Towards a Theory of Nearly Constant Time Parallel Algorithms*

Joseph Gil
The University of British Columbia

Yossi Matias†
The University of Maryland
and Tel Aviv University

Uzi Vishkin†
The University of Maryland
and Tel Aviv University

Abstract

In this paper we demonstrate that randomization is an extremely powerful tool for designing very fast and efficient parallel algorithms. Specifically, a running time of $O(\lg^* n)$ ("nearly-constant"), with high probability, is achieved using $n/\lg^* n$ ("optimal speedup") processors for a wide range of fundamental problems, including: (1) support dictionary operations: insert, delete, lookup; this is the first optimal dictionary algorithm that is in RNC! (2) load balancing (an essential tool for parallel computation); (3) problems considered in [33], including hashing, leaders election, linear approximate compaction and generation of random permutations; (4) simulation of MAXIMUM (a powerful CRCW PRAM model) on TOLERANT (a weak CRCW PRAM model); (5) integer chain sorting. We also give a constant time algorithm which, using n processors, approximates the sum of n positive numbers to within an error which is smaller than the sum by an order of magnitude. A variety of known and new techniques are used. New techniques, which are of independent interest, include estimation of the size of a set in constant time for several settings, and ways for deriving super-fast optimal algorithms from super-fast non-optimal ones.

1 Introduction

An ultimate goal for parallel algorithmics is achieving constant time parallel algorithms whose number of processors equals the serial complexity of the problems being considered. As a compromise, we would be satisfied with "nearly-constant" time parallel algorithms whose time-processor product (i.e., number of operations) equals this serial complexity. In view of several lower bounds for very simple problems, such as computing the parity of n bits, prospects for even approaching this goal for a wide range of problems were unclear.

Nevertheless, we feel that the techniques and results used and presented in this paper in their entirety form "a theory" of nearly-constant time parallel algorithms; the key that enabled this knowledge-base to grow into what deserved to be called a theory is "parallel randomization". The majority of the paper is devoted to description of algorithms concretely emphasizing new contributions. Section 1.2, however, takes a broader perspective.

1.1 Results

We summarize the main results, each preceded by a short description of the problem, and followed by a brief literature review. All algorithms mentioned below are randomized and perform within the stated time bounds with high probability.

Dictionary The *dictionary* problem is to maintain a data structure which supports the *insert*, *delete*, and *membership query* instructions. *Result:* $O(\lg^* n)$ time[1] for a batch of n instructions. using $n/\lg^* n$ processors (optimal speedup). *Previous Work:* a deterministic 2–3 tree parallel data structure, $O(n \lg n)$ operations and $O(\lg n)$ time [35] (non-optimal); a dynamic hashing data structure [11], optimal ($O(n)$ operations) but not in RNC ($O(n^\epsilon)$ time).

Hashing Given a set $S \subseteq U = \{0, \ldots, Q-1\}$, $|S| = n$, the *hashing* problem is to find a one-to-one function $h : S \mapsto [1, dn]$ (for some constant $d \geq 1$) such that h is represented in $O(n)$ space and for any $x \in U$, $h(x)$ can be evaluated in constant time. *Result:* $O(\lg^* n)$ time using $n/\lg^* n$ processors (optimal speedup). *Previous Work:* Parallel hashing: [11, 19, 18, 32, 33]. [33]: $O(\lg^* n \lg(\lg^* n))$ expected time, using an optimal number of processors. [19]: $\Omega(\lg^* n)$ expected time using n processors for a model of parallel computation that fits our algorithm.

Approximate Sum Given n numbers, compute an estimate for their sum which is accurate to within a factor of $(1 + \beta)$ where $\beta = o(1)$. *Result:* Constant time algorithm on an n-processor machine with $\beta = 1/\lg^k n$ for any constant k. Note that the complexity of exact sum is $\Theta(\lg n / \lg\lg n)$; [2]: lower bound (that can be shown to hold for probabilistic algorithms); [10]: deterministic algorithm.

[1]Let $\lg^{(i)} x \equiv \lg(\lg^{(i-1)} x)$ for $i > 1$, and $\lg^{(1)} x \equiv \lg x$; $\lg^* x \equiv \min\{i : \lg^{(i)} x \leq 2\}$. The function $\lg^*(\cdot)$ is extremely slow increasing and for instance $\lg^* 2^{65536} = 5$.

*To Michael O. Rabin for [36].

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

Load Balancing Given m objects distributed among n processors, redistribute the objects so that each processor gets $O(1+m/n)$ objects. *Result:* $O(\lg^* n)$ time for the load balancing as well as for a more general "interval allocation" problem (defined by [26]). *Previous Work:* Nearly-logarithmic time deterministic algorithms: [9, 39]. $O(\lg\lg n)$ time: [15]; $O(\lg\lg n \lg^* n/\lg\lg\lg n)$ time (also for the interval allocation problem): [26].

Integer Chain-Sorting Given n integers from $[1,n]$ create a sorted linked list consisting of all inputs. *Result:* $O(\lg^* n)$ time using $n/\lg^* n$ processors (optimal speedup). *Previous Work:* [27]: definition of the problem and an optimal speedup algorithm of $O(\lg\lg n \lg^* n/\lg\lg\lg n)$ time.

CRCW **Simulations** Simulate the computation of a powerful n-processor CRCW-PRAM sub-model on a weaker one. *Result:* Simulating an n-processor MAXIMUM on an $(n/\lg^* n)$-processor TOLERANT in $O(\lg^* n)$ time and $O(n\lg n)$ space (optimal speedup), and on an $(n/(\lg^* n)^2)$-processor TOLERANT in $O((\lg^* n)^2)$ time and $O(n)$ space (optimal speedup). *Previous Work:* [6, 7, 13, 16, 17, 29, 32]. The best time previously achieved for any non-trivial simulation is $O(\lg\lg n)$.

Compaction Given a set of at most m objects with IDs from $[1,n]$, allocate new IDs from the range $[1,O(m)]$. (This problem is also known as the *linear approximate compaction* (LAC) problem or the *renaming* problem.) *Result:* $O(\lg^* n)$ time using $n/\lg^* n$ processors. This implies a similar result for the *random permutation* problem (see [33]). *Previous Work:* [20, 28, 15, 18, 33, 25]. [33]: $O(\lg^* n)$ time using n processors and $O(\lg^* n \lg(\lg^* n))$ time using an optimal number of processors.

Leaders Election Given a set of n items and their partition into m subsets (some empty) such that each item knows to which subset it belongs, the *leaders election* problem is to select for each subset a unique item ("leader") from the subset, using a space that is bounded by some function of n (but not of input values). *Result:* $O(\lg^* n)$ time using $n/\lg^* n$ processors and $O(n)$ space on ARBITRARY, $O(\lg^* n)$ time using $n/\lg^* n$ processors and $O(n\lg n)$ space on TOLERANT, and $O(\lg^* n)^2$ time using $n/(\lg^* n)^2$ processors and $O(n)$ space on TOLERANT. *Previous Work:* [4, 17, 32, 33]. [4]: $\Omega(\lg n/\lg\lg n)$ time deterministic lower bound for the (easier) problem of *element distinctness*, using n processors on PRIORITY. [17]: $O(\lg\lg n)$ time using n processors, and $O(\lg\lg n \lg^* n)$ time using an optimal number of processors; both algorithms are on TOLERANT and use $O(n)$ space. [33]: $O(\lg^* n \lg(\lg^* n))$ expected time using an optimal number of processors and $O(n)$ space on ARBITRARY.

Optimization Schemes Several families of under-specified parallel algorithms are considered. Given an algorithm that is designed to work with a non-optimal number of processors, an *optimization scheme* enables to convert it into an algorithm that uses an optimal number of processors. An *optimization scheme* emulates efficiently any algorithm taken from a broad family, providing a constructive implementation for the methodology which is guided by Brent's theorem. *Result: (i)* $O(\lg^* n)$ slowdown for a certain (rather general) type of algorithms called "loosely specified"; *(ii)* $O(\lg^* n \lg^*(\lg^* n))$ additive time overhead for "geometric-decaying" algorithms; and *(iii)* $O(\lg^* n)$ additive time overhead for "asynchronous geometric-decaying" algorithms. (This scheme is suitable for all algorithms considered in this paper.) *Previous Work:* [25]: $O(\lg\lg n \lg^* n/\lg\lg\lg n)$ slowdown. [33]: $O(\lg^* n \lg(\lg^* n))$ additive time overhead for "task-decaying" algorithms. *Concurrent Work:* [22]: $O(\lg^* n)$ slowdown. Our scheme for the "loosely-specified" algorithm is more general than the schemes in [22, 25] without compromising performance.

