Converting High Probability into Nearly-Constant Time – with Applications to Parallel Hashing *

(Extended Abstract)

Yossi Matias Uzi Vishkin

University of Maryland
& Tel-Aviv University

Abstract

We present a new paradigm for efficient randomized parallel algorithms that needs $\overline{O}(\log^* n)$ time, where $\overline{O}(x)$ means '$O(x)$ expected'. It leads to: (1) constructing a *perfect hash function* for n elements in $\overline{O}(\log^* n \log(\log^* n))$ time and $\overline{O}(n)$ operations; (2) an algorithm for *generating a random permutation* in $\overline{O}(\log^* n)$ time, using n processors or in $\overline{O}(\log^* n \log(\log^* n))$ time and $\overline{O}(n)$ operations; and (3) an efficient *optimizer*: consider a parallel algorithm that runs in t time using p processors; since at each time unit some of the processors may be idle, we let x, the total number of actual operations, be the sum over all non-idle processors at every time unit; assuming the algorithm belongs to a certain kind, it can be adapted to run in $\overline{O}(t+\log^* n \log(\log^* n))$ time (additive overhead!) using $x/(t+\log^* n \log(\log^* n))$ processors.

We also get an optimal *integer sorting* algorithm. Given integers from the domain $[1..n]$, it runs in $\overline{O}(\frac{\log n}{\log \log n})$ time.

1 Introduction

The quest for fast and processor-efficient parallel algorithms is a demanding intellectual challenge and has contributed towards establishing a tradition of excellence similar to the one implied by the quest for fast serial algorithms. While the ultimate goal is to get constant-time algorithms whose time-processor product (i.e., number of operations) is linear, it is often impossible to get even close to this goal. To keep this abstract short, we refer the reader to [4] for an overview of lower bounds, even for rather elementary problems, and the few sublogarithmic upper bounds known.

Our main contribution is a paradigm for randomized parallel algorithms that enables extremely fast running time. It leads to faster randomized parallel algorithms for some fundamental problems.

<u>*Concrete Results*</u>

- **Hashing**

Input: Given is a set S of n elements from a universe $U = \{0, 1, \ldots, N-1\}$.

The *hashing* problem is to find a one-to-one $h : S \mapsto [1..dn]$ (for some constant $d \geq 1$) such that for any $x \in U$, $h(x)$ can be evaluated in constant time.

Result: $\overline{O}(\log^* n \log\log^* n)$ time and $\overline{O}(n)$ operations (optimal speedup).

Previous work: Sequential hashing: [8, 10, 11]; parallel hashing: [9, 13, 14, 15, 24]. [9] gave an optimal parallel dictionary (i.e., dynamic hashing); its time complexity is $O(n^\epsilon)$ for any constant $\epsilon > 0$. Following an optimal logarithmic-time parallel algorithm [24], [13] gave the two parallel algorithms: one runs in $\overline{O}(\log \log n)$ time using n processors and the other in $\overline{O}(\log \log n \log^* n)$ time using an optimal number of processors. These algorithms were extended to an optimal parallel dictionary with similar complexities [14]. Using a non-standard model of parallel computation, Gil, Meyer auf der Heide and Wigderson [15] gave a hashing algorithm that takes $\overline{O}(\log^* n)$ time if the keys are distinct. Specifically, their model of computation ignores the

*This work was partially supported by NSF grant NSF-CCR-8906949.

cost of counting and the issue of assigning processors to jobs. They also proved an $\Omega(\log^* n)$ time lower bound using n processors for a certain model of parallel computation. Our algorithm fits this latter model.

- **Random Permutation**

Input: Given is the set of n integers $\{1..n\}$.

The *random permutation* problem is to generate a random permutation of this set.

Results: $O(\log^* n)$ time (w.h.p.)[1] using n processors and $O(\log^* n \log\log^* n)$ time (w.h.p.) using an optimal number of processors.

Previous work: [1, 25, 27, 30]. $O(\log n)$ time (w.h.p.) and $n/\log n$ processors [25]. $O(\log n/\log\log n)$ time (w.h.p.) and $n(\log\log n)^2/\log n$ processors [27].

Concurrent work: $O(\log\log n \log^* n/\log\log\log n)$ time (w.h.p.) using an optimal number of processors [18]. The output of this algorithm is a random permutation in a so-called 'padded representation', which is generally insufficient. In order to get a standard representation the time increases to $O(\log n/\log\log n)$ (w.h.p.), using an optimal number of processors.

- **Linear Approximate Compaction (LAC)**

Input: Given is an array A of n cells, at most m of them contain one item each and the others are empty.

The *linear approximate compaction (LAC)* problem is to insert the items into an array $B[1..4m]$ (linear in m).

Results: $O(\log^* n)$ time (w.h.p.) using n processors and $O(\log^* n \log\log^* n)$ time (w.h.p.) using an optimal number of processors.

Previous work: [12, 13, 16, 19, 26]. The fastest known results are $O(\log\log n)$ time (w.h.p.) using n processors [12] and $\overline{O}(\log\log n \log^* n)$ time using an optimal number of processors [13].

Concurrent work: $O(\log\log n \log^* n/\log\log\log n)$ time (w.h.p.) using an optimal number of processors [18].

