Parallel In-Place Algorithms: Theory and Practice*

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Omar Obeya
MIT CSAIL
obeya@csail.mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

Many parallel algorithms use at least linear auxiliary space in the size of the input to enable computations to be done independently without conflicts. Unfortunately, this extra space can be prohibitive for memory-limited machines, preventing large inputs from being processed. Therefore, it is desirable to design parallel in-place algorithms that use sublinear (or even polylogarithmic) auxiliary space.

In this paper, we bridge the gap between theory and practice for parallel in-place (PIP) algorithms. We first define two computational models based on fork-join parallelism, which reflect modern parallel programming environments. We then introduce a variety of new parallel in-place algorithms that are simple and efficient, both in theory and in practice. Our algorithmic highlight is the Decomposable Property introduced in this paper, which enables existing non-in-place but highly-optimized parallel algorithms to be converted into parallel in-place algorithms. Using this property, we obtain algorithms for random permutation, list contraction, tree contraction, and merging that take linear work, $O(n^{1-\epsilon})$ auxiliary space, and $O(n^\epsilon \cdot \text{polylog}(n))$ span for $0 < \epsilon < 1$. We also present new parallel in-place algorithms for scan, filter, merge, connectivity, biconnectivity, and minimum spanning forest using other techniques.

In addition to theoretical results, we present experimental results for implementations of many of our parallel in-place algorithms. We show that on a 72-core machine with two-way hyper-threading, the parallel in-place algorithms usually outperform existing parallel algorithms for the same problems that use linear auxiliary space, indicating that the theory developed in this paper indeed leads to practical benefits in terms of both space usage and running time.

1 Introduction

Due to the rise of multicore machines with tens to hundreds of cores and terabytes of memory, and the availability of programming languages and tools that simplify shared-memory parallel computing, many parallel algorithms have been designed for large-scale data processing. Compared to distributed or external-memory solutions, one of the biggest challenge for using multicores for large-scale data processing is the limited memory capacity of a single machine. Traditionally, parallel algorithm design has mostly focused on solutions with low work (number of operations) and span (depth or longest critical path) complexities. However, to enable data to be processed in parallel without conflicts, many existing parallel algorithms are not in-place, in that they require $\Omega(n)$ auxiliary memory for an input of size n. For example, in the shuffling step of distribution sort (sample sort) or radix sort algorithms, even if we know the destination of each element in the final sorted array, it is difficult to directly move all of them to their final locations in parallel in the same input array due to conflicts. As a result, parallel algorithms for this task (e.g., [17, 19]) use an auxiliary array of linear size to copy the elements into their correct final locations.

While many parallel multicore algorithms are work-efficient and have low span, the $\Omega(n)$ auxiliary memory required by the algorithms can prevent larger inputs from being processed. Purchasing or renting machines multicore machines with larger memory capacities is an option, but for large enough machines, the cost increases roughly linearly with the memory capacity, as shown in Figure 1. Furthermore, additional energy costs need to be paid for machines that are owned, and the energy cost increases proportionally with the memory capacity. Therefore, designing *parallel in-place (PIP) algorithms*, which use auxiliary space that is sublinear (or even polylogarithmic) in the input size, can lead to considerable savings. In addition, in-place algorithms can also reduce the number of cache misses and page faults due to their lower memory footprint, which in turn can improve overall performance, especially in parallel algorithms where memory bandwidth and/or latency is a scalability bottleneck.

There has been recent work studying theoretically-efficient and practical parallel in-place algorithms for sample sort [3], radix sort [36], partition [33], and constructing implicit search tree layouts [6]. These PIP algorithms achieve better performance than previous algorithms in almost all cases. While these algorithms are insightful and motivate the PIP setting, they are algorithms designed for specific problems and have different notions of what "in-place" means in the parallel setting. In this paper, we generalize the ideas

---

*This is the conference version of the paper appearing in the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS), 2021. The full version of this paper can be found on arXiv.
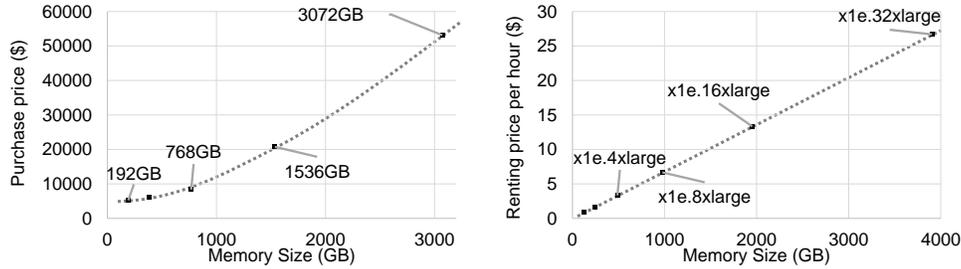
Figure 1: Purchase and rental prices for multicore servers as a function of memory size. The left figure shows the purchase price of an RAX XT24-42S1 server with 72 CPU cores (Xeon Gold 5220) for different DRAM sizes. The right figure shows the rental price of AWS EC2 x1e-series multicore instances vs. the DRAM capacity. In both cases, the DRAM capacity is the dominant part of the overall cost.

Model	Problems	Work-efficient	
Strong PIP Model	Permuting tree layout	✓	[6]
	Reduce, rotating	✓	
	Scan (prefix sum)	✓	*
	Filter, partition, quicksort		
	Merging, mergesort		
	Set operations	✓	[11]
Relaxed PIP Model	Random permutation	✓	*
	List and tree contraction	✓	*
	Merging, mergesort	✓	*
	Filter, partition, quicksort	✓	
	(Bi)Connectivity		*
	Minimum spanning forest		*

Table 1: Algorithms based on the strong PIP model and the relaxed PIP model. "Work-efficient" indicates that the PIP algorithm has the same asymptotic work (number of operations) as the best sequential algorithm with no restriction on auxiliary space. Algorithms marked with * are our main contributions, while other algorithms are either known or require slight changes to existing algorithms. Merging and mergesort in the relaxed PIP model has been presented in [27], but our new algorithm in this paper is much simpler. If a problem has a work-efficient solution in the strong PIP model, then it will not be listed again in the relaxed PIP model.

in previous work into two models, which we refer to as the **strong PIP model** and the **relaxed PIP model**. At a high level, the relaxed PIP model provides similar properties to the classic in-place PRAM model, and the strong PIP model puts further restrictions on memory allocation that allows PIP algorithms to simultaneously achieve small auxiliary space and low span. We provide more details on these models in Section 3.

The main contribution of this paper is a collection of new PIP algorithms in the two models, which include algorithms for solving scan, merge, filter, partition, sorting, random permutation, list contraction, tree contraction, and several graph problems (connectivity, biconnectivity, and minimum spanning forest). The results are summarized in Table 1, and discussed in more detail in Sections 4–6. Some of the algorithms are known, and we summarize them in this paper. The rest are new to the best of our knowledge, and we distinguish them by presenting our results in theorems and corollaries.

The algorithmic highlight in this paper is the Decomposable Property defined in Section 4. The high-level idea is to iteratively reduce a problem to a subproblem of sufficiently smaller size, where the the reduction can be performed using a non-in-place algorithm for the same problem. If we can perform the reduction efficiently, then this leads to an efficient algorithm in the relaxed PIP model. This means that we can convert any existing non-in-place but highly-optimized parallel algorithm to an efficient PIP algorithm. We show many examples of this approach in this paper, including algorithms for random permutation, list contraction, tree contraction, merging, and mergesort. We have also designed other PIP algorithms without using the Decomposable Property, including algorithms for scan, filter, and various graph problems.

We implement five of our in-place algorithms, and compare them to the optimized non-in-place implementations in the Problem Based Benchmark Suite (PBBS) [43]. The running time comparisons for certain input sizes are shown in Figure 2 and we provide more details in Section 7. We show that in addition to lower space usage, our in-place algorithms can have competitive or even better performance compared to their non-in-place counterparts due to their smaller memory footprint, indicating that the theory for PIP algorithms developed in this paper can lead to practical outcomes. Our implementations are publicly-available at `https://github.com/ucrparlay/PIP-algorithms`.

2 Preliminaries

Work-Span Model. In this paper, we use the classic work-span model for fork-join parallelism with binary forking for analyzing parallel algorithms [16, 23]. Unlike machine-based cost models such as the PRAM model [29], this model is a language-based model, and we will justify the use of this model in Section 3. In this model, we assume that we have a set of threads that have access to a shared memory. Each thread supports the same operations as in the sequential RAM model, but also has a `fork` instruction that forks two new child threads. When a thread performs a `fork`, the two child threads all start by running the next instruction, and the original thread is suspended until all of its children terminate. A computation starts with a single root thread and finishes
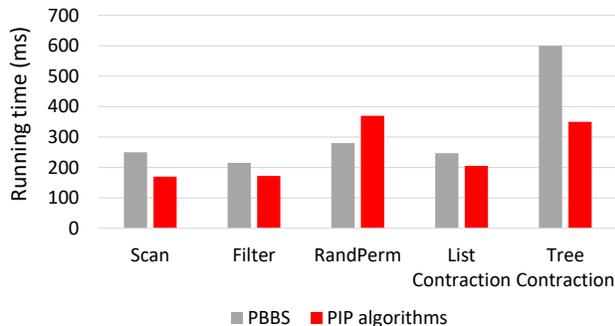
Figure 2: Running times for our new PIP algorithms compared to the non-in-place implementations from PBBS [43]. For scan and filter, the input size is 10^9, and for the other algorithms, the input size is 10^8. The running times are obtained on a 72-core machine with two-way hyper-threading, and more details are presented in Section 7. In all cases, the new PIP algorithms have competitive or better performance, with the additional advantage of using less auxiliary space.

when that root thread finishes. The **work** of an algorithm is the total number of instructions and the **span** (depth) is the length of the longest sequence of dependent instructions in the computation. A thread can allocate a fixed size of memory that is either shared by all threads (referred to as "heap-allocated" memory), or private to this thread and other threads that it forks (referred to as "stack-allocated" memory). The latter case requires freeing the memory allocated after a fork and before the corresponding join. A randomized work-stealing scheduler, which is widely used in real-world parallel languages such as Cilk, TBB, and X10, can execute an algorithm with work W and span D in $W/P + O(D)$ time *whp*[1] on P processors [2, 20].

