Sequential Random Permutation, List Contraction and Tree Contraction are Highly Parallel

Julian Shun* Yan Gu† Guy E. Blelloch‡ Jeremy T. Fineman§ Phillip B. Gibbons¶

Abstract

We show that simple sequential randomized iterative algorithms for random permutation, list contraction, and tree contraction are highly parallel. In particular, if iterations of the algorithms are run as soon as all of their dependencies have been resolved, the resulting computations have logarithmic depth (parallel time) with high probability. Our proofs make an interesting connection between the dependence structure of two of the problems and random binary trees. Building upon this analysis, we describe linear-work, polylogarithmic-depth algorithms for the three problems. Although asymptotically no better than the many prior parallel algorithms for the given problems, their advantages include very simple and fast implementations, and returning the same result as the sequential algorithm. Experiments on a 40-core machine show reasonably good performance relative to the sequential algorithms.

1 Introduction

Over the past several decades there has been significant research on deriving new parallel algorithms for a variety of problems, with the goal of designing highly parallel (polylogarithmic depth), work-efficient (linear in the sequential running time) algorithms. For some problems, however, one might ask if perhaps a standard sequential algorithm is already highly parallel if we simply execute sub-computations opportunistically when they no longer depend on any other uncompleted sub-computations. This approach is particularly applicable in iterative or greedy algorithms that iterate (loop) once through a sequence of *steps* (or elements), each step depending on the results or effects of only a subset of previous steps. In such algorithms, instead of waiting for its turn in the sequential order, a given step can run immediately once all previous steps it depends on have been completed. The approach allows for steps to run in parallel while performing the same computations on each step as the sequential algorithm, and consequently returning the same result. Surprisingly, this question has rarely been studied.

As an example, which we will cover in this paper, consider the well-known algorithm for randomly permuting a sequence of n values [13, 26]. The algorithm iterates through the sequence from the end to the beginning (or the other way) and for each location i, it swaps the value at i with the value at a random target location j at or before i. In the algorithm, each step can depend on previous steps since on step i the value at i and/or its target j might have already been swapped by a previous step. The question is: What does this dependence structure look like? Also, can the above approach be used to derive a highly parallel, work-efficient parallelization of the sequential algorithm?

In this paper, we study these questions for three fundamental problems: random permutation, list contraction and tree contraction [25, 24, 36]. For all three problems, we analyze the dependencies of simple randomized sequential algorithms, and show that the algorithms are efficient in parallel, if we simply allow each step to run as soon as it no longer depends on previous steps. To this end, we define the notion of an *iteration dependence graph* that captures the dependencies among the steps, and then we analyze the depth of these graphs, which we refer to as the *iteration depth*. We also study how to use low-depth iteration dependence graphs to design efficient implementations. This involves being able to efficiently recognize when a step no longer depends on any uncompleted previous step.

Beyond the intellectual curiosity of whether sequential algorithms are inherently parallel, the approach has several important benefits for the design of parallel algorithms. Firstly, it can lead to very simple parallel algorithms. In particular, if there is an easy way to check for dependencies, then the parallel algorithm will be very similar to the sequential one. We use a framework based on deterministic reservations [6] that makes programming such algorithms particularly easy. Secondly, the approach can lead to very efficient parallel algorithms. We show that if a sufficiently small prefix of the uncompleted iterations are processed at a time, then most steps do not depend on each other and can run immediately. This reduces the overhead for repeated checks and leads to work which is hardly any greater than the sequential algorithm. Finally, the parallelization of the sequential algorithm will be deterministic returning the same result on each execution (assuming the same source

*Carnegie Mellon University. jshun@cs.cmu.edu
†Carnegie Mellon University. yan.gu@cs.cmu.edu
‡Carnegie Mellon University. guyb@cs.cmu.edu
§Georgetown University. jfineman@cs.georgetown.edu
¶Intel Labs and Carnegie Mellon University. gibbons@cs.cmu.edu

of random numbers). The result of the algorithm will therefore be independent of how many threads are used, how the scheduler works, or any other non-determinism in the underlying hardware and software, which can make debugging and reasoning about parallel programs much easier [8, 6].

For the random permutation problem we consider the algorithm described above. We show that the algorithm leads to iteration dependence graphs that follow the same distribution over the random choices as do random binary search trees. Hence the iteration depth is $\Theta(\log n)$ with high probability.[1] For this algorithm we also show that recognizing when a step no longer depends on previous steps is easy, leading to a straightforward linear-work polylogarithmic-depth implementation. Therefore the "sequential" algorithm is effectively parallel.

The list contraction problem is to contract a set of linked lists each into a single node (possibly combining values), and has many applications including list ranking and Euler tours [24]. The sequential algorithm that we consider simply iterates over the nodes in random order splicing each one out.[2] We show that for this algorithm each list has an iteration dependence graph that follows the same distribution as random binary search trees. The iteration depth is therefore $O(\log n)$ w.h.p. For this algorithm, determining when a step no longer depends on previous steps is trivial—it is simply when the node's neighbors in the list are both later in the ordering. This leads to a straightforward linear-work parallel implementation of the algorithm.

The tree contraction problem is to contract a tree into a single node (possibly combining node values), and again has many applications [28, 29, 24]. We assume that the tree is a rooted binary tree. The sequential algorithm that we consider steps over the leaves of the tree in random order and, for each leaf, it splices the leaf and its parent out. We show that the iteration depth for this algorithm is again $O(\log n)$ w.h.p. For this algorithm, however, there seems to be no easy on-line way to determine when a step no longer depends on any other uncompleted steps. We show, however, that with some pre-processing we can identify the dependencies. This leads to a linear-work parallelization of the algorithm. We also show how to relax the dependencies such that the contraction is still correct and deterministic, but does not necessarily contract in the same order as the sequential algorithm, giving us a simple linear-work polylogarithmic-depth implementation.

Reducing the randomness required by algorithms is important, as randomness can be expensive. Straightforward implementations of our algorithms require $O(n \log n)$ random bits. By making use of a pseudorandom generator for space-bounded computation by Nisan [31], we show that the algorithms for random permutation and list contraction require only a polylogarithmic number of random bits w.h.p. This result is based on showing that our algorithms can be simulated in polylogarithmic space.

We have implemented all three of our algorithms in the deterministic reservations framework [6]. Our implementations for random permutation and list contraction contain under a dozen lines of `C` code, and tree contraction is just a few dozen lines. We have experimented with these implementations on a shared-memory multi-core machine with 40 cores, obtaining reasonably good speedups relative to the sequential iterative algorithms, on problems for which it is hard to compete with sequential algorithms.

Related Work. Beyond significant prior work on algorithms for the problems that we consider, which is mentioned in the sections on each problem, there has been prior work on understanding the parallelism in iterative (or greedy) sequential algorithms, including work on the maximal independent set and maximal matching problems [7], and on graph coloring [21].

Contributions. The main contributions of this paper are as follows. We show that the standard sequential algorithm for random permutation has low sequential dependence ($\Theta(\log n)$ w.h.p.). For list contraction and tree contraction, we show that the sequential algorithms also have a dependence that is logarithmic w.h.p. given a random ordering of the input. We show natural parallel implementations of these algorithms, where steps are processed as soon as all of their dependencies have been resolved. These parallel algorithms give the same result as the sequential algorithms, which is useful for deterministic parallel programming. We show linear-work parallel implementations of these algorithms using deterministic reservations and activation-based approaches. We also prove that our algorithms for random permutation and list contraction require only a polylogarithmic number of random bits w.h.p., in contrast to $O(n \log n)$ random bits in a straightforward implementation. Finally, our experimental results show that our implementations of the parallel algorithms achieve good speedup on a 40-core machine and outperform the sequential algorithms with just a modest number of cores.

2 Preliminaries

The ***random permutation*** problem takes as input an array A of length n and returns a random ordering of the elements of A such that each of the $n!$ possible orderings is equally likely. The ***list contraction*** problem takes as input a collection of linked lists represented by L, and contracts each list into a single node, possibly combining values on the nodes during contraction. The ***tree contraction***

[1] We use "with high probability" (w.h.p.) to mean probability at least $1 - 1/n^c$ for any constant $c > 0$.

[2] The random order can be implemented by first randomly permuting the nodes, then processing them in linear order.

problem takes as input a tree T and contracts the tree down to the root node, possibly combining values on the nodes during contraction.

In this paper, we state our results in the work-depth model, where ***work*** is equal to the number of operations required (equivalently, the product of the time and processors) and ***depth*** is equal to the number of time steps required. We use the parallel random access machine model (PRAM). We use the exclusive-read exclusive-write (EREW) PRAM, the arbitrary-write and priority-write versions of the concurrent-read concurrent-write (CRCW) PRAM, where a priority-write here means that the maximum value written concurrently is stored. We also use the scan PRAM [5], a variant of the EREW PRAM where scan (prefix sum) operations take unit depth. For the priority-write model we will use a ***writeMax(l,i)*** which writes value i to location l such that the maximum value written to l will end up in that location.

We use the standard definition of a ***random binary search tree***, i.e., the tree generated by inserting a random permutation of the integers $\{0, \ldots, n-1\}$ into a binary search tree.

In this paper, we are concerned with the parallelism available in sequential iterative algorithms. We assume that the iterative algorithm takes n steps, where each ***step*** performs some computation, depending on the results or effects of a subset of previous steps. We are interested in running some of these steps in parallel. What we can run safely in parallel will depend on both the algorithm and the input, which together we will refer to as a ***computation***. We will model the dependencies in the computation as a graph, where the steps $I = \{0, \ldots, n-1\}$ are vertices and dependencies between steps are directed edges, denoted by E.

DEFINITION 1. (ITERATION DEPENDENCE GRAPH) *An **iteration dependence graph** for an iterative computation is a (directed acyclic) graph $G(I, E)$ such that if every step $i \in I$ runs after all predecessor steps in the graph complete, then every step will do the same computation as in the sequential order.*

We are interested in the depth of an iteration dependence graph, which we refer to as the ***iteration depth***, $D(G)$. It should be clear that we can correctly simulate a computation with iteration dependence graph G in $D(G)$ rounds, each running a set of steps in parallel. However, it may not be clear how to efficiently determine for each each step if all of its predecessors have completed. As we will see, and not surprisingly, the method for doing this check is algorithm specific. We will say that a step can be ***efficiently checked*** if we can determine that all its predecessors have completed in constant work/depth, and ***efficiently updated*** if the step itself takes constant work/depth.

