Leaders Election Without a Conflict Resolution Rule - Fast and Efficient Randomized Simulations among CRCW PRAMs

by
Joseph Gil and Yossi Matias

Technical Report 91-21
October 1991

Department of Computer Science
University of British Columbia
Rm 333 - 6356 Agricultural Road
Vancouver, B.C.
CANADA V6T 1Z2

Leaders Election Without a Conflict Resolution Rule – Fast and Efficient Randomized Simulations among CRCW PRAMs *

Joseph Gil †‡§
The University of British Columbia

Yossi Matias ¶‖
University of Maryland and
Tel Aviv University

January 1991**

Abstract

We study the question of fast leaders election on TOLERANT, a CRCW PRAM model which tolerates concurrent write but does not support symmetry breaking. We give a randomized simulation of MAXIMUM (a very strong CRCW PRAM) on TOLERANT. The simulation is optimal, reliable, and runs in nearly doubly logarithmic time and linear space. This is the first simulation which is fast, optimal *and* space-efficient, and therefore grants true comparison of algorithms running on different CRCW PRAMs. Moreover, it implies that the memory to which concurrent read or concurrent write are assumed should *never* be more than linear—the rest of the memory can always be addressed under the EREW convention. The techniques presented in this paper tackle fundamental difficulties in the design of fast parallel algorithms.

Summary

Exclusive-Write PRAM models are known to be strictly weaker than the Concurrent Read Concurrent Write PRAM models (CRCW). While the simple OR function of n bits requires $\Omega(\lg n)$ time on the Concurrent-Read Exclusive-Write (CREW) model, it can be solved in

*A previous version of this paper appeared as part of the first author Ph.D. dissertation [16].

†Part of research was done while author was in the Hebrew University of Jerusalem.

‡Research supported in part by the Leibniz Center for Research in Computer Science.

§Author's address: Department of Computer Science, 6356 Agricultural Road, The University of British Columbia, Vancouver, B.C. V6T 1Z2, Canada.

¶Partially supported by NSF grant NSF-CCR-8906949.

‖Author's address: Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

**Revised: September 1991

constant time on a CRCW. Moreover, there are quite a few more complex problems that can be solved in doubly logarithmic time or even less on the CRCW model.

Any Concurrent-Write model must define a *conflict resolution rule* to account for the case of several processors writing to the same cell. It may be thought that the power of CRCW model results from the ability of a memory cell to perform a "computation", according to write conflict resolution rule, in constant time on unbounded number of inputs. In an attempt to disprove this and in exploration of the borderline between exclusive-write and concurrent-write, we investigate the TOLERANT model, a very weak CRCW model. The TOLERANT model is in fact so weak that it uses no conflict resolution rule, and aside from tolerating an (unsuccessful) *attempt* for concurrent-write, it is identical to CREW.

A fundamental problem that captures the difference in power between the various PRAM models is the *leaders election* problem. Given a partition of the processors into sets, according to the memory cells to which they want to write, the problem is for each set to select a distinguished processor from the set (the "leader"). The decision on which processor to select depends on the model at hand.

We will limit the memory size used by our algorithms to be linear in the number of processors. In addition to efficiency considerations, it is interesting to understand the power of computation in bounded space. Boppana showed that the *element distinctness* problem (deciding whether or not all input elements are distinct) requires $\Omega(\lg n/\lg\lg n)$ time on a deterministic n-processor PRIORITY with bounded space; "bounded-space" here means that the space is not dependent on the input (but may be a function of n). We show that even harder problems can be solved in $O(\lg\lg n)$ time on a randomized TOLERANT with linear space. Another important implication of this result is that the effort of *avoiding* write-conflicts is the main obstacle in obtaining very fast CREW algorithms and not the fact that these models are forbidden from using a strong rule for conflict resolution.

As a fundamental building block we show that picking the processor with the maximal value amongst a set of *anonymous* processors (leader election) can be done in constant time on the randomized TOLERANT model. We generalize and show that n instances of this problem can be solved in $O(\lg\lg n)$ time on an n-processor randomized TOLERANT. Further, we show that an n-processor machine, belonging to any of the CRCW models in common use, is equivalent, up to an $O(\lg\lg n)$ randomized time factor, to an n-processor TOLERANT machine. This includes also the strong and non-standard MAXIMUM CRCW model, in which the conflict resolution rule is a maximum computation. The number of processors in the simulating machine can be reduced to $n/\lg\lg n$ on the (stronger) COLLISION CRCW, or to $n/\lg\lg n\lg^* n$ on the TOLERANT with a further slowdown factor of $O(\lg^* n)$. These simulations are *optimal* in the sense that the time-processor product (i.e., the number of operations) of the simulating and simulated machine are the same, up to a constant factor.

These results improve on all previously published simulation algorithms in one or more of the following aspects: time, optimality, strength of simulating machine, strength of simulating machine, and memory usage. It implies that an algorithm being designed for a strong (and hence more convenient) model of computation such as the MAXIMUM can be *automatically* adapted to work on the relatively weak (and hence closer to reality) TOLERANT model,

with a certain small slowdown but without sacrificing efficiency. In addition to being optimal, our simulation is the first fast simulation that does not introduce a multiplicative factor n increase in memory size. In fact, the extra memory in use is an additive $O(n)$ factor. Moreover, the (possibly large) memory being used by the simulated machine will be now addressed only under the EREW convention. It follows that *the space to be used under the concurrent-read concurrent-write convention (in any standard model) should never be more than linear.*

An application of our result demonstrating the power of the TOLERANT model is an improved optimal and reliable algorithm for the *hashing* problem: given n keys, build a hash table that uses linear space, and supports $O(1)$ time lookup queries. All our algorithms run with high probability, i.e., they successfully terminate within the stated time complexity with probability $1 - n^{-k}$, for any constant k.

The techniques presented in this paper tackle fundamental difficulties in the design of fast parallel algorithms. The potential applications are therefore beyond the scope of the concrete problems considered here.

Postscript The ideas and techniques presented in this paper, together with other ideas, have recently lead to extremely fast new simulations and leaders election algorithms that take $O(\lg^* n)$ time with high probability [GMV91].

1 Introduction

The simplicity and elegance of the Parallel Random Access Machine (PRAM) model for parallel computation have attracted many researchers to use it as a model for the design of parallel algorithms. (Overviews of PRAM theory: motivation, justification, complexity issues, realizations, and algorithms can be found in [43, 11, 28, 30, 44, 27].) The PRAM model is actually a collection of models differing by the rules they impose on concurrent access to memory. In this paper we concentrate on the study of the Concurrent Read Concurrent Write (CRCW) family of sub-models of PRAM. In this family concurrent read and concurrent write accesses by more than one processors to the same memory cell are permitted. Members of the family differ in the way they handle and resolve conflicts arising from simultaneous attempts to write to the same cell.

Cook, Dwork and Reischuk [10] showed that the Concurrent-Write convention is essential even for computing the basic OR function in $o(\lg n)$ time. Beame and Hastad [2] showed that even this convention (which enables computing the OR function in constant time) does not enable to compute any symmetric function in $o(\lg n/\lg\lg n)$ time, using even a polynomial number of processors. Any non-trivial computation that can be done faster would be therefore interesting. In particular are interesting algorithms whose output is an array of size n and that must involve global coordination between the input elements. Our simulation algorithms belong to this class. Some examples for problems that can be solved in doubly-logarithmic time or faster are the maximum finding [41, 39], merging [29], chaining [20, 4, 36], and string matching [3]. Very few doubly logarithmic time or faster randomized algorithms are known. They include "padded" sorting with random input [32], hashing [17], and load balancing [15].

Algorithm designers clearly prefer to work with stronger models of computation. It enables one to avoid tedious implementation efforts that are due to the constraints being posed by the weaker models. On the other hand, it is favorable to have algorithms that work on weaker models of computation. A traditional way to bridge the gap would be to have algorithms that simulate strong models on relatively weak models. Such simulations enable automatic translations between algorithms. An algorithm designed for a strong (and hence more convenient) model, can be translated into an equivalent algorithm that works on a weaker model. Clearly, we would like the simulation algorithms to be efficient in *all* complexity measures.