In this paper, we also analyze the auxiliary space used, and we measure space in units of words.

Work-Efficient and Low-Span Parallel Algorithms. The goal of designing a parallel algorithm is to achieve work-efficiency and low span. Work-efficiency means that the algorithm asymptotically uses no more work than the best (sequential) algorithm for the problem. Low span means that the longest sequence of dependent operations has polylogarithmic length. Achieving low span can lead to practical benefits. For instance, the number of steals in a work-stealing scheduler can be bounded by $O(PD)$, which is proportional to the number of extra cache misses due to parallelism [1, 17]. Low span also leads to fewer rounds of global synchronization, which can lead to significant performance improvements on modern architectures.

The "busy-leaves" property. When using the Cilk work-stealing scheduler, a fork-join program that uses S_1 words of space in a stack-allocated fashion when run on one processor will use $O(PS_1)$ words of space when run on P

processors [20]. This is a consequence of the "busy-leaves" property of the work-stealing scheduler.

Problem definitions. Here we define the problems that are used in multiple places in this paper. Other problems are defined in their respective sections. Consider a sequence $[a_1, a_2, \ldots, a_n]$, an associative binary operator $\oplus$, and an identity element i. **Reduce** returns $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. **Scan** (short for an exclusive scan) returns $[i, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$, in addition to the sum of all elements. An inclusive scan returns $[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_n)]$. **Filter** takes an array A and a predicate function f, and returns a new array containing $a \in A$ for which $f(a)$ is true. **Partition** is similar to filter, but in addition to placing the elements a where $f(a)$ is true at the beginning of the array, elements a for which $f(a)$ is false are placed at the end of the array. We say that a filter or partition is **stable** if the elements in the output are in the same order as they appear in A.

3 Models for Parallel In-Place Algorithms

In the past, PIP algorithms have been designed based the *in-place PRAM model* [25, 26, 28, 34, 37, 46]. However, this model and the PRAM model itself have some limitations, which we describe in Section 3.3. Hence, recent work on PIP algorithms [3, 6, 33, 36] incorporate the PIP setting in the newer work-span model, although they use different notions of "in-place" for the algorithms. In this paper, we generalize the ideas into two models, which we refer to as the *strong PIP model* and the *relaxed PIP model*. At a high level, the relaxed PIP model provides similar properties as the in-place PRAM model, and the strong PIP model puts further restrictions on memory allocation that enables PIP algorithms to achieve small auxiliary space and low span simultaneously. Based on our model definitions, algorithms in [6, 33] can be mapped to the strong PIP model, and algorithms in [3, 36] can be mapped to the relaxed PIP model. In this section, we will first define these two models, and then discuss their relationship to existing PIP models.

3.1 The Strong PIP Model

We start by defining the strong PIP model based on the work-span model for fork-join parallelism.

DEFINITION 1. (STRONG PIP MODEL AND ALGORITHMS) *The **strong PIP model** assumes a fork-join computation only using $O(\log n)$-word auxiliary space in a stack-allocated fashion for an input size of n when run sequentially (with no auxiliary heap-allocated space). We say that an algorithm is **strong PIP** if it runs in the strong PIP model and has polylogarithmic span.*

For a PIP algorithm in the strong PIP model, the Cilk work-stealing scheduler can bound the total auxiliary

---

[1]We say $O(f(n))$ **with high probability (whp)** to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$, where n is the input size.

space to be $O(P \log n)$ words, where P is the number of processors [20]. All strong PIP algorithms presented in this paper, as well as existing ones [6, 33], only use $O(\log n)$-word stack-allocated auxiliary space sequentially. We say a strong PIP algorithm is **optimal** if its work and span bounds match the best non-in-place counterpart.

3.2 The Relaxed PIP Model

Many existing PIP algorithms [3, 25, 26, 28, 34, 36, 37, 46] exhibit a tradeoff between additional space S and span D, such that $S \cdot D = \tilde{\Theta}(n)$.[2] We capture these algorithms in our relaxed PIP model, and refer to these algorithms as relaxed PIP algorithms.

DEFINITION 2. (RELAXED PIP MODEL AND ALGORITHMS) *The **relaxed PIP model** assumes a fork-join computation using $O(\log n)$-word stack-allocated space sequentially and $O(n^{1-\epsilon})$ shared (heap-allocated) auxiliary space for an input of size n and some constant $0 < \epsilon < 1$. We say that an algorithm is **relaxed PIP** if it runs in the relaxed PIP model and has $O(n^\epsilon \cdot \text{polylog}(n))$ span for all values of ϵ.*

The Cilk work-stealing scheduler can bound the total auxiliary space of relaxed PIP algorithms to be $O(n^{1-\epsilon} + P \log n)$ on P processors. For brevity, we refer to the auxiliary space in future references to the relaxed PIP model as just the heap-allocated space. Algorithms in the relaxed PIP model allow sublinear auxiliary space, which is less restrictive than in the strong PIP model. This provides more flexibility in algorithm design, while still being useful in practice as relaxed PIP algorithms still use less space than their non-in-place counterparts. In the next section, we introduce a general property, which allows any existing parallel algorithm with polylogarithmic span that satisfies the property to be easily converted into a relaxed PIP algorithm.

3.3 Relationship to Previous Models

PIP algorithms have been analyzed in the in-place PRAM model for decades. Recent work [3, 6, 33, 36] has designed in-place algorithms into the work-span model, but they only provide algorithms for specific problems rather than focusing on the general parallel in-place setting. In this paper, we formally define the parallel in-place models, and justify the models by discussing the limitations of the previous in-place PRAM, and how our new models overcome it.

The in-place PRAM. Most existing parallel in-place algorithms have been designed in the in-place PRAM [25, 26, 28, 34, 37, 46]. The PRAM has P processors that are fully synchronized between steps, and the running time of an algorithm is the maximum number of steps T used by any processor. In this model, the auxiliary space S is the sum of the total space used across all processors. As pointed out by Berney et al. [6], each processor on a PRAM requires

---

[2]We use $\tilde{O}(f(n))$ to hide polylogarithmic factors.

$\Omega(1)$ (usually $\Omega(\log n)$) auxiliary space to do anything useful (e.g., storing the program counter and using registers). This indicates that if the total auxiliary space S for all processors is bounded to be small, then the parallelism is also bounded by $O(S)$. This is because even if we have an infinite number of processors, no more than S of them can do useful work simultaneously. The overall PRAM time is $\Omega(W/S)$, where W is the overall work in the algorithm. Hence, in the PRAM setting, an algorithm can only achieve high parallelism when S is *asymptotically close* to W. This has been described by Langston et al. [26, 28, 34, 46] as the time-space tradeoff in the PRAM—if the input size is n, then the product of auxiliary space S and PRAM time T is $\tilde{\Omega}(n)$, and an algorithm is optimal on a PRAM when $S \cdot T = \tilde{\Theta}(n)$. This limitation arises because the analysis of parallelism and auxiliary space are intertwined in the in-place PRAM.

Decoupling the analysis between parallelism and auxiliary space. Parallel algorithms with low span have many practical benefits even for small processor counts, due to lower scheduling overhead and improved cache locality, as discussed in Section 2. However, low span cannot be achieved in the in-place PRAM unless we use nearly linear auxiliary space. Our goal is to decouple the analysis of parallelism from the restriction of auxiliary space. In both the strong PIP and relaxed PIP models, the auxiliary space in measured in the sequential setting, whereas the span is analyzed based on the fork-join computation graph. This decouples the space analysis from the span analysis. Furthermore, in the strong PIP model, low span and small auxiliary space can be achieved simultaneously.

To achieve the decoupling, we use the separation of the private "stack-allocated" memory from the shared "heap-allocated" memory in work-span model. The heap-allocated memory is what we usually refer to as the shared memory, and is independent of the number of processors. The stack-allocated memory is per processor, and the "busy-leaves" property guarantees that the overall space usage of a program is $O(PS_1)$ when it is run on P processors, where S_1 is the amount of stack-allocated memory when running the algorithm sequentially. Since P is usually modest in practice, if the stack-allocated memory is small (e.g., $O(\log n)$), then the auxiliary space size $O(PS_1)$ will be negligible on modern machines. As a result, the abstraction of the stack-allocated memory separates the per-processor need from the shared resource, and overcomes the limitation of the in-place PRAM by dynamically mapping the algorithm on a machine with P processors, with the auxiliary space guarantee.

