The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting*

Rajeev Raman†

University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 336

March 1990 (Revised, April 1991)

Abstract

We address the problem of sorting n integers each in the range $\{1, \ldots, m\}$, for $m = n^{O(1)}$, in parallel on the PRAM model of computation. We present a randomized algorithm that runs with very high probability in $O(\log n / \log\log n)$ time with a processor-time product of $O(n \log\log m)$ and $O(n)$ space on the CRCW (COLLISION) PRAM [13]. The improvements that this algorithm makes over existing ones [5, 20, 27] include a weakening of the model of computation used and reducing the space requirement to $O(n)$, without increasing the time needed or work done. For larger values of m our algorithm is better than existing algorithms in several other ways as well. We show that the algorithm can be analyzed using $O(\log^{O(1)})$-wise independence, which implies that the amount of true randomness needed is small.

We also give an improved randomized algorithm for the the problem of *chaining* [22, 29]. An interesting subroutine used is an algorithm for solving a class of processor allocation problems quickly. The algorithms for chaining and integer sorting both make use of efficient algorithms for the construction of the fast priority queue of van Emde Boas [35].

The University of Rochester Computer Science Department supported this work.

*A preliminary version of this paper was presented at the *10th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, December 1990, Bangalore, India.

†email: raman@cs.rochester.edu

1 Introduction and Previous Work

The problem of *integer sorting* in parallel has received a lot of attention in recent years. This is a restricted version of sorting where the keys are integers that are known to lie within a range that is not too large compared to the size of the set to be sorted. This problem is apparently simpler than *general* (comparison-based) sorting, where the keys are not known to have any particular structure, and is a frequently occurring problem in sequential and parallel algorithm design. A special case called the *polynomial* integer sorting problem is of special interest: this is the problem of sorting n integers each in the range $\{0, \ldots, n^{O(1)}\}$. This problem can be solved sequentially in linear time on a RAM using radix sorting [25], in contrast to the well-known $\Omega(n \log n)$ lower bound on comparison-based sorting. We are interested in studying this problem in the parallel setting.

Our models of computation belong to the PRAM family, and we make the usual classification of PRAMs according to their concurrent-writing and reading abilities into the EREW, CREW and CRCW varieties (see, *e.g.*, [16]). We follow current practice in saying that a parallel algorithm achieves *optimal speedup* if its processor-time product is within a constant factor of the best known sequential algorithm (for convenience, we say that a parallel algorithm that achieves optimal speedup is *optimal*). Since the best sequential algorithm for any problem is trivially an optimal parallel algorithm, we are interested in parallel algorithms that optimally achieve a run time that is polylogarithmic in n.

CRCW PRAMs are further classified according to the mode of write-conflict resolution. These include PRIORITY, in which the highest numbered processor succeeds in the event of a write conflict, ARBITRARY, in which an arbitrary processor succeeds in writing and COMMON, which permits a concurrent write only when the values being written are the same. We focus on an interesting model called COLLISION, introduced in [12], which permits only *detection* of write conflicts. In this model, when a write conflict occurs, the contents of the memory location being written to are erased and a special collision symbol appears in that location instead (this is reminiscent of the conflict resolution scheme used by Ethernet). Fich *et al.* showed that COLLISION is strictly weaker than ARBITRARY, *i.e.*, COLLISION cannot simulate ARBITRARY without loss of time or without increasing the number of processors [12, 13]. However, recent work has shown that even the seemingly weak property of collision detection, when used in conjunction with randomization, is quite powerful [7, 23]. As a side issue, our paper modifies some existing randomized sorting algorithms to work on COLLISION, with no degradation in performance, and so may be considered as providing more proof in that direction.

Efforts at finding optimal parallel algorithms for sorting problems have met with mixed success. The general sorting problem for PRAMs has been fully solved: an optimal logarithmic time EREW PRAM algorithm was presented by Ajtai *et al.* and by Cole [1, 8]. Integer sorting has not proved as easy: the only optimal algorithm for integer sorting is a randomized one for special case of sorting n integers in the range $\{1, \ldots, n\}$, due to Rajasekaran and Reif [31]. This algorithm is not *stable*, *i.e.*, the order of records with equal keys may not remain the same in the output as in the input, so this procedure cannot be used to solve the polynomial integer sorting problem by radix sorting. Recently, a fast $o(\log \log n)$ time optimal algorithm for non-stably sorting integers in the range $\{1, \ldots, n\}$ such that the

Author[s]	Model	Time	Space
Hagerup (87)	PRIORITY	$O(\log n)$	$O(n^{1+\varepsilon})$
Bhatt *et al.*(89)	ARBITRARY	$\Theta\left(\frac{\log n}{\log\log n}\right)$	$O(n^{1+\varepsilon})$
Matias and Vishkin (90)	Rand. ARBITRARY	$O(\log n)$ (expected)	$O(n)$
New	Rand. COLLISION	$\Theta\left(\frac{\log n}{\log\log n}\right)$ (w.h.p.)	$O(n)$

Figure 1: CRCW polynomial integer sorting algorithms. (All the above have $PT = O(n \log\log n)$.)

output is in the form of a linked list was described by Hagerup [21], but again, this has no implications for polynomial integer sorting.

The best parallel algorithms known for solving the polynomial integer sorting problem have processor-time (PT) products of $O(n \log\log n)$. Hagerup [20] achieved the above PT bound with a run time of $O(\log n)$. His algorithm runs on PRIORITY and uses $O(n^{1+\varepsilon})$ space, for any fixed $\varepsilon > 0$. This was subsequently improved to $O(\log n/\log\log n)$ time in [5] on the ARBITRARY model, using $O(n^{1+\varepsilon})$ space again. Very recently, Matias and Vishkin [27], using an algorithm similar to that of Bhatt *et al.*, achieved an *expected* $O(\log n)$ running time, $O(n \log\log n)$ PT product and $O(n)$ space on ARBITRARY. Our algorithm has the same PT bound and runs in $O(\log n/\log\log n)$ time with high probability on the COLLISION model, using $O(n)$ space. Since integer sorting is harder than computing any symmetric function, the result of Beame and Håstad [3] shows that $\Omega(\log n/\log\log n)$ is a lower bound on the (expected) run time of any (randomized) algorithm that uses a $poly(n)$ number of processors and runs on PRIORITY. Figure 1 summarizes the known results on CRCW polynomial integer sorting.

Our approach is different from the existing algorithms. First we randomly sample the set of integers to be sorted, and sort the sample (which is small) using existing general sorting algorithms. Then we construct in parallel of a *static* data structure (*i.e.*, one that allows queries only) on the sample that answers predecessor queries quickly, and collect together the elements in the original set that lie in between two consecutive elements in the sample. Sorting these collections turns out to be inexpensive since their size is small with high probability. A key point is that constructing static versions of sophisticated sequential data structures in parallel is relatively easy, and this allows us considerable flexibility. In what follows, an event that occurs "with high probability" occurs with probability $1 - n^{-\beta}$, for any pre-specified integer β. If an algorithm consumes $O(f(n))$ of any resource (time, space, work *etc.*) with high probability, we say that it consumes $\tilde{O}(f(n))$ of that resource (the notation is from [31]). We allow the constant in the big-oh to depend on β, if necessary. Also, when we say CRCW PRAM for the rest of this section, we mean any one of PRIORITY, ARBITRARY and COLLISION (*i.e.*, COMMON will be denoted separately). We now outline the improvements that our algorithm makes over existing ones.

1. Polynomial integer sorting on the CRCW PRAM*:* Our algorithm runs on COLLISION,

a weaker model than the ARBITRARY model used by Bhatt *et al.* and Matias and Vishkin. The algorithms of Bhatt *et al.* and Matias and Vishkin repeatedly use the power of ARBITRARY to do exactly what was proved hard for COLLISION in [13], (*i.e.*, the "representative selection" problem), so weakening the model while preserving work done requires a different approach. As mentioned earlier, we also modify other existing algorithms to run on COLLISION, thus reinforcing belief in the "power of COLLISION".

Our algorithm also uses $O(n)$ space, which is an improvement over the $O(nm^{\epsilon})$ space used by Bhatt *et al.*, where m is the range of integers being sorted. The algorithm of Matias and Vishkin also achieves $O(n)$ space, but it runs in $O(\log n)$ expected time: *i.e.*, with probability $2^{-O(\alpha)}$ it uses $\Omega(\alpha \log n)$ time, as opposed to our high-probability sub-logarithmic running time. Matias and Vishkin achieve linearity of space by using perfect hashing, an idea that was independently discovered by us (the hashing subroutine in [27] is slower but more work-efficient than ours). Our algorithm also avoids the complex "monotonic" list ranking subroutine that is central to the speed of the algorithm of Bhatt *et al.*.