All previous simulation algorithms of strong CRCW models on weak CRCW models are either relatively slow (time is nearly logarithmic) or are highly inefficient. Fast simulations either have a logarithmic increase in the number of processors (and of work) or a polynomial blowup in their memory size. We present a new *fast* and *efficient* randomized algorithm that simulates an n-processor MAXIMUM-CRCW-PRAM on TOLERANT-CRCW-PRAM. Using an optimal number of processors the TOLERANT simulates the MAXIMUM in $O(\lg\lg n \lg^* n)$ time (with high probability). Using n processors, the simulation time is $O(\lg\lg n)$. For the stronger COLLISION model, there exists an $O(\lg\lg n)$ time simulation using an optimal number of processors, with high probability. The simulation algorithms use $O(n)$ space. As a result, the possibly large memory into which the concurrent-read or concurrent-write operations of

the simulated machine are assumed can be addressed by the EREW convention!

The MAXIMUM and TOLERANT models differ in the outcome of the event of several processors concurrently attempt a write operation (of possibly different values) into the same memory cell. In the TOLERANT model the cell contents does not change; in the MAXIMUM model, the maximum value is being written. The MAXIMUM is a powerful CRCW sub-model, while the TOLERANT is a weak one. Several other CRCW sub-models were used and studied in the literature; they are described in the following section. The simulation results apply to most of them as well.

Leaders election A simulation algorithm can be viewed as a *leaders election* algorithm: from each set of processors writing into the same cell the one with the maximum value is elected leader. An easier leaders election problem, in which an *arbitrary* processor is selected to be leader, is closely related to two other fundamental problems: *element distinctness* and *hashing*.

Given n keys drawn from some finite universe the element distinctness problem is to decide whether they are all distinct. A leaders election algorithm can be easily used to solve the element distinctness problem.

Let S be a set of n keys drawn from some finite universe U. Then the hashing problem is to build a data structure for S (a "hash table") that uses $O(n)$ space, and supports *lookup queries* in $O(1)$ operations [14]. A lookup query is: "is $x \in S$?" for any key $x \in U$. Our leaders election algorithm enables to remove duplicate keys. The existence of duplicates was one of the obstacles for achieving high probability ("reliable") hashing algorithm in [17]. Their hashing algorithm runs in $O(\lg\lg n)$ *expected* time on COLLISION$^+$. The presented simulation algorithm may be used to improve this algorithm [18]. The improved hashing algorithm is reliable and runs on TOLERANT.

2 Preliminaries

2.1 The CRCW Models

The following lists the most popular conflict resolution rules and the corresponding names of the sub-models in the CRCW family:

COMMON [31] All processors that simultaneously write into the same cell, *must* be writing the same common value. This results in the value written into the cell.

ROBUST [26] If two or more processors attempt to write into the same cell in a given step, then no assumption is being made about the contents of this cell after this attempt, as long as it is well defined.

TOLERANT [23] If two or more processors attempt to write to the same cell in a given step, then the contents of that cell does not change. This model is called CANCELLATION in [24].

COLLISION [12] If two or more processors attempt to write to the same cell in a given step, then a special collision symbol $\sharp$ is written to that cell.

COLLISION$^+$ [9] If the processors attempting to write to the same cell in a given step all attempt to write the same value, then that value is written in that cell; if at least two values differ, a special collision symbol $\sharp$ is written to that cell.

ARBITRARY [40] If two or more processors attempt to write to the same cell in a given step, then one of them succeeds, but there is no rule assumed on the selection of the successful processor.

PRIORITY [21] If two or more processors attempt to write to the same cell in a given step, then the highest-numbered processor among them succeeds.

RANDOM [33] If two or more processors attempt to write to the same cell in a given step, then the successful processor is selected at random with equal probabilities among them.

MAXIMUM [1] If two or more processors attempt to write to the same cell in a given step, then the processor trying to write the highest value among them succeeds.

FETCH&ADD [42] If two or more processors attempt to write to the same cell in a given step, then their values are added to the value already written in the shared memory location and all prefix sums obtained in the (virtual) serial process are recorded.

The TOLERANT model is similar to the Concurrent Read Exclusive Write (CREW) model, except for the fact that it does not crash when concurrent-write is attempted. Instead, the contents of cells involved in a write conflict remain unaltered. The COLLISION model assumes a specific effect on cells that are subject to a concurrent-write attempt. This model is a straightforward generalization of the Etherenet bus communication protocol [45], in which if more than one bus client concurrently tries to master the bus, the bus contents are corrupted, but the collision is detected. The ROBUST makes no particular assumption about the contents of the cells subject to a concurrent-write attempt and is therefore weaker than both COLLISION and TOLERANT.

The COMMON model is incomparable to COLLISION and to TOLERANT [23]. On one hand, it enables a set of processors to change the content of a cell, while in both TOLERANT and COLLISION only one processor can do that. On the other hand, it does not permit an attempt of concurrent write of different values. The COLLISION$^+$ model is a combination of COMMON and COLLISION.

The ARBITRARY is a stronger model in the sense that even in a concurrent-write attempt of different values one processor succeeds. The RANDOM model is even stronger since it incorporates an assumption about the successful processor. The powerful PRIORITY model has gained popularity despite the assumption of a maximum computation in each conflict resolution. The MAXIMUM model seems to be so much stronger that it has not been generally accepted as a standard model of a CRCW. Indeed, the choices of successful processors in this model are value-dependent and, unlike the other models, involve a computation that is at least as hard as computing the maximum of n arbitrarily large numbers.

2.2 Simulations

Let $\mathcal{M}$ be a generic model (e.g., COMMON, TOLERANT). Then $\mathcal{M}(n,m)$ denotes a machine of this model with n processors which is restricted to use no more than m memory cells; $\mathcal{M}(n)$ denotes a machine of this model that uses n processors (no statement about memory). The comparison of the relative strengths of the sub-models is carried out by studying the time required for a *simulation* of one step of a machine $\mathcal{M}_1$ on another machine $\mathcal{M}_2$.

A machine $\mathcal{M}_2$ simulates $\mathcal{M}_1$ with *slowdown* T, if at most T steps of $\mathcal{M}_2$ are needed to simulate one step of $\mathcal{M}_1$. A simulation of $\mathcal{M}_1(n,m)$ by $\mathcal{M}_2(n',m')$ is *processor-efficient* if the number of processors in the simulating machine is at most the number of processors in the simulated machine, i.e., $n' \leq n$. The simulation is *space-efficient* if the space used by $\mathcal{M}_2$ is larger than the space used by $\mathcal{M}_1$ by at most an *additive* factor of $O(n)$, i.e., $m' = m + O(n)$. Note that it might be the case that $m \gg n$ and we therefore do not allow even a constant blowup in the memory size. The simulation is *efficient* if it is both processor-efficient and space-efficient, and it is also *optimal* if a slowdown T is compensated by an appropriate decrease in the number of processors in the simulating machine; that is, if $n' = O(n/T)$. Finally, we say that a simulation is *fast* if the slowdown is at most polynomial in $\lg\lg n$.

We are interested here only in fast simulations that are efficient and, preferably, optimal. If there exists a simulation of $\mathcal{M}_1$ by $\mathcal{M}_2$ with a constant slowdown that is efficient (and hence optimal) then we say that "$\mathcal{M}_2$ is *as powerful as* $\mathcal{M}_1$" and we denote this by $\mathcal{M}_1 \preceq \mathcal{M}_2$. A similar terminology was used in [8, 9, 26] but they only required the simulations to be processor-efficient.

The following partial ordering can be trivially derived:

$$\text{ROBUST} \preceq \left\{ \begin{array}{l} \text{COMMON} \\ \text{TOLERANT} \\ \text{COLLISION} \end{array} \right\} \preceq \text{COLLISION}^{+} \preceq \text{ARBITRARY} \preceq \left\{ \begin{array}{l} \text{RANDOM} \\ \text{PRIORITY} \end{array} \right. ,$$

$$\text{PRIORITY} \preceq \text{MAXIMUM} \preceq \text{FETCH\&ADD} .$$

Grolmusz and Ragde [23] showed that COMMON and COLLISION are incomparable in the sense of "$\preceq$" and that COMMON and TOLERANT are incomparable in the same sense. We will show in Section 8 that when restricted to linear size memory, COLLISION and TOLERANT are *equivalent*, i.e., each of them is as powerful as the other. We will also show in Section 3.2 that if the definition of $\preceq$ is extended to include random simulations as well then

$$\text{RANDOM} \preceq \text{MAXIMUM} .$$

2.3 On Teams and Anonymous Sets

If each processor of $\mathcal{M}_1$ is emulated by a processor of $\mathcal{M}_2$, then the main difficulty of simulating MAXIMUM is in emulating the max conflict resolution rule. However, this problem

does not reduce to a plain maximum evaluation because a processor attempting a write operation to a certain cell, does not in general know which other processors simultaneously attempt the same operation, consequently, a major obstacle is in identifying the participants in the maximum computation.

