Parallel In-Place Algorithms: Theory and Practice

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 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 nested-parallelism, which better reflect modern parallel programming environments. We then introduce many new parallel in-place algorithms that are simple and efficient both theoretically and practically. The 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, $O(\epsilon n)$ auxiliary space, and $O((1/\epsilon)\text{polylog}(n))$ span for $\epsilon < 1$. We also present new parallel in-place algorithms such as scan, filter, and minimum spanning tree using other techniques.

In addition to theoretical results, we present experimental results for implementations of many new parallel in-place algorithms. We show that on a 72-core machine with two-way hyper-threading, the in-place versions are competitive with or outperform existing parallel algorithms for the same problems that use linear auxiliary space.

ACM Reference Format:

. 2019. Parallel In-Place Algorithms: Theory and Practice. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2020).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SPAA 2020, July 14–17, 2020, Philadelphia, PA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 parallelism, 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 in 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 and span 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 sorting algorithms, even if we know the destination of each element in the final sorted array, it is difficult to directly move them all to their final locations in parallel in the same input array due to conflicts. As a result, parallel algorithms for this task (e.g., [22, 24]) 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 sorting [6], radix sorting [59], and constructing implicit search tree layouts [13]. 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 individual algorithms for specific problems and have different notions of what "in-place" means in the parallel setting. In this paper, we propose two models for the PIP setting, the ***strong PIP model*** and the ***relaxed PIP model*** that generalize the ideas in previous work, which allow polylogarithmic space and sublinear space, respectively. The new models are defined based on nested-parallelism, so not only we restrict the size of auxiliary space, scheduling guarantees based on work-stealing can also be provided, such as the number of steals, parallel running time and I/O-complexity.

We then introduce PIP algorithms based on the two models. Some of them are common practice and they are summarized here, and the cost bounds are given in plain text. The rest are new to the best of our knowledge, and we distinguish them by listing our results in theorems and corollaries. The results of some algorithmic building blocks are summarized in Table 1. The algorithmic highlight in this paper is the Decomposable Property defined in Section 4 The high-level idea is that if we can reduce a problem of size n with work[1] $W(n)$ to a subproblem with size $(1-\epsilon)n$ using $\epsilon \cdot W(n)$ work, then we can have an efficient algorithm on the relaxed PIP model. In many cases, the algorithm for such reduction is the same algorithm (but on a subproblem with smaller size), which means we can convert any existing non-in-place but highly-optimized parallel algorithms to PIP algorithms, which are efficient both theoretically and practically. We show many examples in this paper, such as random permutation, list/tree contraction, merging and mergesort. All these algorithms are simple. For instance, our merging algorithm in Section 4.3 is much simpler than existing work [48] since we can directly use any classic merging algorithm in [51] and turn it into a PIP algorithm. We have also designed other PIP algorithms without using the Decomposable Property, including scan, radixsort, minimum spanning tree, etc.

Model	Problems	Work-efficient	
Strong PIP Model	Permuting tree layout	✓	[13]
	Reduce, rotating	✓	
	Scan (prefix sum)	✓	*
	Filter, partition, quicksort		
	Merging, mergesort		
	Set operations	✓	[16]
Relaxed PIP Model	Random permutation	✓	*
	List/tree contraction	✓	*
	Merging, mergesort	✓	*
	Filter, partition, quicksort	✓	
	(Bi-)Connectivity		[10]
	Minimum spanning tree		*

Table 1: Algorithms based on the strong PIP model and the relaxed PIP model. "Work-efficient" indicates if the PIP algorithm have the asymptotically same work (number of operations) as the best parallel non-in-place algorithm. Algorithms with (*) are new and they highly rely on existing non-in-place parallel algorithms. Other results without citations are either used in practice or require moderate changes from existing algorithms. Merging and mergesort in the relaxed model is discussed in [48] but the new algorithm in this paper is much simpler. If a problem has a work-efficient solution in the strong PIP model, it will not be listed again in the relaxed PIP model.

Since most of the new algorithms in this paper are simple, we carefully engineer five of them, and compare them to the very efficient non-in-place implementation in PBBS [67]. The running time of typical input size is shown in Figure 2 and more details in Section 7. We show that for very simple primitives (e.g., scan, filter), if the input does not need to be kept, we should use the in-place version since they leads to better performance. For more complicated algorithms such as random permutation, list and tree contraction, our new algorithms are highly based on existing non-in-place algorithms. Hence, we show that if we engineer them carefully, the in-place versions can have competitive or even better

[1] The total number of operations.

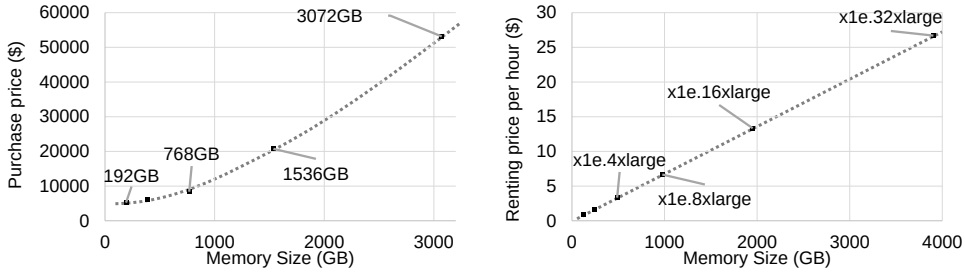

Figure 1: Purchase and rental price for high-end multicore servers. The left figure shows the purchase price of an RAX XT24-42S1 server with 72 CPU cores (Xeon Gold 5220). The DRAM capacity is the decisive part of overall purchase price. The right figure is the rental price of AWS (Amazon Web Services) x1e-series multicore instances vs. the memory capacity.