We also mention here that Gil and Matias [17] and Matias and Vishkin [28] have recently found much faster expected time hashing subroutines that run in almost constant expected time.

2. Arbitrary integer sorting on the CRCW PRAM*:* All the above mentioned algorithms (ours included) can be extended to sort integers in the range $\{1, \ldots, m\}$, for any m, by increasing the PT product to $O(n \log\log m)$ and increasing the running time by an additive factor of $O(\log\log m)$. The assumption made by all of these is that the processors can perform bitwise logical operations, addition and arbitrary left and right shifts of $O(\log m + \log n)$ bit integers in unit time (we call this the *basic* RAM). When $\log\log m$ is both $O(\sqrt{\log n})$ as well as $(1 + \Omega(1))\log\log n$, all these algorithms achieve optimal speedup since the best sequential RAM algorithm for this range of m is that of Kirkpatrick and Reisch [24] which runs in $O(n(\log\log m - \log\log n))$ time. The existing algorithms [5, 20, 27] are not optimal for larger values of m, however, since Fredman and Willard [15] recently gave an $O(n\sqrt{\log n})$ time RAM algorithm for this problem, for all values of m, assuming unit-time multiplication is added to the instruction set of the basic RAM (giving the *augmented* RAM), as well as randomization for reducing space usage. By using their *fusion tree* data structure, our algorithm can be modified to achieve this bound on the augmented RAM and so is currently the only one to achieve optimality for all values of m. A simple change to our algorithm makes it run in the best-possible $\tilde{O}(\log n/\log\log n)$ time on the ARBITRARY model using the basic RAM instruction set with a relatively small amount of extra work for all values of m, *i.e.*, even when the additive factor of $O(\log\log m)$ dominates the running time (theorem 16). It is not clear how to modify the algorithm of Bhatt *et al.* to get similar results.

From a practial point of view, the basic RAM instruction set no more "realistic" than the augmented RAM instruction set, since unit-time integer multiplication is usually available on real machines. In theoretical terms, there would be justification for considering an instruction set comprising only addition, comparison and bitwise logical operations, since all of these are AC^0 operations. The basic RAM includes as a constant-time operation at least one NC^1 operation, *viz.*, the arbitrary shift, and so there seems no immediate reason not to permit multiplication as well on the grounds that it is a complex (*i.e.*, NC^1

Model	Time	Space	Processors (P)/Work (PT)
Basic RAM Instructions			
Previous Best (Bhatt *et. al* [5]).			
ARBITRARY	$O\left(\frac{\log n}{\log\log n} + \log\log m\right)$	$O(nm^\epsilon)$	PT $= O(n\log\log m)$
NEW: Theorems 15, 16 and 17.			
COLLISION	$\tilde{O}\left(\frac{\log n}{\log\log n} + \log\log m\right)$	$O(n)$	PT $= \tilde{O}(n\log\log m)$
COLLISION	$\tilde{O}\left(\frac{\log n}{\log\log n} + \frac{\log\log m}{k}\right)$	$O(nm^\epsilon)$	P $= n2^k$
COMMON	$\tilde{O}(\log n \log^{(3)} n)$	$O(n)$	PT $= \tilde{O}\left(n\left(\frac{\log n}{\log\log n} + \log\log m\right)\right)$
CREW	$\tilde{O}(\log n \log\log n)$	$O(n)$	PT $= \tilde{O}\left(n\left(\frac{\log n}{\log\log n} + \log\log m\right)\right)$
Augmented RAM Instructions			
Previous Best (Matias and Vishkin [27]).			
ARBITRARY	Expected $O(\log n)$	$O(n)$	PT $= O(n\log\log m)$
NEW: Theorem 19			
COLLISION	$\tilde{O}\left(\frac{\log n}{\log\log n}\right)$	$O(n)$	PT $= \tilde{O}(n\min\{\sqrt{\log n}, \log\log m\})$
COMMON	$\tilde{O}(\log n \log^{(3)} n)$	$O(n)$	PT $= \tilde{O}\left(n\frac{\log n}{\log\log n}\right)$
CREW	$\tilde{O}(\log n \log\log n)$	$O(n)$	PT $= \tilde{O}\left(n\frac{\log n}{\log\log n}\right)$

Figure 2: New general integer sorting algorithms. The integers are assumed to lie in the range $\{1, \ldots, m\}$

operation). We also remind the reader that the word size required before Fredman and Willard's algorithm improves upon existing methods is very large, and since we use their data structure this is true of our algorithm as well.

3. Arbitrary integer sorting on COMMON *and the* CREW PRAM*:* Our algorithm can be viewed as a randomized CREW PRAM reduction from the general (stable) integer sorting problem to the problem of (non-stably) sorting integers in the range $\{1, \ldots, n\}$ that uses $O(n\min\{\sqrt{\log n}, \log\log m\})$, and so it may be helpful in solving the integer sorting problem on a CREW PRAM (or on the COMMON PRAM), where even less progress has been made to date. Existing *poly*$(\log(n))$ time algorithms for integer sorting on the CREW PRAM or CRCW (COMMON) PRAM [26, 32] have a PT bound of $O(n\log n/\log\log n)$ and are not much better than general sorting algorithms. For example, an algorithm due to Rajasekaran and Sen [32] stably sorts integers in the range $\{1, \ldots, n\}$ in $O(\log n \log\log n)$ time on a CREW PRAM and in $O(\log n \log^{(3)} n)$ time on COMMON, with the above PT product. Our algorithm can be used to extend these to super-polynomial ranges, with $O(n\min\{\sqrt{\log n}, \log\log m\})$ additional work. Table 1 summarizes the results on general integer sorting.

Given an array of elements $x_1, \ldots, x_n$, m of which are non-zero, the *(unordered) chaining*

problem is to chain all the non-zero elements into a linked list. If the order of the elements in the list is required to be the same as their order of occurrence in the original array, then we get the *ordered chaining* problem. To perform this task n processors, one for each element of the array, are given. One motivation for this problem is the following: database operations involve more than just finding all the records that satisfy a particular predicate: usually they have to be retrieved as well, in order to process them. In the parallel context, we might suppose that one processor is associated with each record, and when a query is presented, each processor determines whether the record it has matches the query predicate or not, following which the processors, in parallel, chain the records so found for further operations. This problem also has applications in parallel algorithms and appears as a subproblem in solving processor reallocation, merging of integers, subset compaction and integer sorting [5, 4, 18, 34]. This problem was studied by Hagerup and Nowak [22] and Ragde [29]. Ragde gives a COMMON algorithm for ordered chaining that runs in $O(\alpha(n))$ time using n processors (where $\alpha(n)$ is a inverse of Ackermann's function). He posed the problem of whether or not algorithms for chaining can be found which run in constant time using a linear number of processors, and was able to show that whenever $m < n^{1/4-\varepsilon}$, for some $\varepsilon > 0$, unordered chaining can be done in constant time. We address this problem, and show that even for values of m which are very close to n, *ordered* chaining can be done in $\tilde{O}(1)$ time. More precisely, we show that whenever $m < n/\log^{(i)} n$, for any fixed i, ordered chaining can be done in $\tilde{O}(1)$ time using n COLLISION processors. The chaining algorithm uses a processor allocation routine that may be of interest in its own right.

2 Preliminaries

2.1 Some Useful Subroutines

The *prefix sum problem* is the following: given n integers $x_1, \ldots, x_n$ in an array, to evaluate all of the sums $ps_i = \sum_{j=1}^{i} x_i$, for $1 \leq i \leq n$. Cole and Vishkin [9] prove that the prefix sum problem can be solved optimally in $O(\log n/\log\log n)$ time on the COMMON model of computation. However, their result only needs the ability to compute the logical OR function in constant time. Since COLLISION permits this as well, we have the following lemma:

Lemma 1 ([9], Theorem 2.2.2) *The prefix sum computation of n integers, each of $O(\log n)$ bits, can be done in $O(\log n/\log\log n)$ time on the* COLLISION *model of computation with optimal speedup.*

We can modify the following algorithm in [31] to run on the COLLISION model of computation (the proof can be found in the appendix):

Lemma 2 *A set of n general keys can be sorted in $\tilde{O}(\log n/\log\log n)$ time using $n(\log n)^{\varepsilon}$* COLLISION *processors, for any constant $\varepsilon > 0$, and $O(n)$ space.*

It is shown in [6] that $n \log n$ COMMON processors can step-by-step simulate any computation of n PRIORITY processors with $O(1)$ time per step slowdown. The simulation, however, increases the memory requirements by a factor of $O(n)$. Thus we get:

Lemma 3 *A set of n general keys can be sorted in $\tilde{O}(\log n/\log\log n)$ time using $n(\log n)^{1+\varepsilon}$* COMMON *processors, for any constant $\varepsilon > 0$, and $O(n^2)$ space.*

It is shown in [33] that the non-optimal sub-logarithmic time algorithm in [31] for the problem of sorting n integers in the range $\{1,\ldots,n\}$ can be modified to achieve optimality with the same speed. Lemma 2 can be used to (easily) show that the model of computation can be weakened to COLLISION:

Lemma 4 ([31], Theorem 4.2 and [33]) *A set of n integers in the range $\{1,\ldots,n\}$ can be sorted in $\tilde{O}(\log n/\log\log n)$ time using $n(\log\log n)/\log n$ processors on the* COLLISION *model of computation.*

The "leftmost one in memory" problem can be stated as follows: given n consecutive memory locations $M_1,\ldots,M_n$, each containing a 0 or a 1, find the least index i such that $M_i = 1$. Fich *et al.* [13] prove the following:

Lemma 5 ([13], pp. 609–610) *"Leftmost one in memory" problems can be solved in $O(1)$ time by n* COLLISION *processors, each associated with one memory location.*

Finally, we will use the following result due to Ragde related to the chaining problem. The original result used the ARBITRARY model of computation, but it is clear that COLLISION could be substituted to obtain the same result.

Lemma 6 ([29], Theorem 1) *Given n memory locations, $M_1,\ldots,M_n$, and a number m, we can in constant time using n* COLLISION *processors, for any value of m, either conclude that there are more than m non-zero items in the memory location, or move the items into M_1 through M_{m^4}.*

2.2 Bounds on Tails of Distributions

We now mention some well-known bounds on the tails of certain distributions. We say that X is $B(n,p)$-distributed if X is a binomially distributed variable with parameters n and p, *i.e.*, X is the random variable that corresponds to the number of successes in n trials, each of which independently has probability of success p. Then:

Lemma 7 (Chernoff bounds, [2]) *Let X a random variable that is $B(n,p)$-distributed, and let $LE(\varepsilon,n,p)$=$\Pr[X \le (1-\varepsilon)np]$. Then:*

$$LE(\varepsilon,n,p) \le e^{-\varepsilon^2 np/3}. \tag{1}$$

Similarly, if $GE(\varepsilon,n,p)$=$\Pr[X \ge (1+\varepsilon)np]$, then:

$$GE(\varepsilon,n,p) \le e^{-\varepsilon^2 np/2}. \tag{2}$$

Sometimes it is necessary to bound the number of successes in a collection of trials such that an upper bound on the probability of success in each trial holds irrespective of the outcome of the other trials. The following folklore lemma (mentioned in [30], *e.g.*), allows us to apply Chernoff bounds in these cases as well (a proof is included for the sake of completeness):

Lemma 8 *Let $t_1, \ldots, t_n$ be a set of Bernoulli trials such that, for any k, $1 \leq k \leq n$, and for all possible outcomes $o_1, \ldots, o_{k-1}$ of trials $t_1, \ldots, t_{k-1}$, $Pr[t_k$ succeeds$|o_1, \ldots, o_{k-1}] \leq p$. Then if X is the random variable that corresponds to the number of successes in these n trials and Y is a binomial random variable with parameters (n,p) then:*

$$\Pr[X \geq r] \leq \Pr[Y \geq r], \quad 0 \leq r \leq n.$$

PROOF. The proof is by induction on n. Let Y_i be a binomial variate with parameters (i, p) and let X_i denote the random variable that represents the number of successes in the first i trials $t_1, \ldots, t_i$. For the base case $i = 1$ it is obvious that $\Pr[X_1 \geq k] \leq \Pr[Y_1 \geq k]$, for any k. For $i = m + 1$, we have that, for all $k \geq 1$:

$$\begin{aligned} \Pr[Y_{m+1} \geq k] &= \Pr[Y_m \geq k] + p\Pr[Y_m = k-1] \\ &= (1-p)\Pr[Y_m \geq k] + p\Pr[Y_m \geq k-1]. \end{aligned} \tag{3}$$

Also,

$$\begin{aligned} &\Pr[X_{m+1} \geq k] \\ &\quad = \Pr[X_m \geq k] + \Pr[(X_m = k-1) \wedge t_{m+1} \text{ succeeds}] \\ &\quad = \Pr[X_m \geq k] + \Pr[X_m = k-1]\Pr[t_{m+1} \text{ succeeds} \,|X_m = k-1] \\ &\quad \leq \Pr[X_m \geq k] + p\Pr[X_m = k-1] \\ &\quad = (1-p)\Pr[X_m \geq k] + p\Pr[X_m \geq k-1] \end{aligned} \tag{4}$$

Comparing (3) and (4) and using the inductive hypothesis, we have the proof.□

For any random variable X let $E[X]$ denote its *expected*, or *mean* value. If $E[X] = \mu$, then the *kth central moment* of X is defined to be $\tau_X^k = E[(X-\mu)^k]$, for k even (τ_X^2 is also known as the *variance* of X). We now mention some well-known facts [11, 30].

Lemma 9 (kth moment inequality) *Let X be a random variable and let $\mu = E[X]$. Then, for all $t \geq 0$ and k even:*

$$\Pr[|X - \mu| > t\sqrt[k]{\tau_X^k}] \leq \frac{1}{t^k}.$$

Lemma 10 *Let $X = \sum_{i=1}^{n} X_i$ where the X_i are k-wise independent random variables, for even k. Then:*

$$\tau_X^k = \sum_{i=1}^{n} \tau_{X_i}^k.$$

Corollary 11 *Let $X = \sum_{i=1}^{n} X_i$ where each X_i is 0 with probability $q = 1 - p$ and 1 with probability p. Also, let the X_i's be k-wise independent, for even k. Then:*

$$\tau_X^k = npq(p^{k-1} + q^{k-1}).$$

PROOF. For each i, $E[X_i] = \mu = p$. Also,

$$\tau_{X_i}^k = E[(X_i - \mu)^k] = qp^k + pq^k = pq(p^{k-1} + q^{k-1}).$$

The corollary follows from lemma 10. □

3 Parallel Construction of a Fast Priority Queue

This section reviews a variant of the priority queue described by van Emde Boas *et al.* [35] (abbreviated as the vEB data structure), and describes methods of constructing this data structure in parallel.

3.1 A Fast Priority Queue

The van Emde Boas data structure stores a set $S \subseteq \{1, \ldots, m\}$, (the elements in S are all distinct), such that predecessor queries can be answered in $O(\log\log m)$ time [35]. The vEB data structure for $\{1, \ldots, m\}$, m a power of 2, is a complete binary tree with depth $\log m$, with all leaves at the same level. The edge leading to the left child of each internal node is labeled with a 0, and the one to the right node with a 1. Every leaf corresponds to the integer that is obtained by concatenating the bits obtained by traversing the path from the root to it (the edges incident upon the root are the MSB's). Every leaf node that corresponds to an integer in S is marked, and an internal node is marked iff one of its children is marked. Every internal node also contains the maximum and minimum element from S stored in the subtree rooted at it. The data structure is stored in an array A of size $O(m)$ in the usual way, with the children of a node stored at $A[i]$ being located at $A[2i]$ and $A[2i + 1]$. In addition, all elements in S are linked together in an ordered list.

This data structure can be used to find $Pred(x)$ and $Succ(x)$ in $O(\log\log m)$ time by doing a binary search on the path from x to the root to find the first marked node along the path. Either the maximum value in the subtree rooted at this node will be $Pred(x)$ or the minimum value in the subtree rooted at this node will be $Succ(x)$. As the elements of S are linked together in a linked list, finding either the predecessor or the successor will enable us to get the other.

Some vertices on a marked path from a leaf to a root are the *left* or *right join* vertices of that path. The left join vertices of a path p are all the vertices v such that p goes to the right child of v and the left child of v is marked (*i.e.*, p is joined by another path from the left at v). The right join vertices of a path are defined similarly. The set of *join* vertices of a path is the union of the sets of its right and left join vertices. Let p be the path from $x \in S$ to the root. The predecessor of x is the maximum element stored at the left child of the deepest left join vertex l along p. Also note that exactly all vertices from x to l will have x as the minimum value in the subtree rooted at them.

3.2 Perfect Hashing

In this section we will review the perfect hashing scheme proposed by Fredman *et al.* [14]. An vEB data structure for a set $S \subseteq \{1, \ldots, m\}$ stored in an array of size $O(m)$ may have many elements that are zero: only $O(|S| \log m)$ locations will contain non-zero values. By storing the useful portions of the array in a hash table, such that the array index is used as a key to retrieve the value stored there, the space requirement can be considerably reduced. Dietzfelbinger *et al.* have used this idea in a sequential context [10].