We define ***aggregate delay***, $A(G)$, of an iteration dependence graph G to be the sum of the heights (one plus the longest directed path to a vertex) of the vertices in G. To understand why this is a useful measure, consider a process in which on every round all steps that have not yet completed check to see if their predecessors are complete, and if so they run and complete, otherwise they try again in the next round. Each round can be run in parallel, and each step is delayed by a number of rounds corresponding to its height in G. Assuming each non-completed step does constant work on each round, then the total work across all steps and all rounds will be bounded by $O(A(G))$.

We will show that all three of our algorithms have steps that can be checked and updated in constant time, and have iteration dependence graphs with $O(\log n)$ depth and $O(n)$ aggregate delay. However, tree contraction requires pre-processing to allow for efficient checking.

```
1: procedure SEQUENTIALRANDPERM(A, H)
2:     for i = n − 1 to 0 do
3:         swap(A[H[i]], A[i])
```

Figure 1: Sequential algorithm for random permutation.

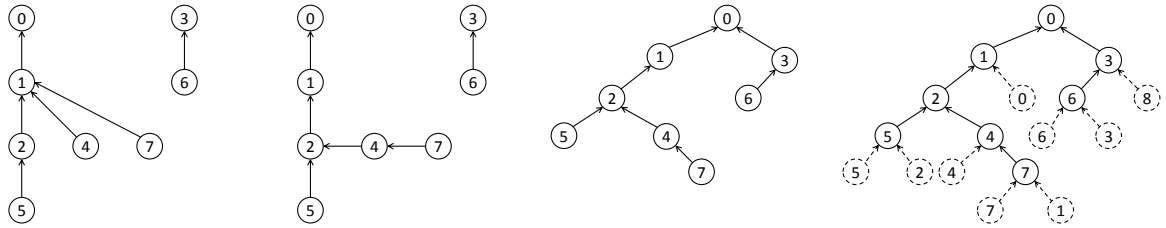
3 Bounding Iteration Depth and Aggregate Delay

In this section, we analyze the iteration depth and aggregate delay for algorithms for the three problems that we are concerned with: random permutation, list contraction and tree contraction. Sections 4 and 5 will then describe how to efficiently check for dependencies and how this leads to efficient parallelizations of the algorithms.

3.1 Random Permutation. Durstenfeld [13] and Knuth [26] discuss a simple sequential algorithm for generating a random permutation which goes through the elements of an array from the end to the beginning (or the other way), and for each element swaps with a random at or earlier position in the array. We assume that the random integers used in the algorithm are generated beforehand, and stored in an array H—i.e., for $0 \leq i < n$, $H[i]$ is a (uniformly) random integer from 0 to i, inclusive. The pseudo-code for Durstenfeld's sequential algorithm is given in Figure 1.

Generating random permutations in parallel has been well-studied, both theoretically [1, 2, 10, 14, 16, 17, 18, 20, 28, 33] and experimentally [9, 19]. Many of these algorithms do linear work and have polylogarithmic depth. As far as we know, however, none of this work has considered the parallelism available in Durstenfeld's sequential algorithm, and none of them return the same permutation as it does, given the same source of randomness.

To analyze the iteration dependence depth of Durstenfeld's algorithm we will use the following definitions. When performing a swap(x, y) we say x is the ***source*** of the swap and y is the ***target*** of the swap. For a given H, we say i ***dominates*** j if $H[i] = j$ and $i \neq j$. We define the



(a) Dominance forest (b) Dependence forest (c) Linked dependence tree (d) Possible locations for $H[8]$

Figure 2: Dominance and dependence forests for $H = [0, 0, 1, 3, 1, 2, 3, 1]$ are shown in (a) and (b), respectively. (c) shows the linked dependence tree for H and (d) shows the possible locations for inserting the 9^{th} node; dashed circles correspond to the value of $H[8]$.

dominance forest of H to be the directed graph formed on n nodes where node i points to node j if i dominates j. Since each node can dominate at most one other node, the graph is a forest. Note that the roots of the dominance forest are exactly the nodes where $H[i] = i$.

Define the ***dependence forest*** of H to be a modification of the dominance forest where the children of each node (from incoming edges) are chained together in decreasing order. In particular, for a node i with incoming edges from nodes $j_1 < \ldots < j_k$ we add an edge from j_{l+1} to j_l for $1 \leq l < k$ (creating a chain) and delete the edges from j_l to i for $l > 1$. Note that the dependence forest is binary, since each node can have at most one incoming edge from the set of nodes pointing to it in the dominance forest, and since it can be part of at most one chain. See Figures 2(a) and 2(b) for an example of the dominance forest and dependence forest for a given H.

LEMMA 3.1. *The dependence forest of H is an iteration dependence graph for* SEQUENTIALRANDPERM.

Proof. We define a step to be ***ready*** if all of its descendants in the dependence forest have been processed. We will show that when a step is ready, its corresponding location in A will contain the same value as it would have when the sequential algorithm processes it. The proof uses induction on the iteration in which a step is processed in the sequential algorithm (i.e., step $n-1$ is the first and step 0 is the last).

The base case is trivial as step $n-1$ is ready at the start of any ordering (no node can point to $n-1$ in the dependence forest) and has the correct value (location $n-1$ cannot be the target of any swap with another element). Consider some step i. Suppose there are multiple steps $j_1, \ldots, j_k$ where $j_1 < j_2 < \ldots < j_k$ with location i as the target of a swap operation. Since $i < j_1 < \ldots < j_k$, by the inductive hypothesis we may assume that steps $j_1, \ldots, j_k$ had the correct value in their corresponding locations in A when they were ready. The sequential algorithm will perform the swaps in decreasing order of the steps (j_k down to j_1), and since $i < j_1$, in the sequential algorithm location i will not be the source of a swap until all of steps $j_1, \ldots, j_k$ have been processed. Any ordering respecting the dependence forest will also process steps $j_1, \ldots, j_k$ in decreasing order, since by definition the dependence forest contains a directed path from j_k to j_1. The fact that $j_1, \ldots, j_k$ have the same value as in the sequential algorithm when they are ready, and that they are processed in the same order as the sequential algorithm implies that the location corresponding to step i will also have the same value as in the sequential algorithm when it is ready (i.e., after all of its incoming steps have been processed). ■

We are interested in showing that the dependence forest is shallow. To do this we will actually add some additional edges to make a tree and then show that this tree has an identical distribution as random binary search trees, which are known to have $\Theta(\log n)$ depth with high probability. We define the ***linked dependence tree*** as the tree created by linking the roots of the dependence forest along the right spine of a tree with indices appearing in ascending order from the top of the spine to the bottom (see Figure 2(c) for an example of the linked dependence tree). The linked dependence tree is clearly also an iteration dependence graph since it only adds constraints.

THEOREM 3.1. *Given a random H, the distribution of (unlabeled) linked dependence trees for H is identical to the distribution of (unlabeled) random binary search trees.*

Proof. We prove this by induction on the input size n. For the base case, $n = 1$, there is a single vertex and the claim is trivially true. For the inductive case note that the linked dependence tree for the first $n-1$ locations is not affected by the last location since numbers at $H[i]$ point at or before i—i.e., the last location will end up as a leaf. By the inductive hypothesis, the distribution of trees on the first $n-1$ locations has the same distribution as random binary search trees of size $n-1$. Now we claim that, justified below, the n^{th} element can go into any leaf position. Since the n^{th} location is a uniformly random integer from 0 to $n-1$ and there are n possible

leaf positions in a binary tree of size $n-1$, all leafs must be equally likely. Hence this is the same process as inserting randomly into a binary search tree.

To see that the n^{th} location can go into any leaf, first note that if it picks itself (index $n-1$), then it is at the bottom of the right spine of the tree, by definition. Otherwise if it picks $j < n-1$, and it will be placed at the bottom of the right spine of the left child of j. This allows for all possible tree positions—to be a left child of a node just pick the parent, and to be a right child follow the right spine up to the top, then pick its parent (e.g., see Figure 2(d)). ■

THEOREM 3.2. *For* SEQUENTIALRANDPERM *on a random H of length n, there is an iteration dependence graph G with $D(G) = \Theta(\log n)$ w.h.p., and $A(G) = \Theta(n)$ in expectation.*

Proof. For the depth, it is a well-known fact that the height of a random binary search tree on n nodes is $\Theta(\log n)$ w.h.p. [12], so Theorem 3.1 implies that the longest path in the iteration dependence graph is $O(\log n)$ w.h.p. To show that this is tight, note that node 0 has $\Theta(\log n)$ incoming edges in the dominance forest w.h.p., and hence the longest path to it in the iteration dependence graph is $\Omega(\log n)$ w.h.p.

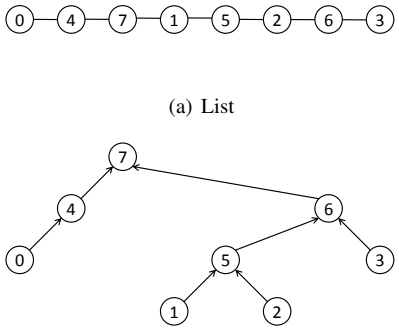
To analyze the aggregate delay we analyze the sum of heights of the nodes in a random binary search tree. Let $W(n)$ indicate the expected sum. The two children of the root of a random binary search tree are also random binary search trees of size i and $n-i-1$, respectively, for a randomly chosen i in $\{0,\ldots,n-1\}$. We therefore have the recurrence: $W(n) = H(n) + \frac{1}{n}\sum_{i=0}^{n-1}(W(i) + W(n-i-1))$, where $H(n) = \Theta(\log n)$ is the expected height of a random binary search tree with n nodes. This solves to $\Theta(n)$ and hence the theorem follows. ■

3.2 List Contraction. List contraction, and the related list ranking, is one of the most canonical problems in the study of parallel algorithms. The problem has received considerable attention both because of its fundamental nature as a pointer-based algorithm, and also because it has many applications as a subroutine in other algorithms. A summary of the work can be found in a variety of books and surveys (see e.g. [25, 24, 36]).