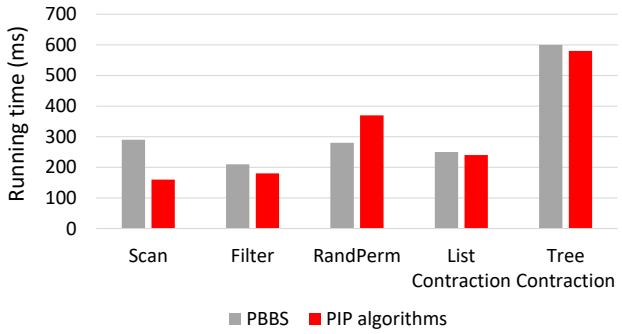

Figure 2: Running time of the PIP algorithms and non-in-place implementations in PBBS [67]. Since most of the PIP algorithms in this paper are simple and practical, we implement some of them and compared them to the PBBS implementations, which are not in-place. For scan and filter the input size is 10^9 and for other problems the input size is 10^8. Running times are on a 72-core machine with two-way hyper-threading, and more details are in Section 7. In all cases, the new PIP algorithms have competitive or better performance, with the additional advantage of using less additional space.

performance. As a result, in addition to smaller auxiliary space, the in-place algorithms can also lead to good performance in many cases due to the smaller memory footprint.

In conclusion, in this paper we bridge the gap between theory and practice for parallel in-place (PIP) algorithms, and the contributions include computational models, new PIP algorithms, algorithm design pattern, implementation and experiment verification. We show that many PIP algorithms can actually be simple and efficient both theoretically and practically. Meanwhile, this paper also leads to interesting future work. For example, most of the algorithms we implemented are in the relaxed PIP model which already have good practical performance, but it is of theoretical interest to design work-efficient algorithms in the strong PIP model.

2 PRELIMINARIES

Work-Span Model. In this paper, we use the classic work-span model for fork-join parallelism in analyzing parallel algorithms [34]. We assume 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 the children terminate. A computation starts with a single root thread and finishes when that root thread finishes. The *work* of an algorithm is the total number of instructions and the *span* (aka. depth) is the longest sequence of dependent instructions in the computation. The Cilk randomized work-stealing scheduler can execute an algorithm with work W and span D in $W/p + O(D)$ time whp[2] on a machine with p processors [5, 26]. Note that since our model is based on binary forking, the span of an algorithm with input size n is $\Omega(\log n)$. Furthermore, when using the Cilk scheduler [25], a fork-join program that uses S_1 space allocated in a stack-allocated fashion[3] when run on one processor will use $O(PS_1)$ space when run on P processors [26].

Problem definitions. Here we define the problems used in multiple places in this paper. Other problems are defined in the corresponding sections. ***Reduce*** takes as input a sequence $[a_1, a_2, ..., a_n]$, an associative binary operator $\oplus$, and an identity element i, and returns $a_1 \oplus a_2 \oplus ... \oplus a_n$. ***Scan*** takes the same input. An inclusive scan returns the ordered set $[a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2 \oplus ... \oplus a_n)]$. An exclusive scan returns $[i, a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2 \oplus ... \oplus a_{n-1})]$, in addition to the sum of all elements. ***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, in the same order as in A. ***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 will appear at the end of the array, in the same order as they appear in A.

3 MODELS FOR PARALLEL IN-PLACE ALGORITHMS

In this section, we will review the existing in-place PRAM model for designing PIP algorithms and describe two new PIP models that are more flexible than the existing model and more accurately reflect modern parallel programming environments.

Existing in-place PRAM model. The existing model for designing parallel in-place algorithms used by most prior work [13, 43, 44,

[2] 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.
[3] Memory allocation in a stack-allocated fashion requires freeing the memory allocated after a fork before the corresponding join.

49, 56, 60, 74] has P processors that are fully synchronized between each step, and the running time of an algorithm is the maximal number of steps T used by any processor. In this model, the space S is the sum of the space used across the processors. An alternate version of this model is defined by Berney et al. [13] based on analyzing the work and span of the algorithm [51], but it is not based on fork-join parallelism and still assumes bulk synchronization between steps. This model requires the computation to use $O(\log n)$-word additional space per processor when mapping to any $P \geq 1$ processors. Algorithms designed based on this variant are strictly stronger than the classic in-place PRAM model since they achieve small auxiliary space and depth simultaneously.

We believe that we should consider PIP models based on fork-join (nested) parallelism for the following reasons. Modern parallel programming languages, such as Cilk and OpenMP, support fork-join parallelism, which significantly simplifies the implementation of parallel algorithms. Provably efficient runtime schedulers have been designed for fork-join parallel programs [5, 27]. Furthermore, modern multicore architectures are not bulk-synchronous, and instead allow parallel processes to run asynchronously. Finally, in environments with multiple jobs running, the number of available processors can vary throughout the computation, which can be handled by the scheduler [5], but this is challenging to handle in the PRAM model.