Suppose that $S \subseteq \{1, \ldots, N\}$, and $|S| = n$. Let p be the smallest prime such that $p > N$ and let $s \geq n$. Fredman *et al.* consider the class of hash functions defined by $\mathcal{H}_{p,s} = \{h^k_{p,s}(x) = (kx \bmod p) \bmod s | 1 \leq k \leq p\}$. They prove the following:

Lemma 12 *Let p and s be as above, and let W^k_i be the number of times the value i is achieved by $h^k_{p,s}$ when restricted to S, that is, $W^k_i = |\{y \in S | h^k_{p,s}(y) = i\}|$. Then*

1. *With probability at least a half, a function $h^k_{p,s}$ chosen uniformly at random from $\mathcal{H}_{p,s}$ satisfies $\sum_{i=1}^{s} (W^k_i)^2 < 5n$.*
2. *With probability at least a half, a function $h^k_{p,2s^2}$ chosen uniformly at random from $\mathcal{H}_{p,2s^2}$ is injective on S.*

These facts are then used to obtain a linear space static data structure for answering membership queries about S. A top-level hash function is chosen that satisfies condition 1 above, which partitions S into at most s buckets. For each bucket containing $m > 1$ elements, $2m^2$ locations are allocated, and a second level hash function is chosen from $\mathcal{H}_{p,2m^2}$ which satisfies condition 2. Thus the composition of the top level function with the appropriate second level hash function is an injective mapping, and membership queries can be answered in $O(1)$ time. Note that the second level tables require a total of $O(n)$ space, since the top-level function is chosen to satisfy condition 1 above.

3.3 Constructing the vEB Data Structure in Parallel

Theorem 13 *An vEB data structure on $S \subseteq \{1, \ldots, m\}$, $|S| = n$ can be constructed in:*

1. *$\tilde{O}(\log n)$ time using $n/\log n$* EREW PRAM *processors and $O(n)$ space, assuming S is given as a sorted array.*
2. *$\tilde{O}(\log n / \log\log n)$ time using $n \log\log n / \log n$* COLLISION *or* COMMON *processors and $O(n)$ space, assuming S is given as a sorted array.*
3. *$O(1)$ time using $n \log m$* COLLISION *processors, using $O(m)$ space, assuming only that the items are distinct and $\log m$ processors are associated with each.*

PROOF. The algorithms are as follows:

ALGORITHM 1: The main idea here is that instead of writing information directly into the array that represents the data structure, the processors will determine what to write by

looking at the input and write ⟨array index,value⟩ pairs in an auxiliary array instead. Then these values are stored in a hash table using the array index as key. A "bucketing" technique is used to obtain optimality and space linearity. Without loss of generality we assume that $m \geq n$ (since the elements of S are distinct). Also, we can assume that $\log\log m = o(\log n)$, since otherwise binary search will perform as well as the vEB data structure.

Step 1: Let the given input be an array X with elements $x_1, \ldots, x_n$. Let $l = \log m$, $r = l\log^2 n$ and $n' = \lfloor n/r \rfloor$. We will place only elements $Y = x_r, x_{2r}, \ldots, x_{n'r} = y_1, \ldots, y_{n'}$ in the vEB data structure. Finding the predecessor in X will consist of finding the predecessor in Y using the vEB data structure, followed by a binary search on a segment of X of length $O(r)$, which takes $O(\max\{\log\log m, \log\log n\})$ $= O(\log\log m)$ time in all.

Step 2: Let the binary representation of each $y_i = b_{i,l} \ldots b_{i,1}$. For each y_i, let $d_i = \max\{j | b_{i,j} \neq b_{i+1,j}\}$. Then if the path from y_i to the root is $v_0, \ldots, v_l$, then exactly the nodes $v_0, \ldots, v_{d_i}$ will have y_i as the maximum value in their subtree. Similarly, each processor can determine the portion of the path along which the value assigned to it is the minimum value stored. We associate l processors with each element in Y, and determine the d_i in $O(\log\log m) = O(\log n)$ time using the obvious algorithm.

Step 3: Let B be an array of size $n'l$. The l processors associated with y_i write the at most l pairs $\langle loc, x_i \rangle$ such that in the actual vEB structure, location loc would have contained x_i as the subtree maximum, into locations $B[il]$ through $B[(i+1)l - 1]$. This can be done in $O(1)$ time, since the values d_i are known from step 2. A similar procedure is followed for subtree minima. Let the set of pairs stored in B be E, and observe that $|E| = O(n/\log^2 n)$.

Step 4: Using the standard prefix sum algorithm (see, *e.g.*, [16]), the processors compact E into $|E|$ consecutive locations in memory, in time $O(\log n)$.

Step 5: Let p be the smallest prime such that $p > m$. The processors divide up into $d_1 \log n$ groups of $P = O(n/(\log n)^2)$ each. In parallel, the processors in each group choose a top-level hash function h independently at random from $\mathcal{H}_{p,2|E|}$.

Step 6: Each group then checks to see if its choice satisfies condition 1 of lemma 12, *i.e.*, that there are not too many collisions at the top level. This is done by first evaluating h at all the elements of E in $O(1)$ time. The values $h(E)$ are sorted to determine the values W_i, $1 \leq i \leq 2|E|$, and a prefix sum computation is used to determine whether h satisfies condition 1. If good function(s) are found, an arbitrary one is chosen in $O(\log\log n)$ time, and if not, the computation is aborted and restarted.

Step 7: Using the prefix sum computed in step 6, it is easy to allocate $O(W_i \log n)$ processors to each bucket i, $1 \leq i \leq 2|E|$, such that $W_i \geq 2$. We can also allocate $O((W_i)^2 \log n)$ space for each such i. Now for each such i, $d_2 \log n$ different "second level" hash functions chosen independently unformly at random from $\mathcal{H}_{p,2(W_i)^2}$ are tried in parallel, trying to find one which is injective (checking for injectiveness is

again accomplished by sorting). If an injective function is not found for all such i, the computation is aborted and restarted.

Analysis: Now we fix the constants d_1 and d_2. If at the first stage we try $d_1 \log n = \beta \log n$ different hash functions, then since each one independently has a probability $1/2$ of being good, with probability at least $1 - n^{-\beta}$ at least one good hash function will be found. At the second level, we try $d_2 \log n = (\beta + 1) \log n$ different functions for each bucket, so that an injective second level hash function will be found for all buckets with probability at least $1 - n \cdot n^{-(1+\beta)} = 1 - n^{-\beta}$.

ALGORITHM 2: For the COMMON model, we make the following modifications to algorithm 1. In step 2, we use the algorithm of lemma 5 to find the values d_i in $O(1)$ time. In step 4 and step 6 we use the fast prefix sum algorithm of lemma 1. For the COLLISION model, in addition to the above modifications, it is possible to simplify the algorithm a little: in step 6 we can use the optimal sub-logarithmic time algorithm of lemma 4 to count collisions and in step 7, checking to see if the hash function chosen is injective or not can be done trivially in $O(1)$ time. This enables us to reduce the value of r from $l \log^2 n$ to $l \log n$ in step 1.

ALGORITHM 3: Here we assume that the $\log m$ processors associated with each element are numbered $1, 2, \ldots, \log m$. Let the given input be $x_1, \ldots, x_n$, $x_i \in \{1, \ldots, m\}$, and let all the x_i's be distinct. Let $path_i$ be the path from x_i to the root.

Step 1: The $\log n$ processors associated with x_i mark all nodes along $path_i$. In constant time the processors determine for each node along $path_i$ whether it is a join vertex or not. Using the "leftmost one in memory" algorithm, in constant time the $\log m$ processors determine the deepest left join and right join vertices along $path_i$.

Step 2: In constant time the processors write the pair $\langle i, x_i \rangle$ as the subtree minimum (maximum) for all vertices below the the deepest left (right) join vertex. Finally, in constant time the processor associated with the deepest left (right) join vertex v looks at v's left (right) child and reads the pair $\langle k, x_k \rangle$ which is the subtree maximum (minimum) there and sets $Pred(x_i) = k$ ($Succ(x_i) = k$). □

Remark: Algorithm 3 uses ideas similar to the constant-time simulation of PRIORITY by COMMON given in [6].