Combined Simulation Result The results in this paper give ways for: *(i)* efficient simulation of a strong model of computation by a weaker one, *(ii)* simulate an algorithm that uses much space by little space (using the dictionary algorithm), and *(iii)* ignore processors allocations issue (using the optimization schemes). This gives much leverage to designer of parallel algorithms. One may choose to design a parallel algorithm on a powerful CRCW model, use large space, and ignore processors allocations issues; still, the algorithm may automatically fit a weaker CRCW PRAM model, become space efficient, *and* take care of allocating processors. The only overhead is, with high probability, a nearly-constant time slowdown.

1.2 The evolving theory

Below we give an overview of the evolving theory of nearly constant time randomized parallel algorithms with forward references to the sections where some of the algorithmic details are described in the paper.

The definitions of the next subsection classify confidence in probabilistic analysis into polynomial and exponential. Precise degrees of polynomials and exponent parameters are of lesser significance.

Micro-level concepts and techniques that are mentioned in Section 3 include: definitions of teams and anonymous sets, the scattering idea, initial estimations, geometric decomposition, maximum finding, and table lookup. Section 4 makes more contributions for estimating the size of an anonymous set (that is, approximating the number of one's in a large array, with a processor allocated to each '1'). This problem enables sometimes replacement of an exact counting that is inherently slow, as implied by the $\Omega(\lg n/\lg\lg n)$ lower bound of Beame and Hastad [2]. Constant time algorithms for estimation under different settings are given. Further advancement for replacing exact sum is made in Section 6, where the sum of n numbers is estimated in constant time to within a small order error.

There are several macro-level techniques. The dominant paradigm that enables $O(\lg^* n)$ ("nearly-constant") time results throughout the paper is presented in Section 3.2. Its use for the compaction (LAC) problem and its extension follows. In Section 5 we apply these techniques, together with other ideas, to get a nearly-constant time algorithm for the load balancing problem—a fundamental difficulty in parallel computation.

As an application, we design an "optimizer" (given in Section 8) that can automatically adapt parallel algorithms of a certain (rather general) type to become processor-efficient. Another optimizer enables to derive $O(\lg^* n)$ time and optimal speedup results for the main problems considered in this paper: Taking advantage of some asynchrony property of algorithms this second optimizer recalls an "older" speedup paradigm. Some algorithms actually have to be redesigned to satisfy this asynchrony property. Leaders election, hashing (Section 7), integer chain-sorting (Section 9), and simulation results (Section 10)—all in nearly-constant time with high probability—come next.

Finally, our most involved result so far is a an algorithm implementing the classical dictionary data type in Section 2. A number of low-level and high-level new ideas and considerations play a role in this algorithm. In some sense this algorithm is the high point of the evolving theory.

Figure 1 demonstrates part of the relationships between the algorithms and techniques considered in this paper (excluding optimization schemes).

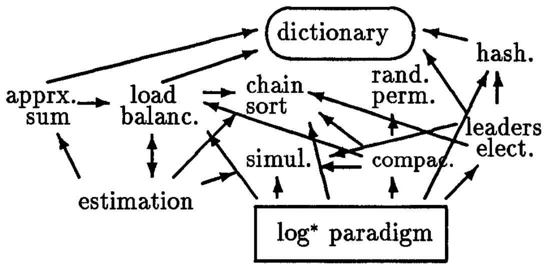

Figure 1: Relations among nearly constant time algorithms.

1.3 Analysis of randomized algorithms

In the present extended abstract we use the following, somewhat coarse, terminology for statements regarding performance of randomized algorithms: an event $A = A(n)$ is *n-negligible* if $\mathbf{Prob}(A) = o(n^{-\epsilon})$ for some constant $\epsilon > 0$. We say that the complement event $\bar{A}$ is ***n*-*polynomial overwhelming***, or n-polynomial, in short, and the event is said to occur with *n-polynomial probability*.

We will be content with showing that algorithms for a problem of size n succeed with n-polynomial probability (although somewhat higher success probabilities are obtainable occasionally). Accordingly, in the algorithms analysis we will ignore events that are n-negligible.

We say that an event $A = A(n)$ is ***n*-*exponential*** if $\mathbf{Prob}(A) \geq 1 - o(2^{-n^{\epsilon}})$ for some constant $\epsilon > 0$, and the event is said to occur with ***n*-*exponential*** *probability*.

Many of the random variables we will be dealing with are such that the probability they deviate significantly from their expected value is very small. This is by large a result of Chernoff and Hoeffding bounds[2], which using the above terminology are given as follows.

Let $x_1, \ldots, x_n$ be independent random variables, $x = \sum_{i=1}^{n} x_i$, and $\bar{x} = \mathbf{E}(x)$.

Fact 1.1. (Chernoff) *If $x_i \in \{0,1\}$ then for all ϵ, $0 < \epsilon < 1$, we have $(1-\epsilon)\bar{x} \leq x \leq (1+\epsilon)\bar{x}$ with $(\epsilon^2\bar{x})$-exponential probability.*

A generalization to the Chernoff bounds is

Fact 1.2. (Hoeffding) *If $x_i \in [0, n^{\delta}]$ for some $\delta < 1$ and $\bar{x} = \Omega(n)$ then, for all fixed λ, $\lambda \geq e$, we have $x \leq \lambda\bar{x}$ with n-exponential probability.*

2 Dictionary

A *dictionary* data structure supports the instructions *insert*, *delete*, and *lookup* of *keys* that are drawn from some finite *universe* U. Let $U = \{0, \ldots, Q-1\}$, where Q is assumed to be prime. A *semi-dictionary* supports only the lookup and insert instructions. A *parallel dictionary* handles one *batch* of keys at a time; each batch consists of an array of *keys* and an instruction: *insert*, *delete*, or *lookup*. The parallel dictionary processes a batch using several processors. For the sake of brevity, henceforth *dictionary* will mean *parallel dictionary*

Theorem 2.1. *There exists a dictionary which, using p processors, has the following features with p-polynomial probability:*

(a) At all times, the total space used by the dictionary is linear in the number of keys currently *stored in the dictionary.*

(b) Any batch of $p\lg^* p$ keys is processed in $O(\lg^* p)$ time (optimal speedup).

(c) A lookup instruction for a batch of p keys is processed in constant time (optimal).

[2] Sometimes, the Martingale Tail Inequality (Azuma's Theorem) is applied for fine analysis of certain algorithmic details.

(d) If an additional storage space of size $O(p)$ is available then a batch of $p/\lg^{(i)} p$ keys is processed in $O(i)$ time.

2.1 Overview

The basic data structure The structure of the dictionary is based on the 2-level hashing scheme of Fredman, Komlós, and Szemerédi [14]. Let $S \subseteq U$ be the set of keys currently stored in the dictionary. A *first-level* function $f : U \rightarrow [1, N]$, $N = O(|S|)$, partitions S into N *buckets*. Bucket $B_i = B_i(f)$ is defined as $f^{-1}(i) \cap S$; in words, the set of keys in S mapped by f into i. Each bucket B_i has a private memory block M_i and a private *second-level* function g_i; the keys in B_i are mapped by g_i into M_i in a one-to-one manner. Given any $x \in U$ the scheme maps it into a memory cell $M_{f(x)}[g_{f(x)}(x)]$. *The mapping is injective with respect to S*, since no two keys from S are mapped into the same memory cell. The scheme stores each key $x \in S$ in its memory cell. Thus, each cell in each of the memory blocks is either empty or has a single key from S. The function g_i, as well as a pointer to the memory block M_i, are stored in a (bookkeeping) array of size N. Given any $x \in U$ the instruction lookup(x) is carried out in constant-time by examining cell $M_{f(x)}[g_{f(x)}(x)]$.