Therefore, this paper proposes two models for designing parallel in-place algorithms that uses fork-join parallelism. Algorithms designed in our new models are processor-oblivious and can be dynamically scheduled to run efficiently on real-world multicore machines. Existing results on multiprogrammed environments [5], cache complexity [1, 20, 22, 69], write-efficiency [9, 17, 18], cache-adaptivity [11, 12, 57], and resource-obliviousness [30, 31] for fork-join programs apply to PIP algorithms designed in our models.

The strong PIP model. We start by defining the strong PIP model, which resembles the definition of in-place in the sequential setting.

Definition 3.1 (Strong PIP model and algorithms). The *strong PIP model* assumes a nested-parallel computation using $O(\log n)$-word auxiliary space sequentially in a stack-allocated fashion for an input size of n. We say that an algorithm is *strong PIP* if it runs on the strong PIP model and has polylogarithmic span.

For a PIP algorithm in the strong PIP model, the Cilk work-stealing scheduler [25] can bound the total auxiliary space to be $O(P \log n)$ words, where P is the number of processors that the computation uses [26]. All strong PIP algorithms discussed in this paper only uses $O(\log n)$-word auxiliary space sequentially, but we believe that relaxing it to $O(\text{polylog}(n))$ words may also be reasonable.

The strong PIP model is equivalent to Berney et al.'s model [13] when mapping to a PRAM, but our model is based on 2-way fork-join parallelism. Similar to Berney et al.'s model, the algorithms based on the strong PIP model can achieve both small auxiliary space and small span, which are much stronger than previous algorithms [43, 44, 49, 56, 60, 74] in the classic in-place PRAM model. The goal of these algorithms is to achieve space-time optimality [56], where the product of auxiliary space S and PRAM time T is $\tilde{\Theta}(n)$ where n is the input size. This means that these algorithms are either space-efficient but not quite parallel, or the other way around. Achieving low span and low auxiliary space simultaneously can have practical benefits even with a small number of processors—the overhead of scheduling fork-join parallelism in practice is proportional to the span of the computation [5, 26].

The relaxed PIP model. Many algorithms designed in the in-place PRAM model [43, 44, 49, 56, 60, 74] aim to have a good tradeoff between additional space S and span D such that $S \cdot D = \tilde{\Theta}(n)$. Here we abstract this goal as the relaxed PIP model and refer to these algorithms as relaxed PIP algorithms.

Definition 3.2 (Relaxed PIP model and algorithms). The *relaxed PIP model* assumes a nested-parallel computation using $O(\log n)$-word stack-allocated space sequentially and $O(\epsilon n)$ shared (heap-allocated) auxiliary space for an input of size n for $n^{-c} < \epsilon < 1$ for some constant $0 < c < 1$. We say that an algorithm is *relaxed PIP* if it runs on the relaxed PIP model and has $O(\text{polylog}(n)/\epsilon)$ span for all values of ϵ.

The Cilk work-stealing scheduler [25] can bound the total auxiliary space to be $O(\epsilon n + P \log n)$ for p processors [26]. Algorithms in the relaxed PIP model allow sublinear space, which is less restrictive than the strong PIP model. However, the relaxed PIP model provides more flexibility in algorithm design, while still being useful in practice as they 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 easily be converted into a PIP algorithm in the relaxed PIP model.

4 DECOMPOSABLE PROPERTY

Designing strong PIP algorithms can be hard. However, if we relax the auxiliary space to sublinear (the relaxed PIP model), then we believe we can find relaxed PIP algorithms for many more problems. In this section, we introduce the ***Decomposable Property***, which enables any algorithm that satisfies the property to be converted into a relaxed PIP algorithm. Meanwhile, if the existing algorithm is work-efficient, then the corresponding relaxed PIP algorithm will also be work-efficient.

Theorem 4.1 (Decomposable Property). *Consider a problem with size n and a parallel algorithm to solve it with work $W(n) = O(n\,\text{polylog}(n))$. If it can be reduced to a smaller problem size $(1-\epsilon)\cdot n$ using $\epsilon \cdot W(n)$ work and space, and polylogarithmic span $D(n)$, then there is a relaxed PIP algorithm for this problem with $W(n)$ work, $O(\text{polylog}(n)/\epsilon)$ span, and $O(\epsilon n)$ auxiliary space.*

Proof. Let r be the value such that $W(n) = nr$. For any given auxiliary space $O(\epsilon n)$ where $n^{-c} < \epsilon < 1$ for some constant $0 < c < 1$, we can iteratively reduce the problem size by $\epsilon n/r$ (this size remains the same throughout the algorithm for r/ϵ rounds), such that each round takes $O(\epsilon n)$ work and space. We restrict the work $W(n)$ to be $O(n\,\text{polylog}(n))$, so that r is polylogarithmic and the auxiliary space ϵn is always larger than r. This means that we can reduce the problem size by at least 1. By applying this reduction for r/ϵ rounds, we have a relaxed PIP algorithm with $W(n)$ work and $D(n)r/\epsilon = O(\text{polylog}(n)/\epsilon)$ span, using $O(\epsilon n)$ 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 $n - n'$

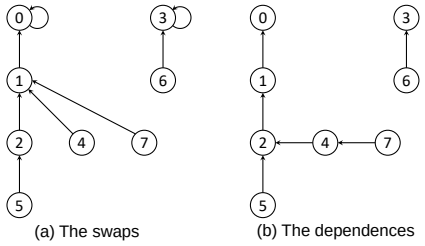

Figure 3: An example when $H = [0, 0, 1, 3, 1, 2, 3, 1]$. Figure (a) indicates the destinations of the swaps shown by H. The dependences of the swaps are shown by Figure (b), indicating the order of the swaps.