Let us use the following definitions for characterizing the two extreme cases of the knowledge state of members of processors sets:

Definition 1 *A set of processors Φ is* anonymous *if every processor knows whether it belongs to Φ or not, but no other information (such as cardinality, ordinal place etc.) is available to the set members.*

Definition 2 *A* team *is a set of processors with* consecutive *indices, such that the starting and the ending indices are known to all the set members.*

A team is *allocated* to an anonymous set Φ if all members of Φ, as well as the first processor in the team (the team *leader*), agree on this allocation.

The regular structure of the team makes it possible to use it as a sub-PRAM, and thus facilitating ordinary parallel computations on it. We assume that each team has a private memory of linear size in the cardinality of the team. This assumption does not add more than $O(n)$ memory to the whole machine.

2.4 Probabilistic Analysis

Our algorithms work with high probability. Hence, each step must *succeed* with high probability, for some proper definition of *success*. It will be convenient to use the following terminology.

Definition 3 *An event $A = A(n)$ is* n-negligible *(or for short* negligible*) if* $\mathbf{Prob}(A) = o(n^{-\epsilon})$ *for some $\epsilon > 0$. We then say that A occurs with* n-negligible *probability. The compliment event $\bar{A}$ is* n-dominant *(or for short* dominant*); we say that it occurs with* n-dominant *probability.*

Using this notation we write Chebyshev inequality in the following convenient form:

Fact 1 *Let X be a positive valued random variable satisfying*

$$\mathbf{Var}(X) \leq (\mathbf{E}(X))^{2-\delta}$$

for some constant $\delta > 0$. Then for any fixed $\epsilon >$, the event

$$(1-\epsilon)\mathbf{E}(X) \leq X \leq (1+\epsilon)\mathbf{E}(X)$$

is $\mathbf{E}(X)$-dominant.

Definition 4 *A probabilistic algorithm for a given problem of size n is* solid *if it solves the problem within the stated complexity bounds with n-dominant probability.*

Note that usually the success probability of a solid algorithm can be amplified by means of repetitions to $1 - n^{-k}$ for *any* constant k.

3 Simulation Results

3.1 Previous Work

Lower bounds Lower bounds for simulation algorithms were considered in [6, 37, 12, 13, 23]. Fich, Ragde and Wigderson [13] proved that if simulating processors are forbidden to communicate with those attempting to write into different cells then the slowdown of the simulation of PRIORITY(n, n) on COMMON(n, n) is $\Theta(\lg n/\lg\lg n)$. This bound holds even if processors are allowed random decisions. Lower bound results concerning simulations of a single-cell machine by a multi-cell one were given by Fich et. al [12]: the simulation time of

(i) PRIORITY$(n, 1)$ on ARBITRARY(n, m),
(ii) ARBITRARY$(n, 1)$ on COLLISION(n, m),
(iii) ARBITRARY$(n, 1)$ on COMMON(n, m),
(iv) COLLISION$(n, 1)$ on COMMON(n, m), and
(v) COMMON$(n, 1)$ on COLLISION(n, m)

is $\Theta(\lg n/\lg(m+1))$. Fich, Meyer auf der Heide and Wigderson proved that the *Element Distinctness* problem requires $\Omega(\lg\lg\lg n)$ time on a deterministic COMMON(n, ∞) for large enough inputs. This bound was improved to $\Omega\left(\sqrt{\lg n}\right)$ in [37], and to $\Omega(\lg n/\lg\lg n)$ by Boppana [6]. The same problem can be solved in constant time by TOLERANT(n, ∞), thus implying lower bounds for deterministic simulations. Grolmusz and Ragde [23] showed that COMMON(n, ∞) and COLLISION(n, ∞) are incomparable and that COMMON(n, ∞) and TOLERANT(n, ∞) are incomparable.

Upper bounds Numerous simulation algorithms among CRCW sub-models appear in the literature. Table 1 gives a summary of processor-efficient simulations of PRIORITY or stronger sub-models. Note that all previous fast simulations are not space-efficient: the space has a blowup by a multiplicative factor of $O(n)$ (!). (Recall that we do not allow even a multiplicative *constant* factor.)

There is no fast processor-efficient simulation of MAXIMUM and the only optimal simulation is slow. Following [34, 17], parallel hashing may be used to modify the fast simulations into space-efficient (but not optimal) simulations with an increase in time. However, the hashing algorithms work only on COLLISION$^+$ and therefore do not help in simulations on COLLISION or on TOLERANT. In fact, one application of our new simulation results is the ability to implement the hashing algorithms on TOLERANT.

Simulated Machine	Simulating Machine	Slowdown	Number of Processors	Additional Space	Reference
FETCH&ADD(n,m)	COLLISION$^+$	$O\left(\frac{\lg n}{\lg\lg n}\right)$ (solid)	$\frac{n\lg\lg n}{\lg n}$ (optimal)	$O(n)$	[34]
FETCH&ADD(n,m)	ARBITRARY	$O\left(\frac{\lg n}{\lg\lg n}\right)$	$n\frac{(\lg\lg n)^2}{\lg n}$	$O(n^{1+\epsilon})$	[34]
PRIORITY(n,m)	COMMON	$O\left(\frac{\lg n}{\lg\lg n}\right)$	n	$O(nm)$	[13]
PRIORITY(n,m)	TOLERANT	$O(\sqrt{\lg n})$	n	$O(nm)$	[9]
PRIORITY(n,m)	COLLISION$^+$	$O\left(\lg\lg n\lg^{(3)} n\right)$	n	$O(nm)$	[9]
PRIORITY(n,m)	ARBITRARY	$O(\lg\lg n)$	n	$O(nm)$	[8]
PRIORITY(n,m)	COLLISION$^+$	$O(\lg\lg n)$ (solid)	n	$O(nm)$	[9]
PRIORITY(n,m)	ROBUST	$O(\lg\lg n)$ (solid)	n	$O(nm)$	[26]
MAXIMUM(n,m)	TOLERANT	$O(\lg\lg n)$ (solid)	n	$O(n)$	**new**
MAXIMUM(n,m)	COLLISION	$O(\lg\lg n)$ (solid)	$\frac{n}{\lg\lg n}$ (optimal)	$O(n)$	**new**
MAXIMUM(n,m)	TOLERANT	$O(\lg\lg n\lg^* n)$ (solid)	$\frac{n}{\lg\lg n\lg^* n}$ (optimal)	$O(n)$	**new**

Table 1: Efficient simulation algorithms of PRIORITY or stronger sub-models

The best previously known simulation results for MAXIMUM(n,m) are those derived from [34]: an $O(\lg n/\lg\lg n)$ expected slowdown simulation on COLLISION$^+(\lg n/\lg n, m+O(n))$ and an $O(\lg n/\lg\lg n)$ slowdown simulation on ARBITRARY$(n\frac{(\lg\lg n)^2}{\lg n}, m+n^{1+\epsilon})$ for any $\epsilon > 0$ (both for the stronger FETCH&ADD).

Comment: The first simulation in the table was originally stated with $O(\lg n)$ expected slowdown on ARBITRARY. The result stated in the table is obtained by using the improved hashing algorithm of [17] and the integer sorting algorithm in [35]. The model of computation can be further weakened by using the results of the present paper. The second simulation in the table was originally stated with $O(\lg n)$ (deterministic) slowdown on PRIORITY. The result stated in the table is obtained by using the integer sorting algorithm of [5]. The simulation of PRIORITY on ROBUST requires $O(\lg n)$ time to detect failure; this is acceptable for slow algorithms but is a major drawback for fast algorithms.

Many simulation algorithms do not increase the number of processors. However, sometimes auxiliary processors are used as well. For example, Kučera showed that if $n' = n^2$ then simulation of PRIORITY on COMMON can be done in constant time [31]. Such an increase in the number of processors is usually unacceptable.

3.2 Results

Our main theorems expand the investigation of the relative power of the different variants of the CRCW-PRAM:

Theorem 2 *One step of* MAXIMUM(n, m) *can be simulated by an* $O(lglgn)$ *time solid algorithm running on* TOLERANT$(n, m + O(n))$.