4 Applications

4.1 Integer Sorting

We will now prove our results on integer sorting. Let $PT(A, n)$, $T(A, n)$ and $S(A, n)$ be, respectively, the processor-time product, time and space required by any algorithm A that sorts integers in the range $\{1, \ldots, n\}$ (not necessarily stably) and runs on on one of the CREW PRAM, COMMON and COLLISION models of computation. We show:

Theorem 14 *Let A be as above and let A use p processors. Then there is an algorithm A' that uses the same number of processors as A, runs on the same model of computation as A and stably sorts integers in the range $\{1,\ldots,m\}$, for any integer m, and has the following complexities:*

1. $PT(A',n) = \tilde{O}(PT(A,n) + n\log\log m)$,
2. $T(A',n) = \tilde{O}(PT(A',n)/p + \log\log m)$ *and*
3. $S(A',n) = S(A,n) + O(n)$. *($S(A',n) = S(A,n) + O(n^2)$ if A runs on* COMMON.*)*

PROOF. We assume the input is given in an array $X[1..n]$. The algorithm A' is as follows.

Step 1: For each i in parallel, compute $Y[i] = n(X[i]-1)+i$. The $Y[i]$'s are distinct and are in the range $\{1,\ldots,nm\}$. Let S be the set of values in the array Y.

Step 2: Choose $S' \subseteq S$ by placing $x \in S$ independently in S' with probability $\log^{-3} n$. If d_1 and d_2 are chosen appropriately, we can ensure that with high probability:

1. $|S'| \leq d_1 n/\log^3 n$ and
2. at most $d_2 \log^4 n$ elements of S have a value that lies in between two consecutive elements of S'.

Step 3: Using lemma 1 or the standard prefix sum algorithm, compact S' into an array.

Step 4: Sort S' using Cole's merge sort [8], lemma 2 or lemma 3 depending on whether A runs on the CREW PRAM, COLLISION or COMMON models.

Step 5: Construct a vEB data structure on S' using Algorithm 1 or 2, as appropriate, from theorem 13.

Step 6: For each element of S, the processors determine its predecessor in S', as well as the rank of its predecessor in S'.

Step 7: S' partitions S into $|S'|+1$ sub-collections in the following way: if the elements of S' are $z_1,\ldots,z_{|S'|}$ in sorted order, then the ith collection $C_i = \{x \in S | z_i \leq x < z_{i+1}\}$, $0 \leq i \leq |S'|$. Associate with each element x of S the index i such that $x \in C_i$, and sort S using these indices as keys, using algorithm A.

Step 8: Now sort the individual sub-collections C_i, using Cole's merge sort.

Analysis: First we show that conditions 1 and 2 in step 2 are both true with high probability. The size of the set S' is a binomial variable with parameters $(n, \log^{-3} n)$. By a routine application of lemma 7 (Chernoff bounds for binomial distributions), we see that for any constant $\varepsilon > 0$, if $d_1 \geq 1+\varepsilon$, the probability that $|S'| > d_1 n/\log^3 n$ is no more than $e^{-\varepsilon^2 n/3\log^3 n}$, which is sufficient for our purposes. Now we fix the value of d_2. Consider S in sorted order. The probability that ℓ consecutive elements of S are not in S' is $(1-\log^{-3} n)^{\ell}$,

and hence $\Pr[|C_i| > \ell] \leq (1 - \log^{-3} n)^{\ell}$. Let $k = (\beta + 1)\log^3 n \ln n$. Using the standard inequality $(1 - 1/x)^x < 1/e$ we find that $\Pr[|C_i| > k] < n^{-(\beta+1)}$. Thus choosing $d_2 = (\beta + 1)\ln 2$ we can ensure that no collection is larger than $d_2 \log^4 n$ with the required probability.

Now we analyze the time required for the individual steps. We will do this by indicating the work and time requirements of each step, and appealing to Brent's theorem whenever necessary.

Step 1: $O(n)$ work, $O(1)$ time.

Step 2: $O(n)$ work, $O(1)$ time.

Step 3: $O(n)$ work, $O(\log n/\log\log n)$ time on COMMON and COLLISION, and $O(\log n)$ time on the CREW PRAM.

Step 4: $\tilde{O}(n)$ work, $\tilde{O}(\log n/\log\log n)$ time on COMMON and COLLISION, and $\tilde{O}(\log n)$ time on the CREW PRAM.

Step 5: $O(n)$ work, $\tilde{O}(\log n/\log\log n)$ time on COMMON and COLLISION, and $\tilde{O}(\log n)$ time on the CREW PRAM.

Step 6: $O(n\log\log m)$ work, $O(\log\log m)$ time.

Step 7: $PT(A, n)$ work, $T(A, n)$ time.

Step 8: Let $s = \max_i\{|C_i|\}$. The work done in this phase is at most $O(\sum_i |C_i|\log s) = O(n\log s)$ and the time requirement is $O(\log s)$. Since $s = O(\log^4 n)$ w.h.p., the work done in this phase is $\tilde{O}(n\log\log n)$ and the time for this phase is $\tilde{O}(\log\log n)$.

Noting that $T(A, n) = \Omega(\log n)$ for the CREW PRAM and $T(A, n) = \Omega(\log n/\log\log n)$ for COLLISION and COMMON, we obtain the theorem. □

Remark: The above algorithm uses ideas that are reminiscent of ones in a randomized simulation of PRIORITY on a COLLISION-like model called COLLISION$^+$ given in [7].

As a consequence of the above theorem and lemma 4, we get:

Theorem 15 *A set of n integers, each in the range $\{1, \ldots, m\}$, can be stably sorted in $\tilde{O}(\log n/\log\log n + \log\log m)$ time with $O(n\log\log m)$ work, using $O(n)$ space, on the* COLLISION *model of computation.*

Another advantage of our algorithm is that even when $\log n/\log\log n = o(\log\log m)$, our algorithm can be sped up to run in $\tilde{O}(\log n/\log\log n)$ time at the expense of extra work. More precisely:

Theorem 16 *Let $k = \lceil \log\log m \log\log n/\log n \rceil$ be greater than 1. Then, n integers, each in the range $\{1, \ldots, m\}$, can be stably sorted in $\tilde{O}(\log n/\log\log n)$ time using $O(2^k n)$ processors, using $O(n)$ space, on the* COLLISION *model of computation.*

PROOF. Firstly, note that when $k > 1$, the algorithm of theorem 15 uses n processors. Suppose we are given $2^k n$ processors. Then, by using 2^k-ary search along the path to the root instead of binary search to find predecessors using the vEB data structure, we can perform step 6 of theorem 14 in $O(\log_{2^k}(\log m))$ time, which is $O(\log n / \log\log n)$ time. Thus this algorithm is non-optimal by a factor of $O(2^k/k)$. Also, when $2^k = \Omega(\log^\varepsilon n)$, for any pre-specified ε, it is more economical to use the general sorting algorithm of lemma 2. □

Finally, we obtain, using Rajasekaran and Sen's integer sorting algorithms [32] as a subroutine, instead of lemma 4 (for the COMMON result we use Cole's merge sort instead of lemma 3 to keep the space linear):

Theorem 17 *A set of n integers, each in the range $\{1,\ldots,m\}$, can be stably sorted with $\tilde{O}\left(n \max\left\{\frac{\log n}{\log\log n}, \log\log m\right\}\right)$ work and $\tilde{O}(\log n \log^{(3)} n)$ time on* COMMON, *or with the same work in $\tilde{O}(\log n \log\log n)$ time on the* CREW PRAM. *Both these algorithms use linear space.*

4.2 Additional Remarks on Integer Sorting

Integer Sorting on the Augmented RAM

We first note that our integer sorting algorithm uses a word size of $O(\log m + \log n)$ and is thus *conservative*, *i.e.*, it does not abuse the unit-cost criterion by generating large operands. Our algorithm uses the basic RAM set of operations everywhere except that for performing hashing in theorem 13, arbitrary multiplication and mod operations are used. For algorithm 1 in theorem 13 (for the CREW PRAM), these operations are completely unnecessary. We can associate $O(\log m)$ processors with each key to be hashed, and this will enable us to do both the above operations in $O(\log\log m) = O(\log n)$ time using basic RAM instructions. For algorithm 2 in theorem 13, however, the above approach increases the run time by an additive $O(\log\log m)$ factor. Note that in step 1 (which has an apparent multiplication) we could instead use $Y[i] = (X[i]-1)2^{\lceil \log n \rceil} + i$, which would serve just as well, and needs only a shift to compute. This makes no end difference to theorem 15, but for theorem 16 we need the use of the augmented RAM if hashing is to be employed. By eliminating hashing, we obtain the same bounds on the basic RAM, but at an increase in the space utilization to $O(nm^\varepsilon)$ for any pre-specified constant ε. Note that there is no need to initialize the memory, since the well-known trick used to avoid initializing memory in the sequential setting can also be used here.