[1] Whenever we state 'w.h.p.', we mean that the algorithm always terminates correctly, but the stated complexity is with probability $\geq 1 - n^{-\alpha}$, for any constant $\alpha > 0$.

- **Integer Sorting**

Input: n integers from the domain $[1..n \log^d n]$ for some constant $d \geq 0$.

The *integer sorting* problem is to sort these n integers.

Result: $O(\frac{\log n}{\log\log n})$ time and $O(n)$ operations (w.h.p.).

Previous work: [27, 30]. $O(\log n)$ time and $O(n)$ operations (w.h.p.); $O(\log n/\log\log n)$ time and $O(n \log\log n)$ operations (w.h.p.) [27]. We note that the Rajasekaran-Reif's Integer Sorting algorithm has the same restriction as above on the input domain.

Concurrent work: A similar result was independently discovered by [17, 29].

- **2-Ruling Set**

Input: A list of n nodes: each node has a pointer to another node such that the pointers define a single linked list.

The *2-ruling set* problem is to find a subset $U \subset V$ of the nodes, such that: (1) if $u, v \in U$ then there is no pointer between u and v; (2) for each $u \in V - U$ there is $v \in U$ such that v points to u or v points to another node w that points to v.

Result: $O(\log^* n \log\log^* n)$ time and $O(n)$ operations (w.h.p.).

Previous work: [6, 7, 20]. $O(\log^* n)$ time and $O(n \log^* n)$ operations [6]; $O(\log^{(i)} n)$ (for any constant i) and $O(n)$ operations [20].

All our results hold for the arbitrary-CRCW PRAM model of parallel computation – where if several processors try to write simultaneously into the same cell, one of them succeeds, and it is not known in advance which one. Some of the results hold also for weaker models.

Section 2 considers the linear approximate compaction (LAC) problem. We selected this relatively simple problem for demonstrating our new approach. We discuss the issues that are involved in extending this approach for the hashing problem in two stages. The first stage is given in Section 3, where an extension to a problem called representative selection (RS) is discussed. The second stage, that is extension to the hashing problem, is discussed in Section 4. Section 5 outlines the algorithm for generating a random permutation. The improvement of this result relative to the previous fastest result is drastic. We feel that the random permutation algorithm offers an elegant way for circumventing what looked like an inevitable barrier. In Section 6 we show how algorithms of a certain type can be modified to run with optimal speed-up. In particular, we use the new technique to derive optimal algorithms from all algorithms that are given in this paper. In Section 7 we consider the multiple linear approximate compaction (MLAC) problem, which generalizes the LAC problem. We outline a simple sublogarithmic time optimal algorithm for this problem that enables to solve the integer sorting problem with similar complexity bounds.

A few of the problems considered in this abstract are well known. Therefore, we allow ourselves to add only a few sentences on extant work. Hashing is known to be a fundamental problem. We refer the reader to [24] for numerous applications of parallel hashing. In particular, parallel hashing turns out to significantly reduce the space requirements in parallel algorithms, by introducing some increase in time (and switching to expected rather than deterministic time bounds). The new hashing algorithm implies only a nearly-constant increase in time. Linear approximate compaction is a fundamental problem in its own right and may serve as an important building block in other algorithms. For example, this problem is needed in the load balancing algorithm of [12]. The LAC algorithm is also used in our random permutation algorithm and for the randomized "optimizer" presented in Section 6. The randomized optimizer is very efficient – it has only a small *additive* overhead. It applies to the well known work-time methodology that is guided by Brent's theorem [21, Chapter 1.3]. The optimizer is applicable for a certain type of algorithms (the same ideas can be adapted also for some other types of algorithms). An optimizer for slow algorithms (at least logarithmic time) was previously given by Shannon [31]. A randomized optimizer for fast algorithms was given independently by Hagerup [18]; the overhead is not as small as in this paper and is multiplicative. Integer sorting frequently appears as a subproblem in parallel algorithms. Our new algorithm may enable to improve these algorithms. A similar result on integer sorting was independently obtained by Hagerup [17] and by Raman [29].

2 Linear approximate compaction

Given is an array A of n cells, with a processor standing by each cell. In addition, at most m items are distributed among the cells. Each cell either contains one item or is empty. The *linear approximate compaction (LAC)* problem is to insert the items into an array $B[1..4m]$ (linear in m).

Theorem 1 *Using n processors the linear approximate compaction problem can be solved in $\overline{O}(\log^* m)$ time. The time complexity of the same algorithm is also $O(\log^* m)$ with probability $\geq 1 - m^{-\alpha}$, for any constant $\alpha > 0$.*

2.1 The algorithm

An item will be identified by the index of its original cell. Initially, each cell of array B is empty (and this is marked by initializing it to 0). Throughout the algorithm we map more and more items to accomplish a one-to-one mapping into array B. An unmapped item is called *active*. Once an item is mapped it becomes *inactive*. Initially all items are active.

The algorithm consists of $O(\log^* m)$ iterations.

Let q_1 be a constant (to be set in the analysis) and $q_{i+1} = 2^{q_i}$ for $i \geq 1$.