In addition to the advantages discussed above, the work-span model simplifies parallel algorithm design and analysis, and algorithm designers do not need to worry about low-level details related to hardware such as memory allocation, caching, and load balancing. Recent papers [6, 33, 36] have made a similar observation on the limitation of the in-place

PRAM model, and analyzed the in-place algorithms using the work-span model. In this paper, we explicitly formalize this discussion and define the two PIP models based on the work-span model.

Other practical considerations. Here we describe additional benefits to use the new PIP models based on the work-span model. Modern parallel programming languages, such as Cilk, OpenMP, TBB, and X10, directly support algorithms designed for the work-span model using fork-join parallelism, with efficient runtime schedulers. In contrast to the PRAM, in which computations have many synchronization points, computations in the work-span model can be highly asynchronous. This is a practical advantage due to the high synchronization overheads on modern hardware [16]. Furthermore, the PIP algorithms in this paper based on our new models have additional guarantees with respect to multiprogrammed environments [2], cache complexity [1, 15, 17], write-efficiency [4, 12, 13], and resource-obliviousness [22].

4 Decomposable Property

Designing strong PIP algorithms is generally challenging (we present several in Section 5, but if we relax the auxiliary space to sublinear (the relaxed PIP model), then we believe that PIP algorithms can be designed for many more problems. In this section, we introduce the ***Decomposable Property***, which enables any existing parallel algorithm that satisfies the property to be converted into a relaxed PIP algorithm. If the existing parallel algorithm is work-efficient, then the corresponding relaxed PIP algorithm will also be work-efficient.

THEOREM 4.1. (DECOMPOSABLE PROPERTY) *Consider a problem with input size n and a parallel algorithm to solve it with work $W(n) = O(n \cdot \mathrm{polylog}(n))$. Let $r = W(n)/n$. If the problem can be reduced to a subproblem of size $n - n^{1-\epsilon}/r$ using $n^{1-\epsilon}$ work and space for some $0 < \epsilon < 1$, and polylogarithmic span $D(n)$, then there is a relaxed PIP algorithm for this problem with $W(n)$ work, $O(n^\epsilon \cdot \mathrm{polylog}(n))$ span, and $O(n^{1-\epsilon})$ auxiliary space.*

Proof. We iteratively reduce the problem size by $n^{1-\epsilon}/r$ (this size remains the same throughout the algorithm), and each round takes $O(n^{1-\epsilon})$ work and space. Since $r = \mathrm{polylog}(n)$, this means $n^{1-\epsilon}$ is asymptotically larger than r, and we can reduce the problem size by at least one on each round. By applying this reduction for rn^ϵ rounds, we have a relaxed PIP algorithm with $W(n)$ work and $D(n) \cdot rn^\epsilon = O(n^\epsilon \cdot \mathrm{polylog}(n))$ span, using $O(n^{1-\epsilon})$ auxiliary space. □

The high-level idea of the Decomposable Property is that, for a problem of size n, if we can reduce the problem size to $n - n'$ using work proportional to n', then we can control the additional space by varying the size of n' to fit in the auxiliary space. This provides theoretically-efficient

relaxed PIP algorithms for parallel algorithms that satisfy this property. On the practical side, we observe that this reduction step usually corresponds to solving a subproblem that is the same as the original problem but with a smaller size. Hence, we can use the best existing non-in-place algorithms for this step. We show in Section 7 that the performance of our relaxed PIP algorithms using this approach is competitive or faster than their non-in-place counterparts. In the rest of this section, we introduce some algorithms that satisfy the Decomposable Property.

4.1 Random Permutation

Generating random permutations in parallel is a useful subroutine in many parallel algorithms. Many parallel algorithms (e.g., randomized incremental algorithms) require randomly permuting the input elements to achieve strong theoretical guarantees. The sequential Knuth [24, 32] shuffle algorithm, shown below, has linear work, where $H[i]$ is an integer uniformly drawn from $[1, \ldots, i]$, and A is the array to be permuted.

---

1 Function KNUTH-SHUFFLE*(A, H)*
2   **for** $i \leftarrow n$ to 1 **do** $A[i] \leftarrow i$
3   **for** $i \leftarrow n$ to 1 **do** swap($A[i]$, $A[H[i]]$)

Recent work [44] has shown that this sequential iterative algorithm is readily parallel. The pseudocode of this parallel algorithm is shown in Algorithm 1, and is both theoretically and practically efficient. The key idea is to allow multiple swaps to be performed in parallel as long as the sets of source and destination locations of the swaps are disjoint. We illustrate the dependence structure on an example in Figure 3. Given an input array H, in Figure 3(a) we create a node for each index, and an edge from the node to the node corresponding to its swap destination. In this example, we can swap locations 6 and 3, 8 and 2, and 7 and 4 simultaneously in the first step since these three swaps do not interfere with each other. To resolve the case where multiple nodes point to the same swap destination, we chain these nodes together, as shown in Figure 3(b). We also remove self-loops. In Algorithm 1, each unfinished swap writes to an auxiliary array R using a max() to reserve both its source and destination locations (Lines 4–6). We assume that max() takes $O(1)$ work, and in practice, it can be implemented using a compare-and-swap loop [42]. We then perform the actual swaps in parallel for the swaps that successfully reserve both of its locations (Line 8). The rest of the swaps will be packed and will try again in the next step. Shun et al. show that Algorithm 1 finishes in $O(\log n)$ rounds *whp* [44]. The work and span can be shown to be $O(n)$ in expectation and $O(\log n)$ *whp*, respectively [16, 44].

We now show that the random permutation algorithm above satisfies the Decomposable Property. The property for the sequential Knuth shuffle is easy to see—after applying

Algorithm 1: PARALLEL KNUTH-SHUFFLE(A, H) [44]

1 $R \leftarrow \{-1, \ldots, -1\}$
2 **parallel for** $i \leftarrow n$ to 1 **do** $A[i] \leftarrow i$
3 **while** swaps unfinished **do**
4   **parallel foreach** unfinished swap $(s, H[s])$ **do**
5 $R[s] \leftarrow \max(R[s], s)$
6 $R[H[s]] \leftarrow \max(R[H[s]], s)$
7   **parallel foreach** unfinished swap $(s, H[s])$ **do**
8   **if** $R[s] = s$ and $R[H[s]] = s$ **then**
       swap($A[H[s]], A[s]$)
9   Reset R and pack the leftover swaps (without modifying the swaps)
10 **return** A
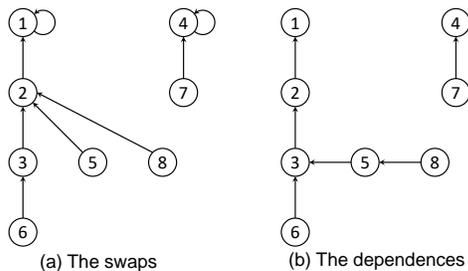
Figure 3: An example for $H = [1, 1, 2, 4, 2, 3, 4, 2]$. (a) indicates the destinations of the swaps according to H. The dependences of the swaps are shown in (b), indicating the order of the swaps.

the first $n^{1-\epsilon}$ swaps, which we refer to as one **round**, the problem reduces to a subproblem of size $n - n^{1-\epsilon}$, which can be solved using the same algorithm. We note that for any $n^{1-\epsilon}$ swaps, up to $3n^{1-\epsilon}$ locations will be accessed in the R and H arrays ($H[s]$, $R[s]$, and $R[H[s]]$ for each swap source s). We can use a parallel hash table to store these values using $O(n^{1-\epsilon})$ space. When the load factor of the hash table is no more than one-half, then each update or query requires $O(1)$ expected work and $O(\log n)$ work *whp* [31, 41]. To guarantee that our algorithm has the same bounds as proved in [16, 44], we always work on the first $n^{1-\epsilon}$ unfinished swaps based on the sequential order. The longest dependence length among the first $n^{1-\epsilon}$ swaps in a phase is bounded by $O(\log n)$ *whp* since it cannot be longer than the overall dependence length for all n swaps, which is bounded by $O(\log n)$ *whp*. The overall span in a phase is $O(\log^2 n)$, where the additional factor of $\log n$ due to hash table insertions and queries. The entire algorithm finishes after n^ϵ rounds and is work-efficient. By applying Theorem 4.1, we obtain the following theorem.

THEOREM 4.2. *There is a relaxed PIP algorithm for random permutation using $O(n)$ expected work, $O(n^\epsilon \log^2 n)$ span whp, and $O(n^{1-\epsilon})$ auxiliary space for $0 < \epsilon < 1$.*

Constant-dimension linear programming and smallest enclosing disks. Based on the relaxed PIP algorithm for random permutation, it is straightforward to design relaxed PIP algorithms for constant-dimension linear programming

Algorithm 2: PARALLEL LIST-CONTRACTION(L) [44]

Input: A doubly-linked list L of size n. Each element l_i has a random priority $p(l_i)$.
1 $R \leftarrow \{0, \ldots, 0\}$
2 **while** elements remaining **do**
3   **parallel foreach** uncontracted element l_i **do**
4   **if** $p(l_i) < p(prev(l_i))$ and $p(l_i) < p(next(l_i))$ **then**
5 $R[i] \leftarrow 1$
6   **parallel foreach** uncontracted element l_i **do**
7   **if** $R[i] = 1$ **then**
8   Splice out element l_i and update pointers
9   Pack the leftover (uncontracted) elements
10 **return** A

and smallest enclosing disks using randomized incremental construction [18, 40]. The randomized algorithms after randomly permuting the input elements take $O(d!n)$ expected work and $O(d \log n)$ span and auxiliary space *whp*, where d is the dimension [18], by using the in-place reduce algorithm that will be discussed in Section 5. By using the relaxed PIP random permutation algorithm, we can obtain parallel in-place algorithms for constant-dimension linear programming and smallest enclosing disks in $O(d!n)$ expected work and $O(n^\epsilon \log^2 n + d \log n)$ span *whp*, using $O(n^{1-\epsilon} + d \log n)$ auxiliary space.