Here we are concerned with analyzing a simple sequential algorithm for list contraction and showing that it has low iteration depth and aggregate delay. We assume the linked list is represented as an array L of nodes, where $L[i]$.prev stores the index of the predecessor of node i (null if none) and $L[i]$.next stores the index of the successor of node i (null if none). A natural sequential iterative algorithm works by splicing out the nodes in order of increasing index, as shown in Figure 3. Each list in L is contracted down to a single node. For simplicity we do not show the values stored on the nodes. If values are stored, then when a node is spliced out its value is combined with its predecessor's value using a combining function, and stored on its predecessor. To perform list ranking, the process is then reversed, adding the nodes back in with the appropriate values. Note that when the combining function is non-associative, then the result depends on the order in which the nodes are spliced out. In such a case, a parallel computation returns the same answer as the sequential algorithm if it satisfies the dependence structure of the sequential algorithm, which we define next.

```
1: procedure SEQUENTIALLISTCONTRACT(L)
2:     for i = 0 to n − 1 do
3:         if L[i].prev ≠ null then
4:             L[L[i].prev].next = L[i].next
5:         if L[i].next ≠ null then
6:             L[L[i].next].prev = L[i].prev
```

Figure 3: Sequential algorithm for list contraction.



(a) List

(b) Dependence forest

Figure 4: (a) An example list, where the numbers represent the position in the input array L, and (b) its dependence forest.

We define the following ***dependence forest*** for an input L. For a list, place the last position k in which any of its links appear at the root r of a tree. Now recursively for the sublists on each side of the node in position k, do the same and make the two roots the children of r. If either sublist is empty, r will not have a child on that side. This defines a tree for each list and a forest across multiple lists. As with the dependence forest for random permutation, the dependencies go up the tree—i.e., each parent depends on its children. An example list along with its dependence forest is shown in Figure 4.

LEMMA 3.2. *The dependence forest of L is an iteration dependence graph for* SEQUENTIALLISTCONTRACT(L).

Proof. For each step i, let j and k be the indices of prev and next nodes when i is spliced out in the sequential order. Clearly j and k must both be larger than i (or null) since they have not yet been spliced out. We need to show that for each i, once all of its descendants in the dependence forest are completed (spliced out), possibly not in the sequential

order, it will point to j and k, and hence will do an identical splice as in the sequential order. By induction we assume this was true for all indices less than i.

Consider the sublist between j and k (not inclusive). The index i must be the largest index on this list because if there were a larger index l, when i is contracted in the sequential order it cannot be linked with both j and k—l must be in the way. By construction of the dependence forest, and because i is the largest on the sublist, it is picked as the root of a tree containing the sublist. Therefore when all descendants are completed (and by induction we assumed they operated correctly) all other nodes on the sublist have been spliced out and i will point to j and k. ■

LEMMA 3.3. *Assuming the ordering of L has been randomized, for each list in L the distribution of (unlabeled) dependence trees is identical to the distribution of (unlabeled) random binary search trees of the same size.*

Proof. The root node of the dependence tree can appear in any position of the list with equal probability, since L is randomly ordered. This property also holds for each sublist of the list. Therefore in each subtree all nodes are equally likely to be the root, which is equivalent to the distribution for random binary search trees. ■

The following theorem now follows from the same argument as in Theorem 3.2 since the iteration dependence graph (for each list) has the same distribution—a random binary search tree. There are no dependencies among different lists.

THEOREM 3.3. *For* SEQUENTIALLISTCONTRACT *on a randomly ordered L of length n, there is an iteration dependence graph G with $D(G) = O(\log n)$ w.h.p., and $A(G) = \Theta(n)$ in expectation.*

3.3 Tree Contraction. As with list contraction, parallel algorithms for tree contraction has received considerable interest [28, 24, 36]. There are many variants of parallel tree contraction. Here we will assume we are contracting rooted binary trees in which every internal node has exactly two children. To represent the tree we use an array T of nodes, each with a parent and two child pointers, with the first n nodes being leaves, and the next $n-1$ being the internal nodes.

We consider an iterative sequential algorithm for tree contraction that rakes the leaves of the tree one at a time, shown in Figure 5. To *rake* a leaf v, we splice it and its parent p out of the tree—i.e, set v's sibling's parent pointer to be v's grandparent, and v's grandparent's child pointer to point to v's sibling instead of p. At the end only the root node remains. As in list contraction, values can be stored on the nodes, and combined during contraction (e.g., for evaluating arithmetic expressions), but we leave

```
1: procedure SEQUENTIALTREECONTRACT(T)
2:     for i = 0 to n − 1 do
3:         p = T[i].parent
4:         if T[p].parent ≠ null then                ▷ p is not root
5:             s = sibling(T, i)
6:             T[s].parent = T[p].parent
7:             switchParentsChild(T, p, s)
8:         else switchParentsChild(T, i, null)       ▷ p is root
```

Figure 5: Sequential algorithm for tree contraction, where *sibling*(T, i) returns the sibling of i in T, and *switchParentsChild*(T, i, v) resets the appropriate child pointer of the parent of i to point to v instead of i.

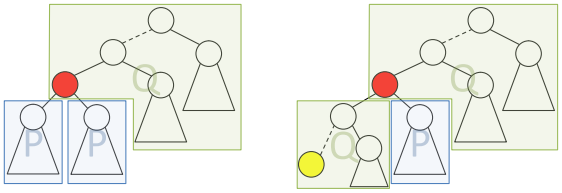
it out of the code. Again, if the combining function is non-associative, then the result depends on the order that we rake the leaves in, and a parallel computation returns the same result as the sequential algorithm if it satisfies the dependence structure of the sequential algorithm.

We define the following labeling of internal nodes, and then define a dependence structure based on it. Let $M(i)$ for each node i be the maximum index of any of the leaves in its subtree, and the *label* of each internal node be $L(i) = \min\{M(j), M(k)\}$, where j and k are the two children of i. The following fact about labels will be useful.

LEMMA 3.4. *In* SEQUENTIALTREECONTRACT *on a tree T the internal node with label i will be raked by the leaf with index i.*

Proof. We prove this by induction. The base case for a tree with a single leaf is trivial as there are no internal nodes. Now assume by induction that this holds for the internal nodes of two separate subtrees, joined together by a new root r. The highest-indexed leaf in each subtree will not appear as a label in the subtrees since the root takes the minimum of the two subtrees, and hence the highest-indexed leaf must be the leaf that remains when the tree is contracted (by induction). Thus, one of the two highest-indexed leaves in the two subtrees must be the node that rakes r. The smaller of these two leaves will be processed first, which is also the label on r by definition. This proves the lemma. ■

The ***dependence tree*** for a tree T is the tree created by taking the maximum label i and placing it at the root. We then partition the tree T by removing the internal node labeled with i, and recursively apply this process to each subtree. The three resulting dependence trees become the children of i. This is repeated until we reach a leaf. Note that this process creates a tree over the leaf indices, since each label corresponds to a leaf index. Also note that this process is similar to how we defined the dependence forest for the list contraction problem, and hence the proof of the lemma below has a similar structure.



(a) Tree decomposition for P-state tree

(b) Tree decomposition for Q-state tree

Figure 6: P-state and Q-state trees used in the proof of Theorem 3.4. The red node is v_s, the interior node corresponding to the leaf with the second largest label. The yellow node is leaf l, the leaf with the largest label.

LEMMA 3.5. *The dependence tree of T is an iteration dependence graph for* SEQUENTIALTREECONTRACT(T).

Proof. For each step i, let j and k be the labels of i's sibling and grandparent when it is raked in the sequential order. We assume leaves have null labels, so the sibling could be null. The labels j and k must both be larger than i (or null) since they have not yet been raked out. We need to show that for each i, once all of its descendants in the dependence tree are completed (raked out), it will have sibling j and grandparent k, and hence will do an identical rake as in the sequential order. By induction we assume this was true for all indices less than i.

Consider the tree between j and k (not inclusive). The label i must be the largest label in this tree since if there were a larger label l, when i is contracted in the sequential order it cannot have both j as a sibling and k as a grandparent—the node with label l is not yet raked out and must be in the way. By construction of the dependence tree, and since i is the largest label in the subtree, it is picked as the root of a dependence tree containing the subtree. Therefore when all descendants are completed (and by induction we assumed they operated correctly), all other nodes on the subtree have been raked out and i will have j as a sibling and k as a grandparent. ■

We are now ready to analyze the iteration depth and work of a dependence tree.

THEOREM 3.4. *For* SEQUENTIALTREECONTRACT *on T with n randomly ordered leaves, there is an iteration dependence graph G with $D(G) = O(\log n)$ w.h.p., and $A(G) = \Theta(n)$ in expectation.*

Proof. The dependence tree for T is based on recursively partitioning T into subtrees. To analyze the depth of the dependence tree, we need to consider two types of subtrees, which have different properties. We define a (sub)tree to be in the ***P-state*** if the distribution of its leaves is uniformly random. We define a subtree to be in the ***Q-state*** if the location of its highest-indexed leaf is fixed. Without loss of generality, we assume a Q-state tree has its highest-indexed leaf on its left spine. We denote the leaf with the largest index in a subtree by l, the leaf with the second largest index by s, and the internal node with label s by v_s.

The initial tree is in the P-state since the ordering of the leaves is uniformly random. For a P-state tree, it is partitioned by v_s into three subtrees, where the two subtrees of the children of v_s are also in the P-state but the final tree is in the Q-state (see Figure 6(a)). This is because as we process v_s's children's subtrees there is no information about the location of the highest-indexed leaf. However after both of the children's subtrees are processed, then leaf l will become a leaf in v_s's original position in (note that leaf l must be in v_s's subtree by definition), hence fixing the location of the highest-indexed leaf in the remaining subtree.

For a tree in the Q-state, it is partitioned by v_s into three subtrees (see Figure 6(b)), where v_s's left child subtree is in the Q-state (as we fixed leaf l to be on the left spine), v_s's right child subtree is in the P-state (we have no information about the location of the highest-index leaf in this subtree), and the remaining subtree is in the Q-state as after v_s's subtree is completely processed, leaf l will become a leaf in v_s's original position.

For a tree with n nodes in the P-state, the size of v_s's subtree is greater than $3n/4$ with at most probability $1/4$. This is because the location of leaf l is random and for v_s's subtree not to contain leaf l, it must appear in the rest of the tree, which has at most $1/4$ probability of occurring if v_s's subtree size is greater than $3n/4$. Hence, at least one of v_s's children's subtree has size greater than $3n/4$ with probability at most $1/4$. By a similar argument, the other subtree (of the Q-state) also has size greater than $3n/4$ with probability at most $1/4$.