High-level description of iteration i:
The main idea is to enhance the mapping of active items by reallocating many processors to them.

Input of iteration i: the number of active items is assumed to be at most m/q_i^6.
The iteration consists of two basic steps:

- *(allocation)* Allocate $7q_i$ processors to each active item. For each active item, this allocation is accomplished with probability $\geq 1 - 2^{-7q_i}$. Each active item which indeed gets $7q_i$ processors is called *participating.*

- *(mapping)* Map each participating item to a different empty cell of B. For each participating item this mapping is accomplished with probability $\geq 1 - 2^{-7q_i}$.

Output of the iteration: The number of active items is at most $m/(2^{q_i})^6 = m/q_{i+1}^6$ with high probability.

The probability-time conversion paradigm:
The main new idea that opened the opportunity for the $O(\log^* m)$ results of the present abstract is in making the probability of success in mapping depend exponentially on q_i. This provides exponential reduction in the number of active items, enabling q_{i+1} to be exponential in q_i. The Allocation step reallocates enough processors that become available and thereby "exponential-ladder" acceleration is accomplished and results in $O(\log^* m)$ rounds. Next, we make the last statement more precise.

Intuition for the analysis
Let m_i be the number of active items at the end of

iteration i. Iteration i *succeeds* if $m_i \leq m/q_{i+1}^6$ and all its preceding iterations succeeded. If all iterations succeed then the series $\{m_i\}$ looks as follows:

$$\frac{m}{q_1^6} \to \frac{m}{2^{q_1^6}} \to \frac{m}{2^{2^{q_1^6}}} \to \frac{m}{2^{2^{2^{q_1^6}}}} \to \cdots$$

Therefore, after $\log^* m$ successful iterations all items become inactive.

Before describing our algorithm in more detail, we sidestep to mention another problem and its solution. These problem and solution are needed for the allocation step. Given is an array of n cells with a processor standing by each of them. At most m cells contain one item each and the other cells are empty. Let $d > 1$ be an integer such that $m^d \leq n$. The *d-polynomial approximate compaction (d-PAC)* problem is to insert the items into an array $[1..m^d]$. Ragde [26] gave a constant-time deterministic algorithm using n processors for the 4-PAC problem. We extended Ragde's ideas to get a constant-time algorithm for the 3-PAC problem, and even for the 2.829-PAC problem. Specifically, we have

Theorem 2 *Using n processors, the 3-PAC problem (where the items are to be inserted into an array $[1..3m^d]$) can be solved in constant time and $O(\sqrt{n}m^2)$ space.*

The proof is omitted from this abstract. The allocation step is described using the 4-PAC algorithm. The 3-PAC algorithm can be used instead, to result with better constants.

Allocation.

The Allocation Step has two substeps. We aim at using the 4-PAC algorithm at the second substep of the Allocations Step. Consider partitioning array A into subarrays of size $7q_i^5$ each. For each subarray with at most q_i active items, the 4-PAC algorithm will compact all active items into the first q_i^4 cells. Since we can afford $7q_i$ processors for each of the first q_i^4 cells of each subarray, this results in assigning $7q_i$ processors to each such active item. The first substep of the Allocation Step will thin-out the number of active items in each subarray and thereby enable application of the 4-PAC algorithm in the second substep. (A similar idea was used by Raman [28].)

- (*First Substep: thinning-out*) We use an auxiliary array with $m/7q_i$ cells, where each cell has $7q_i$ processors standing by. Each active item tries to occupy a cell of the auxiliary array: the item selects a random cell; if no other item selected the same cell, the item occupies the cell; if the cell was selected by other items the item aborts (together with the other items that selected the same cell). Each occupying item gets $7q_i$ processors and thereby becomes participating.
  An active item fails in the thinning-out step with probability polynomially small in q_i (specifically, with probability $\leq \frac{m/q_i^6}{m/7q_i} = 7q_i^{-5}$). Our probabilistic analysis actually provides the following stronger statement. Consider a subarray (of A) of size $7q_i^5$. Following the present substep, the number of its non-participating active elements is $\geq q_i$ with probability $\leq 2^{-7q_i}$.

- (*Second Substep: in-subarray allocation*) Separately for each subarray of size $7q_i^5$: apply Algorithm 4-PAC for compacting the non-participating active items into the first q_i^4 cells. Since the 4-PAC algorithm is deterministic, a compaction will not fail if the subarray contains $\leq q_i$ non-participating active items.

Mapping.

- *(finding empty cells)* For each participating item, each of its allocated processors tries to occupy a cell in B (by selecting a random cell and ending up occupying it if it was empty and was not simultaneously selected by others). If one, or more, processors of a participating item occupied a cell, the item is called *successful*. The probability that a participating item is not successful is exponentially small in q_i (specifically, $\leq 2^{-7q_i}$).

- *(selecting one cell)* For each successful participating item, one of the occupying processors is selected. The item is mapped to the cell occupied by this processor.

Wrapping-up the probabilistic argument.
An active item may fail to become participating in the allocation step. A participating item may fail to be mapped in the mapping step. The probability of each of these failures is exponentially small in q_i (specifically, each is $\leq 2^{-7q_i}$). Consequently, the number of active items at the end of the iteration becomes at most $m/(2^{q_i})^6$, with very high probability. (We only note here that a careful probabilistic analysis is required to cope with minor probabilistic dependence among events that may occur.)