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 can provide theoretically-efficient algorithms once they satisfy this property. On the practical side, we observe that this reduction step is 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 algorithm for this step, so that the overall performance for the entire PIP algorithm can be competitive or even better than the non-in-place algorithm. In the rest of this section, we will introduce some problems satisfying this 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 in order to achieve good theoretical guarantees. Hence, random permutations have been well-studied both theoretically [2, 3, 35, 40–42, 45, 47, 58, 61, 68] and experimentally [33, 46, 68]. Sequentially, Knuth [54] (Durstenfeld's [36]) shuffle (shown below) has linear work, where $H[i]$ is an integer uniformly drawn between 0 and $i-1$, and $A[\cdot]$ is the output random permutation.

```
1 Function KNUTH-SHUFFLE(A, H)
2     for i ← n − 1 to 0 do A[i] ← i
3     for i ← n − 1 to 0 do swap(A[i], A[H[i]])
```

A recent work by Shun et al. [68] has shown that this sequential iterative algorithm is readily parallel. The key idea is to allow multiple swaps to be applied in parallel as long as the sets of source and destination locations of the swaps are disjoint. We show an example in Figure 3. In Figure 3(a) we link the sources and destinations of the swaps. In this example, we can swap location 5 and 2, 7 and 1, and 6 and 3 simultaneously in the first round since these three swaps do not interfere with each other. If the nodes pointing to the same node are chained together and the self-loops are removed, we get the dependences of the computation. An example is shown in Figure 3(b). We can execute the swaps for all leaf nodes and remove them from the tree in a round-based manner, and guarantee to finish in $O(\log n)$ rounds *whp* [68]. The pseudocode of this parallel algorithm is shown in Algorithm 1. The work and span can be shown as $O(n)$ expected and $O(\log^2 n)$ *whp*, respectively [68].

Algorithm 1: PARALLEL-KNUTH-SHUFFLE(A, H) [68]

```
1  R ← {−1, ..., −1}
2  parallel for i ← n − 1 to 0 do A[i] ← i
3  while swaps unfinished do
4      parallel foreach swap (i, H[i]) do
5          R[i] ← max(R[i], i)
6          R[H[i]] ← max(R[H[i]], i)
7      parallel foreach swap (i, H[i]) do
8          if R[i] = i and R[H[i]] = i then swap(A[H[i]], A[i])
9      Reset R and pack the leftover swaps
10 return A
```

We now show the **Decomposable Property** of random permutation. The property for the sequential algorithm is easy to see—after the first ϵn swaps, which we refer to as one **round**, the problem reduces to a subproblem of size $(1-\epsilon)n$ which can be solved using the same algorithm. The parallel algorithm (Algorithm 1) uses an additional array R, and each swap reserves the two slots in R. If a swap successfully reserves the two slots, then it can perform the actual swap, and otherwise it will wait until the next round to try again. We note that for the first ϵn swaps in a round, the overall access for R and H arrays requires $3\epsilon n$ locations ($H[i]$, $R[i]$ and $R[H[i]]$). Theoretically, we can use a parallel hash table to store these values using $O(\epsilon n)$ space. When the load factor of the hash table is no more than a half, each update or query requires $O(1)$ expected cost and $O(\log n)$ *whp* [53, 66]. The longest dependence length among the first ϵn swaps in a phase is bounded by $O(\log n)$ since it cannot be longer than the overall dependence length for all n swaps, which is bounded by $O(\log n)$. The overall span in a phase is $O(\log^2 n)$, with the additional factor of $\log n$ due to hash table insertions queries. The entire algorithm finishes after $1/\epsilon$ rounds and is work-efficient. By applying 4.1, we obtain the following corollary.

COROLLARY 4.2. *There is a relaxed PIP algorithm for random permutation using $O(n)$ expected work, $O((\log^2 n)/\epsilon)$ span* whp, *and $O(\epsilon n)$ auxiliary space.*

Constant-dimension linear programming and smallest enclosing disks. Based on the relaxed PIP algorithm for random permutation, it is straightforward to design the relaxed PIP algorithms for these two problems based on the randomized incremental construction [23, 65]. The randomized algorithms after randomly permuting the input elements takes $O(d \log n)$ space (words), where d is the dimension. By replacing the random permutation to the above algorithm, we can get relaxed PIP algorithms for constant-dimension linear programming and smallest enclosing disks in $O(d!n)$ expected work and $O(\epsilon n)$ auxiliary space with $O((\log^2 n)/\epsilon)$ span *whp*.