For a tree with n nodes in Q-state, the size of v_s's left child's is greater than $3n/4$ with at most probability $1/4$. This is because the location of leaf s must appear in v_s's right subtree by definition, and the location of leaf s is uniformly random, so with at most $1/4$ probability it causes v_s to have a left child of size at least $3n/4$. For the subtree remaining after removing v_s's subtree, it's size is greater than $3n/4$ with probability at most $1/4$ by a similar argument. Note that we have no bound on the size of v_s's right child subtree in the P-state. However, this is fine because once a tree transitions into P-state, it will be divided into small subtrees according to the analysis for P-state trees in the previous paragraph.

We consider paths from the root to each leaf in the dependence tree. Every two steps on such a path will shrink the size of the tree by a factor of $3/4$ with constant probability (by the arguments above). Therefore, using standard arguments, each path will have $O(\log n)$ steps w.h.p., and by a union bound (multiplying the failure probability by n), all path lengths and hence the tree depth will be $O(\log n)$ w.h.p.

To show $A(G)$, we note that a node in the dependence

tree with a subtree of size k will have height $O(\log k)$ in expectation since it is true w.h.p. from the previous discussion. Let $W(n)$ indicate the expected sum of the heights of the nodes in the dependence tree. For a tree of size n, after two levels with constant probability the largest remaining component will be $3/4n$. Assuming the worst case split is $3/4n$ and $1/4n$ when this is true, we have the recurrence $W(n) \leq O(\log n) + p \times \big(W(\frac{3}{4}n) + W(\frac{1}{4}n)\big) + (1-p) \times W(n)$ for some constant $0 < p < 1$. By substitution, we have that $W(n) = O(n)$. ■

4 Algorithmic Design Techniques

We note that we can easily obtain implementations from the iteration dependence graph. If steps in a computation can be efficiently checked and updated, then an algorithm for a problem with iteration depth $D(G)$ can be implemented with $O(nD(G))$ work and $O(D(G))$ depth simply by proceeding in rounds, where in each round all steps check if their predecessors in the iteration dependence graph have been processed, and proceed if so. Since we are interested in work-efficient (linear-work) algorithms, we will prove the following lemma, which we use in Section 5 to obtain linear-work algorithms for the three problems.

LEMMA 4.1. *If steps can be efficiently checked and updated, then an algorithm for a problem with iteration depth $D(G)$ can be implemented with $O(A(G))$ work and $O(D(G)\log n)$ depth on the EREW PRAM, $O(D(G)\log^* n)$ depth w.h.p. on the CRCW PRAM, or $O(D(G))$ depth on the scan PRAM.*

Proof. We define a step to be ***ready*** if all of its predecessors in the iteration dependence graph have been processed. The algorithm proceeds in rounds, where in each round all remaining steps check if they are ready. If a step is ready, it proceeds in executing its computation. After processing the ready steps, consider them as having been removed from the iteration dependence graph, and hence the iteration depth of the remaining iteration dependence graph is 1 less than before. The initial iteration depth is $D(G)$, so $D(G)$ rounds suffice. In each round, we pack out the successful steps so that no additional work is done for them in later rounds. The pack requires linear work in the number of remaining steps. Since each round removes the leaves of the iteration dependence graph, and the steps can be efficiently checked and updated, the work done on each step is proportional to its height in the graph. The total work is proportional to the sum of the heights of all steps in the iteration dependence graph, which is the aggregate delay $A(G)$. The depth of the algorithm is $O(D(G)P(n))$, where $P(n)$ is the depth of the pack, which is $O(\log n)$ on the EREW PRAM, $O(\log^* n)$ w.h.p. on the CRCW PRAM using approximate compaction [17] and $O(1)$ on the scan PRAM. This proves the lemma. ■

We now describe two techniques that we use to obtain algorithms for the three problems in Section 5. The deterministic reservations method checks all remaining steps in each round, executing the ones whose dependencies have all been satisfied, and gives algorithms satisfying the bounds of Lemma 4.1. The activation-based approach directly activates a step when it is ready.

4.1 Deterministic Reservations. Deterministic reservations is a framework introduced by Blelloch et al. [6] for designing deterministic parallel algorithms. It gives a way for steps in a parallel algorithm to check if all of their dependencies have been satisfied using shared data structures. Deterministic reservations proceeds in rounds, where each round consists of a *reserve phase*, followed by a synchronization point, and then a *commit phase*. In the reserve phase, all (or a prefix) of the steps first write to locations in shared data structures corresponding to the steps it conflicts with in the iteration dependence graph. After synchronizing, then all (or a prefix) of the steps each check whether it can proceed with its computation in the commit phase. Steps that fail to proceed in the commit phase (it has a conflict with a step earlier in the ordering) survive to the next round. This is repeated until no steps remain. In deterministic reservations, one can either execute all of the steps in a round, or just a prefix of them. In the prefix-based approach, each prefix is processed until completion before moving on to the next prefix. In practice, this gives a nice trade-off between extra work and parallelism by adjusting the size of the prefix.

In the framework, each algorithm specifies only a RESERVE function and a COMMIT function for the steps, executed in the reserve and commit phase, respectively. This yields very concise code. Each function takes the step number as an argument. RESERVE returns 0 if the step drops out; otherwise it applies the reservation and returns 1. COMMIT returns 0 if the step successfully commits (and drops out), and 1 otherwise. The steps that drop out are removed at the end of the round. The deterministic reservations approach directly gives algorithms satisfying the bounds in Lemma 4.1.

4.2 Activation-based Approach. The activation-based approach directly "wakes-up" (activates) each step exactly when it is ready [7, 21]. In particular, the predecessors in the iteration dependence graph are responsible for activating the step. At the beginning, we identify all the steps that do not depend on any others (in our examples, these can be determined easily). Then on each round, each active step executes its computation, and then detects whether it is the last predecessor of a successor; if so, it wakes up the successor. The approach is work-efficient since it only runs steps exactly when they are needed. As we will see, the implementations are problem-specific.

```
1: H = swap targets
2: R = {−1, ..., −1}
3: procedure RESERVE(i)
4:     writeMax(R[i], i)                    ▷ reserve own location
5:     writeMax(R[H[i]], i)                 ▷ reserve target location
6:     return 1
7: procedure COMMIT(i)
8:     if (R[i] = i and R[H[i]] = i) then
9:         swap(A[H[i]], A[i])              ▷ swap if reserved
10:        return 0
11:    else return 1
```

Figure 7: RESERVE and COMMIT functions and associated data for random permutation.

5 Parallel Algorithms

In this section, we describe parallel algorithms for random permutation, list contraction and tree contraction designed using the deterministic reservations approach and the activation-based approach, discussed in Section 4.

5.1 Random Permutation. To implement the random permutation algorithm using deterministic reservations, we specify the RESERVE and COMMIT functions shown in Figure 7. The implementation uses an array R, initialized to contain all -1, to store reservations. The RESERVE function for index i simply calls writeMax to the two locations $R[i]$ and $R[H[i]]$ with value i and returns 1. The COMMIT function simply checks if both writeMax's were successful (i.e., both $R[i]$ and $R[H[i]]$ store the value i) and if so, swaps $A[H[i]]$ and $A[i]$ and returns 0; otherwise it returns 1. This process guarantees that a step will successfully commit (swap) if only if its children in the dependence forest have finished in a previous round of deterministic reservations. This is because if any child were not finished, then it would have competed in the writeMax and won since it has a higher index. In particular, the left child as shown in Figure 2(b) will win on $R[i]$ and the right child in that figure will win on $R[H[i]]$.

THEOREM 5.1. *For a random H, deterministic reservations using the* RESERVE *and* COMMIT *functions for random permutation runs in $O(n)$ expected work and $O(\log n \log^* n)$ depth w.h.p. on the priority-write CRCW PRAM.*

Proof. Apply Theorem 3.2 and Lemma 4.1. The RESERVE and COMMIT functions take constant work/depth, so the steps of the computation can be efficiently checked and updated. The writeMax requires the priority-write CRCW PRAM. ■

Activation-Based Implementation. We now discuss a linear-work activation-based implementation of the parallel random permutation algorithm. The implementation keeps track of the nodes ready to be executed of the dependence graph, processes and deletes these nodes from the graph in each round, and identifies the new nodes that are ready for the next round. It relies on constructing the dependence forest, and the following lemma states that this can be done efficiently.

LEMMA 5.1. *The dependence forest for a given H can be constructed in $O(n)$ expected work and $O(\log n)$ depth w.h.p. on the CRCW PRAM.*

Proof. Building the dependence forest of random permutation for a given H requires sorting all of the nodes which point to the same node in the forest. We do this by (1) using a non-stable integer sort in the range $[1, \ldots, n]$ [33] to group all the nodes, and then (2) sorting the nodes within each group using a parallel comparison sort [24]. (1) can be done in $O(n)$ work and $O(\log n)$ depth on the CRCW PRAM. The depth for (2) is $O(\log \log n)$ w.h.p. since the largest group is of size $O(\log n)$ w.h.p. The total work for (2) is $\sum_{i=0}^{n-1} cs_i \log s_i$ where s_i is the number of nodes pointing to node i and c_1 is a constant. To show that $\sum_{i=0}^{n-1} c_1 s_i \log s_i = O(n)$, we use a similar argument used in the analysis of perfect hash tables [30]. Let $X_{ij} = 1$ if $H[i] = H[j]$ and $X_{ij} = 0$ otherwise.

$$
\begin{aligned}
\sum_{i=0}^{n-1} c_1 s_i \log s_i &\le \sum_{i=0}^{n-1} c_2 s_i^2 && \text{for some constant } c_2 \\
&= c_2 \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} X_{ij} \\
&= c_2 (n + 2\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} X_{ij}) && \text{consider } X_{ij} \text{ where } i < j \\
&\le c_2 (n + 2\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \frac{1}{i+1}\frac{1}{j+1}) && (*) \\
&\le c_2 (n + 2\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} \frac{1}{(i+1)^2}) \\
&\le c_2 (n + 2n\sum_{i=1}^{n} \frac{1}{i^2}) \\
&< c_2 (n + 2n \cdot \frac{\pi^2}{6}) \\
&= O(n)
\end{aligned}
$$

(*) follows because $H[i]$ and $H[j]$ are independent.

After sorting, creating the pointers in the dependence forest takes $O(n)$ work and $O(1)$ depth. ■

We now use Theorem 5.1 to design an activation-based random permutation algorithm.