Finally, we note that there is an initialization step, in which the number of active items is decreased to at most m/q_1^6 (q_1 is set in the analysis part which is omitted) with high probability.

3 Representative Selection

Given is an array A of n cells, with a processor standing by each cell. At most m cells contain one item each and the other cells are empty. Each item has a *value* from a universe $\{0, 1, \ldots, N-1\}$. The *value-set* of a value v is the set consisting of all items whose value is v. The *representative selection (RS)* problem is to map the items into an array $B[1..cm]$ (for some constant $c \geq 1$), so that any two items are mapped to the same cell of B if and only if they belong to the same value-set. The mapping will provide a *representative* for each value-set. In other words, a one-to-one function from the set of input values to a range whose size is linear in the number of items, is to be found.

Since the RS problem is closely related to the LAC problem of the previous section, we would like to modify the algorithm for LAC into an algorithm for RS. For this, the following concern must be addressed.
Consistency: items from the same value-set must agree on the same representative. To accomplish this, the mapping will be a function of the value.

We change only the Mapping step. The Allocation step of the LAC algorithm remains unchanged.

To ensure consistency each iteration selects at random a function that maps items based on their values. Specifically, the function is selected from a class of universal hash functions of the simplest kind (i.e., 2-universal [5]).
Two specific considerations should be made:
1. If a participating item x has k processors, then it is replaced by k pairs $< x, 1 >, < x, 2 >, \ldots, < x, k >$. Each pair is mapped by the function into array B, and thereby "tries to find a representative". In case several pairs (of the some value v) succeed, all items whose value is v must consistently pick the same representative.
2. In case an active item is not participating in an iteration it still "needs to know" in later iterations whether other item from its value-set were mapped in the iteration.

Probabilistic arguments
As in the LAC algorithm, an active item remains active at the end of iteration i with probability exponentially small in q_i (specifically, $\leq 2^{-7q_i}$). However, items may be highly dependent (they may have the same value). Therefore, we cannot claim a global decrease in the number of active items with high probability, but only with a constant probability (e.g. $\geq 1/2$). It is essential that *all* iterations are successful. To increase the success probability of each iteration to the appropriate level, each iteration is repeated $O(\log\log^* m)$ times (i.e., q_i is increased only once every $O(\log\log^* m)$ iterations).

We have

Theorem 3 *There exists an algorithm for the representative selection problem that runs in* $\overline{O}(\log^* m \log\log^* m)$ *time using n processors.*

4 Hashing

Given is a set S of n elements from a universe $U = \{0, 1, \ldots, N-1\}$. The *hashing* problem is to find a one-to-one $h : S \mapsto [1..dm]$ (for some constant $d \geq 1$) such that for any $x \in U$, $h(x)$ can be evaluated in constant time.

We will construct a hash function that is based on a 2-level scheme, as suggested by [11]. The first level consists of a single function from the input set into $[1..n]$. It partitions the input set into *buckets*: bucket i consists of the input elements that are mapped to i. In the second level, each bucket is allocated with a private memory region, called *block*, of an appropriate size. Then, a one-to-one function is found for each bucket that maps the elements of the bucket into its block.

As in previous parallel hashing algorithms ([13, 24]), the main effort is in finding a one-to-one mapping for each bucket in the second level. The problem is twofold. First, an appropriate block should be allocated to each bucket. Then, the appropriate function into this block should be found.

We give an algorithm that follows the paradigm that is implied by the LAC algorithm. Assume that the number of buckets at the beginning of iteration i is at most n/q_i^c, for some constant c. In the Mapping step, each bucket fails to find an appropriate mapping with probability exponentially small in q_i. To make this statement about probabilities possible, the first level function is selected at random from a class of hash functions, suggested in [10]. The important property of these functions is that a bucket is of size q (i.e., it contains q elements) with probability exponentially small in q.

Each bucket is allocated with q_i blocks of size q_i^2 each, and with q_i processors – one per block. Each processor selects at random a function that maps the bucket's elements into its block. So, each bucket has q_i functions, and we would like to find out whether at least one of them is one-to-one, and if so pick such function. We use the same functions as in [11]. To check whether a function is one-to-one, we allocate q_i processors to each element (one per each function). If the bucket is of size $\leq q_i$ then each function is one-to-one with probability $\geq 1/2$; therefore, the probability that none of the functions is one-to-one is exponentially small in q_i. Finally, the bucket is of size $\geq q_i$ with probability exponentially small in q_i.

Since we use the representative selection algorithm of the previous section as one of the steps of the hashing algorithm (that was not specified in the present abstract), we have

Theorem 4 *There exists an algorithm for the hashing problem that runs in* $\overline{O}(\log^* n \log(\log^* n))$ *time using n processors.*

5 Generating random permutation

The two known algorithms for computing a random permutation [25, 27] consist of two basic steps. First, the items $1, \ldots n$ are mapped at random into a linear size array. Their order in the array gives an implicit random permutation. In order to compute the permutation explicitly, the items are compressed into an array $[1..n]$, by using a prefix sums algorithm.