Updates to the data structure Deletions are easy: A key in S is deleted by simply emptying its associated memory cell. Insertions, however, will add new keys to buckets, and some second-level functions may map several keys to the same memory cell, violating the injectiveness of the mapping. Let S' be a set of new keys to be inserted ($S' \cap S = \emptyset$). We say that a key $x \in S'$ *affects* B_i if $i = f(x)$. All affected buckets are discarded from the dictionary and all their keys are retrieved for reinsertion. These buckets are allocated new memory blocks and new second level functions. Other buckets are not changed. Let $S'' = S' \cup \{B_{f(x)} \mid x \in S'\}$ be the set of keys which are either new or belong to affected buckets. The insertion of S' is done by building a new dictionary for S'', using the existing first-level function. For the actual implementation the following issues need to be addressed: (a) how to perform the retrieval; (b) how to build the new dictionary for S''. We first describe the high level management of the dictionary, and then elaborate on these issues.

2.2 Macro level maintenance

Rebuilding the data structure There are two cases where the dictionary algorithm starts to rebuild the whole data structure from scratch. *Case 1. Time-performance degradation.* The first-level function is selected at random from an appropriate class of hash functions, and with high probability satisfies certain properties that guarantee certain time performance. We keep track of the algorithm time performance. If it exceeds a certain limit, indicating a possible failure in the first-level function, we select a new first-level function and start building the dictionary from scratch. *Case 2. Space-performance degradation.* The first-level function is the dominant factor in determining the amount of space used by the dictionary. Since we would like the space used by the algorithm to be linear in the number of keys actually stored, it becomes necessary to replace the first-level function whenever the number of stored keys exceeds or drops under certain limits. Specifically, when too many cells are emptied as a result of delete instructions, the memory usage by the dictionary should reflect this, and for this purpose reorganization is needed. In addition, any first-level function has a certain upper bound on the number of keys that it can support and still maintain specific performance bounds; so once the total number of keys exceeds a certain bound it should be replaced by a new function. When replacing the first-level function, the entire dictionary must be reconstructed afresh.

Monitoring the number of keys The number of keys in the dictionary is changing dynamically, and for Case 2 above, we need to monitor it often. The next few sections develop tools that enable estimation of the number of keys, as discussed below. Each processor holds a count of the number of insert and delete instructions that it had so far (for multiple keys, only one representative takes the key into account). At each step, we compute the approximate sum of this count, by using Theorem 6.1. As long as the number of steps is at most polylogarithmic in p, this will give *at each step* a linear estimate of the size of the dictionary, with high probability. To support a dictionary that runs longer than that, we run in the background a logarithmic time algorithm for exact sum that enables to update the size of the dictionary once every $O(\lg n)$ steps. *Comment:* In fact, even the (simpler) estimation algorithm of Corollary 4.1 can be used instead of the approximate sum.

How to mask offline activity in an online algorithm In cases 1 and 2 above, where a new data structure is being built, it takes considerable time from the moment the dictionary algorithm starts to rebuild an alternative data structure until this new data structure is ready to replace the existing one. Specifically, rebuilding the dictionary takes at least $|S|/p$ time while our objective is to have *each* batch of m arbitrary instructions be executed with high probability in $O(m/p + \lg^* p)$ time.

The idea is to spread the overhead between *all* insertion steps, by slowing each one of them by a constant factor and run a rehash computation in the "background". Related ideas are reviewed in [34].

We first consider a *semi-dictionary*. We will have two dictionaries in parallel: Suppose that at some point in time $|S| = 2^i + 1$ for some integer i. A typical situation is where we have a data structure that can handle up to 2^{i+1} keys; we call it the *current* dictionary. At this point in time a second data structure will start to be built. It will be designed to handle up to 2^{i+2} keys; this is the *support* dictionary. When the size of $|S|$ exceeds 2^{i+1}, the support dictionary will be ready and at this point in time it will become the current dictionary. A new support dictionary will start to be constructed then. During the time in which S increases from $2^i + 1$ to 2^{i+1}, the current dictionary runs in a fraction α of its speed—operating only in one out of every $1/\alpha$ time units. In the other time units the construction of the support dictionary progresses. The exact value of α is set to satisfy certain timing goals. Eventually, the last keys to make the set of size 2^{i+1} will be inserted simultaneously by both dictionaries. This will enable the support dictionary to become a current dictionary.

Similar ideas can be applied to cope with a shrinking dictionary, following delete operations, and with dictionaries that may expand and shrink in an unpredictable way.

2.3 Inserting new and retrieved keys

As explained above, a batch of insertion instructions is implemented by building a new dictionary for a set consisting of new and retrieved keys, using the existing first-level function. The new dictionary will use space linear in the set size and will be built using a parallel hashing algorithm. More precisely, only part of a parallel hashing algorithm is used: assuming a first-level function is fixed, it finds second-level functions for the elements in the set (namely, for all elements in affected buckets), and allocates a memory block to every affected bucket.

An $O(\lg^* n \lg(\lg^* n))$ expected time hashing algorithm was given in [33]. It can be used in the dictionary but only to get *expected amortized* time bounds. In order to get $O(\lg^* n)$ execution time with high probability, we give in Section 7 an improved parallel hashing algorithm, that will also enable the dictionary algorithm work on weaker models of CRCW PRAM.

2.4 Retrieving the affected buckets

For each affected bucket, we need to show how to: (1) retrieve all its keys, and (2) allocate a memory block which is big enough.

Number of retrieved keys By Fact 7.1, the number of retrieved keys is $O(|S'|)$ with $|S'|$-polynomial probability (where S' is the set of new keys to be inserted).

How to retrieve old keys Among the new keys of each affected bucket, one will be selected to represent the bucket and be "responsible" for retrieving the old keys in this bucket. This can be done by a leaders-election algorithm (Theorem 7.2). The main difficulty in retrieval is that old keys are scattered in a memory block that might be much larger than the bucket size (in fact, it is not uncommon that it will be exponential in its size). A representative processor (one which is standing by the representative new key) will retrieve the entire memory block of the affected bucket, as follows: the memory block is considered as an array of tasks, in the custody of the retrieving processor. A *load balancing* algorithm is used (Theorem 5.1) to distribute the tasks evenly among all the processors. In the process, dummy tasks (i.e., empty cells in the memory block) are discarded and only old keys remain. By computing the maximum actual load of a single processor (in constant time [38]) we derive the size of memory that will be used for the static hashing algorithm.

The total size of a retrieved memory blocks The distribution of buckets sizes in the first level function, as proved in Fact 7.1, is used to prove the following.

Lemma 2.1. *The sum of the sizes of retrieved memory blocks (of affected buckets) is $O(|S'|)$ with $|S'|$-polynomial probability.*

2.5 Reducing memory space requirements of algorithms

A standard application of a data structure that supports the semi-dictionary operations is as follows. Take any (parallel) algorithm, and implement its memory using a semi-dictionary. The amount of space being used will be linear in the number of the memory cells actually used. Each time unit of the algorithm will be implemented within the time bound for one batch of semi-dictionary instructions.

3 Basic Concepts and Techniques

3.1 On teams and anonymous sets

Teams In a p-processor PRAM, processors are numbered $\mathrm{P}_1, \ldots, \mathrm{P}_p$. This numbering is often used as a cooperation tool in parallel algorithms. A *team* is defined as a set of processors with *consecutive* indices, such that the starting and the ending indices are known to all members of the set. A team is *allocated* to a task by appointing its first processor (the team "leader") to the task. In many cases it will be implicitly assumed that the p PRAM's processors are divided into teams and that each team has a private memory linear in its size. The private memory assumption does not add more than $O(p)$ memory to the whole machine, and it allows the team to use its regular structure in functioning like a sub-PRAM.

An exponential team constant time paradigm Suppose a small problem is given to a very large team on a CRCW. Then the range of input values gives crude limits on the set of possible output sequences. The algorithm proceeds by separately examining each output sequence against the input. We demonstrate the paradigm with the prefix sum problem.