THEOREM 5.2. *For a random H, an activation-based implementation of random permutation runs in $O(n)$ expected work and $O(\log n \log^* n)$ depth w.h.p. on the CRCW PRAM.*

Proof. We form the dependence forest for a given H,

which by Lemma 5.1 can be done in $O(n)$ expected work and $O(\log n)$ depth w.h.p. on the CRCW PRAM.

We first identify the leaves of the dependence forest and maintain the set of leaves at each step (these are the steps that are ready to be processed). Then we repeatedly process the leaf set, remove it and its edges from the graph, and identify the new leaf set until the dependence forest has been completely processed. Since we are satisfying all dependencies in the dependence forest, by Lemma 3.1, this guarantees correctness. We assume that the neighbors of a node are represented in an array, and partitioned into incoming edges and outgoing edges. To identify the new leaf set at each step, nodes that are removed perform a check on its parent to see if it has any incoming edges remaining. The check can be done in $O(1)$ work and time per neighbor since each node has at most two incoming edges.

After all checks are completed, nodes with no incoming edges are added to the next leaf set. Duplicates can be eliminated by filtering in work linear in the size of the new leaf set since each node can be duplicated at most once (each node has at most 2 incoming edges). The new leaf set is packed with approximate compaction, requiring work linear in the leaf set size and $O(\log^* n)$ depth w.h.p. Each step is processed a constant number of times, so the total work is $O(n)$. Each round reduces the iteration depth of the iteration dependence graph on the remaining steps by 1, and since the initial iteration depth is $\Theta(\log n)$ w.h.p. by Theorem 3.2, the overall depth is $O(\log n \log^* n)$ w.h.p. ■

Adapting to the CRQW PRAM. We adapt our random permutation algorithms to the concurrent-read queue-write (CRQW) PRAM [14, 15], which closely models cache coherence protocols in multi-core machines. In this model, concurrent reads to a memory location are charged unit cost but concurrent writes to a memory location have a contention cost equal to the total number of concurrent writes to the location. In each step, the maximum contention over all locations is charged to the depth.

Lemma 5.1 also applies for the CRQW PRAM as integer sorting can be done in $O(n)$ work and $O(\log n)$ depth w.h.p. on the CRQW PRAM [14], and comparison sorting can be implemented on an EREW PRAM (a weaker model than the CRQW PRAM). Packing on the CRQW PRAM can be done in linear work and $O(\sqrt{\log n})$ depth w.h.p. [15], so an activation-based implementation of the sequential algorithm can be made to run in $O(n)$ expected work and $O(\log^{1.5} n)$ depth w.h.p.

The deterministic reservation-based implementation of random permutation can also be adapted to the CRQW PRAM, using prefix sums for packing. The only place in the algorithm that requires concurrent writes is the call to writeMax. However since the dominance forest has in-degree $O(\log n)$ w.h.p., there can be at most $O(\log n)$ concurrent calls to writeMax to a given location, leading to $O(\log n)$ contention. This requires $O(\log n)$ additional slackness (depth) per step. Using prefix sums for packing, each round already requires $O(\log n)$ depth, so this slackness does not affect the overall bounds, and we are left with an algorithm that does linear work and $O(\log^2 n)$ depth w.h.p. on the CRQW PRAM.

```
1:  R = {0, ..., 0}                                   ▷ boolean array
2:  procedure RESERVE(i)
3:      if i < L[i].prev and i < L[i].next then
4:          R[i] = 1                                  ▷ reserve own location
5:      return 1
6:  procedure COMMIT(i)
7:      if (R[i] = 1) then
8:          if L[i].prev ≠ null then
9:              L[L[i].prev].next = L[i].next
10:         if L[i].next ≠ null then
11:             L[L[i].next].prev = L[i].prev
12:         return 0
13:     else return 1
```

Figure 8: RESERVE and COMMIT functions and associated data for list contraction.

Random Permutation via Rotations. Here we describe another parallel implementation of the sequential algorithm, using the fact that the values at the locations of the nodes pointing to the same node in the dominance forest just get rotated. In particular, if $i_1, \ldots, i_k$ with $i_l < i_{l+1}$ point to j, then after all other dependencies to $i_1, \ldots, i_k$ are resolved, then $A[j] = A[i_k], A[i_1] = A[j]$ and $A[i_{l+1}] = A[i_l]$ for $1 \leq l < k$. We can build the dominance forest using an integer sort to group the nodes and then a comparison sort within each group in $O(n)$ work and $O(\log n)$ depth w.h.p. on the CRCW PRAM by the same analysis as done in the proof of Lemma 5.1. Then we can process the forest level by level, starting with the leaves, and rotating the values of each group of leaves and the target node. The level numbers for the nodes can be computed using leaffix operations or Euler tours [24] in linear work and $O(\log n)$ depth. Rotating the values can be done in work proportional to the number of nodes processed, and $O(1)$ depth. As the height of the dominance forest is $\Theta(\log n)$ w.h.p., this gives an algorithm with $O(n)$ work and $O(\log n)$ depth w.h.p. on the CRCW PRAM. It can also be implemented in the same bounds, as the integer sort can be done in $O(n)$ work and $O(\log n)$ depth w.h.p. on the CRQW PRAM. We note that however this approach is less practical than the approach using deterministic reservations.

5.2 List Contraction. The deterministic reservations implementation (pseudo-code shown in Figure 8) of list contraction maintains a boolean array R initialized to

all 0's. The RESERVE function for index i checks if $i < L[i]$.prev and $i < L[i]$.next, and if so, writes a value of 1 to $R[i]$. The COMMIT function for index i checks if $R[i]$ is equal to 1 and if so, splices out the node $L[i]$ and returns 0; otherwise it returns 1. These functions preserve the ordering imposed by the iteration dependence graph of L throughout its execution. To see this, note that if neither of its current neighbors in the list is lower-indexed, then step i will be a leaf in the iteration dependence graph by definition (both neighbors will be selected as roots before i in the dependence graph construction process, so i will have no descendants). Only in this case will $R[i]$ be set to 1 in the RESERVE step and step i executes its COMMIT step. Otherwise, step i will not proceed. Therefore, by Lemma 3.2, it generates the same result as the sequential algorithm.

The RESERVE and COMMIT functions take constant work/depth, so the steps of the computation can be efficiently checked and updated. By applying Theorem 3.3 and 4.1, we obtain the following theorem for list contraction. We can implement list contraction on the EREW PRAM because reads and writes of the neighbors inside the RESERVE and COMMIT steps can be separated into a constant number of phases such that there are no reads or writes to the same location in a phase.

THEOREM 5.3. *For a random ordering of L, deterministic reservations using the* RESERVE *and* COMMIT *functions for list contraction runs in $O(n)$ expected work and $O(\log^2 n)$ depth w.h.p. on the EREW PRAM, $O(\log n \log^* n)$ depth w.h.p. on the CRCW PRAM, or $O(\log n)$ depth on the scan PRAM.*

Activation-Based Implementation.

THEOREM 5.4. *For a random ordering of L, an activation-based implementation of list contraction runs in $O(n)$ work and $O(\log^2 n)$ depth w.h.p. on the EREW PRAM, $O(\log n \log^* n)$ depth w.h.p. on the CRCW PRAM, or $O(\log n)$ depth on the scan PRAM.*

Proof. For each node, we store a counter keeping track of the number of lower-indexed neighbors it has in the list. These counters can be initialized in linear work and constant depth. Then we identify the "roots", which are the nodes whose counters are 0 (they have no lower-indexed neighbors). In each round, we process all roots, and update the counters of their neighbors as follows. For a root v, let v_{next} be the successor node of v and v_{prev} be the predecessor node of v. We first analyze the case where $v_{next} > v_{prev}$. We also have that $v_{prev} > v$ by definition of a root. After splicing out v, v_{next} becomes a neighbor of v_{prev} so we decrement the counter of v_{prev}. If the counter of v_{prev} reaches 0, then we add it to the next set of roots. The counter of v_{next} is left unchanged as its new neighbor is still a lower-indexed neighbor. In the case where $v_{prev} > v_{next}$, we decrement the counter of v_{next}, and check whether it reaches 0. By splitting the reads and updates of neighbors into a constant number of phases, no concurrent reads or writes are required.

This algorithm satisfies the iteration dependence graph by noting that a node will only be spliced out if both of its neighbors in the list have higher indices, and appealing to the same argument made for the correctness of the deterministic reservations-based implementations of list contraction. Each round processes all leaves in the dependence graph, so by Theorem 3.3, $O(\log n)$ rounds are sufficient w.h.p. to process all of the nodes. On each round, $O(P(n))$ depth is required for packing the new roots into an array, leading to a total of $O(P(n) \log n)$ depth w.h.p. across all rounds. $P(n)$ is $O(\log n)$ if using prefix sums on the EREW PRAM, $O(\log^* n)$ w.h.p. if using approximate compaction on the CRCW PRAM, and $O(1)$ on the scan PRAM. The work spent on each node is constant, since its counter is decremented a constant number of times. The work for packing is linear in the number of nodes. Thus the total work is $O(n)$.

5.3 Tree Contraction. With a pre-processing phase, we can label each internal node with the highest-indexed leaf in its sub-tree using a parallel leaffix operation (with the minimum operator) in $O(n)$ work and $O(\log n)$ depth. Then each internal node stores the smaller of the two computed labels of its children. Since the minimum operator does not have an inverse, we must do this with tree contraction. Note that, however, minimum is associative, so the result of this pre-processing phase would be consistent with any tree contraction algorithm. After pre-processing, we can run the parallel algorithms described in this section with any operator (does not have to be associative), and get the same answer as the sequential algorithm (Algorithm 5). With the internal nodes labeled, the ***neighborhood*** of a leaf is defined as the leaves labeled on its parent and its grandparent nodes. Only if the labels on these two internal nodes are greater than or equal to the leaf's ID can it proceed in raking.