The new linear approximate compaction (LAC) algorithm enables an $\overline{O}(\log^* n)$ execution time of the first step towards the computation of a random permutation. However, the time complexity for the compression step is lower bounded by $\Omega(\log n / \log\log n)$ [2].

We suggest an alternative approach that leads to a randomized algorithm for generating random permutation in $\overline{O}(\log^* n)$ time (!)

The key idea is to take the implicit random permutation that results from the LAC step, and refer to it as a "permutation in the canonical form for a cyclic notation", as explained below.

Following [22], we describe the cyclic notation for permutations and their canonical form. A permutation π in a *cyclic notation* is a partition of the numbers $[1..n]$ into ordered sets. These sets are called *cycles* and denoted $(x_1 x_2 \ldots x_j)$ since the order $(x_1 x_2 \ldots x_j)$ is defined to be equivalent to the order $(x_2 x_3 \ldots x_j x_1)$; and, by induction, we get equivalence among all cyclic orders. This notation defines a permutation in the following way. A cycle $(x_1 x_2 \ldots x_j)$ implies: $x_1 \xrightarrow{\pi} x_2 \xrightarrow{\pi} \cdots x_j \xrightarrow{\pi} x_1$; that is, $\pi(x_1) = x_2$, $\pi(x_2) = x_3$, and so on. The *canonical form* for the cyclic notation is achieved as follows.

- Each cycle is written with its smallest number first.
- The cycles are ordered in decreasing order of the first number in the cycle.

There are two nice properties of this representation that we use: (1) For each permutation, there is a unique representation in the canonical form. Therefore, there is a 1-1 correspondence between permutations in the standard form and permutations in the canonical form. (2) Consider a sequence $x_1, x_2, ..., x_n$ of the n numbers $[1..n]$ that represents a canonical cyclic notation. Each x_i which is first in its cycle is a "prefix-min"; where x_i is defined to be a *prefix minimum* if $x_i = MIN\{x_j | 1 \leq j \leq i\}$. Thus, the end points of a cycle do not have to be marked; elements which are first in their cycle can be identified by finding whether they are prefix-minima.

We are now ready to present the random permutation algorithm.

Algorithm RP

1. Randomly insert the items $\{1, \ldots, n\}$ into an array $A[1..4n]$. Let the value of each empty cell in A be $n+1$.
2. Identity each number that is a prefix-minimum.
3. For each item find its left and right nearest items in A. For each item that is a prefix-min, find the nearest prefix-min on the left.
4. Compute a random permutation π. Let j be right nearest item to i in A. $\pi[i]$ is the item j unless j is a prefix minimum. If j is a prefix minimum then $\pi[i]$ is the left nearest prefix-min of i.

Claim: Array π contains a random permutation of $[1..n]$.

Complexity: Step 1 can be done in $O(\log^* n)$ time (w.h.p.) using Algorithm LAC. Step 2 can be done in $O(\log^* n)$ time by using the prefix-min algorithm of [3], that is designed for inputs from the domain $[1..n]$. Step 3 can be done in $o(\log^* n)$ time by applying the nearest-one algorithm of [4] or of [26]. Step 4 can be done in additional constant time by the available information.

We have

Theorem 5 *There exists an algorithm for computing a random permutation that, using n processors, runs in $\overline{O}(\log^* n)$ time. The time complexity of the same algorithm is also $O(\log^* n)$ with probability $\geq 1 - n^{-\alpha}$, for any constant $\alpha > 0$.*

6 Making algorithms optimal

A scheme for modifying fast algorithms into optimal ones that is applicable to a certain type of algorithms is presented. It uses the linear approximate compaction algorithm as a key procedure. The scheme can be viewed as automatizing the known work-time methodology (which is guided by Brent's theorem) [21, Chapter 1.3] under certain circumstances. As a result, all the algorithms in this paper become optimal with $O(\log^* n \log(\log^* n))$ time (w.h.p.).

Assume that we are given an algorithm $\mathcal{A}$ that, using n processors, takes time t. Further assume that if a processor becomes inactive at some step, it remains inactive in all subsequent steps. An active processor in a step is counted as an *operation*. We say that a simulation of the algorithm by n_1 processors in t_1 time is optimal if the time-processor product, $n_1 t_1$, is proportional to the number of operations of the algorithm. We have:

Theorem 6 (The randomized optimizer) *Algorithm $\mathcal{A}$ has an optimal simulation in $O(t + \log^* n \log(\log^* n))$ time (w.h.p.).*

In a deterministic version, a prefix sums algorithm is used instead of the LAC algorithm, and the *additive* time overhead is $O(\log n)$. Specifically, we have,

Theorem 7 (The deterministic optimizer) *Algorithm $\mathcal{A}$ has an optimal simulation in $O(t+\log n)$ (worst case) time.*

Proof of Theorem 6 (sketch): The processors of Algorithm $\mathcal{A}$ are called *virtual processors*. We emulate Algorithm $\mathcal{A}$ by p *real* processors. In each step of Algorithm $\mathcal{A}$ a real processor is responsible for a subset of the *active* virtual processors. It emulates the processors in its subset sequentially. An active virtual processor at a certain step is a *task* of the responsible real processor. The *load* of a processor is the number of tasks that it has. If the maximum load of all processors is constant, then we are done: Algorithm $\mathcal{A}$ can proceed without any modification.