Fact 3.1. *Using (a team of) 2^k processors, the prefix sum of $k^{O(1)}$ integers of $k^{O(1)}$ bits each can be computed in constant time.*

Anonymous sets In the algorithms presented here, we often run into situations where the challenge is to use processors that are available in a concerted manner. One form in which processors are available leads to the following definition: A set of processors Φ is *anonymous* if each processor in the set knows that it belongs to Φ and no processor outside the set claims that it belongs to the set; no other information, such as cardinality, ordinal place, and membership of other processors is assumed to be available to a processor. A *partition* of the processors into several anonymous sets is such that each processor belongs to exactly one of the sets. An anonymous set is *allocated* a resource (e.g., memory, team of processors) if all processors in it know this allocation, and no other processor claims this resource.

Scattering A *scatter* of an anonymous set of processors into an array is a primitive operation that does the following. Each processor selects independently and with uniform distribution an array position into which it tries to write its name. A processor *collides* if some other processor selected the same cell. If a write-collision in such an attempt occurs, the colliding processors know about that. A scatter is *injective* if no processor collides; it is *covering* if all the array cells are selected.

Lemma 3.1. *In a scatter of an anonymous set Φ, $|\Phi| = \mu$, into an array of size k,*
(a) **Prob** *(scatter is injective)* $\geq 1 - \mu^2/k$.
(b) **Prob** *(scatter is covering* $|$ $k \leq \mu/4) \geq 1 - ke^{-\mu/6k}$.

Estimations An *estimation* of a positive quantity μ is denoted by $\widehat{\mu}$. The *relative error* of the estimation is $|\widehat{\mu} - \mu|/\mu$. We say that $\widehat{\mu}$ is an *ϵ-estimate* if its relative error is at most ϵ. $\widehat{\mu}$ is a *linear-estimate* if $\max(\widehat{\mu}/\mu, \mu/\widehat{\mu})$ is bounded by a constant; $\widehat{\mu}$ is a *polynomial-estimate* of μ if $\lg\widehat{\mu}$ is a linear-estimate of $\lg\mu$.

Lemma 3.2. (self estimation) *Suppose that an anonymous set Φ, $|\Phi| = \mu$, is allocated an array of size $2\lg n$. Then Φ can compute in constant time an estimate $\widehat{\mu}$, such that $\widehat{\mu}$ is a linear estimate with constant probability and a polynomial estimate with μ-polynomial probability.*

The algorithm for Lemma 3.2 is inspired by [23, 29, 43, 45]. A tool called *geometric decomposition* is used (under different names) in all these papers.

Geometric decomposition A *geometric decomposition* of a set of processors Φ is performed by having each processor in Φ choose to belong to exactly one subset Φ_i with probability 2^{-i}, for $i \geq 1$. For implementation purpose, the number of subsets $\Phi_1, \Phi_2, \ldots$, into which the decomposition is done, is bounded by some d, and with probability 2^{-d}, no subset is selected.

Maximum finding

Fact 3.2. ([17], generalizing [38]) *Let Φ be an anonymous set, $|\Phi| \leq k$, such that every $\mathrm{P} \in \Phi$ has a value $val(\mathrm{P})$. If Φ is allocated with an array of size k^ϵ, for some constant $\epsilon > 0$, then* $\max\{val(\mathrm{P}) \mid \mathrm{P} \in \Phi\}$ *can be computed in constant time, with k-polynomial probability.*

3.2 The log-star paradigm

Consider an abstract problem whose input is an array of n *active items*. A single processor can *deactivate* an active item with constant probability in a single time unit; q (independent) processors can *deactivate* an active item with q-exponential probability in a single time unit. The problem is to deactivate all items.

The *log-star paradigm* consists of $\lg^* n$ basic rounds. The number of active items at the beginning of round i is at most n/q_i^c, where $\{q_i\}$ is a sequence defined by $q_{i+1} = 2^{q_i^\epsilon}$, $q_1, c > 0$ are sufficiently large constants, and $\epsilon > 0$ is a constant.

Round i consists of two steps:

allocation: Allocate to each active item a team of size q_i. The allocation succeeds for all but $\frac{1}{2}n/q_{i+1}^c$ active items with n-polynomial probability. Only allocated items *participate* in the next step.

deactivation: Deactivate each participating item. The deactivation succeeds for all but $\frac{1}{2}n/q_{i+1}^c$ active items with n-polynomial probability.

How is the paradigm used? Given a concrete problem, only the deactivation step needs to be implemented. The allocation step of [33] can be used as is. The deactivation step is implemented using a *deactivation routine*; in a deactivation routine a participating item becomes inactive with q_i-exponential probability (the "deactivation probability").

In the algorithms that apply the log-star paradigm the following situation always exists: The participating items are partitioned into subsets. The deactivation probability of a participating item is pseudo-independent with respect to all participating items outside its subset; this means that this deactivation probability is at most q_i-exponential irrespectful of which subset of these items became deactivated. However, for our analysis we need an additional condition: an upper bound, of say $z = n^\delta$ where $\delta < 1$ is some constant, on the size of each subset.

Then, one iteration of the deactivation routine suffices for implementing the deactivation step (by Hoeffding bounds).

To achieve an upper bound of z on each subset, the following *dependency limiting* step is performed, prior to application of the log-star paradigm. 'Large' subsets (of size $> n^\delta$ only) are handled, using the fact that only (relatively) few such sets ($< n^{1-\delta}$) may exist.

The paradigm's framework was first used in [19] for an algorithm in a nonstandard model of computation. The paradigm was first used for PRAM algorithms in [33]. Allocation of processors o jobs was a fundamental problem that had to be overcome.

3.3 Compaction

A basic problem that was solved by the log-star paradigm (and was actually used to demonstrate it) is the compaction problem. Given is an anonymous set Φ. Let μ denote $|\Phi|$, and assume that $m \geq \mu$ is known. The *compaction* problem is to find an injective (one-to-one) mapping from Φ into an array of size $O(m)$.

Fact 3.3. **([33])** (global-team compaction) *Using a team of size n, the compaction problem can be solved in $O(\lg^* \mu)$ time, with n-exponential probability.*

The deactivation step of the compaction algorithm satisfies the following:

Fact 3.4. *Assume that there are n/q active items and an array of size $O(n)$, in which at least a constant fraction of the cells are empty. Then, using a global team of n processors, each active item can find a private cell (and thus become inactive) in constant time, with q-exponential probability.*

This step is sometimes used as a basic sub-step of the deactivation routine in other algorithms (see Section 9).

In Fact 3.4 the n processors are required to be given in a team in order to enable reallocation throughout the algorithm run. If sufficiently many processors are given to each item a priori then we have

Lemma 3.3. (allocated-teams compaction) *Assume that to each member of Φ there is an allocated team of size $\Omega(\lg\mu)$. Then, the compaction problem can be solved in $O(\lg^* \mu)$ time, with n-exponential probability.*

A generalization of the compaction problem is solving several such problems simultaneously for several anonymous sets. In particular, given k anonymous sets Φ_i and k arrays $D_i[1 \ldots 4m_i]$, where $m_i \geq |\Phi_i|$, $i = 1, \ldots, k$, the ***multiple compaction*** problem is to find for each set Φ_i an injective mapping into array D_i. In this abstract, we only need the case $k = \lg n$. We have

Lemma 3.4. (multiple compaction) *Let $n = \sum_{i=1}^{k} |\Phi_i|$ for $k \leq \lg n$. Then, using n processors, the multiple-compaction problem can be solved in $O(\lg^* n)$ time, with n-polynomial probability.*

The algorithm of Lemma 3.4 treats separately small sets, of size at most $\lg n$, and large sets. For large sets, a variant of the compaction algorithm (Fact 3.3) is used. Small sets are treated separately, using an algorithm for ***polynomial approximate compaction*** (see [33, 37]) and Fact 3.1.

Team size per item	Team size per set	Accuracy	Success probability
—	—	linear	μ-polynomial
—	exponential	exact	μ-exponential
—	polynomial	$1/\lg^k \mu$	μ-polynomial
—	logarithmic	linear	$\lg\mu$-polynomial
v	—	linear	v-exponential

Table 1: Constant time estimations of μ—the size of an anonymous set.