Theorem 3 *One step of* MAXIMUM(n, m) *can be simulated by an* $O(lglgn)$ *time solid algorithm running on* COLLISION$(n/lglgn, m + O(n))$ *(optimal speedup).*

Theorem 4 *One step of* MAXIMUM(n, m) *can be simulated by an* $O(lglgnlg^*n)$ *time solid algorithm running on* TOLERANT$(n/lglgnlg^*n, m + O(n))$ *(optimal speedup).*

These results improve on all previously known simulation results in one or more of the following aspects:

Time The $O(\lg\lg n)$ time (with high probability) is considerably better than the $O(\lg n/\lg\lg n)$ (expected) time of the best previously known simulation of MAXIMUM.

Strength of Simulated Machine The simulated machine is MAXIMUM which is stronger than PRIORITY in at least two aspects:

(a) The range of values among which the maximum is selected is unbounded.
(b) Written values need not be distinct.

Strength of Simulating Machine Our simulations are done by the TOLERANT model. The previous efficient simulation of MAXIMUM works on COLLISION$^+$ that is strictly stronger than TOLERANT [23].

Memory Blowup All previous fast simulations of PRIORITY use very large space. The memory size of the simulating machine is $O(nm)$, where m is the memory size of the simulated machine. (Note that m might be much larger than n.) Our simulations require only an *additional* space of size $O(n)$. Furthermore, regardless the memory size in the simulated machine, only an $O(n)$ segment of the memory in the simulating machine is accessed under the CRCW convention. The rest of the memory is accessed under the EREW convention.

Optimality Many simulation algorithms did not preserve optimality, i.e., their time-processor product was not $O(n)$. Optimal speedup simulations were first introduced in [34] who argued that preserving optimal-speedup is a desirable property also for a simulation algorithm. Our algorithm achieves optimal-speedup by using an additional pre-processing simulation algorithm stage. This stage is different in nature than the main algorithm, and we feel that it is also interesting in its own right.

The algorithms behind the theorems above are actually described for the COLLISION model. We use the fact that the simulation algorithms use only linear memory together with the following theorem to carry the simulations on the TOLERANT model as well.

Theorem 5 *If* $m = O(n)$ *then* COLLISION$(O(n), m) \preceq$ TOLERANT(n, m).

The above simulations are valid also for the RANDOM model as can be seen from the following simple lemma:

Lemma 6
There exists a solid simulation algorithm of RANDOM(n, m) *on* MAXIMUM$(n, m + O(n))$ *with* $O(1)$ *slowdown.*

Proof The simulation is carried out as follows. Before writing into a cell, each processor chooses at random a number from a range of size $O(n^3)$, and tries to write this value into the cell. Only processors that succeeded in this writing will continue and write their actual value. It is easy to verify that, with dominant probability, no value will be chosen by more than one processor, and that the MAXIMUM write-conflict resolution rule ensures that the processors selected are indeed chosen at random. ∎

Comment: A related idea can lead to a solid algorithm simulating RANDOM on PRIORITY in $O(\lg^* n)$ time using an optimal number of processors. This is carried out by first computing a random permutation using an algorithm from [35], and then processors shuffle their values according to the permutation. A simple write step follows.

Boppana [6] gave an $\Omega(\lg n/\lg\lg n)$ time lower bound for any deterministic algorithm for the element distinctness problem which runs on an n-processor PRIORITY with bounded memory. "Bounded memory" means that the memory size is an arbitrary function of n but not of the input values range. A consequence of our simulation result is that randomization techniques allow to solve the element distinctness problem faster and on a much weaker model using only linear memory:

Corollary 7 *There exists a solid algorithm for the leaders election problem, running in* $O(lglgn)$ *time on* TOLERANT$(n, O(n))$. *This algorithm enables solving the element distinctness problem and also the problem of removing all duplications from a given multi-set of size* n.

Postscript Recently, [19] presented a new simulation algorithm in the $\lg^* n$ time level. Specifically, they gave an algorithm that simulates MAXIMUM(n, m) on TOLERANT$(n/\lg^* n, m + O(n\lg n))$ in $O(\lg^* n)$ time (optimal speedup), and on TOLERANT$(n/(\lg^* n)^2, m+O(n))$ in $O((\lg^* n)^2)$ time (optimal speedup), both with high probability. Their simulation algorithms use some techniques pioneered by this paper.

4 Statement of the Problem

In this section we identify the basic complication arising when simulating MAXIMUM, and outline the rest of the paper.

The *Find_Max* Problem Consider the following problem: Let Φ be an anonymous set of processors, each processor $P \in \Phi$ has a value $val(P)$, and all values are distinct. The *Find_Max*(Φ) problem is to single out a processor $P^* \in \Phi$ such that

$$\forall P \in \Phi,\ val(P^*) \geq val(P) \ .$$

If Φ is empty then the solution is vacuous. Denote by $Find\_Max_{n,m}$ the problem of simultaneously solving m instances of *Find_Max* for the sets (some possibly empty) $\Phi_1, \Phi_2, \ldots, \Phi_m$, such that

$$\sum_{i=1}^{m} |\Phi_i| \leq n \ .$$

Simulating the write step of MAXIMUM(n,m) can be viewed as $Find\_Max_{n,m}$: Φ_i is the set of all processors that wish to write to cell M_i; $val(P_j) = \langle V_j, j \rangle$, where V_j is the value P_j should write in this step. Lexicographic ordering is to be used in comparisons of $val(P_j)$.

Our main task is to design an algorithm for $Find\_Max_{n,m}$. One of the difficulties is to identify the non-empty instances, or equivalently reducing m to $O(n)$. Another is that it cannot generally be assumed that there are processors available for solving *Find_Max*(Φ) other than those in Φ.

In Section 5 we show that if an estimate is known on $|\Phi|$ then *Find_Max*(Φ) can be solved using a team of external processors whose size depends on the estimate. The algorithm presented runs in $O(1)$ time with very high probability. This partial solution is used as a subroutine in Section 6 to give a solid algorithm for $Find\_Max_{n,n}$ running in $O(\lg\lg n)$ time. In Section 7 pseudo-random hash functions are used to extend this algorithm to solve $Find\_Max_{n,m}$ within the same time frame. The above mentioned algorithms run on the COLLISION model. Section 8 shows that the algorithms run on the TOLERANT model as well by showing a linear memory equivalence of COLLISION on TOLERANT. We explain how to achieve optimal speedup in Section 9.

5 The Basic Algorithm

The following is a generalization of the $O(1)$ time randomized algorithm for computing the maximum on a team of Reischuk [38].

Theorem 8 *Let d_1, d_2, β be any positive integers. Let Φ be an anonymous set, $|\Phi| \leq \beta^{d_1}$. Then there exists a probabilistic algorithm,* RandMax, *which using a team of β processors on* COLLISION, *solves Find_Max(Φ) in $O(1)$ time with probability $\geq 1 - \beta^{-d_2}$.*

Comment: Note that the AND and OR functions can be computed on COLLISION in $O(1)$ time, by a deliberate writing of the special collision symbol $\sharp$. One reservation applies: the cell used for the AND/OR computation must be properly initialized. If the processors having the inputs are an anonymous set, then this initialization is not simple. This will not be an obstacle for us since AND/OR will be computed by teams only.

Proof outline In each iteration of RandMax, random allocation is used to map a sample of size $\beta^{O(1)}$ from the anonymous set Φ to the team. The maximum value of the sample is computed and Φ is then adjusted to contain only processors with value not less than this maximum. The rest of this section is devoted to the details of the proof.

5.1 Building Blocks

Lemma 9 *Let d_2 be fixed. Let Φ be allocated with a team of β^6 processors, for $\beta \geq |\Phi|$. Then Find_Max(Φ) can be solved in $O(1)$ time with probability $> 1 - \beta^{-d_2}$.*

Proof An array $A[1, \beta^3]$ is used by the team. Each $\mathrm{P} \in \Phi$ picks at random $i \in [1, \beta^3]$, and tries to write $val(\mathrm{P})$ into $A[i]$. The team computes $\max\{A[i] : 1 \leq i \leq \beta^3\}$. This is implemented in constant time by doing all $O(\beta^6)$ possible comparisons and finding the value that won all comparisons. Note that computation involves AND/OR functions.

The algorithm fails only if there was a collision in the random write step. The probability of a collision in a single cell is at most $\binom{\beta}{2}\beta^{-6}$ and the probability of a collision in any cell is hence no more than $\beta^3\binom{\beta}{2}\beta^{-6} < \frac{1}{\beta}$.