4.2 List/Tree Contraction

List ranking [4, 7, 32, 50–52, 62, 63, 71–73] is one of the most canonical problems in the study of parallel algorithms. The problem is given a set of linked lists, to return to each element in each list its position in the list. The problem is fundamental because it has many applications as a subroutine in other algorithms, while the problem

Algorithm 2: List-Contraction(L) [68]

Input: A doubly-linked list L of size n. Each element l_i has a random priority $p(l_i)$.

```
R ← {0, ..., 0}
while element left do
    parallel foreach uncontracted element i do
        if p(l_i) < p(prev(l_i)) or p(l_i) < p(next(l_i)) then
            R[i] ← 1
    parallel foreach uncontracted element i do
        if R[i] = 1 then Splice element i out and update
          pointers
    pack the leftover (uncontracted) elements
return A
```

itself seems inherently sequential. 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 also work on ϵn elements in a round, and apply existing parallel list contraction algorithms [4, 7, 32, 51, 52, 62, 63, 71–73] to contract these ϵn elements. These algorithms contract a list but can be easily adapted to rank the list by adding a second phase that expands the contracted nodes in a reverse order. Shun et al. [68] showed a simple randomized algorithm (Algorithm 2) [68], which takes linear work and $O(\log^2 n)$ span *whp*. For a problem of size n, we can work on ϵn elements and contract them using the algorithm by Shun et al., which requires $O(\epsilon n)$ space and expected work, and $O(\log^2 n)$ span *whp* (no more than the span for n elements). Then the problem reduces to a subproblem with size $(1-\epsilon)n$. We can iteratively apply this for $1/\epsilon$ rounds (fixing the number of elements in each round), and get an relaxed PIP algorithm for list contraction.

Tree contraction can be considered as a generalization of list contraction and has ample applications for many tree and graph applications [9, 51, 58, 64, 68]. Here we will assume we are contracting rooted binary trees in which every internal node has exactly two children. The **Decomposable Property** for tree contraction holds similarly since 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 ϵn tree nodes in each round and contract them using existing algorithms, and repeat for $1/\epsilon$ rounds. We can use the parallel tree contraction algorithm by Shun et al. [68] that requires linear expected work and $O(\log^2 n)$ span *whp* per round.

Corollary 4.3. *There are relaxed PIP algorithms for list contraction and tree contraction using $O(n)$ expected work, $O((\log^2 n)/\epsilon)$ span* whp*, and $O(\epsilon n)$ auxiliary space.*

4.3 Merging and Mergesort

Merging two sorted arrays of size n (stored in an array of size $2n$) is another canonical primitive in parallel algorithm design. The in-place merging algorithms have been studied in the sequential and the PRAM settings [44, 48], although they are very complicated. There are lots of sophisticated subtleties in these algorithms in maintaining constant space in total or per processor on a PRAM.

However, by viewing the algorithm in the relaxed PIP model and using the Decomposable Property, we can actually base our algorithms on existing non-in-place merging algorithm and add some details from the sequential in-place algorithm [48]. The key idea in [48] to merge in-place is to cut both input arrays into chunks of size k, and sort the chunks based on the last elements of the chunks. Then we can merge the first chunks from both input arrays until the elements in one chunk is used up, and grab the next chuck for that input array and continue.

To get a relaxed PIP algorithm, we set the chunk size to be $k = \epsilon n$ and we also have $O(\epsilon n)$ auxiliary space. With this space, we can use out-of-place algorithm to merge a prefix of size $O(\epsilon n)$ and repeat for $O(1/\epsilon)$ rounds. To start, we also sort the chunks based on the last elements, and move each chunk to their final destination in parallel by using the $O(\epsilon n)$ auxiliary space as a buffer, which will take $O((\log n)/\epsilon)$ span to finish sorting the chunks. In the merging phase, we move the two chunks from both arrays to the auxiliary space, use any existing algorithm to merge them until either we run out of the elements in one chunk (and we load the next chunk to the auxiliary space), or we gather a full chunk of merged numbers (so we write it back to the original array). At any time, there can be at most three chunks in the auxiliary space—two chunks for input arrays and one for the merged output, so the required auxiliary space is $O(\epsilon n)$. We can use any existing out-of-place algorithms [51] to merge in the auxiliary space that takes linear work and logarithmic span. Such merging happens for at most $2n/k$ times (after loading new chunks to auxiliary space) plus $2n/k$ times (when the output chunk is full and needs to flush), and so the overall span is $O((\log n)/\epsilon)$.

Corollary 4.4. *Merging two list of size n uses $O(n)$ work, $O((\log n)/\epsilon)$ span, and $O(\epsilon n)$ auxiliary space.*

Here when $\epsilon < n^{-1/2}$, we do not have sufficient auxiliary space to sort all $2/\epsilon$ chunks at the beginning, and so we can sort ϵn chunks at a time and repeat when they are used up. This will not affect the cost bounds. With the relaxed PIP merging algorithm, a mergesort algorithm can be relaxed PIP with $O(n \log n)$ work, $O(\epsilon n)$ auxiliary space, and $O((\log^2 n)/\epsilon)$ 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 ϵn using linear work and logarithmic span, and repeat for $1/\epsilon$ rounds. The only additional work is to move the filtered elements to the beginning of the array, which requires a parallel for-loop. Therefore, a relaxed PIP algorithm for filter takes $O(n)$ work, $O((\log n)/\epsilon)$ span, and $O(\epsilon n)$ auxiliary space. We can implement partition similarly, but instead of moving the filtered elements to the beginning, we swap the elements so that at the end of the algorithm, the rest of the elements are moved to the end of the array. This algorithm has the same cost as filter, but the partition result is not stable. With the partition algorithm, we can have a relaxed PIP algorithm for quicksort with $O(n \log n)$ expected work and $O((\log^2 n)/\epsilon)$ span *whp*, using $O(\epsilon n)$ auxiliary space.

5 STRONG PIP ALGORITHMS

The strong PIP model is restrictive because of the polylogarithmic requirement for auxiliary space. To date, only a few non-trivial and work-efficient strong PIP algorithms have been proposed (e.g., reduce and rotating an array as trivial results, and certain fixed permutations [13]). In this section, we will review existing strong PIP algorithms for reduce and rotation, and present new algorithms for scan (prefix sum), filter, merging, and sorting.