3.4 Models of computation

As a model of computation we use the concurrent-read concurrent-write parallel random access machine (CRCW PRAM) family. The members of this family differ by the outcome of the event where more than one processor attempt to write simultaneously into the same shared memory location: In the TOLERANT ([24]) the content of that cell does not change (it may be viewed as a CREW with an additional ability to *try* concurrent write); in the ARBITRARY ([40]) one of the processors succeeds, and it is not known in advance which one; in the PRIORITY ([21]) the lowest-numbered processor succeeds; in the MAXIMUM ([1]) the processor trying to write the maximum value succeeds.

Whenever the model of computation is not specified it is assumed to be ARBITRARY.

4 Size Estimates of Anonymous Sets

The main contribution of this section is an estimation procedure that is suitable for employment in the log-star paradigm. Given a team of q processors allocated to each set member, we aim at an estimation algorithm that succeeds with q-exponential probability. Building towards this, an additional setting, where a global team is allocated to the set as a whole, is considered. Interesting consequences are improved estimation algorithms for a setting where no external processing power is available (see Corollary 4.1). Parameters of interest are: accuracy, error probability, and time complexity. Table 1 summarizes all estimation results.

Global team estimations Let Φ be an anonymous set, $|\Phi| = \mu$, which is allocated a team of size v. Let $\delta, k > 0$ be any constants.

Lemma 4.1. *If $v \geq 2^{\delta\mu}$ then Φ can compute in constant time $\widehat{\mu}$, such that $\widehat{\mu} = \mu$ with v-polynomial probability.*

Proof (Sketch) The algorithm is based on finding an injective scatter of Φ into an array A of size $2k^2$, where $k = \lg v/\delta \geq \mu$. This is done by scattering Φ (Lemma 3.1(a)): first to allocate processors to each member of Φ and then to use the allocated processors for finding an injective mapping into A. Once Φ is represented in A, Fact 3.1 is used to compute μ. ∎

Lemma 4.2. *If $v \geq \mu^\delta$ then Φ can compute in constant time $\widehat{\mu}$, such that the relative error of $\widehat{\mu}$ is at most $1/\lg^k v$ with v-polynomial probability.*

Proof (Sketch) We assume that $\mu \geq \lg^{2k+1} v$ (otherwise Lemma 4.1 may be applied). The anonymous set Φ is partitioned, using a geometric decomposition, into $O(\lg v)$ subsets $\Phi_1, \Phi_2, \ldots$. Each sub-set is then allocated a sub-team of size $\Omega(v/\lg v)$; this sub-team tries to compute exactly $|\Phi_i|$ with v-polynomial probability, by applying Lemma 4.1 and a *guess* that $|\Phi_i| \leq \lg^{2k+1} v$. Let $\ell = \min\{i : |\Phi_i| \text{ was computed}\}$, then clearly ℓ can be computed in constant time. Our estimate is $2^\ell |\Phi_\ell|$.

We have $\mathbf{E}(|\Phi_i|) = \mu/2^i$. By Chernoff bounds (Fact 1.1), the relative error of $|\Phi_i|$ with respect to $\mathbf{E}(|\Phi_i|)$ is ϵ with $\epsilon^2 \mathbf{E}(|\Phi_i|)$-exponential probability. For the purpose of the proof we choose $\epsilon = 1/\lg^k v$. It can be shown that with v-polynomial probability, ℓ will be such that $|\Phi_\ell| \approx \lg^{2k+1} v$. ∎

Theorem 4.1. *If $v \geq \lg^{1+\delta} \mu$ then Φ can compute in constant time $\widehat{\mu}$, such that the relative error of $\widehat{\mu}$ is at most $1/\lg^k v$ with v-polynomial probability.*

Proof (Sketch) The proof outline is the similar to that of Lemma 4.2; instead of applying exponential team estimation for computing $|\Phi_i|$ exactly, the *estimation* algorithm of Lemma 4.2 is tried on all $|\Phi_i|$. ∎

Self estimations We consider settings similar to the above with the exception that no external team is available.

Corollary 4.1. *The anonymous set Φ, with an allocated array of size μ^δ, can compute in constant time $\widehat{\mu}$, such that the relative error of $\widehat{\mu}$ is at most $1/\lg^k v$ with v-polynomial probability.*

Proof (Sketch) Compute a polynomial estimate $\widehat{\mu}$ of μ, with μ-polynomial probability (Lemma 3.2), such that $\widehat{\mu} \leq \mu$. Scatter Φ over an array of size $\min(\widehat{\mu}^{1/3}, v^\delta)$. By Lemma 3.1(b), the scatter is covering with μ-exponential probability, and thus provides an external team of size v^δ. We then apply Theorem 4.1. ∎

Estimation using a team for each member We turn now to deal with a setting where a team of size q is allocated to each member of Φ. This can be thought of as having q *copies* of Φ. The first step is to show how to *check* an estimate. This checking is useful for the following reason. Using Lemma 3.2 a linear estimate can be computed with constant probability. Then, the probability of success can be amplified by running q estimation algorithms in parallel and then checking each estimate. Several checking results were obtained, all using the fact that an injective mapping into a "sufficiently large" array can be computed in constant time with high probability, while an injective mapping into a "sufficiently small" array does not exist. These results, together with Lemma 3.2 and using a load balancing step (Corollary 5.1) lead to

Theorem 4.2. (multiple estimation) *Given is a partition of the processors $\text{P}_1, \ldots, \text{P}_n$ into anonymous sets, such that each processor is allocated a team of size q. Then, there exists a constant time algorithm that for each set Φ, $|\Phi| = \mu$, computes a linear-estimate with $q\lg\mu$-exponential probability.*

5 Load Balancing

Assume that m independent unit-time tasks are partitioned among n processors of a PRAM. The input to a processor P_i consists of m_i, the number of tasks allocated to this processor (its "load"), together with a pointer to an array of task's representations; no other information about the global partition is available. The *load balancing* problem is to redistribute the tasks among the processors such that each processor has at most $O(1 + m/n)$ tasks. We have

Theorem 5.1. (load balancing) *The load balancing problem can be solved in $O(\lg^* \min(n, m))$ time with n-polynomial probability.*

Proof We first assume that $m = n$. The load balancing is achieved by finding an embedding of the processors' tasks-arrays into distinct segments of a global array A of size $O(n)$; each tasks-array is mapped into a segment linear in its size. Part of the algorithm is based on the $O(\lg\lg n \lg^* n/\lg\lg\lg n)$ time load balancing algorithm of Hagerup [26] and it uses his idea of "logarithmic standard lengths" [27]. The main steps are given below. All steps succeed with n-polynomial probability, and all except for the last step take constant time.

1. Each processor P_i rounds m_i, its load, to the nearest larger power of 2. This step does not increase the total number of tasks by more than a factor of 2.

2. Define the sequence of anonymous sets $\{\Phi_i\}$, by $\Phi_i = \{\text{P}_j \mid m_j = 2^i\}$ and allocate each set Φ_i with a team of size $\lfloor n/\lg n \rfloor$. Let $\mu = |\Phi_i|$. Using the teams, compute for each μ_i an estimate $\widehat{\mu}_i$, which, with n-polynomial probability, is a linear-estimate and greater than μ_i (Theorem 4.1).

3. Compute the prefix sums of the sequence $\{\widehat{\mu}_i 2^i\}$ (Fact 3.1). Note that the total sum is still $O(n)$. Based on the prefix sums, partition array A into sub-arrays A_i, such that $|A_i| = 4\widehat{\mu}_i 2^i$.

4. Partition each sub-array A_i into $4\widehat{\mu}_i$ segments of size 2^i each; each segment represents a team; the segments are represented by an array $D_i[1 \ldots 4\widehat{\mu}_i]$.

5. Use Lemma 3.4 to find injective mappings from Φ_i into D_i, for $i = 1, \ldots, \lg n$. If $\text{P}_j \in \Phi_i$ is mapped into

$D_i[k]$ for some k, then the team of size 2^i represented by $D_i[k]$ will take care of the $m_j \leq 2^i$ tasks of P_j.