The COLLISION model allows for failure detection. By repeating the algorithm d_2 times, the failure probability becomes $< \beta^{-d_2}$. ∎

The above algorithm may be useful even if it fails. Only processors with values exceeding the maximum of the written values should continue, thus decreasing the size of Φ. This suggests a way for solving the *Find_Max* for larger sets:

(a) Select a random sample Ψ from Φ.
(b) Solve *Find_Max*(Ψ).
(c) Let Φ' be the set of processors from Φ with values not smaller than $\max \Psi$.
(d) Continue the process with Φ'.

It seems natural to expect that the decrease in $|\Phi|$ will be by a factor equal to the sample size. However, there is still a constant probability that this will not occur. The following lemma gives a less ambitious estimate for the decrease. This estimate is accurate with sufficiently high probability.

Lemma 10 *Let Ψ be a random sample of size β drawn from a set Φ, and Φ' be defined as in the above. Then*

$$\mathbf{Prob}\left(|\Phi'| > |\Phi|/\sqrt{\beta}\right) < 1/\beta \quad .$$

Proof Rank processors in Φ according to their value, with the processor with highest value ranked 1. The event $|\Phi'| > |\Phi|/\sqrt{\beta}$ occurs exactly when the sample includes none of the processors ranking $1, 2, \ldots, |\Phi|/\sqrt{\beta}$. The probability for this is smaller than in the case of sampling *with replacements* for which this probability is exactly $\left(1 - 1/\sqrt{\beta}\right)^{\beta} < 1/\beta$ for $\beta \geq 2$. (Note that the proof assumes that the values of Φ are distinct.) ∎

5.2 Algorithm RandMax

Suppose that $|\Phi| \leq \beta^{d_1}$, where $\beta \geq 2$ and d_1 are known. Then the following recursive probabilistic algorithm implements the sampling idea to solve *Find_Max*(Φ) in $O(1)$ time. The failure probability will be at most β^{-d_2}. For simplicity, the algorithm is presented using a team of size $\beta^{O(1)}$. It is easy to see that the implementation using a team of size β is merely a matter of change in notations.

Algorithm RandMax(Φ, d_1, d_2);
INPUT: *An instance* Φ *of Find_Max.*
OUTPUT: *A set* $\Phi' \subseteq \Phi$ *such that Find_Max*(Φ') = *Find_Max*(Φ).

Begin
if $d_1 \leq 1$ **then**
Apply the algorithm of Lemma 9 to solve *Find_Max*(Φ) with failure probability β^{-d_2}, and return the result.
fi;
loop $\lceil \frac{d_2+1}{2} \rceil + 1$ **times**
$\Phi \leftarrow$RandMax($\Phi, d_1 - 1, d_2 + 1$);
if $|\Phi| = 1$ **then**
return Φ;
fi;
$\Psi \leftarrow \emptyset$;
Select a random sample Ψ of Φ where each element is chosen with probability C/β^{d_1-3}, where C is a constant to be chosen later.
Find the maximum value of the processors in Ψ.
$V \leftarrow \max\{val(\mathrm{P}') : \mathrm{P}' \in \Psi\}$;
$\Phi \leftarrow \{\mathrm{P} \in \Phi : val(\mathrm{P}) \geq V\}$;
end loop;
return Φ;
End RandMax;

Theorem 8 follows from

Lemma 11 *If the input to* RandMax *satisfies* $|\Phi| \leq \beta^{d_1}$ *then*

$$\textbf{Prob}\left(|\Phi'| > 1\right) \leq \beta^{-d_2} \quad .$$

Proof The Lemma clearly holds for $d_1 \leq 1$. Induction on d_1 is used to show that it is correct for the entire range of $|\Phi|$. It may be assumed that in the first $\lceil \frac{d_2+1}{2} \rceil$ iterations $|\Phi| > \beta^{d_1-1}$ holds before the sample is taken. Otherwise, by the inductive hypothesis, the recursive call in the next iteration will solve *Find_Max*(Φ) with sufficiently high probability. Thus we can write

$$\beta^{d_1-1} < |\Phi| \leq \beta^{d_1} \quad , \tag{1}$$

and since each element in Φ is selected to be in Ψ with probability C/β^{d_1-3}

$$C\beta^2 < \mathbf{E}(|\Psi|) \leq C\beta^3 \quad . \tag{2}$$

We say that an iteration is *successful* if it reduces $|\Phi|$ by a factor of β. The possible iteration *failures* are:

1. *Find_Max*(Ψ) was not of sufficiently high rank to decrease Φ sufficiently: It follows from Lemma 10 that if $|\Psi| \geq 4\beta^2$ this kind of failure occurs with probability at most $1/4\beta^2$.

2. The sample size was too small for the conditions of Lemma 10: by using Chernoff bounds [7] and (2) the constant C can be selected so that

$$\mathbf{Prob}\left(|\Psi| \leq 4\beta^2\right) \leq 1/4\beta^2 \quad .$$

3. Computing $\max\{val(P') : Processor' \in \Psi\}$ in constant time failed: By Lemma 9 if $|\Psi| = 8\beta^3$ then using a team of size $\beta^{O(1)}$ the failure probability can be made $< 1/4\beta^2$.

4. The sample size was too large for the algorithm of Lemma 9 to work: by using Chernoff bounds the constant C can be selected so that

$$\mathbf{Prob}\left(|\Psi| \geq 8\beta^3\right) \leq 1/4\beta^2 \quad .$$

It follows that the total failure probability for an iteration is at most $1/\beta^2$. With probability at least $1 - 1/\beta^{d_2+1}$, one of the first $\lceil \frac{d_2+1}{2} \rceil$ iterations will be successful, in which case the inductive hypothesis guarantees that the following recursive call will fully solve *Find_Max*(Φ) with probability at least $1 - 1/\beta^{d_2+1}$. The total success probability is therefore no less than

$$\left(1 - \frac{1}{\beta^{d_2+1}}\right)^2 > 1 - \frac{2}{\beta^{d_2+1}} \geq 1 - \frac{1}{\beta^{d_2}} \quad ,$$

which completes the inductive step.

Note that whenever Lemma 9 is applied, the algorithm may safely assume that the set size is $< 8\beta^3$, and hence a team of size $\beta^{O(1)}$ will suffice. ∎

5.3 Computing the Maximum Without a Team

Perhaps surprisingly, the memory array of a team is more significant than the processors of the team for the purpose of Algorithm RandMax.

Lemma 12 *Let Φ be an anonymous set. Let $\epsilon > 0$ be fixed. Then there exists a probabilistic algorithm which, using an auxiliary array of size $\beta \geq \Phi^{\epsilon}$, solves Find_Max($\Phi$) in $O(1)$ time with β-dominant probability.*

Proof The algorithm is carried out by emulating Algorithm RandMax using a *virtual* team of size β, which is created by the processors in Φ, as follows. Assume, without loss of generality, that $\epsilon < 1/2$. Each processor selects a random $i \in [1, \beta]$, and will function as the ith processor in the team. This step is successful if each $i \in [1, \beta]$ was selected by at least one processor $P \in \Phi$. This occurs with β-dominant probability. Clearly, a success in this step creates an *actual* team on ARBITRARY.

We observe that the only write operations done by team's processors are for computing OR or AND functions. This can be done by a virtual team on COLLISION, provided that the given array is initialized to values other than the collision values.

Comment: (i) It is easy to verify whether or not the creation of a (virtual) team is successful. (ii) The algorithm can be used even if ϵ is unknown: if it fails for β, we repeat it with an array of size $\sqrt{\beta}$, etc.

■

6 Simulating a Linear Memory Machine

In this section we show how Algorithm RandMax can be employed using random allocation methods for solving many instances of *Find_Max*.

Theorem 13 *There exists a solid algorithm,* MultiMax, *running on* COLLISION$(O(n), O(n))$ *which solves Find_Max$_{n,n}$ in $O(lglgn)$ time.*

In particular MultiMax can be used to produce a simulation of MAXIMUM(n, n) on COLLISION$(O(n), O(n))$.

Allocation of teams to instances of the problem is carried out using the following simple observation: as the number of not yet solved instances of the problem decreases, the amount of resources available for each remaining instance increases, and hence considerably more instances can be solved. In our case, the main resource for solving the instances is the processors array of size $O(n)$ which can be thought of as a team pool. We run MultiMax in iterations: in iteration t, the processors array is partitioned into w_t teams of size β_t. The iteration itself consists of the following steps:

Step1: *(recruit)* Each "active" instance of *Find_Max* selects a team at random. Using the collision detection property an instance can detect if it has selected a private team and, if this is the case, the team is *recruited* for solving that instance. Instances that failed in this step carry on to the next iteration.