Deterministic reservations-based implementation. Figure 9 defines the RESERVE and COMMIT functions and associated data required for deterministic reservations. $N(i)$ corresponds to the neighborhood of step i, which includes the leaf labeled on its parent (if it has one) and the leaf labeled on its grandparent (if it has one). These functions preserve the ordering imposed by the iteration dependence graph of T (defined in Section 3.3) throughout its execution because if the i^{th} leaf is spliced out, the RESERVE step guarantees that if $R[i]$ is set to 1, and guarantees that there are no lower-indexed leaves in the neighborhood of step i (i.e. step i has no children in the dependence forest). Only in this case does step i rake itself out in the COMMIT step

```
1: R = {0, ..., 0}                                   ▷ boolean array
2: procedure RESERVE(i)
3:     if i < j, ∀j ∈ N(i) then
4:         R[i] = 1                                  ▷ reserve own location
5:     return 1
6: procedure COMMIT(i)
7:     if (R[i] = 1) then
8:         p = T[i].parent
9:         if T[p].parent ≠ null then                ▷ p is not root
10:            s = sibling(T, i)
11:            T[s].parent = T[p].parent
12:            switchParentsChild(T, p, s)
13:        else                                      ▷ p is root
14:            switchParentsChild(T, i, null)
15:            return 0
16:    else return 1
```

Figure 9: RESERVE and COMMIT functions and associated data for tree contraction. *sibling*(T, i) returns the sibling of i in T, and *switchParentsChild*(T, i, v) resets the appropriate child pointer of the parent of i to point to v instead of i.

(the procedure for raking is the same as in the sequential algorithm shown in Algorithm 5).

Again, the steps can be efficiently checked and updated because the RESERVE and COMMIT functions take constant work/depth. By applying Theorem 3.4 and 4.1, we obtain the following theorem for tree contraction. Again, we can use the EREW PRAM because reads and writes of the neighbors inside the RESERVE and COMMIT steps can be separated into a constant number of phases such that there are no reads or writes to the same location in a phase.

THEOREM 5.5. *For a random ordering of T, deterministic reservations using the* RESERVE *and* COMMIT *functions for tree contraction runs in $O(n)$ expected work and $O(\log^2 n)$ depth w.h.p. on the EREW PRAM, $O(\log n \log^* n)$ depth w.h.p. on the CRCW PRAM, or $O(\log n)$ depth on the scan PRAM.*

The tree contraction used for pre-processing can be done deterministically in linear work and $O(\log n)$ depth on the EREW PRAM, which is within the stated complexity bounds of Theorem 5.5.

Activation-based implementation.

THEOREM 5.6. *An activation-based implementation of Algorithm 5 runs in $O(n)$ work and $O(\log^2 n)$ depth w.h.p. on the EREW PRAM, $O(\log n \log^* n)$ depth w.h.p. on the CRCW PRAM, or $O(\log n)$ depth on the scan PRAM.*

Proof. The activation-based implementation of list contraction described in Theorem 5.4 can be adapted for tree contraction. The "roots" are the steps with no lower labels on its parent and grandparent, which implies that it has no lower-indexed steps in its neighborhood. A root that is successfully processed potentially updates the counters of the steps in its neighborhood. The counter of each step is initialized to the number of lower-indexed steps that are in its neighborhood. Overall this takes linear work and constant depth. This algorithm satisfies the dependencies of the iteration dependence graph defined in Section 3.3 because the roots are the steps that have no more dependencies. Again, the reads and updates are split into a constant number of phases to avoid concurrency. Since the iteration depth is $O(\log n)$ w.h.p. by Theorem 3.4, and each round of the algorithm reduces the iteration depth of the remaining dependence graph by 1, $O(\log n)$ rounds are required w.h.p. Therefore, we have a depth of $O(P(n) \log n)$ w.h.p., where $P(n)$ is $O(\log n)$ on the EREW PRAM, $O(\log^* n)$ w.h.p. on the CRCW PRAM and $O(1)$ on the scan PRAM. The work is linear because each step is processed a constant number of times. ■

6 Limited Randomness

The parallel algorithms that we described in Section 5 use $O(\log n)$ random bits per input element, thus requiring $O(n \log n)$ bits of randomness in total. In this section, we describe how to reduce the amount of randomness to a polylogarithmic number of random bits while preserving the iteration dependence depth for random permutation and list contraction.

To show that limited randomness suffices, we use Nisan's [31] pseudorandom generator for space-bounded computation, which uses $O(S \log n)$ truly random bits to generate pseudorandom bits that are capable of fooling an S-space machine. More accurately, the probability of failure event given the generated stream of pseudorandom bits differs by at most (an additive) ϵ from the failure probability given truly random bits, where the *bias* ϵ can be driven down to $O(1/n^c)$ for any constant c by increasing the number of truly random bits by a constant factor. Thus, a result that holds with high probability using truly random bits also holds with high probability using the pseudorandom bits, provided that the failure event can be tested by an S-space machine.

For our purposes, it suffices to show that a space-S computation can verify the iteration depth of the dependence graph. As long as the low-space computation uses the same mapping from random bits to steps, the actual computation would have the same dependence graph. The challenge in designing these low-space verifiers and applying Nisan's theorem is that the verifier must consume the random bits as a one-pass stream of bits. By exhibiting such $O(\log n)$-space and $O(\log^2 n)$-space verifiers for the iteration depths of random permutation and list contraction, respectively, we prove that $O(\log^2 n)$ random bits suffice for random permutation and $O(\log^3 n)$ random bits suffice for list contraction.

THEOREM 6.1. *Using Nisan's generator with a seed of $O(\log^2 n)$ random bits, the iteration depth of the*

dependence graph for random permutation is $O(\log n)$ w.h.p.

Proof. Consider a single step i. Theorem 3.2 states that if each step chooses uniformly random numbers, then for any constant c the probability of step i exceeding depth $O(c \log n)$ is $O(1/n^c)$. Assuming we can verify the depth bound for step i in $O(\log n)$ space, Nisan's theorem states that the probability of exceeding the depth bound using the generated pseudorandom bits is at most $O(1/n^c) + \epsilon = O(1/n^c)$. Taking a union bound over all steps, the probability of choosing a seed that causes any step to have high depth is $O(1/n^{c-1})$.

The following is an $O(\log n)$-space procedure for calculating the depth of step i, using a single pass through the stream of random bits. Scan from step i down to step $H[i]$ in the input array, counting the number of intervening steps k such that $H[k] = H[i]$. These steps form a chain in the dependence forest directed from i to $H[i]$. Repeat this process starting from $i' = H[i]$ down to $H[i']$, until reaching the root of this tree (i.e., the starting node i' has $H[i'] = i'$). The sum of the lengths is equal to the depth of i in the dependence forest. This process requires $O(\log n)$ space to maintain a few pointers and the sum.

One additional detail is that the permutation algorithm expects random values in the range $[0, \ldots, i]$, but what we have access to is a stream of (pseudo)random bits. Without loss of generality, assume n is a power of 2. To generate a number in the range $[0, \ldots, i]$, for any constant c first generate a number x in the range $[0, \ldots, n^c - 1]$. For values $x < (i+1)\lfloor n^c/(i+1) \rfloor$, use $H[i] = x/(\lfloor n^c/(i+1) \rfloor)$. If any larger value is generated, the algorithm fails. The probability of failure for a particular value is at most $n/n^c = 1/n^{c-1}$, and using a union bound over all values, the failure probability becomes $O(1/n^{c-2})$. ■

We note that the random permutation produced using limited randomness is not truly random.

For list contraction, we assume that each node is assigned a random number, which we call a ***priority***, from the random bits of Nisan's generator. The random ordering of the list L can be viewed as the ordering in which the priorities are sorted in increasing order. By choosing random numbers from the range $[0, \ldots, n^c - 1]$ for constant $c > 1$, the priorities are distinct w.h.p. and Theorem 3.3 applies.

THEOREM 6.2. *Using Nisan's generator with a seed of $O(\log^3 n)$ random bits to assign each node a (pseudo)random priority, the iteration depth of the dependence graph for list contraction is $O(\log n)$ w.h.p.*

Proof. As in Theorem 6.1, we will exhibit an algorithm that can verify the depth of a node/step in the dependence tree using a single pass through the random priorities. Since the probability of the depth bound being exceeded is polynomially small, a union bound over all steps completes the proof.

To verify the depth of node x in the dependence forest, the verifier simulates the incremental insertion of nodes, in input order, into the dependence forest. After each step, the structure of the dependence tree containing x is identical to a treap using the same priorities and node comparisons respecting list-order. We begin the simulation by inserting the node x, assuming pessimistically that it has minimum priority (which only increases its depth). Throughout the process, we maintain the root-to-leaf path down to x. When inserting a new node z, the idea is to simulate the treap insertion process with respect to the path down to x. To insert z, step down the path until finding the first (highest) node y such that either x and z are in different subtrees of y, or $y = x$. If z has lower priority than y, then the path to x is unchanged. Otherwise, splice in z to be the parent of y, and repeatedly rotate z and its parent until z has lower priority than its parent. This rotation process may result in the path shortening and/or the ancestors being rearranged, depending on the list-order comparisons among nodes.

List-order comparisons can be performed in $O(\log n)$ space using a constant number of pointers and traversing the list. As long as the depth of a node never exceeds $O(\log n)$, then the space used by the simulation is $O(\log^2 n)$. If the depth ever exceeds $O(\log n)$, then the simulation stops and reports a high-depth node. By Theorem 3.3, this is a low probability event. ■

We now discuss the work and depth required to generate the random numbers from Nisan's pseudorandom generator. The generator uses $O(\log n)$ independent hash functions $h_1, \ldots, h_S$, each requiring $O(S)$ random bits, and a seed x with $O(S)$ random bits [31]. Define $G_0(x) = x$ and $G_t(x) = (G_{t-1}(x), h_t(G_{t-1}(x)))$ for $t \geq 1$. The output of the generator is $G_{t'}(x)$, where $t' = O(\log(n \log n/S))$, which has $O(n \log n)$ bits.

LEMMA 6.1. *The output of Nisan's pseudorandom generator can be computed in $O(nS/\log n)$ work and $O(\log n \log(1 + S/\log n))$ depth.*

Proof. We construct $G_{t'}(x)$ recursively using the definition above. Level t of the recursion requires $O(2^t(S/\log n)^2)$ work and $O(\log(1 + S/\log n))$ depth, as the hash functions can be evaluated in $O((S/\log n)^2)$ work and $O(\log(1 + S/\log n))$ depth using naive multiplication (we can evaluate $O(\log n)$ bits with one unit of work). To generate $O(n \log n)$ pseudorandom bits, we perform $O(\log(n \log n/S))$ levels of recursion. The total work is $\sum_{t=0}^{\log(n \log n/S)} O(2^t(S/\log n)^2) = O(nS/\log n)$ and depth is $O(\log n \log(1 + S/\log n))$. ■

By plugging in the space bounds for random permutation and list contraction into Lemma 6.1, we obtain the following corollary.