The extension of the algorithm for the case where m is unknown is as follows. If a linear-estimate $\widehat{m}$ for m is known then each set of $\lceil \widehat{m}/n \rceil$ tasks is considered as a unit-task, thus reducing the problem to the case $m = n$. It remains to show how to compute a linear-estimate $\widehat{m}$ for the sum m of n numbers. The maximum load x is computed with n-polynomial probability in constant time (Fact 3.2). The sum m is between x and nx. Each number y is replaced by $\lceil yn/x \rceil$. The new numbers are now bounded by n, and a linear-estimate for their sum can be computed by steps 1–3 above. It is easy to verify that by multiplying this estimate by x/n, we get a linear-estimate for m. ■

The load balancing algorithm provides an injective mapping from the set of processors into a linear size array, such that a processor P_j is mapped into a sub-array of size $\geq m_j$. It is therefore a generalization of the compaction problem, which can be thought of as load balancing with $m_j = 1$ for all j. Sometimes the load balancing will take place as a basic sub-step in another algorithm that uses the log-star paradigm (see Section 9). By using Fact 3.4 in Step 6 we have

Corollary 5.1. *Assume that each participating processor has an allocated team of size q. Then using linear space and constant time, each participating processor with load y can be allocated with a sub-array of size $\geq y$ with q-exponential probability.*

6 Approximate Sum

It follows from the proof of Theorem 5.1 that a linear-estimate for the sum of n integers drawn from the range $[0, n^{O(1)}]$ can be computed in constant time with n-polynomial probability. This section presents a more accurate sum estimation algorithm that works for unbounded integer inputs as well as for real numbers.

Definition 6.1. *Given n numbers $x_1, \ldots, x_n$, the* approximate sum *problem is to find an estimate $\widehat{X}$ for $X = \sum_{i=1}^{n} x_i$ such that the relative error of $\widehat{X}$ is $o(1)$. The estimate $\widehat{X}$ is the* approximate-sum.

Theorem 6.1. *There exists a constant time algorithm that, using n processors, solves the approximate sum problem with n–polynomial probability. Specifically, the relative estimation error is $\leq 1/\lg^k n$ for any constant k.*

Overview The algorithm consists of three parts. In the first part (steps 1-2) the input numbers are modified to integers from $[1 \ldots n^2]$. In the second part (step 3) the input is further modified to get a more favorable distribution of values, where the relatively larger values can only exist in "big" groups; the reason being that misetimating larger values contributes more to an error, and having them in bigger groups enables evaluating their number with higher probability. The third part (steps 4-6) is a computation similar to steps 1-3 in the load balancing algorithm, with a modification that enables higher accuracy.

The algorithm uses an array $r[1 \ldots n]$ of registers. Denote by $r_k[i]$ the value of $r[i]$ at the end of step k of the algorithm, and let $R_k = \sum_{i=1}^{n} r_k[i]$. We assume that initially $r[i] = r_0[i] = x_i$.

1. Find $\mathbf{r} = \max_i r[i]$ (Fact 3.2), and let $\delta = \mathbf{r}/n^2$. Let $r[i] \leftarrow r[i]/\delta$, for $i = 1, \ldots, n$. Then, $R_1 = R_0/\delta = X/\delta$, $\max_i r_1[i] = n^2$ and $n^2 \leq R_1 \leq n^3$.

2. Let $r[i] \leftarrow \lceil r[i] \rceil$, for $i = 1, \ldots, n$. Then, $R_1 \leq R_2 < R_1 + (n-1) < (1+1/n)R_1$, and $r_k[i] \in [0 \ldots n^2]$, $i = 1, \ldots, n$.

3. For $j = 0, 1, \ldots, 2\lg n$ let Φ_j be the anonymous set of all processors P_i such that $\lfloor \lg r[i] \rfloor = j$. The input is so modified that for all j, $1.5\lg n \leq j \leq 2\lg n$, Φ_j is either empty or of size $\geq n^{1/12}$. This is done by "breaking" large numbers to many small numbers, similarly to the Disperse procedure of [15]. For this, injective scatters of the small sets (of size $\leq n^{1/12}$) into arrays of size $n^{1/4}$ is computed (Lemma 3.1); each member of these sets is thereby allocated with a team. Each allocated number is "dispersed" among its team members.
This transformation does not change the sum, i.e., $R_3 = R_2$, and adds less than n new numbers to the input.

4. Each $r[i]$ is rounded and set to $r_4[i]$ which is either 2^j or 2^{j+1}, where $2^j \leq r_3[i] < 2^{j+1}$. The rounding is chosen at random to satisfy $\mathbf{E}(r_4[i]) = r_3[i]$. Specifically, $\mathbf{Prob}\left(r_4 = 2^j\right) = 2 - r_3[i]2^{-j}$ and $\mathbf{Prob}\left(r_4 = 2^{j+1}\right) = r_3[i]2^{-j} - 1$. Clearly, $\mathbf{E}(R_4) = R_3$. It can be shown by Chernoff bounds (proof omitted) that $|R_4 - R_3|/R_3 \leq 1/n$ with n-polynomial probability.

5. Compute estimates $\widehat{n}_j$ for $|\Phi'_j|$, for $j = 0, 1, \ldots, 2\lg n$, by applying Theorem 4.1 with a relative error $\leq 1/\lg^k n$.

6. Compute the sum $\widehat{N} = \sum_{j=0}^{2\lg n} \widehat{n}_j 2^j$ (Fact 3.1). The approximate sum is $\widehat{X} = \delta \widehat{N}$.

Analysis All steps above take constant time, and they all succeed with n-polynomial probability. The relative errors accumulated in the algorithm are: $1/n$ in Step 2, $1/n$ in Step 4, and $1/\lg^k n$ in Step 5. The total relative estimation error is therefore $\leq 1/\lg^k n$.

7 Hashing

The hashing problem is solved in two steps. First, a leaders election algorithm is used to remove duplicates from the array of keys. Then, a hashing algorithm that assumes distinct keys is used. Both leaders election and hashing algorithms in [33] use the log-star paradigm. They take $O(\lg^* n \lg(\lg^* n))$ expected time, since the deactivation probabilities may be dependent without any limitation. In the leaders elections algorithm, the dependency comes from items being in the same anonymous set. We will therefore employ a dependency limiting step, in which large sets will be handled separately. In the hashing algorithm, the assumed dependency was caused by lack of knowledge about the first level function. We show stronger properties of the class of hash functions from which the first level function is selected. These properties imply that there is only a limited dependency. As a result we get

Theorem 7.1. *There exists an algorithm for the hashing problem that, using* $n\lg^{(i)}n$ *processors, takes* $O(i)$ *time and* $O(n\lg^{(i)}n)$ *space, for any* $i \geq 0$, *with n-polynomial probability.*

The hashing algorithm is used in the dictionary algorithm of Section 2. [32] list several algorithms where hashing is sufficient to reduce to space requirement to linear space or close to it. Yet another application is to get a more efficient algorithm for constructing a function that is $\lg n$-wise independent [42].

7.1 Leaders election

Theorem 7.2. *Using* $n\lg^{(i)}n$ *processors, the leaders election problem can be solved in* $O(i)$ *time with n-polynomial probability on* ARBITRARY, *using* $O(n)$ *space, and on* TOLERANT, *using* $O(n\lg n)$ *space. In particular, using n processors, the time complexity is* $O(\lg^* n)$. *It can also be solved on* TOLERANT, *using* $O(n)$ *space, in* $O(\lg^* n)^2$ *time, with n-polynomial probability.*

To get the algorithm on ARBITRARY, it is sufficient to add the dependency limiting step below to the algorithm in [33]. In order to have it on TOLERANT, the arbitrary convention is replaced by having at step i, q_i geometric decompositions into arrays of size $\lg n$. To obtain the algorithm on TOLERANT using linear space, some other ideas need to be employed.

The dependency limiting step We select a random sample with the following properties: *(i)* The size of the sample is "small"; *(ii)* For each "large" set, there is at least one item from the set in the sample. Property *(i)* enables to easily solve the leaders election problem for the items in the sample, since we can use an array that is much larger than the sample size. Property *(ii)* guarantees that after solving the problem for the sample, all "large" sets can be eliminated from the input.