With the augmented RAM instruction set, however, we can actually improve upon theorem 16, by constructing the *fusion tree* data structure of Fredman and Willard [15] rather than a vEB data structure in step 5 of theorem 14. The fusion tree is a data structure for a RAM that stores a set $S \subseteq \{1,\ldots,m\}$, $|S| = n$, such that predecessor queries can be answered in $O(\min\{\sqrt{\log n}, \log\log m\})$ time. Their data structure uses $O(n)$ space and assumes an augmented RAM with word size $O(\log m)$. We note the following, which only uses the fact that the sequential construction of the (static) fusion tree is easily parallelizable, if S is given as a sorted array:

Fact 18 *If S is given as a sorted array, the fusion tree data structure can be constructed optimally in parallel in $O(\log n/\log\log n)$ time on either* COMMON *or* COLLISION *and in $O(\log n)$ time on a* CREW PRAM, *with the augmented instruction set.*

Thus, by using the above fact in place of steps 5 and 6 of theorem 14, we obtain, using either lemma 4 or Rajasekaran and Sen's integer sorting algorithms [32] as a subroutine:

Theorem 19 *A set of n integers, each in the range $\{1,\ldots,m\}$, can be stably sorted with $\tilde{O}(n\sqrt{\log n})$ work in $\tilde{O}(\log n/\log\log n)$ time on* COLLISION, *$\tilde{O}(n\log n/\log\log n)$ work and $\tilde{O}(\log n\log^{(3)} n)$ time on* COMMON, *or with the same work in $\tilde{O}(\log n\log\log n)$ time on the* CREW PRAM. *All the above algorithms use the augmented RAM set of instructions and use linear space*

Reducing the Amount of Randomness Used

In this section we note that we can analyse the running time of our integer sorting algorithm using only $\log^{O(1)} n$-wise independent random variables. First we consider step 2 of theorem 14. Let $k = d_2\log^4 n$, for some constant d_1. For each $x \in S$, we select a random number in the range $\{1,\ldots,\log^3 n\}$, uniformly from a k-wise independent distribution, and place $x \in S'$ if the number chosen is 1. We would like to show that with high probability:

1. $|S'| \leq d_1 n/\log^3 n$ and
2. at most k elements of S have a value that lies in between two consecutive elements of S'.

The second part follows as before. We consider S in sorted order. The probability that ℓ consecutive elements of S are not in S' is $(1-\log^{-3} n)^{\ell}$, for any $\ell \leq d_2\log^4 n$. Therefore, for all i, $\Pr[|C_i| > k] < e^{-d_2\log n}$, which is less than $n^{-(\beta+1)}$ if d_2 is large enough. Thus, with probability at least $1-n^{-\beta}$, all the C_i's are small. Now we show that S' is not large either. From corollary 11, we find that

$$\sqrt[k]{\tau^k_{|S'|}} = \left[\frac{n}{\log^3 n}pq(p^{k-1}+q^{k-1})\right]^{1/k}$$

where $p = \log^{-3} n$ and $q = 1-p$. Since $p \ll q < 1$, we find that $\sqrt[k]{\tau^k_{|S'|}}$ is $1 \pm o(1)$, depending upon the value of k above. We thus get that, for any fixed $\varepsilon, \beta > 0$, the following is true for sufficiently large n:

$$\Pr[|S'| > \frac{(2+\varepsilon)n}{\log^3 n}] < \left(\frac{\log^3 n}{n}\right)^k < n^{-\beta}.$$

For the perfect hashing phase in the construction of the EKZ data structure, we can again clearly make do with at most $O(\log n)$-wise independence. It therefore follows that we can execute the algorithm using at most $O(\log^5 n)$ truly random bits and still obtain the same performance.

4.3 The Chaining Problem

Now we turn to the chaining problem. Recall that the parameters here are n, the size of the array and m the number of non-zero elements in the array. Since constructing an EKZ data structure on a set S also permits us to chain the elements of S together, we obtain as an immediate corollary of theorem 13:

Corollary 20 *Ordered chaining can be done in $O(1)$ time using $m \log n$ processors on the* COLLISION *model of computation, provided there are $\log n$ processors numbered 1 through $\log n$ at each non-zero location.*

Though the number of processors used is $O(n)$ whenever $m = O(n/\log n)$, this algorithm cannot be said to solve the chaining problem, since it makes the strong assumption that each non-zero element has $\log n$ associated auxiliary processors that are consecutively numbered as well. However, using this algorithm as a subroutine, we are able to get processor- and time-efficient algorithms for the chaining problem. To do so we have to solve the processor allocation problem. Let the set of indices associated with non-zero elements be NZ ($|NZ| = m$).

Let $\log^{(i)} n$ be the ith iterate of the log function (*i.e.*, $\log^{(0)} n = n$, and for $i > 0$, $\log^{(i)} n = \log(\log^{(i-1)} n)$). For the problem of ordered chaining we show the following:

Theorem 21 *Ordered chaining can be done in $\tilde{O}(1)$ time using n* COLLISION *processors provided $m < n/(\log^{(t)} n)^6$, for any fixed $t > 0$.*

PROOF. The proof is inductive. For the base case $t = 1$ we will solve the allocation problem by grouping the given processors into $G = n/\log n$ groups of size $\log n$, and then creating a mapping $A : NZ \mapsto \{1, \ldots, G\}$, such that for each $g \in G$, at most two distinct $i \in NZ$ will have $A(i) = g$. The mapping will be constructed in two phases: the first will be randomized and the next, deterministic. In the first step, all non-zero elements will be assigned to a group at random, and those that get hold of a unique group will allocate that group to themselves. A substantial number of non-zero elements will remain unallocated, but the remainder will be "uniformly scattered" in the input array. In particular, it will be the case that with very high probability, every sufficiently large chunk of the input array will contain only a "polynomially sparse" number of unsatisfied non-zero elements. The compaction algorithm of lemma 6 can now be used to locally solve the allocation problem.

Step 1: Each processor $p \in NZ$ chooses a random integer $g(p)$ from $\{1, \ldots, n/\log n\}$. Let U be the set of processors with unique integers $g(p)$,*i.e.*, $U = \{p \in NZ | (\forall p' \in NZ \setminus \{p\})\ g(p) \neq g(p')\}$. Every $p \in U$ sets $A(p) = g(p)$. Let $NZ_1 = NZ \setminus U$.

Step 2: Now divide the input array into contiguous chunks of size $s = \log^6 n$, with the ith chunk $C(i)$ consisting of the indices $si + 1, \ldots, s(i + 1)$, for $1 \leq i \leq n/s$. Let $NZ_1(i) = NZ_1 \cap C(i)$. For each i, s processors attempt to compress $NZ_1(i)$ into locations $si + 1, \ldots, si + s/\log n$ using the algorithm of lemma 6. This will work iff $C(i)$ is *sparse*, *i.e.*, $(|NZ_1(i)|)^4 < s/\log n$.

Thus in each chunk $C(j)$ that is sparse, every $p \in NZ_1(j)$ is assigned a unique integer $g_1(p)$ in the range $\{1, \ldots, \ldots, s/\log n\}$, and sets $A(p) = sj/\log n + g_1(p)$. We will show that with high probability all the chunks $C(j)$ are sparse.

Step 3: For each $p \in NZ$, the $\log n$ processors $A(p)$ are used to solve the ordered chaining problem using corollary 20.

Analysis: Since each index in NZ is in NZ_1 with probability at most $\log^{-5} n$ independently of all other indices, lemma 8 implies that $|NZ_1(i)|$ is upperbounded by a random variable that is binomially distributed with parameters $(s, \log^{-5} n)$. Lemma 7 now states that $|NZ_1(i)| > d_1 \log n$ with probability at most $2^{-d_2 \log n} = n^{-d_2}$ for any prespecified number d_2, if d_1 is large enough. This implies that all chunks will be sparse with the required probability, if d_2 is large enough.

For the inductive step we assume that the proposition is true for $t = 1, \ldots, r$. During steps 1 and 2, an attempt is made to allocate $\Omega(\log^{(r+1)} n)$ processors with each index in NZ, in a manner similar to steps 1 and 2 in the base case. Then the input array is divided into segments, and the ordered chaining problem is solved locally within these segments. Then representatives are chosen from within each segment. With high probability, the number of representatives will be small enough so that a recursive call can be made to the algorithm of the inductive hypothesis. We divide the n processors into $G = n/\log^{(r+1)} n$ groups of $\log^{(r+1)} n$ each.

Step 1: Each processor $p \in NZ$ chooses a random integer $g(p)$ from $\{1, \ldots, n/\log^{(r+1)} n\}$. Let U be the set of processors with unique integers $g(p)$: each $p \in U$ sets $A(p) = g(p)$. Let $NZ_1 = NZ \setminus U$.