4.2 List Contraction and Tree Contraction

List ranking [29, 30, 38] is one of the most important problems in the study of parallel algorithms. The problem takes as input a set of linked lists, and returns for each element its position in its list. List contraction is used to contract a linked list into a single node, and is used as a subroutine in list ranking.

We now discuss the Decomposable Property of list contraction. The order of contracting elements does not matter as long as all elements are eventually contracted. Therefore, similar to random permutation, we can process $n^{1-\epsilon}$ elements in a round, and apply existing parallel list contraction algorithms [29, 30] to contract these $n^{1-\epsilon}$ elements. To show an example, we discuss Shun et al.'s non-in-place list contraction algorithm [44] and how to turn it into a relaxed PIP algorithm. This is also the algorithm that we implemented in this paper.

The pseudocode and the high-level idea of this algorithm is given in Algorithm 2. A careful implementation of this algorithm takes worst-case linear work and $O(\log n)$ span *whp* [16, 44]. This algorithm assigns a random priority to each list element (Figure 4(a)), contracts all elements that have priority lower than both of its neighbors' priorities (Figure 4(b)), packs the leftover elements, and iterates until the list is empty. The number of rounds of this algorithm is the length of the longest dependence among the nodes, which is $O(\log n)$ *whp* [44]. Figure 4(d) shows the dependences in the example (a node depends on all of its descendants in
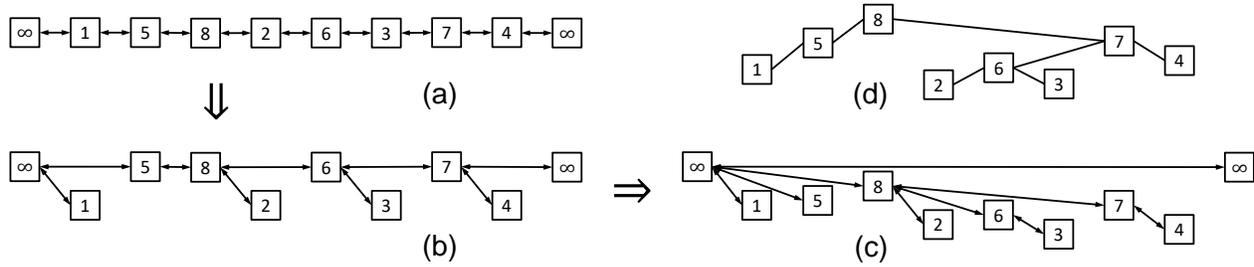
Figure 4: A list contraction example for Algorithm 2 with the priorities shown in the boxes. (a) shows the original input, with an extra ∞ element at each end of the list. The first round of the algorithm contracts the nodes with priorities 1, 2, 3, and 4, as shown in (b). After all nodes are contracted, a tree structure is formed, as shown in (c). The dependences of the algorithm are shown in (d), where a node depends on all of its descendants in the tree. For instance, the contraction of the node with priority 6 needs to be done after the contraction of the nodes with priorities 2 and 3.

the tree shown)—here the algorithm finishes in 4 rounds (the height of the tree).

As discussed, the order of the contraction does not matter. Hence, for a problem of size n, we can work on $n^{1-\epsilon}$ elements and contract them using this algorithm, which requires $O(n^{1-\epsilon})$ work, $O(\log n)$ span *whp* (no more than the span for n elements), and $O(n^{1-\epsilon})$ space. Then the problem reduces to a subproblem of size $n - n^{1-\epsilon}$. We can iteratively apply this for n^ϵ rounds, which yields a relaxed PIP algorithm for list contraction.

After the n elements are spliced out, list contraction algorithm generates a tree, and the tree for the example in Figure 4(a) is shown in Figure 4(c). The remaining work in list ranking after list contraction is referred to as "reconstruction" [29], which distributes the values down the tree. Therefore, once we obtain this tree structure, the classic algorithms [29, 39] for reconstruction take worst-case linear work and $O(\log n)$ span *whp*. Representing the tree only requires n pointers, which fit into the $2n$ pointers in the input linked list if we are allowed to overwrite the input. For our new relaxed PIP algorithm, we can store the $n^{1-\epsilon}$ tree pointers in each round by overwriting the $2n^{1-\epsilon}$ pointers of the elements being processed in the current round. After we recursively solve the smaller subproblem, we can use the classic reconstruction algorithm for the $n^{1-\epsilon}$ elements in the current round, which takes worst-case linear work and logarithmic span *whp*. In total, the reconstruction step has the same work, span, and auxiliary space bounds as list contraction.

Tree contraction is a generalization of list contraction and has many applications in parallel tree and graph algorithms [29, 35, 38, 44]. Here we will assume that we are contracting rooted binary trees in which every internal node has exactly two children. As in list contraction, the ordering of contracted tree nodes does not matter as long as a parent-child pair is not contracted in the same round. For a problem of size n, we can work on $n^{1-\epsilon}$ tree nodes each round and contract them using existing tree contraction algorithms, and repeat for n^ϵ rounds. Therefore, the Decomposable Property is satisfied for tree contraction. We can convert the paral-

lel tree contraction algorithm of Shun et al. [16, 44] that is not in-place, but theoretically and practically efficient, to a relaxed PIP algorithm that requires $O(n^{1-\epsilon})$ expected work and $O(\log n)$ span *whp* per round, and $O(n^{1-\epsilon})$ space.

We obtain the following theorem for list contraction and tree contraction.

THEOREM 4.3. *There are relaxed PIP algorithms for list contraction and tree contraction that take $O(n)$ work, $O(n^\epsilon \log n)$ span whp, and $O(n^{1-\epsilon})$ auxiliary space for $0 < \epsilon < 1$.*

4.3 Merging and Mergesort

Merging two sorted arrays of size n and m (stored consecutively in an array of size $n + m$) is another canonical primitive in parallel algorithm design. We assume without loss of generality that $n \geq m$. Parallel in-place merging algorithms have been studied for the PRAM model [25], using $O(n \log n)$ work, $O(n^\epsilon \log n)$ span, and $O(n^{1-\epsilon})$ auxiliary space when mapped to the relaxed PIP model. However, this algorithm is quite complicated and unlikely to be practical. By using the Decomposable Property, we can design a much simpler algorithm based on any existing textbook parallel non-in-place merging algorithm, combined with some features of the sequential in-place merging algorithm [27]. The key idea in [27] for in-place merging is to split both input arrays into chunks of size k, and sort the chunks based on the last element of each chunk. Then, the algorithm merges the first remaining chunk from each of the two input arrays, and when one chunk is used up, the algorithm replaces it with the next chunk in the corresponding array.

To obtain a relaxed PIP algorithm, we set the chunk size to $k = n^{1-\epsilon}$, so that we can process two chunks using $O(n^{1-\epsilon})$ auxiliary space. With this space bound, we can use a non-in-place merging algorithm to output the smallest $k = n^{1-\epsilon}$ elements and repeat for $O(n^\epsilon)$ rounds.

The first step of our algorithm is the same as [27], which sorts the chunks based on only their last elements, and moves each chunk to their final destination in parallel by using the $O(n^{1-\epsilon})$ auxiliary space as a buffer. Sorting all of the chunks takes $O(n^\epsilon \log n)$ span. Then, in the merging phase, we move

the first chunk from each array to the auxiliary space, use any existing parallel merging algorithm to merge them, until we either run out of the elements in one chunk, at which point we load the next chunk of the corresponding array to the auxiliary space, or until we gather a full chunk of merged elements, at which point we flush it back to the original array and empty the buffer. At any time, there can be at most three chunks in the auxiliary space—two chunks from the input arrays and one chunk for the merged output, and so the required auxiliary space is $O(n^{1-\epsilon})$. We can use any existing non-in-place parallel algorithm [10, 29] to perform the merge in the auxiliary space, which takes linear work and logarithmic span. Such calls to merge in the algorithm can happen at most $2(n+m)/k$ times—$(n+m)/k$ times after loading new chunks to the auxiliary space and $(n+m)/k$ times after the output chunk is full and is flushed. Each merge takes work linear in the output size, and $O(\log k) = O(\log n)$ span. The overall work is therefore $O(n)$, and the span is $O((n/k)\log n) = O(n^\epsilon \log n)$. This gives the following theorem.