7.2 Hashing distinct keys

The first level function The hashing algorithm imposes requirements on the distribution of buckets size, as determined by the first level function. These requirements are met with high probability by random functions. However, a random function cannot be represented efficiently, nor can it be evaluated in constant time. Siegel [42] and then Dietzfelbinger and Meyer auf der Heide [12] described classes of functions that are, for our purpose, as good as random functions. These functions can be represented efficiently and be evaluated in constant time. The hashing algorithm uses the construction of [12], i.e., a class $\mathcal{R}$ of functions from U to $[1, N]$, which is suitable for any universe size. The deactivation step in round i of the hashing algorithm only deals with keys from buckets of size $\leq q_i$. The property of the class $\mathcal{R}$ that makes it suitable for the log-star paradigm is [12]: For f randomly chosen from $\mathcal{R}$, $\forall r$, $\mathbf{E}\left(|\{B_i(f) : |\{B_i(f)\}| \geq r\}|\right) \leq |S|\left(e^{r-1}/r^r\right)^{1/d}$ for some constant d. To show that the dependency is limited, we prove a stronger property, using the Martingale Tail Inequality (Azuma's Theorem) [31].

Fact 7.1. *Let* $S \subseteq U$, $|S| \leq N$, *be fixed.*

Let f be randomly chosen from $\mathcal{R}$. *Then there exists a constant c, such that with* $|S|$*-polynomial probability*

$$\forall i, r \quad \#\{x \in B_i(f) : |B_i(f)| \geq r\} < c|S|2^{-r} .$$

8 Optimizing Parallel Algorithms

Many PRAM algorithms are designed using the well known work-time methodology. (See [30, Chapter 1.3] for a detailed discussion; first use in a PRAM algorithm in [41].) Guided by Brent's theorem [5], the methodology suggests to first describe a meta-algorithm in terms of a sequence of rounds; each round may include any number of independent constant time operations. Second, the meta-algorithm is implemented on a p-processor PRAM using a scheduling principle: if the number of operations in round r is w_r, then each of the p processors should execute a set of $O(w_r/p)$ operations ("optimal simulation", below). Let $W = \sum_r w_r$ be the total number of operations of the meta-algorithm. If T, the number of rounds of the meta-algorithm, satisfies $T = O(W/p)$ then the resulting time T_p will satisfy $pT_p = O(W)$.

Traditionally, the scheduling principle above is being implemented in an ad-hoc manner. However, in this section, we advance towards implementing it automatically, which would wishfully mean automatic implementation of any meta-algorithm on a PRAM. We note that this goal does not appear to be well-defined since *any meta-algorithm* is an unclear notion. Our results will apply to a family of so called *loosely specified* algorithms.

8.1 Loosely-specified algorithms

Assume that we are given a parallel algorithm A which is specified as follows: It consists of rounds; in each round several unit-time tasks are to be performed. All the tasks for round 1 are given in an array. The tasks for a round $i > 1$ can be specified in any of the following forms: In an array; or, upon its completion, each task in a round $j < i$ can "appoint" an array of tasks for a single later round. We assume that a task in round i is being specified exactly once. A parallel algorithm which is so specified is called ***loosely specified***. Assume that the total number of tasks is at most W and the number of rounds is at most T. We say that a simulation of the algorithm by p processors in time T_p is ***optimal*** if the time-processor product, pT_p, is within a constant factor from W. We show

Lemma 8.1. *The problem of optimally simulating one round of the loosely specified algorithm* A *can be reduced in constant time and* $O(T \cdot W)$ *space to the load balancing problem.*

By using the fast load balancing algorithm (Theorem 5.1) and the fast dictionary (Theorem 2.1) for space reduction we have

Theorem 8.1. (The randomized optimizer) *Algorithm* A *has an optimal simulation in time* $T_p = O(T\lg^* p)$, *with p-polynomial probability, and* $O(W) = O(pT)$ *space.*

Comment: In a concurrent work [22], Goodrich considered an alternative randomized optimizer and demonstrated quite a few applications to parallel computational geometry. Theorem 8.1 provides a more general optimizer.

8.2 Task-decaying algorithms

Assume that we restrict the loosely specified algorithm as follows: each task in round i can appoint at most one task and only for round $i+1$. In particular, this guarantees the number of tasks will not increase as the algorithm proceeds. We call these algorithms ***task-decaying*** algorithms. Quite a few parallel algorithms are of this type. [33] showed how to simulate task-decaying algorithms using an optimal number of processors by using the compaction algorithm. The overhead in their simulation result is an ***additive*** factor of $O(\lg^* n \lg(\lg^* n))$. It appears, however, that often task-decaying algorithms actually satisfy a geometric decaying. Namely, an upper bound on the number of tasks in round i decreases at least geometrically.

Let A be a geometric-decaying algorithm that takes time T, where the upper bound on the number of tasks in round 1 is n. [18] showed that such algorithms can be simulated with an optimal number of processors $p = n/T_p$, by having $O(\lg^* T_p)$ calls to a load balancing algorithm. By using the $O(\lg\lg n)$ load balancing algorithm of [15] their simulation had an ***additive*** overhead of $O(\lg\lg n \lg^* n)$ time. By using the new load balancing algorithm (Theorem 5.1), we have

Corollary 8.1. (The additive optimizer) *Algorithm* A *can be implemented in time* $T_p = O(T + \lg^* n \lg^*(\lg^* n))$, *with n-polynomial probability, and an optimal number of processors.*

Corollary 8.1 can be used to get optimal speedup results for the problems considered in this paper. The algorithms presented in this paper are in general "fast" (usually of $O(\lg^* n)$ time), using n processors. These algorithms are therefore non-optimal. Each of the algorithms given can be easily transformed into a geometric-decaying one. The additive optimizer is used to implement these (optimal) geometric-decaying algorithms and reduce the number of active items to less than $n/\lg^* n$. The fast (and non-optimal) algorithms can now be employed. This general strategy is called ***accelerating cascades*** in [8].

Is this a tight analysis? Take an iteration of one of our algorithms. Using $p = n$ processors the step that dominates its running time consists of having n items thrown into a linear number of cells. A fraction of the items will occupy singleton cells. Observing that $p' \leq p$ processors are used to emulate the p processors in $\lceil p/p' \rceil$ rounds, our situation improves since at each round only p' items are being thrown into the cells. Note that the word emulation is correct only in a weak sense since the execution by p' processors might be different than the execution by the p processors. A key observation, however, is that the execution by p' processor is still be correct. Algorithms that result in correct execution regardless of the order in which the p processors are emulated are called "asynchronous". It turns out that for the compaction algorithm this idea is enough to get a considerable improvement. For the other algorithms some additional effort is needed. Specifically, selecting the subsets of p' processors at random at each round among the p processors (until all p processors are exhausted) would yield a similar improvement. Some algorithms (e.g., the hashing algorithm) require some adaptation to become asynchronous.

Comment: In the additive optimizer (Corollary 8.1) the processors are synchronized in rounds; therefore, geometric decaying is the only requirement from the algorithm.

To get random subsets, we apply a random permutation to the set of p processors and then emulate them by p' processors in a round robin fashion. We refer the reader to section 4 in [44], where a similar application of random permutations has been used and analyzed. Finally, we note that a padded representation of a randomized permutation (as provided in [33]) suffices. We can show

Lemma 8.2. *The following problems have asynchronous geometric-decaying algorithms: Hashing, leaders election, load balancing, compaction, random permutation, simulating* MAXIMUM *on* TOLERANT, *and integer chain-sorting.*

We therefore have

Theorem 8.2. *For the following problems we have algorithms that run in* $O(\lg^* n)$ *time with n-polynomial probability using* $n/\lg^* n$ *processors (op-*

timal speedup): Hashing, leaders election, load balancing, compaction, random permutation, simulating MAXIMUM *on* TOLERANT, *and integer chain-sorting.*

9 Integer Chain Sorting

Given n input numbers, the *chain sorting* problem is to chain them in a linked list such that for each element, its successor in the list is of greater value. The *integer chain sorting* problem is when the input numbers are from the range $[1 \ldots n]$. We have

Theorem 9.1. *The integer chain sorting problem can be solved in $O(\lg^* n)$ time, with n-polynomial probability, using $n/\lg^* n$ processors (optimal speedup).*

Each set of input numbers with the same value is considered as an anonymous set: Φ_i is the set of elements with value i. We may view the integer chain sorting problem as two different chaining problems. One is to chain the items *within* each set Φ_i, and the other is to chain *between* the sets $\Phi_1, \ldots, \Phi_n$. The latter problem can be easily reduced to the *nearest-one* problem: given an array of zeros and ones, find for each cell the nearest cell with the value '1'. This problem has a deterministic $o(\lg^* n)$ time algorithm [3, 37] . It therefore remains to chain-sort only within each bucket.