COROLLARY 6.1. *The random bits of Nisan's pseudo-random generator for our random permutation and list contraction algorithms can be computed in $O(n)$ work and $O(\log n)$ depth, and $O(n \log n)$ work and $O(\log n \log\log n)$ depth, respectively.*

7 Experiments

We implement the deterministic parallel iterative algorithms for random permutation, list contraction and tree contraction. For tree contraction, we use a version that does not do a pre-processing step, and each leaf simply checks its nearby leaves to see if there are any conflicts. This version is described in the Appendix, and while it does not return the same answer as the sequential algorithm (but is still deterministic), it is more efficient as it does not require a pre-processing step. All of our parallel implementations use the prefix-based version of deterministic reservations [6], which performs better in practice than the version which processes all remaining steps in each round. In the Appendix, we prove complexity bounds for these versions. In our implementations, each prefix is processed once, and the unsuccessful steps are moved to the next prefix. For random permutation, we chose a prefix size of $n_i/50$ where n_i is the number of remaining steps. For list contraction we chose a fixed prefix size of $n/100$ and for tree contraction we chose a fixed prefix size of $n/50$. These were experimentally determined to give the best performance. Our implementations are all very simple—the random permutation and list contraction implementations use under a dozen lines of `C` code and the tree contraction implementation uses a few dozen lines. For comparison, we also implement the sequential iterative algorithms.

We run our experiments on a 40-core (with two-way hyper-threading) machine with 4×2.4GHz Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. We ran all parallel experiments with two-way hyper-threading enabled, for a total of 80 threads. We compiled all of our code with `g++` version 4.8.0 with the `-O2` flag. The parallel codes use Cilk Plus [27] to express parallelism, which is supported by the `g++` compiler that we use. The writeMax operation used in random permutation is implemented using a compare-and-swap loop [38]. We obtain randomness via hash functions, although more sophisticated random number generators could be used.

The number of elements for random permutation, number of nodes for list contraction, and number of leaves for tree contraction is 10^9. For random permutation, the data array A stores 32-bit integers and we randomly generated the swap targets (the H array). For list contraction, to generate the input, we first generated a random permutation, giving us a collection of cycles on the nodes, and then deleted one edge on each cycle, giving us a collection of linked lists. For tree contraction our input was a random binary tree with 10^9 randomly-indexed leaves, giving us a total of $2 \times 10^9 - 1$ nodes. Often, list and tree contraction are used as a part of a larger algorithm, so the pre-processing step of randomly permuting the elements only needs to be applied once. In our experiments, we do not store values on the nodes for list contraction and tree contraction. A summary of the timings for each of the three algorithms are shown in Table 1. The times that we report are based on a median of three trials.

Algorithm	(1)	(40h)	(seq)
Random permutation	92.1	4.62	38.8
List contraction	160	3.97	46
Tree contraction	350	10.0	172

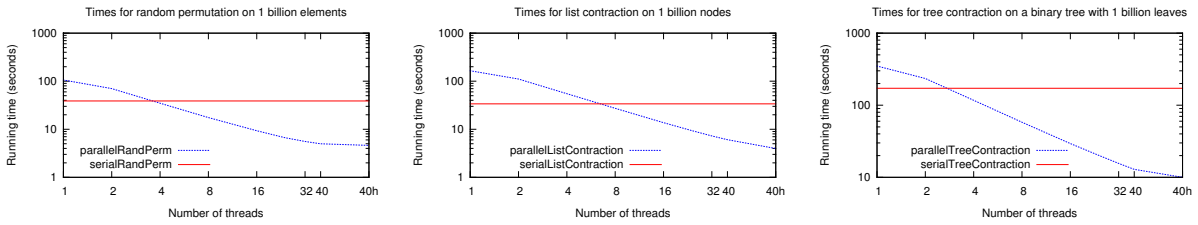
Table 1: Times (seconds) for $n = 10^9$ on 40 cores with hyper-threading. (1) indicates 1 thread, (40h) indicates 80 hyper-threads, and (seq) is the sequential iterative implementation.

Plots of running time versus number of threads in log-log scale for each of the three algorithms are shown in Figure 10. We see that the parallel implementations all get good speedup, and outperform the corresponding sequential implementation with a modest number of threads.

For random permutation, the parallel implementation outperforms the standard simple sequential implementation [26] with 4 or more threads. We also compared it to a sorting-based random permutation algorithm that we implemented, where we create pairs $(A[i], r_i)$ where each r_i is a random number drawn from $[1, \ldots, n^2]$, and the key to sort on is the second value of the pair. Note that this does not give the same permutation as the sequential algorithm. We used a parallel sample sort, which is part of the Problem Based Benchmark Suite [39]. On 80 hyper-threads the sorting-based algorithm took 5.38 seconds, and on a single thread it took 204 seconds. Both of these timings are inferior to the random permutation algorithm implemented with deterministic reservations.

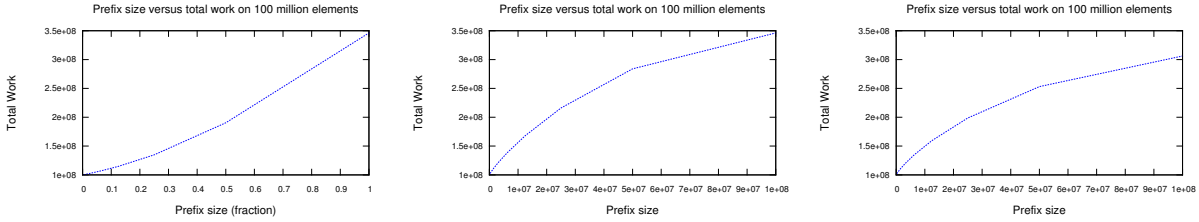
We note that an experimental study of other parallel random permutation algorithms has recently been conducted by Cong and Bader [9], which compares algorithms based on sorting [33], dart-throwing [28, 14, 16] and an adaptation of Sander's distributed algorithm [37]. None of these algorithms generate the same permutation as the sequential algorithm. It is difficult to directly compare with their reported numbers because their numbers include the cost for generating random numbers, while our numbers do not, their input sizes are much smaller (the largest size was 20 million elements), and the machine specifications are different.

For list contraction, the parallel implementation out-



(a) random permutation (b) list contraction (c) tree contraction

Figure 10: Running time vs. number of threads for $n = 10^9$ on 40 cores with hyper-threading (log-log scale). (40h) indicates 80 hyper-threads.



(a) random permutation (b) list contraction (c) tree contraction

Figure 11: Total work vs. prefix size for $n = 10^8$.

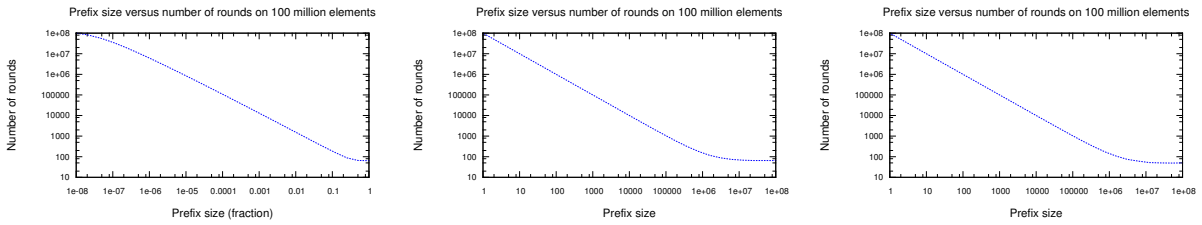
performs the serial implementation with 8 or more threads. We also implemented a parallel version of list contraction where the random numbers are regenerated in each round. In this strawman implementation, we cannot directly apply the prefix processing idea because the priorities of the nodes are not fixed. Therefore all remaining nodes are processed in each iteration. On 80 hyper-threads, the implementation took 6.46 seconds to finish. This is slower than our parallel implementation, which takes 3.97 seconds on the same input. The reason is that there is more wasted work in processing all of the nodes on each iteration, and also an added cost of regenerating random numbers on each iteration. In addition, this implementation does not return the same answer as the sequential implementation.

List ranking algorithms have been studied experimentally in the literature [35, 40, 32, 11, 22, 23, 3, 34]. None of these implementations return the same answer as a sequential ordering of processing the nodes would. The most recent experimental work on list ranking for multi-cores is by Bader et al. [3]. However since they used a much older machine, and they are solving list ranking instead of list contraction, it is hard to compare.

Finally, for tree contraction the parallel implementation outperforms the sequential implementation with 4 or more threads. Again, we compare it with a parallel strawman version that processes all remaining leaves and regenerates the random numbers on each iteration. On 80 hyper-threads this implementation took 23.3 seconds, compared to 10 seconds for our parallel implementation. As in list contraction, this is due to the wasted work of processing all leaves on each iteration and the added cost of regenerating the random numbers.

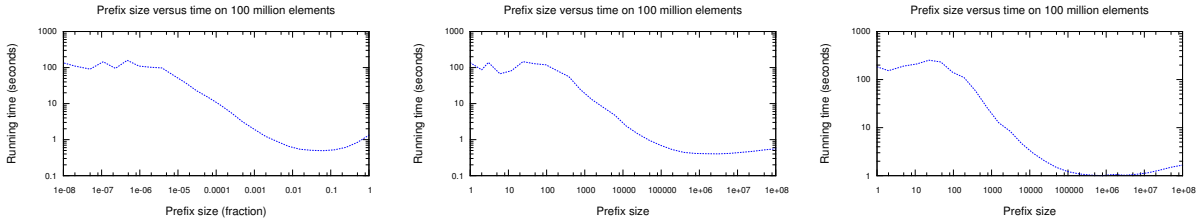
The most recent experimental work on tree contraction on multi-cores is by Bader et al. [4]. They present an implementation of tree contraction based on the standard algorithm that only rakes leaves [24]. The algorithm is more complicated than ours as it involves using Euler tours and list ranking to label the leaves to allow non-conflicting leaves to be raked in parallel. Furthermore, it does not return the same answer as a sequential algorithm. Again, because they use a much older machine and they solve the more expensive arithmetic expression computation, it is hard to compare.