Step 2: The input array is divided into contiguous chunks $C(i)$ of size $s = (\log^{(r+1)} n)^6$ and we let $NZ_1(i) = NZ_1 \cap C(i)$ as before. Using s processors, for each i, an attempt is made to compress $NZ_1(i)$ into locations $si + 1, \ldots, si + s/\log^{(r+1)} n$ by running the algorithm of lemma 6. This will work iff $C(j)$ is *sparse*, *i.e.*, iff $|NZ_1(j)|^4 < s/\log^{(r+1)} n$. Let $S = \{j | C(j) \text{ is sparse}\}$. Thus for every $j \in S$, every $p \in NZ_1(j)$ is assigned a unique integer $g_1(p)$ in the range $\{1, \ldots, s/\log^{(r+1)} n\}$, and sets $A(p) = sj/\log^{(r+1)} n + g_1(p)$. Let $NZ_2 = NZ_1 \setminus \cup_{j \in S} NZ_1(j)$.

Step 3: Now we divide the input array into contiguous segments $S(i)$, $1 \leq i \leq n/(\log^{(r)} n)^7$ of size $(\log^{(r)} n)^7$. Call a segment $S(i)$ *allocated* if $S(i) \cap NZ_2 = \emptyset$, and *non-allocated* otherwise. For any $S(i)$, let $T(i) = S(i) \cap NZ$ be the set of non-zero elements within $S(i)$. Let A be the set of indices that represent allocated segments and NA the set of indices representing non-allocated segments. By the definition of A, we know that for each $i \in A$, each $p \in T(i)$ has $\Omega(\log^{(r+1)} n)$ processors allocated to it, and so we can use the algorithm of corollary 20 to link each element in $T(i)$ to its successor in $T(i)$. After this is done, the first and last elements from $T(i)$, $i \in A$, are chosen to be *representatives* for that segment. Let the set of all representatives be R.

Step 4: Note that $|R| \leq 2n/(\log^{(r)} n)^7$. We will show that $|NZ_2| = o(n/(\log^{(r)} n)^{13})$ with very high probability. Since $|NA| \leq |NZ_2|$, this implies that $|\cup_{i \in NA} T(i)|$

is $o(n/(\log^r n)^7)$ and hence a recursive call suffices to chain the elements of $R \cup (\cup_{i \in NA} T(i))$ together in order. A little care must be taken in the recursive call not to reset successors for non-zero elements whose successors are already known from a previous higher-level invocation, and corollary 20 can easily be modified to handle this. (Otherwise, the representatives may end up not pointing to their successors after this recursive call.)

Analysis: Consider each chunk $C(i)$, which has at most $s = (\log^{(r+1)} n)^6$ non-zero elements in it initially. The probability that each $p \in NZ$ is also in NZ_1 is at most $(\log^{(r+1)} n)^{-5}$, and so the probability that $|NZ_1(i)| > d_1 \log^{(r+1)} n$ is at most $2^{-d_2 \log^{(r+1)} n} = (\log^{(r)} n)^{-d_2}$, for any d_2, if d_1 is large enough. Thus, for any chunk i, $\Pr[i \notin S] \leq (\log^{(r)} n)^{-d_2}$, which implies $|\{i | i \notin S\}| < n/(\log^{(r)} n)^{14}$ with the required probability, if d_2 is large enough, and thus $|NZ_2|$ is $o(n/(\log^{(r)} n)^{13})$ with high probability. □

Corollary 22 *Unordered chaining can be done in $\tilde{O}(1)$ time using n* COLLISION *processors, provided $m < n/\log^{(i)} n$, for any fixed i.*

We would like to point out here that the technique above can be used to solve a generalized version of the chaining problem. Define the *c-color* chaining problem to be the following: suppose we are given an array of values $x_1, \ldots, x_n$, such that some array elements have a value in the range $\{1, \ldots, c\}$, and the rest are uncolored. Let S_k, $k \in \{1, \ldots, c\}$, be the set of array indices i such that $x_i = k$. The c-color chaining problem is then to output c linked lists $l_1, \ldots, l_c$ such that l_i contains all the elements in S_i exactly once. This generalization has a natural interpretation in the context of database operations: it enables us find all the records that match a particular predicate and then subclassify them based on a secondary field as well, and is at the heart of the stable integer sorting problem. The algorithm for the "base" case of theorem 21 above can clearly be used to solve the c-color chaining problem as well, and thus we can solve the c-color chaining problem in $O(1)$ time whenever $\sum_{i=1}^{c} |S_i| < n/(\log n)^6$. (The induction step cannot be made to work because at the end of step 3, we will have to choose representatives from each segment *for each color*, and thus the set of representatives may not be small enough for the recursive call to work.) However, by a simple modification, we can improve this slightly:

Theorem 23 *The c-color chaining problem can be solved in $\tilde{O}(1)$ time with using n* COLLISION *processors whenever $\sum_{i=1}^{c} |S_i| < n/(\log n)^{1+\varepsilon}$ for any constant $\varepsilon > 0$.*

PROOF. As before, we combine processors into groups of size $\log n$ each. If each $p \in NZ$ attempts to grab one of these groups at random, then the probability of failure will be at most $(\log n)^{-\varepsilon}$. This means that the number of non-allocated elements will be attenuated by a factor of $O((\log n)^{-\varepsilon})$, and it is easy to see that after $O(1)$ such stages, the number of unallocated non-zero elements will be less than $n/(\log n)^6$ with very high probability, and the processor allocation algorithm of theorem 21 can be applied. For non-constant ε the run time is approximately $O(\log(1/\varepsilon))$.□

We note that the above processor allocation problem can be restated in a more general manner. Suppose we have an array of size n, and there are m "active" locations in the array,

with one processor associated with each such location (this is referred to as the *allocated* PRAM in the literature). Furthermore, there is one processor associated with each (active or inactive) location in the array. Now, suppose that we want to allocate to each active processor k other processors to help it in some computation. More precisely suppose each active processor has a distinguished set of k consecutive memory locations. We would like, at the end of the allocation algorithm, that each active processor's locations contain k processor ids, such that no processor id appears in more than $O(1)$ such locations. Clearly, we need $n = \Omega(mk)$, and we can always arrange it so that the *advantage* $a = n/mk$ is at least 4, just by creating multiple copies of the available processors. Let $\phi(n,k) = \min\{i | \log^{(i)} n \leq k\}$. We can prove the following easily from the above discussion:

Lemma 24 *The above problem can be solved in* $O(\log(\log k/\log a)+\phi(n,k))$ *time on* COLLISION.

Remarks: Theorem 21 implies that ordered chaining can be done with high probability in $O(\log^* n)$ time using $O(n)$ processors for all values of m, which is inferior to Ragde's result, and thus it is still an open question whether the chaining problem can be solved in constant time for all values of m. Also, since our algorithms are randomized, they cannot easily determine when an unexpectedly high number of non-zero elements appear in the input, since an aborted computation could also be the result of an unfavorable sequence of coin tosses.

Also, the probability that the chaining algorithm terminates within the stated time bounds can be improved to $1 - 2^{-n^\beta}$ for any constant $\beta < 1/4 - \varepsilon$. This can be done as follows: all except the "base case" algorithm actually have success probability of the required order of magnitude. To improve the base case, after step 2, we observe that at most n^β elements will remain with probability $1 - 2^{-n^\beta}$, and we allocate processors to these by compressing the remaining non-zero elements into the first $n/\log n$ locations of the array. The significance of being able increase the probability this way is that the multiplicative constant within the $\widetilde{O}$ need not depend on β.

5 Conclusions and Open Problems

To conclude, we state our main results once again. We have described a randomized algorithm that sorts n integers in the range $\{1,\ldots,m\}$, $m = n^{O(1)}$, that runs in time $O(\log n/\log\log n)$ time with high probability, using $O(n)$ space and $O(n\log\log n)$ work. The model of computation used is the CRCW (COLLISION) PRAM. This algorithm runs as fast as (or faster than) the best currently known algorithms [5, 20, 27] and does no more work, while using only linear space and running on a weaker model of parallel computation. The same holds for intermediate values of m ($\log\log m = (1+\Omega(n))$ and $\log\log m = O(\sqrt{\log n})$, and for still larger values, a modification of our algorithm is the only optimal one until now. The approach also gives improved algorithms for (large) integer sorting on the CREW and COMMON models. Another virtue of our algorithm is that it is somewhat simpler than existing ones. Finally, we note that the model of computation may be weakened to the TOLERANT CRCW PRAM [19], which has an even weaker collision-detection mechanism, whereby concurrent writes to a location do not change the value of that location. (This was also suggested to us by Torben Hagerup.)