THEOREM 4.4. *Merging two sorted arrays of size n and m (where $n \geq m$) stored consecutively in memory takes $O(n)$ work, $O(n^\epsilon \log n)$ span, and $O(n^{1-\epsilon})$ auxiliary space for $0 < \epsilon < 1$.*

When $\epsilon > 1/2$, the auxiliary space $O(n^{1-\epsilon})$ is insufficient for sorting all $O(n^\epsilon)$ chunks at the beginning, and so we sort $O(n^{1-\epsilon})$ chunks at a time until all chunks have been processed. As done in [27], we use dual binary search to find the smallest $O(n^{1-\epsilon})$ chunks to merge, and repeat our above algorithm for $O(n^\epsilon)$ rounds. This will not affect the cost bounds.

With the relaxed PIP merging algorithm, we can obtain a relaxed PIP mergesort algorithm with $O(n \log n)$ work, $O(n^{1-\epsilon})$ auxiliary space, and $O(n^\epsilon \log^2 n)$ span.

4.4 Filter, Unstable Partition, and Quicksort

It is easy to see that we can work on a prefix of the filter problem of size $n^{1-\epsilon}$ using linear work and logarithmic span, and repeat for n^ϵ rounds. The only additional work is to move the unfiltered elements to the beginning of the array, which can be done in linear work and $O(\log n)$ span for each prefix. This gives a relaxed PIP algorithm for filter that takes $O(n)$ work, $O(n^\epsilon \log n)$ span, and $O(n^{1-\epsilon})$ auxiliary space. We can implement partition similarly, and when moving the unfiltered elements to the beginning, we swap the elements so that at the end of the algorithm, the filtered elements are moved to the end of the array. This algorithm has the same cost as filter, although the partition result is not stable. With the relaxed PIP partition algorithm, we can obtain a relaxed PIP algorithm for (unstable) quicksort that takes $O(n \log n)$ expected work and $O(n^\epsilon \log^2 n)$ span *whp*, and $O(n^{1-\epsilon})$ auxiliary space.

5 Strong PIP Algorithms

The strong PIP model is restrictive because of the polylogarithmic auxiliary space requirement. To date, only a few non-trivial and work-efficient strong PIP algorithms have been proposed: reducing and rotating an array, which are trivial, certain fixed permutations [6], and two-way partitioning [33]. In this section, we review existing strong PIP algorithms for reduce and rotation, and present new algorithms for scan (prefix sum), filter, merging, and sorting.

5.1 Existing Algorithms

Reduce. The classic divide-and-conquer algorithm for reduce is already strong PIP. It is implemented by dividing the input array by two equal sized subarrays, recursively solving the two subproblems in parallel, and finally summing together the partial sums from the two subproblems. This algorithm requires $O(\log n)$ sequential stack space, $O(n)$ work and $O(\log n)$ span, and so it is an optimal strong PIP algorithm.

Rotating an array. Given an array $[a_1, a_2, ..., a_n]$ and an offset o, the output is a rotated array $[a_{o+1}, \ldots, a_n, a_1, \ldots, a_o]$. This can be implemented by first reversing $[a_1, \ldots, a_o]$, then reversing $[a_{o+1}, \ldots, a_n]$, and finally reversing the entire array. Reversing can be implemented with a parallel loop, which requires $O(\log n)$ stack space when run serially. This algorithm requires $O(n)$ work and $O(\log n)$ span, and is therefore an optimal strong PIP algorithm.

5.2 Scan

Scan (prefix sum) is probably the most fundamental algorithmic primitive in parallel algorithm design. Here we assume $\oplus$ is $+$ (addition) for simplicity, but the results in this section also apply to other associative binary operators. Non-in-place implementations of scan have been designed since the last century, and the work-efficient version is generally referred to as the Blelloch scan [8]. The Blelloch scan contains two phases. The first phase is referred to as the "up-sweep", which partitions the array into two halves, computes the sum recursively for each half, then uses the prefix sums for each half to calculate the prefix sums for the entire sequence, and finally stores this result in auxiliary space. Then the algorithm applies a "down-sweep" phase, which propagates the sums from the first phase down to each element recursively—for a subproblem with a prefix sum of p ($p = 0$ for the subproblem corresponding to the whole sequence), we recursively solve the left half with prefix sum p, and the right half with prefix sum p plus the sum of the left half, in parallel. This algorithm takes $O(n)$ work and $O(\log n)$ span, but unfortunately, it requires linear auxiliary space to store all of the partial sums.

Making existing approaches in-place. We first discuss a solution to make the Blelloch scan in-place. We partition the array into two equal-sized halves, recursively solve each

---

Algorithm 3: IN-PLACE-SCAN

    Input: An array $A_{1\dots n}$ of size n, assuming $A_0 = 0$.
    Output: The exclusive prefix-sum array of A, and sum σ.

1 UP-SWEEP($A, 1, n$)
2 $\sigma \leftarrow A_n$
3 DOWN-SWEEP($A, 1, n, 0$)
4 **return** (A, σ)

5 **Function** UP-SWEEP(A, s, t)
6   **if** $s = t$ **then return**;
7   **In parallel:**
8   UP-SWEEP($A, s, \lfloor (s+t)/2 \rfloor$)
9   UP-SWEEP($A, \lfloor (s+t)/2 \rfloor + 1, t$)
10    $A_t \leftarrow A_t + A_{\lfloor (s+t)/2 \rfloor}$

11 **Function** DOWN-SWEEP(A, s, t, p)
12   **if** $s = t$ **then** $A_s \leftarrow p$, **return**;
13   *LeftSum* $= A_{\lfloor (s+t)/2 \rfloor}$
14   **In parallel:**
15   DOWN-SWEEP($A, s, \lfloor (s+t)/2 \rfloor, p$)
16   DOWN-SWEEP($A, \lfloor (s+t)/2 \rfloor + 1, t, p + \textit{LeftSum}$)

---

half, and apply a parallel for-loop to add the sum of the left half to every element in the right half. Directly applying this algorithm leads to $O(n \log n)$ work, since the recursion tree has $\log_2 n$ levels, and on each level we need to perform $O(n)$ additions, which takes $O(n)$ work and $O(\log n)$ span. We can reduce the work overhead by stopping the recursion when we reach a subproblem of size no more than $\log_2 n$ (these subproblems constitute the base cases), and apply a sequential in-place scan for these subproblems, and store the partial sums in the last elements of the subproblem arrays. We then run scan on the sums of the $m = O(n/\log n)$ base cases using the aforementioned algorithm. This scan takes $O(m \log m) = O(n)$ work and computes the prefix sum before the beginning of each base case. Lastly, we add this prefix sum to the elements in each base case subproblem to obtain the final result for scan. This algorithm uses $O(n)$ work, $O(\log^2 n)$ span, and $O(\log(n/\log n)) = O(\log n)$ auxiliary space, which is the recursion depth.

Another approach is to use the Brent-Kung adder [21], which is a circuit to solve the scan problem with $O(\log n)$ span, $O(n)$ gates, and $O(n \log n)$ area. We can change the circuit to an algorithm that contains $O(\log n)$ parallel for-loops and each for-loop simulates the gates at one level. The work of this algorithm is linear, which is the same as the number of gates, and the span is $O(\log^2 n)$—$O(\log n)$ parallel for-loops each taking $O(\log n)$ span for forking the tasks. The output of the original circuit is an inclusive scan (i.e., the output is $[a_1, \dots, (a_1 \oplus a_2 \oplus \dots \oplus a_n)]$). The circuit can be modified to compute the exclusive scan in the same bounds. In conclusion, we can make the the existing approaches in-place, but their span would not be optimal.

A new optimal strong PIP algorithm. Our new strong PIP algorithm is almost as simple as the non-in-place Blelloch scan, and has the same work and span bounds. The new algorithm as shown in Algorithm 3, and illustrated in Figure 5. In the pseudocode, we assume $A_0 = 0$ when it is referenced, but the algorithm does not actually need to store this. The new strong PIP algorithm also contains two phases: the up-sweep and the down-sweep phases, both of which are recursive. The key insight in our new algorithm is to maintain all of the intermediate results in the input array of n elements, and use stack space in the down-sweep phase to pass down the partial sums. For each recursive subproblem corresponding to a subarray from index s to t, we partition it into two halves, s to k and $k+1$ to t, where $k = \lfloor (s+t)/2 \rfloor$. In the up-sweep phase, we first recursively solve the two subproblems, and then add the value at index k to the value at index t. These additions are shown as arrows on the left side of Figure 5. In the down-sweep phase, we keep the prefix sum p of each subproblem. Similar to the Blelloch scan, we compute the prefix sum of the right subproblem by adding the sum of the left subproblem to the current prefix sum. In our algorithm, the sum of the left subproblem is stored at $A_{\lfloor (s+t)/2 \rfloor}$. Both recursions stop when $i = j$, and at the end of the down-sweep, we obtain the exclusive scan result. The down-sweep process and its output on an example are shown on the right side of Figure 5.

Correctness and efficiency. The correctness and efficiency of this algorithm is based on the following observation. In the down-sweep phase, the value of A_t in any recursive call is not being used (except for the root where A_n is the total sum). Hence, in our algorithm, we reuse the space for A_t to store the sum for the next level. The reduction tree (left side of Figure 5) has $2n - 1$ nodes: n nodes for the input and $n - 1$ internal nodes storing the partial sums. We note that in the down-sweep phase, only the sums of the left subproblems are used, and there are $n - 1$ of them. They are stored in $A_{1,\dots,n-1}$ by the end of up-sweep, while A_n stores the total sum. With all of these values, we can run the down-sweep phase in the same way as in the Blelloch scan. The partial sums stored in $A_{1,\dots,n-1}$ are passed to the output by the argument p in the down-sweep function call, which is stored in the stack space. Hence, the new strong PIP scan algorithm uses $O(n)$ work, $O(\log n)$ span, and $O(\log n)$ sequential auxiliary stack space, and is therefore an optimal strong PIP algorithm.