In Figure 11, we plot the total work performed by the three algorithms as a function of the prefix size for $n = 10^8$. Since the prefix size is a constant fraction for random permutation, in the plots, the x-axis shows the fraction used. For list contraction and tree contraction, the prefix size is fixed across rounds, so the x-axis shows the actual size of the prefix. We see that the work goes up as we increase the prefix size as there is more wasted work due to failed steps. Note that a prefix size of 1 corresponds to the work performed by the sequential algorithm. In Figure 12, we plot the number of rounds of deterministic reservations as a function of prefix size in log-log scale. We see the opposite effect here—a larger prefix size leads to fewer rounds because there is more parallelism. These plots show the trade-off between work and parallelism. Finally, in Figure 13 we plot the parallel running time as a function of the prefix size in log-log scale. We see that the best running time uses a prefix size somewhere in between 1 and n.



(a) random permutation (b) list contraction (c) tree contraction

Figure 12: Number of rounds vs. prefix size for $n = 10^8$ (log-log scale).



(a) random permutation (b) list contraction (c) tree contraction

Figure 13: Running time vs. prefix size for $n = 10^8$ on 40 cores with hyper-threading (log-log scale).

8 Conclusion

We have shown that simple "sequential" iterative algorithms for random permutation, list contraction and tree contraction are surprisingly parallel. We prove that the iteration dependence depth for these problems is logarithmic with high probability, and describe linear-work polylogarithmic-depth parallel algorithms for solving these problems. For random permutation and list contraction, we show that the iteration depth bounds are maintained with high probability even when using only a polylogarithmic number of random bits. Using limited randomness in tree contraction is left for future work. We show experimentally that our implementations for the three problems get good speedup and outperform the sequential implementations with a modest number of cores. The simplicity, practical efficiency, determinism, and theoretical guarantees of our algorithms make them very attractive.

Acknowledgements

This work is supported by the National Science Foundation under grant numbers CCF-1314590 and CCF-1314633, the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and a Facebook Graduate Fellowship. We thank Bradley Kuszmaul for helpful discussions.

References

[1] L. Alonso and R. Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.

[2] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Symposium on Parallelism in Algorithms and Architectures*, pages 95–102, 1990.

[3] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *International Conference on Parallel Processing*, pages 547–556, 2005.

[4] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In *High Performance Computing*, pages 63–75. Springer, 2002.

[5] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11):1526–1538, 1989.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[7] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Symposium on Parallelism in Algorithms and Architectures*, pages 308–317, 2012.

[8] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[9] G. Cong and D. A. Bader. An empirical analysis of parallel random permutation algorithms on SMPs. In *Parallel and Distributed Computing and Systems*, pages 27–34, 2005.

[10] A. Czumaj, P. Kanarek, M. Kutylowski, and K. Lorys. Fast generation of random permutations via networks simulation. *Algorithmica*, pages 2–20, 1998.

[11] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 25(1):1–16, 1997.

[12] L. Devroye. A note on the height of binary search trees. *J.*

ACM, 33(3):489–498, 1986.

[13] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964.

[14] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, 1996.

[15] P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28(2):3–29, 1999.

[16] J. Gil. Fast load balancing on a PRAM. In *Symposium on Parallel and Distributed Processing*, pages 10–17, 1991.

[17] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Foundations of Computer Science*, pages 698–710, 1991.

[18] J. Gustedt. Randomized permutations in a coarse grained parallel environment. In *Symposium on Parallelism in Algorithms and Architectures*, pages 248–249, 2003.

[19] J. Gustedt. Engineering parallel in-place random generation of integer permutations. In *Workshop on Experimental Algorithmics*, pages 129–141, 2008.

[20] T. Hagerup. Fast parallel generation of random permutations. In *International Colloquium on Automata, Languages and Programming*, pages 405–416. Springer, 1991.

[21] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Symposium on Parallelism in Algorithms and Architectures*, pages 166–177, 2014.

[22] D. Helman and J. JaJa. Designing practical efficient algorithms for symmetric multiprocessors. *Algorithm Engineering and Experimentation*, pages 37–56, 1999.

[23] D. Helman and J. JaJa. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[24] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[25] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990.

[26] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[27] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[28] G. Miller and J. Reif. Parallel tree contraction and its application. In *Foundations of Computer Science*, pages 478–489, 1985.

[29] G. Miller and J. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[30] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.

[32] J. Patel, A. Khokhar, and L. Jamieson. Scalable parallel implementations of list ranking on fine-grained machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 1006–1018, 1997.

[33] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[34] M. S. Rehman, K. Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the GPU. In *International Conference on Supercomputing*, pages 235–243, 2009.

[35] M. Reid-Miller. List ranking and list scan on the CRAY C90. *J. Comput. Syst. Sci.*, 53(3):344–356, 1996.

[36] J. H. Reif. *Synthesis of Paralell Algorithms*. Morgan Kaufmann, 1993.

[37] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Inf. Process. Lett.*, 67(6):305–309, 1998.

[38] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Symposium on Parallelism in Algorithms and Architectures*, pages 152–163, 2013.

[39] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *Symposium on Parallelism in Algorithms and Architectures*, pages 68–70, 2012.

[40] J. F. Sibeyn. Better trade-offs for parallel list ranking. In *Symposium on Parallelism in Algorithms and Architectures*, pages 221–230, 1997.

A Appendix

A.1 Prefix-based deterministic reservations for random permutation. For our experiments, we use the prefix-based version of deterministic reservations. The idea is that selecting a small enough prefix ensures that the amount of wasted work is small in practice. We analyze a version of the algorithm which processes each prefix until completion before moving on to the next prefix. This differs from our implementation, but our implementation only process more steps per round, so can only have lower depth. If we select a constant fraction (e.g., half) of the remaining elements as the prefix, then there are $O(\log n)$ iterations. Processing each prefix requires $O(\log n \log^* n)$ depth by Theorem 5.1 (the dependence forest of the prefix can only be shallower than the dependence forest of all the steps). Hence the total depth is $O(\log^2 n \log^* n)$ w.h.p. Since processing all steps per round gives expected linear-work by Theorem 5.1, and processing a prefix only decreases the amount of wasted work (work spent on processing failed steps), the work is still $O(n)$ in expectation.

A.2 Prefix-based deterministic reservations for list contraction. Similar to the analysis for random permutation in Section A.1, a prefix-based version of list contraction with a prefix consisting of a constant fraction of the remaining steps leads to an algorithm with $O(n)$ expected work and polylogarithmic depth by applying Theorem 5.3.

A.3 Variant of tree contraction algorithm. Here we analyze the simpler version of tree contraction that does not require a pre-processing phase. It does not return

the same answer as the sequential algorithm, but is still deterministic.

The deterministic reservations-based algorithm still uses the functions shown in Figure 9, but with a modified definition of the neighborhood $N(i)$. Define the ***region*** of leaf i to contain i's parent (if it exists) and i's grandparent (if it exists). Define $N(i)$ to contain the leaves j, $j \neq i$, such that the intersection of i's region and j's region is non-empty. The dependence forest for a tree T is the tree on the n leaf nodes such that leaf i has a directed edge to leaf j if $i < j$ and j is in i's neighborhood.

LEMMA A.1. *For any i, $N(i) \leq 4$.*

Proof. There can be at most 3 other regions that contain i's grandparent and at most 2 other regions that contain i's parent. If the number of other regions that contain i's grandparent is exactly 3 then the 4 grandchildren of i's grandparent are all leaves and $N(i) = 3$. In all other cases, there are at most 2 other regions containing i's grandparent and at most 2 other regions containing i's parent, so $N(i) \leq 4$. ■

We now show that for a random ordering of the leaves of T, the algorithm does not have a very long chain of dependencies.

THEOREM A.1. *For a random ordering of the leaves of T, the parallel implementation of greedy tree contraction requires $O(\log^2 n)$ rounds to terminate w.h.p.*

Proof. We analyze the prefix-based version of the algorithm, which can only be slower than the fully parallel version. Consider the dependence forest induced by the δn lowest indexed leaves, where δ is the fraction of the remaining elements to take as the prefix. If there processing the prefix takes k rounds, then there must be a k-length (undirected) path in the dependence forest. The probability of a k-length path is δ^k. By Lemma A.1, each node has at most 4 neighbors so the maximum number of k-length paths starting at any node in the dependence forest is 4^k. By a union bound over all nodes, the probability of a k-length path in the prefix is at most $n \cdot 4^k \cdot \delta^k$. For $\delta = 1/8$ and $k = 2 \log n$, this gives a high probability bound. Since δ is a constant fraction, the number of prefixes is $O(\log n)$, giving $O(\log^2 n)$ rounds w.h.p. overall. ■

A straightforward implementation of this algorithm using deterministic reservations has $O(n \log^2 n)$ work and $O(\log^2 n)$ depth w.h.p. The algorithm can also be implemented in linear work as the following theorem shows.

THEOREM A.2. *For a random ordering of the leaves of T, prefix-based deterministic reservations using the* RESERVE *and* COMMIT *functions for tree contraction runs in $O(n)$ expected work and $O(\log^3 n)$ depth w.h.p. on the EREW PRAM.*

Proof. Define a ***maximal path*** in the dependence forest to be a path whose length cannot be extended following either forward edges or backward edges. Since each iteration processes all nodes in each maximal path of the dependence forest, we need to show that the size of each maximal path in the dependence forest is constant. Let S_i be the expected size of the i^{th} maximal path. The work for each maximal path is proportional the square of its length, as each iteration processes all nodes and removes one node. For a prefix of size δn, the probability of a k-length path in the prefix starting at any node is at most δ^k and number of paths is at most 4^k. Therefore, we have $E[S_i^2] = \sum_{k=1}^{\delta n} k^2 \cdot 4^k \cdot \delta^k$. For $\delta = 1/8$, $E[S_i^2] = O(1)$. Note that our analysis is loose since we over-count the work for maximal paths that intersect.

We pack out successful steps after each round, so that the expected amount of work spent processing a prefix is linear in its size, giving us overall expected $O(n)$ work. The pack requires $O(\log n)$ depth per round on the EREW PRAM, and by Theorem A.1, there are $O(\log^2 n)$ rounds w.h.p., so the overall depth is $O(\log^3 n)$ w.h.p. The reads and writes are split into a constant number of phases to avoid concurrency. ■

The version we use in our experiments is slightly different, but it only processes more steps per round so the depth bound still applies. We do not analyze its work bound here.