Reduce returns the sum based on associated binary operator $\oplus$ of a given array of size n. The classic divide-and-conquer algorithm for reduce is already strong PIP. It is implemented by dividing the input array by two equal size subarray, recursively summing them in parallel, and finally summing together the partial sums from the two subproblems. Sequentially running this algorithm takes $O(\log n)$ stack space, and so it is an optimal strong PIP algorithm, using optimal $O(n)$ work and $O(\log n)$ span.

Rotating an array. Given an array $[a_1, a_2, ..., a_n]$ and an offset o, the output is $[a_{o+1}, \ldots, a_n, a_1, \ldots, a_o]$. 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. This strong PIP algorithm is also optimal, and requires $O(n)$ work and $O(\log n)$ span.

5.1 Scan (Prefix Sum)

Scan (prefix sum) is probably the most fundamental algorithmic primitive in parallel algorithm design. ***Scan*** takes the input of an ordered set $[a_1, a_2, ..., a_n]$, associative binary operator $\oplus$, and an identity i, and returns $[i, a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2 \oplus ... \oplus a_{n-1})]$, in addition to the total sum of all elements. Here we assume $\oplus$ is + (addition) for simplicity, but all results in this section apply to general cases. The non-in-place versions and implementations have been introduced since the last century, and the work-efficient version is generally referred to as Blelloch scan [15]. Blelloch scan contains two phases. The first phase is referred to as the "up-sweep". The algorithm is recursive, and for every subproblem it partitions the range into two halves and computes the two partial sums recursively, then uses the two partial sums to calculate the sum for the subproblem, and finally stores it in auxiliary space. Then the algorithm applies another "down-sweep" that propagates the sums down to each node—for a subsequence, we recursively solve the left half, and the right half plus the sum of the left half, in parallel. This algorithm has $O(n)$ work and $O(\log n)$ span, but unfortunately, it requires linear auxiliary space to store all of the partial sums.

Simple fixes to the classic algorithm. There are a few ways to make Blelloch scan in-place. For example, we can partition the array into two equal-size chunks, recursively solve them, and apply a parallel for-loop to add the sum of the left chunk to every element in the right chunk. 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 $n/2$ additions. We can reduce the overhead by stopping the recursion when we reach a subsequence of size no more than $\log_2 n$ (base cases). We then apply sequential scan for base cases and store the sum at the end. We then scan on the sum of the base cases, which takes linear 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 chunk to get the final scan result. This algorithm uses $O(n)$ work, $O(\log^2 n)$ span, and $O(\log n)$ auxiliary space.