This paper does not resolve the main open problem of sorting n integers in a range polynomial in n optimally in $poly(\log n)$, though it achieves optimal speedup for a larger range of m than existing algorithms. It would be interesting to see if a deterministic algorithm with the same time complexity can be constructed for the COLLISION model, and some hope may be found from the remarks at the end of section 4.1, which suggest that the amount of randomness needed by the algorithms is small. It appears difficult to weaken the model of computation any further, and in particular to achieve $O(n \log\log n)$ processor-time product and $O(\log n)$ running time on the COMMON model of computation.

In the case of the chaining problem, we have addressed the open question that Ragde [29] posed, namely whether or not chaining can be done in constant time with a linear number of processors. Ragde gave a partial answer to this question: our paper gives a much improved, but still partial, answer to it. In addition, our algorithms are easily modified to solve a generalized version of the chaining problem. A complete solution to Ragde's open problem still has to be found.

6 Acknowlegdements

We would like to thank Paul Dietz for the explanation of the version of the vEB data structure used in this paper, Sanjay Jain for his proof of lemma 8 and Torben Hagerup for his comments on a draft of this paper. We would also like to thank Paul Dietz, Sanjay Jain, Danny Krizanc, Lata Narayanan and Joel Seiferas for helpful discussions.

References

[1] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[2] D. Angluin and L. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, 1979.

[3] P. Beame and J. T. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, 1989.

[4] O. Berkman and U. Vishkin. Recursive *-tree parallel data structure. In *Proc. 30th IEEE FOCS*, 1989.

[5] P. Bhatt, K. Diks, T. Hagerup, V. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. Technical Report TR 15/89, Fachbereich Informatik, Universitat des Saarlandes, November 1989. Accepted for publication in *Information and Computation.*

[6] B. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between CRCW PRAM models. In *Proceedings, 13th Symposium on Mathematical Foundations of Computer Science*, pages 231–239. Lecture Notes in Computer Science Series No. 324, Springer-Verlag, Berlin, 1988.

[7] B. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. In *Proceedings, 7th Inernational Conference on Fundamentals of Computation Theory*, pages 95–104. Lecture Notes in Computer Science Series No. 380, Springer-Verlag, Berlin, 1989.

[8] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17:770–785, 1988.

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

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnhert, and R. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Proc. 29th IEEE FOCS*, pages 524–531, 1988. The idea of compressing the van Emde Boas data structure (with applications to partially persistent arrays) was mentioned in the talk.

[11] W. Feller. *An Introduction to Probability Theory and its Applications, Vol I.* John Wiley, 1965.

[12] F. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. In *Proc. 3rd. Annual ACM PODC*, pages 179–189, 1984.

[13] F. E. Fich, P. Ragde, and A. Wigderson. Relations among concurrent write models of parallel computation. *SIAM Journal of Computing*, 17(3):606–627, June 1988.

[14] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[15] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier using FUSION trees. In *Proc. 22nd ACM STOC*, pages 1–7, 1990.

[16] A. M. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[17] J. Gil and Y. Matias. Fast hashing on a PRAM — designing by expectation. In *Proceedings, 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), San Francisco*, pages 71–280, January 1991.

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

[19] V. Grolmusz and P. Ragde. Incomparability in parallel computation. In *Proc. 28th IEEE FOCS*, 1987.

[20] T. Hagerup. Towards optimal parallel integer sorting. *Information and Computation*, 75:39–51, 1987.

[21] T. Hagerup. Constant-time parallel integer sorting (extended abstract). In *Proc. 23rd ACM STOC*, 1991. To appear.

[22] T. Hagerup and M. Nowak. Parallel retreival of scattered information. In *Proc. 16th Annual ICALP*, pages 439–450. Lecture Notes in Computer Science Series No. 372, Springer-Verlag, Berlin, 1989.

[23] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 117–124, 1990.

[24] D. Kirkpatrick and S. Reisch. Upper bounds for sorting intgers on a random access machine. *Theoretical Computer Science*, 28:263–276, 1984.

[25] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Massachussetts, USA, 1973.

[26] C. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. In *Proc. ICPP*, pages 869–876, 1986.

[27] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. Technical Report CS-TR-2935, University of Maryland, January 1990. An extended abstract appeared in *Proc. 17th ICALP*, Springer LNCS 443, July 1990.

[28] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time — with applications to parallel hashing. In *Proc. 23rd ACM STOC*, 1991. To appear.

[29] P. Ragde. The parallel simplicity of compaction and chaining. In *Proceedings 17th Annual ICALP*. Lecture Notes in Computer Science Series No. 443, Springer-Verlag, Berlin, July 1990.

[30] P. Raghavan. Lecture notes in randomized algorithms. Technical Report RC 15340, IBM, December 1989.

[31] S. Rajasekaran and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal of Computing*, 18(3):594–607, 1989.

[32] S. Rajasekaran and S. Sen. On parallel integer sorting. Technical Report TR-CS-DUKE-1987, Duke University, 1987.

[33] R. Raman. Optimal sub-logarithmic time integer sorting on a CRCW PRAM (note). Submitted to *Information Processing Letters*, January 1991.

[34] L. Rudolph and W. Steiger. Subset selection in parallel. In *Proc. ICPP*, pages 11–14, 1985.

[35] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

A General Sort on COLLISION

We now prove lemma 2. All the steps in the original algorithm can be trivially modified to work on COLLISION except the algorithm for the following "estimation" problem:

Lemma 25 *Let $S = \{1, \ldots, n\}$ be a set of indices and let each element of S belong to exactly one of $\sqrt{n}$ groups $G_1, \ldots, G_{\sqrt{n}}$, such that $\max_i\{|G_i|\} \leq \sqrt{n}\log n$, and such that in constant time, for each index in S we can determine the group it belongs to. Then we can compute numbers $N_1, \ldots, N_{\sqrt{n}}$ such that $\sum_{i=1}^{\sqrt{n}} N_i = O(n)$, and such that with high probability, $N_i \geq |G_i|$ for each $i \in \{1, \ldots, \sqrt{n}\}$, in $\tilde{O}(\log n/\log\log n)$ time using $n(\log n)^{\varepsilon}$* COLLISION *processors, for any constant ε.*

PROOF. The algorithm is a modification of the original one. We use a common array B of size $n/\log n$, partitioned into $\sqrt{n}$ contiguous groups $B_1, \ldots, B_{\sqrt{n}}$ of size $\sqrt{n}/\log n$ each. For each location $l \in B$, we also have an array A_l of size $(\log n)^{1+\varepsilon}$.

Step 1: $n/\log^2 n$ processors in parallel each choose a random index from $\{1, \ldots, n\}$.

Step 2: For each index i so chosen, let $g(i)$ be such that $i \in G_{g(i)}$. The processors compute, for each i, L_i which is the number of elements from G_i which are chosen in step 1. In [31] it is shown that setting $N_i = d_2 \log^2 n \max(1, L_i)$ for some sufficiently large constant d_2 suffices. This is done in the following manner: each of the $n/\log^2 n$ processors with index i chooses a random location within $B_{g(i)}$. Let S_j be the set of processors that chooses location j in the array B. Clearly, $L_j = \sum_{k \in B_j} |S_k|$. In [31] the $|S_j|$s are computed by showing that, with high probability, $\max_i\{|S_j|\} = O(\log n/\log\log n)$, and then by sequentially computing, using the properties of ARBITRARY, the exact value of $|S_j|$. An alternative method is for each of the processors in S_j allocate itself a unique location the array A_j, and then using $(\log n)^{1+\varepsilon}$ processors, one for each location in A_j, to compute the numbers $|S_j|$ in $O(\log\log n)$ further time. The allocation of indices can be done in the following way: each processor in S_j chooses a location at random in A_j. Processors that succeed in choosing a location not chosen by any other, allocate that location to themselves. Similarly, for s more stages, each 'unallocated' processor attempts to randomly choose a location that has neither been previously allocated nor has been chosen by some other processor in the same stage: if it succeeds, it allocates that location to itself and does nothing for the remainder of the stages. The probability of failure of any processor in any stage is at most $(\log n)^{-\varepsilon}$, which implies that if $s = d_1 \log n/\log\log n$, for some sufficiently large constant d_1, then with sufficiently high probability all the elements of S_j will have succeeded in finding assignments.

Step 3: A prefix sum computation on the values $|S_j|$ is then performed to compute the quantities L_i, and N_i is set to $d_2 \log^2 n \max(1, L_i)$.

The analysis is as in [31] and will not be repeated here. □