Step2: *(solve)* Recruited teams are employed in algorithm RandMax with the following parameters:

- Team of size β_t.
- A *guess* that $|\Phi| < {\beta_t}^{d_1}$.
- Failure probability of $\beta_t^{-d_2}$.

All sets for which the RandMaxfailed continue to the next iteration.

The dependency of w_t (and consequently that of β_t) on t as well as the exact values of d_1 and d_2 are set in the analysis.

Analysis We carry out the analysis of MultiMax by showing that the number of "living instances" of the problem decreases at a doubly logarithmic rate. Specifically, let $N(t)$ denote the number of the non-empty anonymous sets for which *Find_Max* has not yet been solved at the beginning of iteration t ($t = 0, 1, \ldots$) of algorithm MultiMax, and let $\mathbf{N}(t)$ be a predetermined function of t:

$$\mathbf{N}(t) = 2^{2-2\cdot 1.5^t} n \quad .$$

Using induction on t we show that for a proper setting of w_t, β_t, d_1 and d_2, the event

$$N(t) \leq \mathbf{N}(t) \tag{3}$$

for all $t \geq 1$ is n-dominant.

Observe that if $N(t)$ is known to be less than $\sqrt[4]{n}$ then MultiMax can finish in $O(1)$ time: divide the processors' array into $n^{3/4}$ teams of size $\sqrt[4]{n}$ each. In the recruit step, with n-dominant probability, each instance is allocated with a team; RandMax is then invoked with *(i)* $|\Phi| \leq n$ *(ii)* failure probability n^{-2}. It is easy to verify that with n-dominant probability *all* instances of RandMax will succeed in solving *Find_Max*. We carry out the analysis under the assumption that $\mathbf{N}(t) \geq \sqrt[4]{n}$.

For the inductive base, $t = 0$, we have $\mathbf{N}(t) = n \geq N(t)$. To prove the inductive step, consider the reasons for an instance of *Find_Max* to remain active at the end of this iteration t:

1. The instance may have failed to recruit a team in Step1: The number of such failures can be shown to be $\leq 2N(t)^2/w_t$ with w_t-dominant probability [16]. In our setting of the parameters, $\mathbf{N}(t)$ is always $\leq w_t$ and hence this probability is n-dominant as well.

2. A set may be too large for the activation of RandMax in Step2, i.e., the guess was wrong: A simple counting argument shows that there are no more than $n/{\beta_t}^{d_1}$ wrong guesses.

3. The activation of RandMax in Step2 may fail even if its input is not too large: Recall that the failure probability is $\beta_t^{-d_2}$, and that all invocations of RandMax are independent. It follows from Chernoff bounds that the number of these failures is $\leq 2\mathbf{N}(t)\beta_t^{-d_2}$ with $\mathbf{N}(t)$-dominant probability, and hence with n-dominant probability as well.

By the inductive hypothesis, $N(t) \leq \mathbf{N}(t)$. Therefore, with dominant probability,

$$N(t+1) \leq \frac{2\mathbf{N}(t)^2}{w_t} + n{\beta_t}^{-d_1} + 2\mathbf{N}(t)\beta_t^{-d_2}$$

and since $\mathbf{N}(t) \leq n$,

$$N(t+1) \leq \frac{2\mathbf{N}^2(t)}{w_t} + 3n{\beta_t}^{-\min(d_1,d_2)} \quad . \tag{4}$$

We set $d_1 = d_2 = 4$, $\beta_t = 2^{1.5^t - t/5}$, and $w_t = 2^{4-t/1.5}n/\beta_t$. Note that the number of teams decreases at a doubly exponential rate, which however is slower than the decrease of $\mathbf{N}(t)$, and that the team size increases at an almost doubly exponential rate. Also note that $w_t\beta_t$, the total number of processors used in the teams, decreases in an exponential rate; hence, each processor participates in a team $O(1)$ times during the algorithm.

By examining the first term in the bound of Inequality (4) we get

$$\frac{2\mathbf{N}^2(t)}{w_t} = 2 \cdot 2^{-3\cdot 1.5^t - t/1.5}n$$

and as can be easily verified

$$\frac{2\mathbf{N}^2(t)}{w_t} > 3 \cdot 2^{-4\cdot 1.5^t}n = 3n{\beta_t}^{-\min(d_1,d_2)} \quad .$$

Hence,

$$N(t+1) \leq \frac{4\mathbf{N}^2(t)}{w_t} = \frac{1}{4}2^{1.5^t}2^{4-4\cdot 1.5^t}n = 2^{2-3\cdot 1.5^t}n = \mathbf{N}(t+1)$$

which completes the inductive step in proving Inequality (3) and Theorem 13.

7 The General Simulation

Conducting Algorithm MultiMax for the general case $m \geq n$ is possible provided only that each non-empty Φ_i, $i \leq m$, acts as one in the recruiting step. A possible solution for $Find\_Max_{m,n}$ can thus be done by appointing a leader (i.e., a team of size 1) to each non-empty anonymous set Φ_i. As shown in [17], an $O(\lg\lg n)$ hashing algorithm can be used for (external) leader selection: using memory addresses as keys, the range m can be reduced to $O(n)$. However, when keys are not distinct (as it is the case in our problem), this hashing algorithm uses the (relatively strong) COLLISION$^+$ model.

We observe that unanimous random operation is somewhat easier than arbitration and it can be achieved without a leader by using pseudo-random functions. Specifically, in iteration t we build a global hash function h_t: $[0, m-1] \mapsto [0, w_t - 1]$ for coordinating the recruit step (Step1) of processors that belong to the same instance of *Find_Max*; all the processors in the set Φ_i will select the team numbered $h_t(i)$.

It was shown in [18] that it is possible to construct in constant time a function for which the number of collisions is at most $4N(t)^2/w_t$ with n-dominant probability. In order for the analysis of MultiMax to remain valid, we use twice as many teams of the same size in each iteration (i.e., $w'_t = 2w_t$ and $\beta'_t = \beta_t$). This increase in the assumed number of processors can be translated to doubling the running time.

Algorithm MultiMax has to be further modified because collision will be sensed by all sets with more than one element as well. We therefore apply Step2 in all cases and add another step which implements late detection of recruiting failure:

Step3 The processor selected by Step2, writes the name of the set it belongs to into a designated place in the team private memory region. Sets that detect that this processor was not among their members, continue to the next iteration.

7.1 Eliminating concurrent memory access

A simulation of a write step of MAXIMUM$_{n,m}$ is based on first running RandMax for each anonymous set, and then having the selected leader write its value. It is clear that there is never a concurrent write into any of the m memory cells of the simulated machine. (The only concurrent write operations take place in the $O(n)$ memory used by the simulation algorithm.) We now seek a method for eliminating concurrent reads from these cells.

Note that whenever a processor decides during the execution of algorithm MultiMax that it does not have the maximum value, it does so after reading a cell with a greater value belonging to a processor from the same anonymous set. The algorithm never reuses this cell, and hence it can be used *later* for data broadcast from a processor with a high value to all processors subsumed by it. It follows that MultiMax generates a channel of broadcast from every elected leader to all of its sets members. Transmitting data along this channel takes $O(\lg\lg n)$ time.

Suppose that the read step is also simulated using MultiMax, a processor P_i using i

as processor value. Then the only processor which needs to access a cell of the simulated machine is the selected leader of the anonymous set of readers from this cell. This set leader can now broadcast the cell value to all set members.

8 Implementing the Simulations on TOLERANT

The simulation algorithm as presented assumed the COLLISION model. Grolmusz and Ragde in [23] showed that in general, COLLISION is strictly stronger than TOLERANT. However, the fact that only linear memory was used, enables an implementation on TOLERANT with no time increase. In this section we show that *any* algorithm running on COLLISION which uses linear memory can be run with no time increase on TOLERANT. All algortihms presented in this section are determinstic. Similar algorithms were obtained independently in [24].

8.1 Simulating COLLISION on TOLERANT

Grolmusz and Ragde in [23] described an algorithm illustrating the strength of the TOLERANT model in computing the OR function in $O(1)$ time. Extending their algorithm we show

Lemma 14 COLLISION$(n, n) \preceq$ TOLERANT$(2n, n)$.