THEOREM 5.1. *The new strong PIP scan algorithm is optimal, using $O(n)$ work, $O(\log n)$ span, and $O(\log n)$ sequential auxiliary space.*

5.3 Other Strong PIP Algorithms

Filter, unstable partition, and quicksort. Consider a k-way divide-and-conquer algorithm for filter, where we partition the array into k chunks of equal size, filter each chunk,
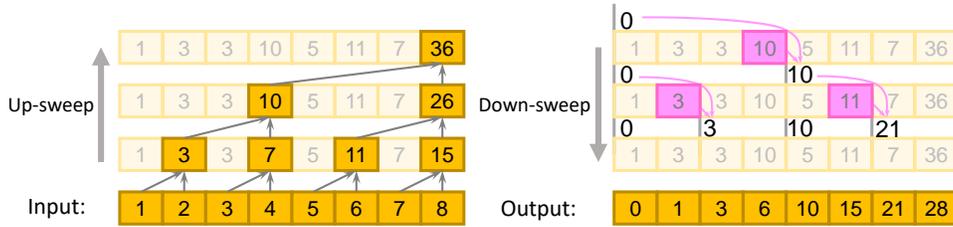
Figure 5: Our new strong PIP scan algorithm. It has an up-sweep phase (left) and a down-sweep phase (right). Each pair of arrows pointing to the same element indicates an addition.

and pack the unfiltered results together. For one level of recursion, this takes linear work and $O(k \log n)$ span if chunks are processed one at a time, but within each chunk we move the elements in parallel. This algorithm only requires a constant amount of extra space to store pointers. The number of levels of recursion is $O(\log_k n)$, and so the overall work is $O(n \log n / \log k)$, and the overall span is $O((k/\log k) \log^2 n)$. Similar to Section 4.4, we can use this filter algorithm to implement an unstable partition algorithm, with the same cost bounds. In theory, we can plug in any constant for k, which gives a strong PIP algorithm with $O(\log^2 n)$ span and $O(\log n)$ auxiliary space, although it is not work-efficient. Alternatively, we can achieve work-efficiency by setting $k = n^\epsilon$. This does not achieve polylogarithmic span, but has good performance in practice. We implement this filter algorithm and present experimental results in Section 7.

We can obtain an unstable quicksort algorithm that applies the partition algorithm for $O(\log n)$ levels of recursion *whp*. We note that Kuszmaul and Westover [33] recently developed a work-efficient strong PIP algorithm for partition, which gives a work-efficient and polylogarithmic-span quicksort algorithm.

Merging and mergesort. We again consider merging two sorted arrays of size n and m, which are stored consecutively in an array of size $n + m$. Again, we can use a two-way divide-and-conquer approach, where we use a dual binary search to find the median among all $n + m$ elements, and in parallel swap the out-of-place elements in two arrays. This swap can be implemented by the strong PIP algorithm for array rotation, discussed in Section 5.1. Then, we recursively run merging on the two subproblems, each of size $(m + n)/2$. The subproblem size shrinks by a factor of 2 on each level of recursion, and so the recursion depth is bounded by $\log_2(n + m)$. The work to swap the elements at each level is $O(n+m)$, and so the overall work is $O((n+m) \log(n+m))$. The span and auxiliary space is $O(\log(n + m))$, which is proportional to the recursion depth. This gives a strong PIP algorithm for merging. A strong PIP mergesort algorithm can be obtained by plugging in this merging algorithm, although it is not work-efficient.

Set operations. We now consider computing the union, intersection, and difference of two ordered sets of size n and $m \leq n$. If the two sets are given in a binary tree format, then existing algorithms for these operations [11, 45] are already strong PIP, work-optimal ($O(m \log(n/m + 1))$ work), and have $O(\log^2 n)$ span. We now describe how to implement these operations if the sets are given in arrays stored contiguously in memory. For union, we can first use the merging algorithm described above, and then the filter algorithm described above to remove duplicates. Therefore, computing the union on arrays is strong PIP. For intersection and difference, we can run binary searches to find each element in the smaller set inside the larger set, and then apply the filter algorithm described above to obtain the output, which takes $O(n \log n)$ work. The resulting algorithms are not work-efficient, since our strong PIP merging and filter are not work-efficient.

6 Relaxed PIP Graph Algorithms

In this section, we introduce new relaxed PIP algorithms for graph connectivity, biconnectivity, and minimum spanning forest. There has been some related work on sequential in-place graph algorithms, which we discuss in the full version of this paper.

Connectivity and Biconnectivity. The standard output size for graph connectivity and biconnectivity is $O(n)$ and $O(m)$, respectively. Recent work by Ben-David et al. [5] introduces a compressed scheme for storing graph connectivity information. For any $1 \leq k \leq n$, it requires $O(k \log n + m/k)$ output size with an $O(k)$ expected query work for connectivity and $O(k^2)$ expected query work for biconnectivity. Constructing such a compressed (bi)connectivity oracle takes $O(km)$ expected work and $O(k^{3/2} \log^3 n)$ span *whp*. By setting $k = m^\epsilon$, we have the following theorem.

THEOREM 6.1. *A (bi)connectivity oracle can be constructed using $O(m^{1-\epsilon})$ auxiliary space, $O(m^{1+\epsilon})$ expected work, and $O(m^{3\epsilon/2} \log^3 n)$ span whp for $1/2 < \epsilon < 1$. A connectivity query can be answered in $O(m^\epsilon)$ expected work, and a biconnectivity query can be answered in $O(m^{2\epsilon})$ expected work.*

The high-level idea in the algorithms is to select a subset of the vertices as the "centers" and only keep information for these center vertices. Each vertex has a $1/k$ probability of being selected as a center. This is referred to as the ***implicit decomposition*** of the graph. For a query to a non-center vertex v, we apply a breadth-first search from v

to the first center c, which takes $O(k)$ expected work [5]. For connectivity, v's label is the same as c's label. It is also possible that a search does not reach any center, but Ben-David et al. [5] show that the expected size of a connected component without a center node is small ($O(k)$ in expectation), and so the cost to traverse all vertices in such a component is also $O(k)$ in expectation. For biconnectivity, an additional step of local analysis is required to obtain the output for v from c, which requires $O(k^2)$ expected work.

Theorem 6.1 gives algorithms that is almost relaxed PIP, other than having an extra factor of $O(m^{\epsilon/2})$ in the product of the space and span bounds. Alternatively, we can obtain new relaxed PIP connectivity and biconnectivity algorithms by using the minimum spanning forest algorithm that will be discussed next, at a cost of additional work.

Minimum Spanning Forest. The idea of implicit decomposition can be extended to the minimum spanning forest (MSF) problem. For simplicity, we assume that the graph is connected, but disconnected graphs can also be handled using an approach described by Ben-David et al. [5].

We note that the MSF is unique for a graph (assuming that ties are broken consistently). Therefore, for a query to vertex v, instead of using a breadth-first search on *all* edges to find the center in connectivity, we need to search out to a center using *only* the MSF edges. This can be achieved by using a Prim-like search algorithm from v. This increases the work by a factor of $O(\log k)$ to compute the implicit decomposition of the graph and for the query cost (the queue will contains $O(k)$ vertices on average for each search).

We can generate an implicit decomposition of the graph using a similar approach as for connectivity and biconnectivity. We then compute the MSF across the m/k centers of the decomposition. The output size of this spanning forest is $O(m/k)$. To compute the MSF in parallel, we can use Borůvka's algorithm. We start with every cluster being in its own component, and enumerate all edges for $O(\log n)$ rounds until the entire graph is connected. On each round, we run Borůvka's algorithm to find the minimum outgoing edges from each component. This takes $O(mk \log k)$ work—we check all m edges in a Borůvka's round and each edge takes $O(k \log k)$ work to find the clusters of both of its endpoints. By setting $k = m^\epsilon$, we obtain the following theorem.

THEOREM 6.2. *Given a graph with n vertices and m edges, a data structure for minimum spanning forest can be computed in $O(m^{1+\epsilon} \log^2 n)$ expected work, $O(m^\epsilon \operatorname{polylog}(n))$ span whp, and $O(m^{1-\epsilon})$ auxiliary space. Querying if an edge is in the MSF takes $O(m^{1-\epsilon} \log n)$ work.*

This MSF algorithm is relaxed PIP. We can also obtain relaxed PIP algorithms for connectivity and biconnectivity using the MSF as the spanning forest required in those algorithms. Detailed analyses for correctness and the cost

bounds of these algorithms are provided in the full version of this paper.

7 Implementations and Experiments

In the previous sections, we have designed parallel in-place algorithms with strong theoretical guarantees. Many of these algorithms are relatively simple, and in this section we describe how to implement these algorithms efficiently so that they can outperform or at least be competitive with their non-in-place counterparts, while using less space. We present implementations for five algorithms: scan, filter, random permutation, list contraction, and tree contraction. The implementations for the first two are fairly simple, and the last three are based on the deterministic reservations framework of Blelloch et al. [14].