Without loss of generality, we assume that the total number of operations is equal to the number of virtual processors n. (We can always add processors that are inactive from the beginning.) Let n_i be the number of active processors in round i. Then, $n_0 = n$ and $\sum_{i=1}^{t} n_i = n$. We will first assume that there is a geometric decrease in the number of active processors in each round, i.e. that $n_{i+1} = n_i/2$. It will be later shown how to handle the general case.

We keep the invariant that, in round i, each real processor is responsible to at most $32n_i/p$ virtual processors. Thus, from round $\log(n/p)$ and on, each real processor is responsible for at most $O(1)$ virtual processors. Simulating round i will take at most $32n_i/p$ steps, and all rounds will be simulated in at most $\sum_{i=1}^{\log(n/p)} 32n_i/p = O(n/p)$ time.

Assume that in step $i-1$ the load of each processor is at most $32n_{i-1}/p$. We show how to ensure that in round i the load of each processor will be at most $32n_i/p$ tasks (a similar technique to reduce the load was used in [12]). Note that the average load in round i is n_i/p. A processor is *over-loaded* in round i if its load is at least $16n_i/p$. The number of over-loaded processors is at most $\frac{n_i}{16n_i/p} = \frac{p}{16}$.

By using Algorithm LAC, the over-loaded processors are mapped into an array $B[1..4\frac{p}{16}]$. At each cell of array B, 4 processors are standing by. The tasks of each over-loaded processor are distributed among these 4 processors – each one of them copies its new tasks into its local array of tasks. By assumption, the load of each over-loaded processor is at most $32n_{i-1}/p = 64n_i/p$. Therefore, each processor copies at most $\frac{1}{4} \times 64n_i/p = 16n_i/p$ new tasks. The load of each processor now consists of at most $16n_i/p$ old tasks and at most $16n_i/p$ new tasks. Thus, the load of each processor is now at most $32n_i/p$, as required.

Copying the new tasks for a processor in each round takes at most $16n_i/p$ steps, summing to a total of $O(n/p)$ steps in all rounds. Algorithm LAC is employed $O(\log(n/p))$ times, summing to a total of $O(\log^* n \log(n/p))$ time (w.h.p.). Therefore, the time required to have at most p active virtual processors is $O(n/p + t + \log^* n \log(n/p))$ (w.h.p.). Taking $p = n/(\log^* n \log\log^* n)$, the theorem follows (for the case where a geometric decrease in the series $\{n_i\}$ is assumed).

We now deal with the general case, where the only thing we know about the series $\{n_i\}$ is that $n_i \le n_{i-1}$. Throughout the simulation we keep an upper-bound estimate η_i on n_i. (Specifically, $\eta_i \ge n_i/32$.) The load of each processor in round i is bounded by $32\eta_i/p$. Initially, $\eta_0 = n_0$. For each $i \ge 0$, we will have either $\eta_{i+1} = \eta_i$ or $\eta_{i+1} = \eta_i/2$. Once every several rounds, we try to update the estimate. An *update attempt* in round i is carried out by assuming that $n_i \le \eta_{i-1}/2$, and trying to reduce the load of all processors to at most $16\eta_{i-1}/p$. This is done by using Algorithm LAC in a similar way to what we showed above. There is one difference though: since we do not know if Algorithm LAC should successfully terminate (it depends on the actual value of n_i) we run it for $O(\log^* n)$ steps and halt, even if it does not terminate.

If Algorithm LAC successfully terminates then we have a *successful* update: the estimate is updated by letting $\eta_i = \eta_{i-1}/2$; otherwise we have an *unsuccessful* update attempt: the estimate remains the same, i.e. $\eta_i = \eta_{i-1}$. Note that if indeed $n_i \le \eta_{i-1}/2$ then we have a successful update with high probability. If there is no update attempt in round i then the estimate remains the same, i.e. $\eta_i = \eta_{i-1}$.

The only thing that remains is to decide how often to do the update attempts. Recall that an update attempt takes $O(\log^* n)$ time. An update attempt is followed by a *phase* of $\log^* n$ steps of the *real* processors; however, a phase never ends in the middle of a round of Algorithm $\mathcal{A}$: additional steps by the real processors are added to finish the simulation of the round of Algorithm $\mathcal{A}$. Then, the next update attempt is being done. As a result, each phase takes about the same time as the update attempt that follows it. After the estimate becomes less than p, we do not make any more update attempts. This simple strategy leads to $O(n/p+\log^* n \log(n/p))$ time (w.h.p.), as shown below.