Proof The first n processors of the simulating machine, are used for a step by step simulation. The other n processors are allocated one for each memory cell. Let M be a fixed cell, P_M the processor allocated to it, and Φ the set of processors in the simulated machine trying to write into that cell. The write step of COLLISION is simulated by the following steps:

1. P_M saves the previous value of M, and then clears it.
2. Each processor in Φ writes its value into M. Simultaneously, P_M writes some value into M.
3. If M is non-blank then $\Phi = \emptyset$ and P_M restores the old value of M. This terminates the simulation for that cell in this case.
4. All processors in Φ write their values into M.
5. If M remains blank then $|\Phi| > 1$, and P_M writes $\sharp$ into that cell, otherwise $|\Phi| = 1$, and M has the correct value.

■

Comment: The algorithm assumes that there is a reserved "blank" value which none of the processors in Φ tries to write into M.

8.2 Simulating TOLERANT on TOLERANT

Simulating TOLERANT on TOLERANT with fewer processors is not immediate as it is for the other models. The difficulty arises from the fact that simultaneous writing on TOLERANT behaves differently than sequential writing, and this difference is used in the above algorithm, so emulating two processors by one is not immediate. The following lemma shows how to circumvent this obstacle.

Lemma 15 *For any positive integer k,* $\text{TOLERANT}(kn, n/2) \preceq \text{TOLERANT}(n, n/2)$.

Proof The first $n/2$ processors of the simulating machine will each play the parts of $2k$ processors in the simulated machine. The remaining $n/2$ processors are allocated to the $n/2$ memory cells, one for each cell. Let M, P_M and Φ be as in Lemma 14. Each write step is simulated by the following stages:

1. P_M saves the previous value of M, and then executes $M \leftarrow 0$.
2. Emulators of members of Φ together with P_M cooperate to perform the assignment

$$M \leftarrow \begin{cases} |\Phi| & \text{If } |\Phi| \leq 1 \\ 2 & \text{otherwise} \end{cases} .$$

3. If the value of M is 1 then the simulation for that cell terminates after the single member of Φ writes its value. The stage consists of $2k$ emulation rounds in which each emulating processor acts for his $2k$ emulated processors.
4. P_M restores the old value of M if its content immediately after Stage 2 was 0 or 2.

Stage 2 is carried out in $2k$ rounds. In each round the technique described in the above proof enables P_M to determine whether the $n/2$ processors emulated in the round contain 0, 1 or at least 2 processors. By the end of the $2k$ rounds P_M knows enough about Φ to perform the assignment. ∎

If memory is larger than $n/2$, this procedure can be activated for each $n/2$ memory segment in turn, giving:

Corollary 16 *If $m = O(n)$ then* $\text{TOLERANT}(O(n), m) \preceq \text{TOLERANT}(n, m)$.

Combining this with Lemma 14 completes the proof of Theorem 5.

Comment: Related results were independently described by Hagerup. The interested reader is refered to [25] for further discussion and study of the self-simulation question.

9 Achieving Optimal Speedup

In order to get a simulation algorithm that uses an optimal number of processors, it is enough to describe a pre-processing algorithm in which the number of "active" items decreases at least geometrically. Given such algorithm the number of active items can be reduced to $p = n/\lg\lg n$ in $o(\lg\lg n)$ time, using p processors, by applying an "optimizer" [35]. After this stage, the non-optimal algorithm can be employed.

In each iteration of the algorithm a constant fraction of the active elements become inactive. An iteration consists of the following. For each of the m instances of *Find_Max* we allocate a *virtual array* of size $2\lg n$. We assume first that the virtual arrays reside in a space of size $2m\lg n$. Later on we show how they can be implemented using $O(n)$ memory. Into each array we do a *geometric decomposition* of the anonymous set associated with it: each processor selects the ith array cell with probability 2^{-i}; with probability $2^{-2\lg n}$, no cell is selected. A processor which selects a cell tries to write its name into that cell. If exactly one of the processors succeeds in writing its name, it is called a *pivot*. Note that if a pivot exists then it is selected at random. The following fact was proved in [22].

Fact 17 *There exists a constant $C > 0$, such that for a set Φ, $|\Phi| \leq n$, a pivot is selected with probability at least C.*

After the geometric decomposition, the pivot writes its name in a special predetermined cell (for the set) called a *throne*, and all processors with values less than the pivot's value deactivate themselves. If elements other than the pivot remain active in the set, they notify that to the pivot. If the pivot is the only remaining element, it is the elected set leader, and it deactivates itself.

Lemma 18 *Suppose that a random pivot has been selected for a set with k active elements. Let k' be the number of elements that remain active. Then*

$$\mathbf{E}(k') = \frac{k}{2} + \frac{1}{2} - \frac{1}{k} \ .$$

Proof Rank active processors by value, with the highest valued processor ranked 1. Then

$$\mathbf{E}(k') = \frac{1}{k}\left(0 + \sum_{i=2}^{k} i\right) = \frac{(k-1)(k+2)}{2k} = \frac{k}{2} + \frac{1}{2} - \frac{1}{k} \ .$$

■

Comment: The variance of k' is given by

$$\begin{aligned} \mathbf{Var}(k') &= \mathbf{E}\left(k'^2\right) - \mathbf{E}^2(k') \\ &= \frac{1}{k}\left(0 + \sum_{i=2}^{k} i^2\right) - \mathbf{E}^2(k') \end{aligned}$$

$$= \frac{1}{k}\left(\frac{k(k+1)(2k+1)}{6} - 1\right) - \left(\frac{k}{2} + \frac{1}{2} - \frac{1}{k}\right)^2$$
$$= \frac{k^2}{12} + \frac{11}{12} - \frac{1}{k^2} \quad .$$

Combining Fact 17 and Lemma 18 we get

Corollary 19 *There exists a constant $\alpha < 1$ such that if a set is of size k then the expected number of set elements remaining active after the geometric decomposition is at most αk.*

The analysis of the total number of active items is given assuming that sets are not too large. Observe that our only concern is when $K \geq n/\lg\lg n$, otherwise we are done.

Lemma 20 *Let K be the total number of active elements at the beginning of an iteration, and assume that $K > n/lglgn$. Let K' be their number by the end of the iteration. Suppose that all anonymous sets are of size at most $n^{3/4}$. Then, there exists a constant $\gamma < 1$, such that $K' \leq \gamma K$ with n-dominant probability.*

Proof Denote by k_i (respectively k_i') the number of active elements in the set Φ_i at the beginning (respectively at the end) of the iteration. Then $K' = \sum_{i=1}^{m} k_i'$, and since all the k_i' are independent

$$\mathbf{Var}\,(K') = \sum_{i=1}^{m} \mathbf{Var}\,(k_i') \quad .$$

Also, $\mathbf{Var}\,(k_i') \leq k_i^2$, and hence (by convexity arguments) the above sum is maximized if all the non-zero k_is equal $n^{3/4}$. Thus

$$\mathbf{Var}\,(K') \leq n^{1/4}(n^{3/4})^2 = n^{7/4} \quad .$$

We have $\mathbf{E}\,(K') \leq \alpha K$ and $K \geq n/loglgn$. The lemma follows by using Chebyshev inequality. ∎

Bounding the size of sets To obtain $k_i \leq n^{3/4}$ for all $i = 1, \ldots, m$, we apply the following $O(\lg\lg n)$ time pre-processing step before the iterations. A sample (with replacements) of size $n/\lg\lg n$ is randomly taken from the input, by having each of the $n/\lg\lg n$ simulating processors select one of the n simulated processors at random. We run the non-optimal algorithm for this sample. For each anonymous set, consider the leader selected for its subset in the sample, if this subset is not empty. Deactivate all processors in the set whose values are less than that of the subset leader. To implement this step, we build a hash table for the leaders, using the hashing algorithm of [17], with the $O(n/\lg\lg n)$ leader IDs being the input set. Since this set consists of distinct keys, the hashing algorithm can be implemented on COLLISION$(n/\lg\lg n, O(n/\lg\lg n))$ and hence, by Theorem 5, on TOLERANT$(n/\lg\lg n, O(n/\lg\lg n))$. Its time complexity is $O(\lg\lg n)$ with n-dominant probability [18].