The algorithm consists of $O(\lg^* n)$ basic iterations. At iteration i, we allocate to each anonymous set Φ_j a private sub-array D_j in a global array D, and map a subset $\Phi_j^i \subset \Phi_j$ into D_j. Since all the elements in Φ_j^i are in the same sub-array, they can be chained using a nearest-one algorithm, applied for the global array D. On the other hand, the subsets $\Phi_j^1, \Phi_j^2, \ldots$ can be chained in a simple manner (sequentially in time). As a result, the required linked list will be derived. A leaders election algorithm (Theorem 7.2) is employed in a pre-processing step to enable allocation of sub-arrays to anonymous sets.

We use the log-star paradigm. Assume that each participating item is allocated with a team of size q^2. The deactivation step consists of the following sub-steps, each taking constant time: (1) An estimate μ_j for the number of participating items in each set Φ_j is computed; each estimate is a linear-estimate with q-exponential probability. (2) To each set Φ_j try to allocate a sub-array D_j of size $O(q_i\mu_j)$, using Corollary 5.1. (3) Try to map each participating item in Φ_j into sub-array D_j, using Fact 3.4.

The dependency limiting step: Each set is partitioned into $n^{1-\delta}$ subsets by scattering into an array $[1..n^{1-\delta}]$. All subsets are of size $O(n^\delta)$ with n-polynomial probability. We only need to chain within the subsets of the same set. A leader is selected for each set, and an integer chain sorting algorithm is used to chain the leaders in each (original) set. For $\delta > 1/2$ the number of leaders in each set is $\leq n^{1-\delta} < n^\delta$, as required.

10 Simulating MAXIMUM on TOLERANT

The concurrent write operation of a CRCW machine partitions the processors into anonymous sets in a natural way. An anonymous set is defined as all processors that try to write into the same memory cell. Using the techniques developed for computing on anonymous sets we have

Theorem 10.1.
One step of an n-processor MAXIMUM *can be simulated on an $(n/\lg^* n)$-processor* TOLERANT *in $O(\lg^* n)$ time with n-polynomial probability.*

Comment: In the simulation algorithm, $O(n\lg n)$ auxiliary space is used by the simulating machine; the original memory used by the MAXIMUM is addressed with an *exclusive-write* convention.

In a pre-processing step, a leaders election algorithm is employed. The dependency limiting step of this algorithm, slightly modified, will serve also the simulation algorithm.

We use the log-star paradigm. Assume that each participating item is allocated with a team of size q^2. The deactivation step uses ideas from the load balancing algorithm and consists of the following sub-steps, each taking constant time: (1) An estimate for the number of participating items in each set is computed by Theorem 4.2; each estimate is a linear-estimate with q-exponential probability. (2) The sets are (implicitly) grouped into $\lg n$ super-sets, where super-set Ψ_i consists of all sets whose estimate is between 2^{i-1} and 2^i. (3) An estimate μ_i for the size of each super-set Ψ_i is computed by Theorem 4.2(b); all estimates are linear-estimates with n-polynomial probability. (4) Each super-set Ψ_i is allocated with a sub-array of size $O(q^2\mu_i 2^i)$, using Fact 3.1. This sub-array is partitioned into segments of size 2^i each, represented by an array of size $O(q^2\mu_i)$. (5) For each super-set Ψ_i we try to find an injective mapping into an array of size $O(q\mu_i)$ by having q copies (with distinct identifiers) for each participating set in the super-set Ψ_i and applying a hash function selected at random from a 2-universal class of hash functions into the range $O(q\mu_i)$. For each participating set, there exists at least one copy that does not collide when applying the hash function, with q-exponential probability. An injective mapping enables to allocate a team of size $q2^i$ to each participating set in Ψ_i, if μ_i is a linear-estimate. It is not known however if a scatter is injective or not. The following step is done by assuming for each scatter that it is injective. (6) For each set and for each of its potential allocated teams we do q maximum computations using Fact 3.2. For a set with a polynomial-estimate and with an allocated team at least one of the computations will succeed with q-exponential probability.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. In *ICPP '83*, pages 175–179, 1983.

[2] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *STOC '87*, pages 83–93, 1987.

[3] O. Berkman and U. Vishkin. Recursive *-tree parallel data structure. In *FOCS '89*, 1989.

[4] R. B. Boppana. Optimal separations between concurrent-write parallel machines. In *STOC '89*, pages 320–326, 1989.

[5] R. P. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, 21:302–206, 1974.

[6] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *MFCS '88*, pages 231–239, 1988.

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

[8] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *FOCS '86*, pages 478–491, 1986.

[9] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17:128–142, 1988.

[10] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Info. Comp.*, 81:334–352, 1989.

[11] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *SPAA '89*, pages 360–368, 1989.

[12] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hahshing in real time. In *ICALP '90*, pages 6–19, 1990.

[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. *JACM*, 31:538–544, July 1984.

[15] J. Gil. Fast load balancing on PRAM. Manuscript, 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 and efficient simulations between CRCW PRAMs. Manuscript, 1990.

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

[19] J. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *STOC '90*, pages 244–253, 1990.

[20] J. Gil and L. Rudolph. Counting and packing in parallel. In *ICPP '86*, pages 1000–1002, 1986.

[21] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *JACM*, 29:1073–1086, 1982.

[22] M. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *this proceedings*, 1991.

[23] A. G. Greenberg and R. Ladner. Estimating the multiplicity of conflicts in multiple access channels. In *FOCS '83*, pages 384–392, 1983.

[24] V. Grolmusz and P. L. Ragde. Incomparability in parallel computation. In *FOCS '87*, pages 89–98, 1987.

[25] T. Hagerup. Fast parallel generation of random permutations. Manuscript (Nov. 1990), also in ICALP '91, to appear.

[26] T. Hagerup. Private communication, Oct. 1990.

[27] T. Hagerup. Constant-time parallel integer sorting. In *STOC '91*, pages 299–306, 1991.

[28] T. Hagerup and M. Nowak. Parallel retrieval of scattered information. In *ICALP '89*, pages 439–450, 1989.

[29] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *SPAA '90*, pages 117–124, 1990.

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

[31] R. Karp. Probabilistic analysis of algorithms. Class notes, Univ. California, Berkeley, 1989.

[32] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. In *ICALP '90*, pages 729–743, 1990. Also to appear in Journal of Algorithms.

[33] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. In *STOC '91*, pages 307–316, 1991. Also in UMIACS-TR-91-65, Univ. of Maryland.

[34] K. Mehlhorn. *Data Structures and Algorithms*. Springer-Verlag, Berlin Heidelberg, 1984.

[35] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *ICALP '83*, pages 597–609, 1983.

[36] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[37] P. L. Ragde. The parallel simplicity of compaction and chaining. In *ICALP '90*, pages 744–751, 1990.

[38] R. Reischuk. A fast probabilistic parallel sorting algorithm. In *FOCS '81*, pages 212–219, 1981.

[39] G. E. Shannon. Optimal on-line load balancing. In *SPAA '89*, pages 235–245, 1989.

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

[41] Y. Shiloach and U. Vishkin. An $O(n^2 \lg n)$ parallel Max-Flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[42] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *FOCS '89*, pages 20–25, 1989.

[43] P. G. Spirakis. Optimal parallel randomized algorithms for sparse addition and identification. *Info. Comp.*, 76:1–12, 1988.

[44] U. Vishkin. Randomized speed-ups in parallel computations. In *STOC '84*, pages 230–239, 1984.

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