7.1 Experimental Setup

We run all of our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4×2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. We compile the code using the g++ compiler (version 5.4.1) with the -O3 flag, and use Cilk Plus for parallelism.

We compare our PIP algorithms to the non-in-place versions in the Problem Based Benchmark Suite (PBBS) [43], which is a collection of highly-optimized parallel algorithms and implementations and widely used in benchmarking. The implementations of random permutation, list contraction, and tree contraction in PBBS are from [44].

7.2 Scan and Filter

For scan, we implement Algorithm 3 and switch to a sequential in-place scan when the subproblem size is less than 256. For filter, we implement the PIP algorithm from Section 5.3, but we keep the implementation work-efficient by setting the branching factor $k = \sqrt{n}$, and only apply one level of recursion. However, this increases the span to $O(\sqrt{n} \log n)$ and has $O(\sqrt{n})$ rounds of global synchronization (the k chunks are processed one after another), which is a significant overhead. We use the following optimization to significantly reduce this overhead in practice. We move the elements from multiple consecutive chunks in parallel as long as the destination of the last chunk is before the original location of the first chunk. We apply a binary search in each round to find the maximum number of chunks that can be moved in parallel. If the unfiltered elements are distributed relatively evenly in the input and the output size is a constant fraction of the input, then the algorithm requires logarithmic rounds to finish.

We compare our PIP algorithms to the non-in-place versions in PBBS. The PBBS scan is the classic Blelloch scan implementation [7] and the filter is similar to our implementation, but the output is stored in a separate array. In
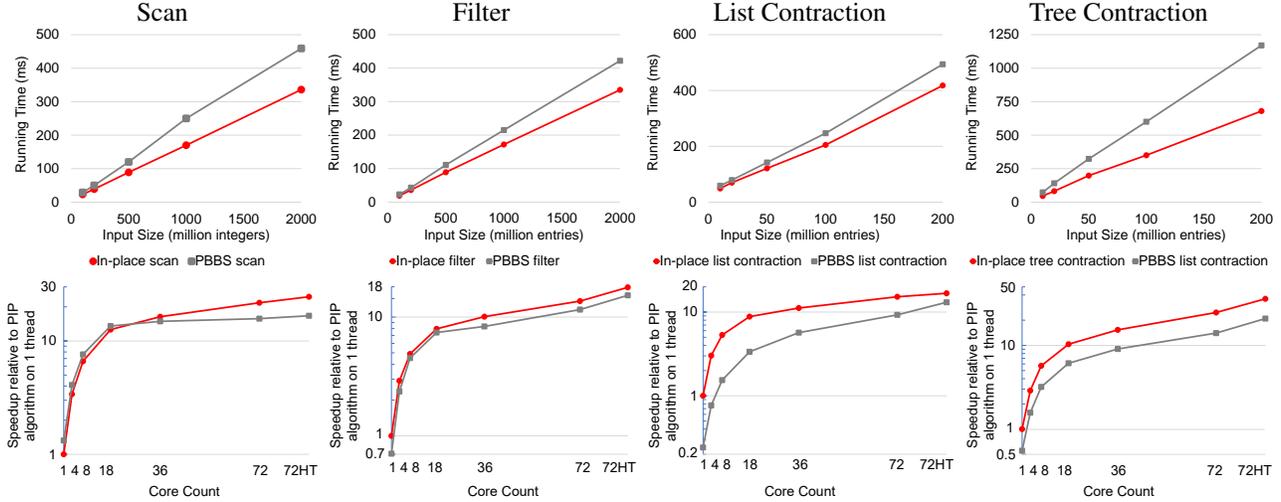
Figure 6: Running times and speedups for scan, filter, list contraction, and tree contraction. The top figures show the running times of our parallel in-place algorithms and their non-in-place counterparts from PBBS for varying input sizes. The bottom figures show the parallel speedups compared to the parallel PIP implementations on 1-thread, and the core count varies from 1 core to all 72 cores with two-way hyper-threading (72HT).

Input Size	Scan		Filter	
(million)	PBBS	PIP	PBBS	PIP
100	29	23	23	19
200	50	39	43	36
500	120	89	111	89
1000	250	170	215	172
2000	459	336	422	335

Table 2: Running times (in milliseconds) of the PBBS algorithms and our new PIP algorithms for scan and filter on 72 cores with hyper-threading.

Core	Scan		Filter	
count	PBBS	PIP	PBBS	PIP
1	3150	4170	2770	2150
4	1020	1230	861	701
8	548	630	524	393
18	308	331	301	244
36	280	255	254	191
72	265	192	222	179
72HT	250	170	215	172

Table 3: Running times (in milliseconds) of the PBBS algorithms and our new PIP algorithms for scan and filter on varying core counts.

Output fraction	12.5%	25%	50%	75%	87.5%
PBBS filter	189	202	215	234	252
PIP filter	94	118	172	212	254

Table 4: Running times (in milliseconds) of the PBBS filter algorithm and our new PIP filter algorithm, with varying output fraction on 72 cores with hyper-threading. The input is 1 billion integers.

the PBBS filter, it first filters each $\sqrt{n}$-sized chunk in parallel while each chunk is processed sequentially, and then moves the remaining elements to a separate output array in parallel.

The running times and scalability (parallel speedup relative to the best algorithm on 1 thread, which was our PIP algorithm in all cases) for scan and filter on different input sizes are shown in Figure 6 and Tables 2–3. For filter, 50% of the input entries are kept in the output. Our in-place scan is 30–45% faster and our in-place filter is about 25–30% faster than their non-in-place counterparts due to having a smaller memory footprint. The speedups are also competitive or better than the non-in-place versions. For filter, the fraction of elements in the output affects the performance of both our algorithm and the PBBS algorithm. A larger output fraction increases the number of rounds for movement and global synchronization in the PIP filter algorithm. In Table 4, we vary the output fraction and show that our new algorithms range from about 2x faster (12.5% output) to having about the same performance (87.5% output).

This experiment indicates that using the PIP scan and filter algorithms can improve both the running time and memory usage over the non-in-place filter algorithm, and is preferable when the input can be overwritten. We note that

ParlayLib [9], the latest version of PBBS, also includes the in-place versions of scan and filter, and we plan to compare with these in the future.

7.3 Random Permutation, List Contraction, and Tree Contraction

Implementing the PIP algorithms for random permutation, list contraction, and tree contraction is more challenging since they are more complicated than scan and filter. However, the Decomposable Property can greatly simplify the implementation of these algorithms, and we only need to design an efficient implementation working on a prefix of the problem and run it iteratively. Interestingly, the original implementations of these algorithms in [44] are based on a

Phase:	Reserve	Commit	Cleaning
RP-PBBS	2/1	2/1	0/0
RP-Naïve	0/3	0/3	3/0
RP-Flat	1/2	1/2	2/0
RP-OneRes	1/1	1/2	1/0
RP-Final	1/1	2/1	1/0

Table 5: Approximate number of sequential/random access per swap on each round for the five implementations of random permutation.

Input size:	10M	30M	100M	300M	1000M
RP-PBBS	43.7	89.2	283	781	2680
RP-Naïve	134	256	910	2580	9330
RP-Flat	98.1	187	644	1880	6280
RP-OneRes	77.1	133	422	1370	5250
RP-Final	65.6	131	388	1160	3960

Table 6: Running time (in milliseconds) of the five implementations of random permutation on 72 cores with hyper-threading.

Additional space	0.4%	1%	2%	4%
Running time (ms)	537	425	411	388

Table 7: Running time with different restrictions on additional space for RP-Final on 72 cores with hyper-threading. The input is 100 million 64-bit integers.

framework named as the *deterministic reservations* [14] that runs similarly in rounds, where each round processes a prefix of the remaining elements. Hence, we carefully modify the original implementations and obtain new PIP algorithms that are competitive for random permutation, and faster for list contraction and tree contraction. Due to space constraints, we present detailed descriptions of our implementations in the full version of the paper.

Deterministic Reservations. Deterministic reservations is a framework for iterates in a parallel algorithm to check if all of their dependencies have been satisfied through the use of shared data structures, and executing the ones that have been satisfied [14]. Deterministic reservations proceeds in rounds, where on each round, each remaining iterate tries to execute. Iterates that fail to execute will be packed and processed again in the next round. To achieve good performance in practice, instead of processing all iterates on every round, the framework only works on a prefix of the remaining iterates (usually around 2% of the input iterates). This naturally meets our requirement for controlling the number of elements to process in the relaxed PIP algorithms.

Random Permutation. We implement the PIP random permutation algorithm (Algorithm 1) based on deterministic reservations. We have four implementations (RP-Naïve, RP-Flat, RP-OneRes, and RP-Final), and each one improves upon the previous one.