Algorithm 3: In-Place-Scan

Input: An integer array A 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  σ ← A_n
3  Down-Sweep(A, 1, n, 0)
4  return (A, σ)
5  Function Up-Sweep (array A, s, t)
6      if s = t then return
7      In parallel:
8          Up-Sweep(A, s, ⌊(s + t)/2⌋)
9          Up-Sweep(A, ⌊(s + t)/2⌋ + 1, t)
10     A_t ← A_t + A_⌊(s+t)/2⌋
11 Function Down-Sweep (array A, s, t, p)
12     if s = t then A_s = p, return
13     In parallel:
14         Down-Sweep(A, s, ⌊(s + t)/2⌋, p)
15         Down-Sweep(A, ⌊(s + t)/2⌋ + 1, t, p + A_⌊(s+t)/2⌋)
```

Another way is to use the "Brent-Kung adder" [28] which is a circuit to compute prefix sum 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 in 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 $O(\log n)$ for forking off the tasks. Also, the output of the original circuit is inclusive (i.e., the output is $[a_1, ..., (a_1 \oplus a_2 \oplus ... \oplus a_n)]$), but this can be changed to the exclusive version by using additional work and space. Inspired by this algorithm, we design the following strong PIP scan algorithm with optimal work and span bounds.

A new optimal strong PIP algorithm. Similar to Blelloch scan, our new algorithm as shown in Algorithm 3 contains two phases, the up-sweep and the down-sweep phases, and both of which are recursive. The key insight in our new algorithm is to maintain all intermediate results on the array of n elements (the input array), similar to the circuit, but use the stack space in the down-sweep to pass down the desired value. For each recursive problem 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 recurse and then add the value in index k to t. These additions are shown as arrows in the left side of Figure 4. In the down-sweep phase, we keep the prefix sum p of each subproblem. Similar to Blelloch scan, we add the sum of the left subproblem to the prefix sum. Both recursions stop when $i = j$. An illustration of this algorithm is shown in Figure 4.

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, until it is

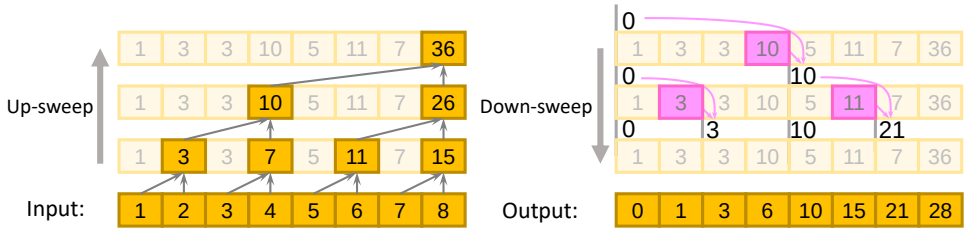

Figure 4: A new strong PIP scan algorithm that has $O(\log n)$ span. It also has an "up-sweep" phase and a "down-sweep" phase. Each pair of arrows pointing to the same element indicates an add.

the sum of the left recursion and will never be rewritten thereafter. By doing so, we squeeze the useful part of a reduce tree, with size exactly n, into the input array, and expand them back to the prefix sum by using p in the stack space. Hence, the new strong PIP scan algorithm uses $O(n)$ work, and $O(\log n)$ span and $O(\log n)$ sequential auxiliary space. The span is optimal since doing $O(n)$ operations requires $\Omega(\log n)$ span.

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.*

Note that compared to the circuit-based algorithms, our new algorithm uses the stack space to pass the prefix sum down to each recursive subproblem, so Algorithm 3 are not required to be in a 2-way recursion. In fact, picking a slightly larger fan-out can be helpful in practice to reduce the overhead caused by function calls since the recursion tree will be shallower.

5.2 Other Strong PIP Algorithms

Filter, unstable partition, and quicksort Consider a k-way divide-and-conquer algorithm for filter—equally partition the array into k chunks, filter each chunk, and pack the filtered results together. For one level of recursion, it takes linear work and $O(k \log n)$ span for packing if we sequentially work on each chunk, but within each chunk 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 $O(n \log_k n)$ and the span is $O(k \log n)$ per level for a total of $O((k/\log k)\log^2 n)$ span. In theory we can plug in k as any constant, which gives a strong PIP algorithm with $O(\log^2 n)$ span and $O(\log n)$ auxiliary space, although it is not work-efficient. In practice, we can guarantee work-efficiency by picking $k = n^\epsilon$ where n is the input size. This does not achieve polylogarithmic span but has good performance in practice. Similar to Section 4.4, we can use this filter algorithm to implement an unstable partition algorithm (with the same cost bounds as filter), and a quicksort algorithm that applies partition for $O(\log n)$ recursive levels *whp*.

Merging and mergesort Now, let us consider merging two sorted arrays of size n (stored in an array of size $2n$). Again, we can use a 2-way divide-and-conquer approach—use a dual binary search to find the median of all $2n$ elements, and in parallel swap the out-of-place elements in two arrays, and recursively merge the two subproblems each with overall size n. We note that in the recursive subproblems, the two input arrays do not necessarily have the same size. The recursion depth is $\log_2 n$, and the work to swap elements in each level is upper bounded by $O(n)$. A mergesort can be implemented accordingly with the merging algorithm. Therefore, we have strong PIPs algorithm for merging and mergesort with polylogarithmic span and auxiliary space although again they are not work-efficient.

Set operations We now consider computing the union, intersect, and difference of two sets of size n and $m < n$. If the two sets are given in a tree format, then the existing algorithms [16, 21, 70] are already strong PIP and work-optimal ($O(m \log(n/m + 1))$ work). If the sets are given in an array, and the output is also a consecutive array, then the work is $\Omega(n + m)$. We can use filter and merging to implement set operations in this case, so they can be strong PIP. However, the resulting algorithms are not work-efficient, since our strong PIP filter and merging algorithms are not work-efficient.

6 OTHER RELAXED PIP GRAPH ALGORITHMS

In this section, we will introduce several relaxed PIP graph algorithms, including graph connectivity, biconnectivity, and minimum spanning tree/forest.

Designing graph algorithms for the strong PIP model is hard since the input graph $G = (V, E)$ contains the vertex set and edge set, but the output is some other information related to the graph and the output size is usually proportional to the vertex set size (e.g., single-source shortest distances, spanning tree, and strongly connected components). Therefore, it does not seem reasonable to represent the output only using polylogarithmic space. The exception is s-t connectivity, which has been studied in the sequential in-place setting [8, 29, 38, 39, 55], but these random-walk based algorithms are inherently sequential.

On the other hand, it seems reasonable to study graph algorithms in the relaxed PIP model if we do not require the output to be explicitly written out. In this case, we could design an algorithm that uses $O(\epsilon n)$ space and for arbitrary $\epsilon < 1$, and returns a data structure from which the output of the graph algorithm can be obtained with additional work.

6.1 Connectivity and Biconnectivity