Assume that an update attempt is done in round i. If the update attempt fails then (w.h.p.) $n_i \ge \eta_{i-1}/2$. Recall that the load of each processor in round i is bounded by $32\eta_i/p$ (w.h.p.). Consider all the rounds of Algorithm $\mathcal{A}$ that fall in phases that are followed by unsuccessful update attempts. Then, simulation of all such rounds takes a total of $\sum_{i=1}^{t} 32\eta_i/p \le \sum_{i=1}^{t}(32 \cdot 2n_{i+1}/p) = O(n/p)$ time (w.h.p.). On the other hand, note that after each successful update the estimate is halved. Therefore, the number of phases that are followed by a successful update is at most $\log(n/p)$, each of them takes the maximum among $O(\eta_i/p)$ (where i is the first step in the phase) and $O(\log^* n)$ time. Since the series of η_i following successful phases decreases geometrically, the theorem follows. $\Box$

Applications.

The randomized optimizer is used to get optimal algorithms for the problems discussed in this paper, using the non-optimal algorithms given above. For each of the problems we need to show an algorithm that reduces the number of active keys to the desired level, and that meets the specifications of Theorem 6. In fact, such algorithms are already used as initialization procedures for each of the algorithms given in the previous sections. (Their role is to reduce the number of active items to at most m/q_1^6.) These initialization procedures even meet the stronger requirement of geometric decrease in the number of active processors. To keep this abstract short, we will not describe these algorithms.

We mention one additional application for the optimizer: an optimal algorithm for the 2-ruling set problem. This problem has an $O(\log^* n)$ time algorithm using n processors [6], and an optimal algorithm that takes $O(\log^{(i)} n)$ time, for any constant $i \geq 1$ Han [20]. We explain how to use the randomized optimizer to obtain a faster (though randomized) optimal algorithm.

A simple $\overline{O}(\log n)$ time algorithm for the 2-ruling set was presented in [32]. In each round, some of the nodes are selected to be in the 2-ruling set, and other nodes are disqualified from being in the set. The rest of the nodes remain *active*. With high probability, there is a geometric decrease in the number of active nodes in each round. Therefore, using p processors, the optimizer can be used to reduce the number of active nodes to at most $O(p)$ in $O(n/p + \log^* n \log(n/p))$ time (w.h.p.). At this stage, the active nodes form a collection of disjoint linked lists. The deterministic $O(\log^* n)$ time algorithm of [6] can be used for each of these lists separately.

We summarize the results on optimal algorithms in the following theorem.

Theorem 8 *The following problems can be solved in $\overline{O}(\log^* n \log(\log^* n))$ time, using an optimal number of processors.*

- *Hashing.*
- *Representative selection.*
- *Linear approximate compaction.*
- *Random permutation.*
- *2-ruling set.*

7 Sublogarithmic optimal integer sorting

The *multiple linear approximate compaction (MLAC)* problem:
Input: An array A of n cells. Each cell contains one item that is labeled by a non-negative integer from the range $[0..n]$. Let G_i be the set of items labeled by i. The input provides also a sequence $m_1, m_2, \ldots, m_n$, where $\sum_{i=1}^{n} m_i = cn$ for some constant $c > 0$, and m_i is an upper bound on $|G_i|$.
Output: All n items inserted into an array $D[1..4cn]$ (linear in n) such that all the items in G_1 appear first, all the items in G_2 appear next, and so on.

Given are n integers from the domain $[1..n \log^d n]$ for some constant $d \geq 0$. The *integer sorting* problem is to sort these n integers. The integer sorting algorithm of Rajasekaran and Reif [27] consists essentially of reducing the problem into the MLAC problem (they do not use this name), and then solving it. It is easy to implement their reduction in $O(\log n / \log\log n)$ time (w.h.p.), using $n \log\log n / \log n$ processors. Therefore, improving the MLAC algorithm leads to an improved algorithm for integer sorting. In this section we show how MLAC can be solved, with high probability, in $O(\log n / \log\log n)$ time and an optimal number of processors. As a result we have

Theorem 9 *The integer sorting problem of n keys can be solved in $O(\log n / \log\log n)$ time, using $n \log\log n / \log n$ processors, with probability $\geq 1 - n^{-\alpha}$, for any constant $\alpha > 0$.*

We first give a short overview of Rajasekaran-Reif's algorithm and indicate what are the changes being made. They first show that n keys from the range $[1..\log n]$ can be sorted in $O(\log n)$ time (w.h.p.), using $n/\log n$ processors. This sorting is stable, i.e., the order between keys with the same value remains the same. It is easy to see that in a very similar way, n keys from the range $[1..\log n / \log\log n]$ can be stably sorted in $O(\log n / \log\log n)$ time (w.h.p.), using $n \log\log n / \log n$ processors. Thus, the idea of radix sort [22] can be implemented to reduce the sorting problem to the problem of sorting n keys from the range $[1..n/\log^d n]$, for any constant $d > 0$. The range will be assumed to be $[1..n/\log^3 n]$.