The overall goal in our implementations is to reduce the number of memory accesses in the algorithm. The original RP-PBBS implementation from PBBS needs roughly 4 sequential accesses and 2 random accesses per swap on each round. In our new PIP algorithm, we need a data structure to hold all associated memory accesses in the prefix for auxiliary arrays R and H. In RP-Naïve, we simply use concurrent hash tables [41], and this implementation incurs roughly 3 sequential accesses and 6 random accesses per swap on each round. As an improvement, RP-Flat uses an array to replace the hash table for the H array, which changes the number of sequential and random accesses to 4 and 4, respectively. RP-OneRes removes one of the reservations (the one on line 6 of Algorithm 1) and reduces the number of random accesses to 3, at the cost of an additional cleaning phase at the end. Our final version, RP-Final, uses an array instead of a hash table for the part of R that is accessed contiguously by the iterates. RP-Final incurs

4 sequential accesses and 2 random accesses per swap on each round, which is the same as in the non-in-place version. More discussion about the implementation details are given in the full version of this paper, and the numbers of memory accesses for each implementation are given in Table 5.

We test the performance of our implementations on inputs of size 10 million to 1 billion 64-bit integers, and compare them with the best non-in-place counterpart, which is from PBBS (RP-PBBS). The actual running times are shown in Figure 7 and Table 6. In Figure 7 (left), we see that all of the implementations have similar and consistent scalability with respect to input size. In Figure 7 (right), we show the running times relative to RP-PBBS. RP-Final only has a modest overhead of 30–40% over RP-PBBS, while only using 4% of the auxiliary space required by RP-PBBS. We can further reduce the additional space by shrinking the prefix size, at the expense of having more rounds, and hence more global synchronization. In Table 7, we present the running times under different amounts of additional space for RP-Final.

List Contraction and Tree Contraction. Similar to random permutation, for list contraction and tree contraction, we design our PIP implementations based on the non-in-place implementations from PBBS [43, 44]. In this case, the auxiliary space in list contraction and tree contraction is the R array (shown in Algorithm 2), which has linear size. In list contraction (Algorithm 2) and tree contraction, $R[i]$ represents whether node i can be contracted in the current round. The deterministic reservations framework also needs to keep this information stored in another form, since it needs to pack the remaining iterates for the next round. We optimized the implementations to compute the remaining iterates directly from the information in the R array, and we provide more details in the full version of the paper.

We test the performance of our implementations on inputs of between 10 million to 200 million entries, each of which contain two 64-bit pointers. The running times and speedups over the the PIP algorithm on 1 thread as a function of core counts are shown in Figure 6 and Tables 8–
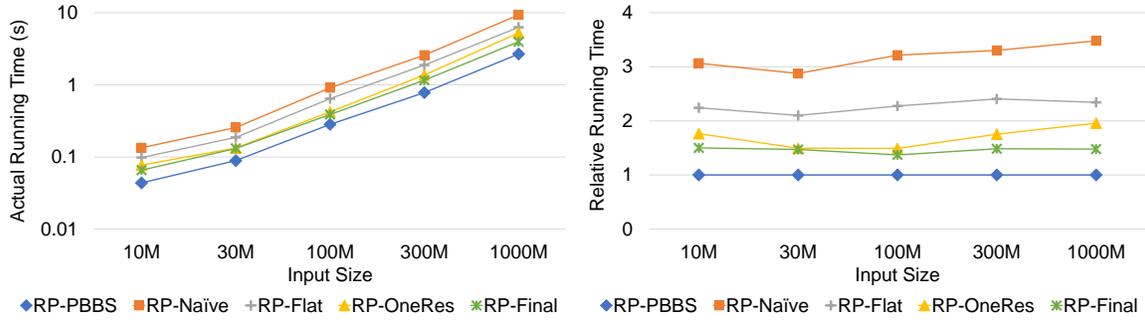
Figure 7: The actual running times (left) and running times relative to `RP-PBBS` (right) of different implementations for random permutation on 72 cores with hyper-threading. The input size varies from 10 million to 1 billion integers.

| Input Size | List contraction | | Tree contraction | |
(million)	PBBS	PIP	PBBS	PIP
10	59	49	73	46
20	79	70	141	82
50	142	131	323	198
100	247	205	600	350
200	494	418	1170	680

Table 8: Running times (in milliseconds) of the PBBS algorithms and our new PIP algorithms for list and tree contraction on 72 cores with hyper-threading.

| Core | List contraction | | Tree contraction | |
count	PBBS	PIP	PBBS	PIP
1	13900	3350	22800	12600
4	4370	1110	8020	4370
8	2180	636	3950	2210
18	1010	384	2060	1220
36	592	302	1390	1050
72	362	220	899	512
72HT	247	205	603	350

Table 9: Running times (in milliseconds) of the PBBS algorithms and our new PIP algorithms for list and tree contraction on varying core counts.

9. Since we eliminated the use of the R array in our PIP algorithms and simplified the logic in the implementation (which required changes to the deterministic reservations framework), the parallel execution time improved by 15–20% for list contraction and 60–70% for tree contraction compared to non-in-place version in PBBS by Shun et al. [44]. It is interesting to observe from Figure 6 and Table 9 that the speedups of the PIP algorithms over the PBBS implementations on one thread is larger than on 72 cores with hyper-threading. We conjecture that on one thread, the simpler logic improves prefetching, whereas when using all hyper-threads, prefetching does not help as much due to the memory bandwidth already being saturated.

7.4 Additional Space Usage

Table 10 shows the input size and total memory usage of our PIP algorithms. We see that for our two strong PIP algorithms (scan and filter), the auxiliary space overhead is

Problem	Input size (MB)	Memory usage (MB)	Over-head
Scan	7629.4	7636.2	<0.1%
Filter	7629.4	7636.9	<0.1%
Random permutation	762.9	791.2	3.7%
List contraction	762.9	773.5	1.4%
Tree contraction	1144.4	1154.9	0.9%

Table 10: Memory usage of our algorithms on the five experiments from Figure 2.

negligible (less than 0.1%), and for the three relaxed PIP algorithms (random permutation, list contraction, and tree contraction), the best performance is achieved when the space overhead is between 0.9%–3.7%, which is still much smaller than the input size. We can further reduce the space overhead for the relaxed PIP algorithms at the cost of higher running time (e.g., see Table 7). In contrast, the existing non-in-place algorithms for these problems require additional space proportional to the input size.

8 Conclusion

In this paper, we defined two models for analyzing parallel in-place algorithms. We presented new parallel in-place algorithms for scan, filter, partition, merge, random permutation, list contraction, tree contraction, connectivity, biconnectivity, and minimum spanning forest. We implemented several of our algorithms, and showed experimentally that they are competitive or outperform state-of-the-art non-in-place parallel algorithms for the same problems.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *TCS*, 35(3), 2002.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), Apr 2001.

[3] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. In-place parallel super scalar samplesort (ips4o). In *ESA*, 2017.

[4] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *SPAA*, 2016.

[5] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IPDPS*, 2018.

[6] K. Berney, H. Casanova, A. Higuchi, B. Karsin, and N. Sitchinava. Beyond binary search: parallel in-place construction of implicit search tree layouts. In *IPDPS*, 2018.

[7] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Computers*, 38(11), 1989.

[8] G. E. Blelloch. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[9] G. E. Blelloch, D. Anderson, and L. Dhulipala. ParlayLib—a toolkit for parallel algorithms on shared-memory multicore machines. In *SPAA*, 2020.

[10] G. E. Blelloch, L. Dhulipala, and Y. Sun. Introduction to parallel algorithms 15-853: Algorithms in the real world. 2018.

[11] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *SPAA*, 2016.

[12] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *SPAA*, 2015.

[13] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Efficient algorithms with asymmetric read and write costs. In *ESA*, 2016.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *PPoPP*, 2012.

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, 2011.

[16] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *SPAA*, 2020.

[17] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *SPAA*, 2010.

[18] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. In *SPAA*, 2016.

[19] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *SPAA*, 1991.

[20] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

[21] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE transactions on Computers*, (3), 1982.

[22] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Trans. Parallel Comput.*, 3(4), Mar. 2017.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

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

[25] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, 1991.

[26] X. Guan and M. A. Langston. Parallel methods for solving fundamental file rearrangement problems. *Journal of Parallel and Distributed Computing*, 14(4), 1992.

[27] B.-C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3), 1988.

[28] B.-C. Huang and M. A. Langston. Stable duplicate-key extraction with optimal time and space bounds. *Acta Informatica*, 26(5), 1989.

[29] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[30] 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.

[31] D. E. Knuth. Notes on "open" addressing. 1963.

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

[33] W. Kuszmaul and A. Westover. Brief announcement: Cache-efficient parallel-partition algorithms using exclusive-read-and-write memory. In *SPAA*, 2020.

[34] M. A. Langston. Time-space optimal parallel computation. In *Parallel Algorithm Derivation and Program Transformation*. Springer, 1993.

[35] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *FOCS*, 1985.

[36] O. Obeya, E. Kahssay, E. Fan, and J. Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *SPAA*, 2019.

[37] A. Pietracaprina, G. Pucci, F. Silvestri, and F. Vandin. Space-efficient parallel algorithms for combinatorial search problems. *Journal of Parallel and Distributed Computing*, 76, 2015.

[38] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[39] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[40] R. Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*. 1993.

[41] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *SPAA*, 2014.

[42] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, 2013.

[43] 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 *SPAA*, 2012.

[44] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *SODA*, 2015.

[45] Y. Sun, D. Ferizovic, and G. E. Blelloch. PAM: Parallel augmented maps. In *PPoPP*, 2018.

[46] S. Q. Zheng, B. Calidas, and Y. Zhang. An efficient general in-place parallel sorting scheme. *The Journal of Supercomputing*, 14(1), 1999.