while when q is significantly away from $e^{\sqrt{1/2 \ln n \ln \ln n}}$, then work of any instance of the algorithm with this displaced q must be considerably higher than otherwise.

There are several follow-up research directions which will be interesting to explore. Any AWT algorithm has a progress tree with internal nodes of fanout q. One could consider generalized AWT algorithms where fanout does not need to be uniform. Suppose that a processor that visits a node of height i, uses a collection $R^i_{q(i)}$ of $q(i)$ permutations on $[q(i)]$. Now we could choose different values of $q(i)$ for different heights i. Does this technique enable any improvement of work as compared to the case when $q = q(1) = \ldots = q(h)$? What are the best values for $q(1), \ldots, q(h)$ as a function of n? Suppose that we are given a relative cost κ of performing a write to the cell of the array w, compared to the cost of executing any other basic action. What is the shape of the progress tree that minimizes work? These questions give rise to more complex optimization problems, which would be interesting to solve.

The author developed a result related to the study presented in this paper. Specifically, the author showed a work-optimal deterministic algorithm for the asynchronous CWA problem for a nontrivial number of processors $p \ll n$. An extended abstract of this study will appear as [12], and a full version will appear in the upcoming doctoral dissertation of the author.

Acknowledgements. The author would like to thank Charles Leiserson for an invitation to join the Supercomputing Technologies Group, and Dariusz Kowalski, Larry Rudolph, and Alex Shvartsman for their comments that improved the quality of the presentation.

References

1. Anderson, R.J., Woll, H.: Wait-free Parallel Algorithms for the Union-Find Problem. Extended version of the STOC'91 paper of the authors, November 1 (1994)
2. Anderson, R.J., Woll, H.: Algorithms for the Certified Write-All Problem. SIAM Journal on Computing, Vol. 26(5) (1997) 1277–1283 (Preliminary version: STOC'91)
3. Aumann, Y., Kedem, Z.M., Palem, K.V., Rabin, M.O.: Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. 34th IEEE Symposium on Foundations of Computer Science FOCS'93, (1993) 271–280
4. Buss, J., Kanellakis, P.C., Ragde, P.L., Shvartsman, A.A.: Parallel Algorithms with Processor Failures and Delays. Journal of Algorithms, Vol. 20 (1996) 45–86 (Preliminary versions: PODC'91 and Manuscript'90)
5. Chlebus, B., Dobrev, S., Kowalski, D., Malewicz, G., Shvartsman, A., Vrto, I.: Towards Practical Deterministic Write-All Algorithms. 13th Symposium on Parallel Algorithms and Architectures SPAA'01, (2001) 271–280
6. Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. Distributed Computing, Vol. 14(2) (2001) 75–81
7. Kanellakis, P.C., Shvartsman, A.A.: Efficient Parallel Algorithms Can Be Made Robust. Distributed Computing, Vol. 5(4) 1992 201–217 (Preliminary version: PODC'89)
8. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)
9. Kedem, Z.M., Palem, K.V., Raghunathan, A., Spirakis, P.G.: Combining Tentative and Definite Executions for Very Fast Dependable Parallel Computing. 23rd ACM Symposium on Theory of Computing STOC'91 (1991) 381–390
10. Knuth, D.E.: The Art of Computer Programming Vol. 3 (third edition). Addison-Wesley Pub Co. (1998)
11. Lovász, L.: Combinatorial Problems and exercises, 2nd edition. North-Holland Pub. Co, (1993)
12. Malewicz, G.: A Work-Optimal Deterministic Algorithm for the Asynchronous Certified Write-All Problem. 22nd ACM Symposium on Principles of Distributed Computing PODC'03, (2003) to appear
13. Martel, C., Park, A., Subramonian, R.: Work-optimal asynchronous algorithms for shared memory parallel computers. SIAM Journal on Computing, Vol. 21(6) (1992) 1070–1099 (Preliminary version: FOCS'90)
14. Naor, J., Roth, R.M.: Constructions of Permutation Arrays for Certain Scheduling Cost Measures. Random Structures and Algorithms, Vol. 6(1) (1995) 39–50