The integer sorting problem with the new range is reduced now to the MLAC problem as follows. First, an estimate is computed for the size of each set G_i, the set of keys with value i. Then, the MLAC algorithm is used to insert the keys properly into an array of size $O(n)$. Finally, a prefix sums algorithm is used to compress the sorted keys in the array. Since a prefix sums can be done in $O(\log n / \log\log n)$ time, using an optimal number of processors [6], we only need to discuss the estimation step. This step is based on a sampling technique: a sample of size $n/\log^2 n$ of the keys is selected at random. This sample is small enough to be processed by a sorting algorithm with a non-linear operation count, and is large enough to provide the required estimates. The number of keys from each set G_i in the sample is the basis for the estimate of $|G_i|$. These numbers are computed by sorting the sample: an $O(\log n)$ general sorting algorithm is used in [27] (note that the number of processors is $n/\log n$ and the size of the sample is $n/\log^2 n$). Rajasekaran and Reif gave also an integer sorting algorithm that sorts k keys in $O(\log k/\log\log k)$ time (w.h.p.), using $k(\log\log k)^2/\log k$ processors. We use this algorithm to sort the sample in a sub-logarithmic time.

A high-level description of the MLAC algorithm follows. It consists of three main parts. In the first part, *most* of the items are inserted into an array B of size $2cn$. In the second part, the remaining items are sorted into an array C. The third part consists of merging arrays B and C into array D.

Part 1. (i) Array B is partitioned into subarrays $B_1 ... B_n$; subarray B_i has size $2cm_i$, $1 \leq i \leq n$. This is done using a prefix sums algorithm.

(ii) Each processor is given a pool of $\log n/\log\log n$ items; the processor tries to find for each item in its pool a private cell in the appropriate subarray. This is done in $4\log n/\log\log n$ steps. In a step, the processor takes an item that has not yet found a private cell; let G_i be the set of the item; the processor randomly selects a cell in B_i and tries to write its index into the cell; if the processor succeeds then it inserts the item into the cell, and moves to the next item in its pool in the next step; otherwise, the processor will try to find a private cell for the same item in the next step, as well.

(iii) A prefix sums algorithm is used to compress the items in array B.

Lemma 1 *Following step (ii), the number of active items is $n/\log^2$, with probability $\geq 1 - n^{-\alpha}$, for any constant $\alpha > 0$.*

Proof (sketch): The probability of success in each attempt is at least 1/2. By Chernoff bounds, the probability that a processor did not successfully insert all its items is exponentially small in $\log n/\log\log n$. The number of such processors is therefore, with high probability, at most $n/\log^3 n$. Thus, the number of active items is, with high probability, at most $n/\log^2 n$. $\Box$

Part 2. The active items are sorted into an array C, by using the $\log n/\log\log n$ (non-optimal) integer sorting algorithm of Rajasekaran and Reif [27, Theorem 4.2].

Part 3. The final step is to merge arrays B and C into an array D. Merging can be done in $O(\log\log n)$ using an optimal number of processors [23].

Therefore, with high probability, the algorithm terminates in $O(\log n/\log\log n)$ time using an optimal number of processors.

Acknowledgments. We would like to thank Omer Berkman and Yossi Gil for fruitful discussions.

References

[1] R.J. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proc. 2nd Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 95–102, 1990.

[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*, pages 83–93, 1987.

[3] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin. Some triply-logarithmic parallel algorithms. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 871–881, 1990. Also in Fast Routing around a rectangle – in preparation.

[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] J.L. Carter and M.N. Wegman. Universal classes of hash functions. *J. Computer and System Sciences*, 18:143–154, 1979.

[6] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[7] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[8] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Proc. of the 29th IEEE Annual Symp. on Foundation of Computer Science*, pages 524–531, 1988.

[9] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. 1st Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[10] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 6–19, 1990.

[11] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. of the Association for Computing Machinery*, 31:538–544, 1984.

[12] J. Gil. Fast load balancing on PRAM. Manuscript, 1990.

[13] J. Gil and Y. Matias. Fast hashing on a PRAM. In *Proc. of the 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, 1991.

[14] J. Gil, Y. Matias, and U. Vishkin. A fast parallel dictionary. In preparation, 1990.

[15] J. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 244–253, 1990.

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

[17] T. Hagerup. Constant-time parallel integer sorting. In *this proceedings*, 1991.

[18] T. Hagerup. Fast parallel generation of random permutations. In *to be presented at the 18th ICALP*, 1991.

[19] T. Hagerup and M. Nowak. Parallel retrieval of scattered information. In *Proc. of 16th ICALP, Springer LNCS 372*, pages 439–450, 1989.

[20] Y. Han. Matching partition a linked list and its optimization. In *Proc. 1st Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 246–253, 1989.

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

[22] D.E. Knuth. *The art of computer programming*, volume 3, Sorting and searching. Addison-Wesley, Reading, 1973.

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

[24] 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 *J. of Algorithms.*

[25] G.L. Miller and J.H. Reif. Parallel tree contraction and its application. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 478–489, 1985.

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

[27] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18:594–607, 1989.

[28] R. Raman. The power of collision: Randomized parallel algorithms for chaining and integer sorting. In *Proc. of the 10th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, December 1990. Also in TR-336, Univ. of Rochester Computer Science Dept., revised January 1991.

[29] R. Raman. Optimal sub-logarithmic time integer sorting on a CRCW PRAM (note). Submitted for publication, 1991.

[30] J.H. Reif. An optimal parallel algorithm for integer sorting. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 496–503, 1985.

[31] G.E. Shannon. Optimal on-line load balancing. In *Proc. 1st Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 235–245, 1989.

[32] U. Vishkin. Randomized speed-ups in parallel computations. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 230–239, 1984.