Lemma 21 *Let k_i be the number of active elements in Φ_i after the pre-processing step. Then with n-dominant probability, for all $i = 1, \ldots, m$, we have $k_i \leq n^{3/4}$.*

Proof For a set Φ_i

$$\textbf{Prob}\left(k_i \geq n^{3/4}\right) = \left(1 - \frac{n^{3/4}}{n}\right)^{n/\lg\lg n} < e^{-n^{3/4}/\lg\lg n} \quad .$$

The probability that there exists i for which $k_i \geq n^{3/4}$ is at most

$$ne^{-n^{3/4}/\lg\lg n} = o(1/n) \quad .$$

■

Implementing virtual arrays It remains to be shown how the virtual arrays can be implemented in $O(n)$ space. Let $\bar{K}$ be a (known) bound on the number of active elements at the beginning of an iteration. Note that no more than $\bar{K}$ cells in the virtual arrays are actually used. We map these cells into an array of size $O(\bar{K})$, by means of hash function. (The sequence of $\bar{K}$s will be shown to be decreasing geometrically in time, implying that the total memory used for the hashing is $O(n)$.) We call the cell into which the pivot is writing a *throne*. Whenever this mapping causes a throne to collide, the pivot associated with this throne "fails" and all active elements in the cell remain active. It follows from [18] that if all sets have fewer than $n^{3/4}$ elements, then a hash function can be constructed in constant time such that, using this function, at most a constant fraction of the active elements have colliding pivots, with n-dominant probability. Combining with Lemma 20 we get that with high probability, a constant fraction of the remaining active elements are deactivated. The required geometric decrease is thus achieved.

Model of computation The algorithm described above can be implemented in a straightforward manner on the COLLISION model of computation, thus achieving an optimal double logarithmic solid simulation of MAXIMUM on COLLISION. Implementation on TOLERANT, however, is more subtle. The difficulty is that it is not known, *in general*, how to simulate TOLERANT(n) on TOLERANT(p) in $O(n/p)$ time. Since not all write operations of the simulated processors can be implemented simultaneously, it might occur that a simulated processor succeeds in writing even when other simulated processors are writing to the same cell. This may obstruct collision detection in the algorithm. Indeed, the optimizer in [35] assumes that OR computation can be done in constant time. It is not known how to compute the OR function efficiently in constant time on TOLERANT without collision detection. In addition, the selection of pivots in the algorithm above uses collision detections—the pivot is *the only* element in a set that does not collide after the geometric decomposition. We show below how to implement an optimizer and the pivots selection step on TOLERANT.

Optimizer For an implementation on TOLERANT we replace the optimizer. It has been shown in [17] that if the number of active elements decreases at least geometrically, a load balancing algorithm can be used as an optimizer. Specifically, load balancing should only be invoked $O(\lg^* n)$ times. Using the $O(\lg\lg n)$ time load balancing algorithm of [15], which runs on ROBUST (a model weaker than TOLERANT), we get an optimal $O(\lg\lg n \lg^* n)$ time solid implementation of the simulation algorithm on TOLERANT.

Pivots selection Assume that a step of the simulated machine is executed in r rounds. A set Φ_i is partitioned into r subsets $\Phi_i^1, \ldots, \Phi_i^r$ where Φ_i^j consists of the processors in Φ_i that are simulated in round j. The idea is to consider the collection of rm subsets $\left\{\Phi_i^j\right\}$ instead of the m sets $\{\Phi_i\}$. The pivot selection process for each subset is done in one round and is therefore identical to the algorithm described above. The analysis of decrease in the number of active elements is therefore valid also for the collection $\left\{\Phi_i^j\right\}$. One issue is still common to all subsets: a pivot must learn whether it is alone in its set, in which case it is selected leader and deactivates itself. In order to enable this, the throne will be common to all subsets, by using the same hash function in all rounds. The implementation is then straightforward: the common throne is used first to keep only the highest value pivot for each set and then, by executing a second sequence of rounds, to notify the pivot if it is not alone in its set. A consequence is that collisions may now occur with thrones occupied in *previous* rounds. Yet this is of no obstacle as in the analysis (of the non-optimal algorithm) such collisions were already taken into account.

Acknowledgments

We thank Faith Fich for valuable comments on a previous version of this paper. Stimulating discussions with Uzi Vishkin are gratefully acknowledged.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *Proc. International Conference on Parallel Processing*, pages 175–179, 1983.

[2] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing (STOC '87)*, pages 83–93, 1987.

[3] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing (STOC '89)*, 1989.

[4] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989.

[5] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. Technical Report TR 15/1989, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, W. Germany, November 1989.

[6] R. B. Boppana. Optimal separations between concurrent-write parallel machines. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing (STOC '89)*, pages 320–326, 1989.

[7] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–507, 1952.

[8] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *Proc. of 13th Symposium on Mathematical Foundations of Computer Science, Springer LNCS 324*, pages 231–239, 1988.

[9] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. In *FCT '89*, pages 95–104, 1989.

[10] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[11] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[12] F. E. Fich, P. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, June 1988.

[13] F. E. Fich, P. L. Ragde, and A. Wigderson. Simulations among concurrent-write PRAMs. *Algorithmica*, 3:43–51, 1988.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31:538–544, July 1984.

[15] J. Gil. Fast load balancing on PRAM. Manuscript, also in Technical Report 91-14, The University of British Columbia, Nov. 1990.

[16] J. Gil. *Lower Bounds and Algorithms for Hashing and Parallel Processing.* PhD thesis, The Hebrew University of Jerusalem, Israel, Nov. 1990.

[17] J. Gil and Y. Matias. Fast hashing on a PRAM. In *SODA '91*, pages 271–280, Jan. 1991.

[18] J. Gil and Y. Matias. Polynomial hash functions are reliable. Manuscript, Aug. 1991.

[19] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *to appear in the Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, Oct. 1991.

[20] J. Gil and L. Rudolph. Counting and packing in parallel. In *Proc. International Conference on Parallel Processing*, pages 1000–1002, 1986.

[21] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *J. ACM*, 29(4):1073–1086, July 1982.

[22] A. G. Greenberg and R. Ladner. Estimating the multiplicity of conflicts in multiple access channels. In *Proc. of the 24th IEEE Annual Symp. on Foundation of Computer Science*, pages 384–392, 1983.

[23] V. Grolmusz and P. L. Ragde. Incomparability in parallel computation. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 89–98, 1987.

[24] V. Grolmusz and P. L. Ragde. Incomparability in parallel computation. *Discrete Applied Mathematics*, 29:63–78, 1990.

[25] T. Hagerup. Self-simulation on the PRAM. Manuscript, Oct. 1990.

[26] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '90)*, pages 117–124, 1990.

[27] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1991.

[28] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS) U. C. Berkeley, 1988.

[29] C. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. on Comp*, C-32:942–946, 1983.

[30] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Comput. Sci.*, 71:95–132, 1990.

[31] L. Kučera. Parallel computation and conflicts in memory access. *Inf. Process. Lett.*, 14:93–96, 1982.

[32] P. MacKenzie and Q. Stout. Ultra-fast expected time parallel algorithms. In *SODA '91*, 1991.

[33] C. U. Martel and D. Gusfield. A fast parallel quicksort algorithm. *Inf. Process. Lett.*, 30:97–102, 1989.

[34] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 729–743, 1990. Also to appear in *Journal of Algorithms.*

[35] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing (STOC '91)*, pages 307–316, 1991. Also in UMIACS-TR-91-65, Inst. for Advanced Computer Studies, Univ. of Maryland.

[36] P. L. Ragde. The parallel simplicity of compaction and chaining. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 744–751, 1990.

[37] P. L. Ragde, W. Steiger, E. Szemerédi, and A. Wigderson. The parallel complexity of element distinctness is $\Omega\left(\sqrt{\lg n}\right)$. *SIAM Journal on Disceret Mathematics*, 1(3):399–410, Aug. 1988.

[38] R. Reischuk. A fast probabilistic parallel sorting algorithm. In *Proc. of the 22nd IEEE Annual Symp. on Foundation of Computer Science*, pages 212–219, 1981.

[39] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2:88–102, 1981.

[40] Y. Shiloach and U. Vishkin. An $O(\lg n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[41] L. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[42] U. Vishkin. On choice of a model of parallel computation. Technical Report TR 61, Dept. of Computer Science, Courant Institute, New York University, 1983.

[43] U. Vishkin. Synchronous parallel computation – a survey. Technical Report TR 71, Dept. of Computer Science, Courant Institute, New York University, 1983. Also: Annual Paper Collection of "Datalogforeningen" (The Computer Science Association of Aarhus, Denmark), 1987, 76-89.

[44] U. Vishkin. Research on parallel algorithms. In *UMIACS 1989 Annual Report.* , 1990.

[45] D. Willard. Log-logarithmic selection resolution protocols in a multiple access channel. *SIAM J. Comput.*, 15:468–477, 1986.