A recent work by Ben David et al. [10] introduced a compressed scheme for graph connectivity information. The standard output size for graph connectivity and biconnectivity is $O(n)$ and $O(m)$ respectively. The new results in [10] requires $O(k \log n + m/k)$ output size with an $O(k)$ and $O(k^2)$ expected query cost respectively. The cost to construct such a compressed (bi)connectivity oracle takes $O(km)$ expected work and $O(k^{3/2}\log^3 n)$ span. Hence, by applying $k = 1/\epsilon$, graph (bi)connectivity uses $O(\epsilon m)$ auxiliary space, $O(m/\epsilon)$ expected work, and $O(\log^3 n/\epsilon^{3/2})$ span for $n^{-1/2}\log n < \epsilon < 1$. This algorithm is almost relaxed PIP other than a factor of $O(\epsilon^{-1/2})$ more for the span bound. The algorithm is not work-efficient as

compared to the non-in-place version, but the work-space tradeoff holds even for the simpler *s-t* connectivity in the sequential setting [37] (arbitrary (bi)connectivity is strictly harder than *s-t* connectivity).

The high-level idea in these algorithms is to select a subset of the vertices as the "centers" with probability $1/k$ and only keep information for these center vertices. 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 is expected to search for $O(k) = O(1/\epsilon)$ vertices. Clearly for connectivity, v has the same output value as c. For biconnectivity, we need additional local analysis to get the output for v from c, which requires $O(k^2)$ work.

6.2 Minimum Spanning Tree/Forest

The previous idea of implicit decomposition can be extended to minimum spanning tree (MST) or forest (MSF), and we will introduce it in this paper. For simplicity we assume the graph is connected, and we can use the similar approach in [10] for disconnected cases.

The difference between MST and spanning tree is that MST is unique for a graph (break ties consistently). Therefore, for a query to vertex v, instead of using a BFS to find the center, we need to search out to a center using the MST edges. This can be achieved by using a Prim's-like best-first search algorithm from v that requires a priority queue rather than a FIFO queue. This gives an additional $O(\log k)$ term for the work to compute the implicit decomposition of the graph (the queue will contains $O(k)$ vertices on average for each search).

Based on the decomposition, we can compute the "spanning tree" for the m/k clusters that each contains one center and a few other vertices. The output size of this spanning tree is $O(m/k)$. To compute the spanning tree in parallel, we can use Borůvka's algorithm—enumerating all edges for $O(\log n)$ rounds until the entire graph is connected. In each round, we run classic Borůvka's algorithm to find the minimum outgoing edges from each connected component, which takes $O(mk \log k)$ work—$O(k \log k)$ work to find the cluster of each endpoint of an edge for all m edges. By setting $k = 1/\epsilon$, we the following theorem.

Theorem 6.1. *Given a graph with n vertices and m edges, the MST/MSF can be computed using $O((m \log n \log 1/\epsilon)/\epsilon)$ expected work, $O(\epsilon n)$ auxiliary space, and $O(\text{polylog}(n) \log 1/\epsilon)/\epsilon)$ span.*

A more detailed analysis for correctness and cost bounds can be found in the full version of this paper.

7 IMPLEMENTATION AND EXPERIMENTS

In the previous sections we show parallel in-place algorithms with good theoretical guarantees. We note that many new algorithms in this paper are simple and the motivations for PIP algorithms are due to practical concerns. Hence, in this section we discuss how to implement these algorithms efficiently so that they can outperform or at least be competitive to the non-in-place algorithms. Here we discuss implementations for five algorithms: scan, filter, random permutation, list contraction, and tree contraction. The implementations for the first two are fairly straightforward, and the last three are based on deterministic reservation [19].

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. Parallelism is supported by Cilk Plus.

7.2 Scan and Filter

For scan, we implement Algorithm 3 and coarsen at subproblem size 256 (switch to sequential scan, the threshold barely affect the running time). We implement the PIP filter algorithm in Section 5.2, but we keep the implementation work-efficient by setting the branching factor k as $\sqrt{n}$ and apply one round of recursion. We compare to the non-in-place versions in PBBS in which scan is the classic implementation [14] and filter is similar to our implementation but the output is in a separate array. In PBBS filter, it packs each $\sqrt{n}$ chunk to the output array and packs again across the chunks. We note that PBBS does this is because PBBS is a library so that it provides the non-destructive implementations. The reason for such comparison is to show that if we allow rewriting the input, then there is a significant performance gain due to smaller memory footprint.

The running time and scalability (relative speedup) for scan and filter are shown in Figure 6. For filter, 50% entries are in the output. The in-place version of scan is about 50% faster consistently due to smaller memory footprint. Filter is about 15% faster, which is less than scan since the output size is smaller so the save of memory footprint is also less. This experiment indicates that using the in-place version can improve the performance and should be used if the input does not need to be kept.

7.3 Deterministic Reservations

As mentioned in Section 4, algorithms designed based on the Decomposable Property can use the existing highly optimized non-in-place implementations. Our random permutation, list and tree contraction implementations are based on deterministic reservations, which is a framework introduced by Blelloch et al. [19]. Shun et al. [68] implemented the fastest non-in-place algorithms for these three problems.

This framework gives a framework for iterates 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. Algorithm 1 and 2 are shown in such a framework that the first parallel for-loop writes to locations in shared data structure R corresponding to the steps (iterates). After synchronizing, then the second parallel for-loop checks whether an iterate can execute in the commit phase. Iterates that fail to execute will be automatically packed by the framework and retry in the next round. This is repeated until no iterates remain.

To achieve the best practical performance, instead of try all iterates simultaneously, the framework only works on a prefix of all the iterates. After each round, the failed iterates are packed and new iterates are added so that sufficient tasks are processed in the

next round. In practice, we pick the prefix containing 2% of the overall elements. This gives a nice trade-off between extra work and parallelism by adjusting the size of the prefix, and at the same time it naturally meets our requirement for controlling the execution size in the relaxed PIP algorithm.

For PIP algorithms, we usually need an additional phase in each round, which we refer to as the *cleaning phase*. In classic parallel algorithms based on deterministic reservations, we only initialize the reservation array R at the beginning of the algorithm (line 1 in Algorithm 1 and 2). For PIP algorithms, we need to clean the data and reuse the space for the next round. In practice, we will discuss how to clean the data since different approaches of this phase lead to significant varied performance.

7.4 Random Permutation

We implement the PIP random permutation algorithm (Algorithm 1) based on deterministic reservation. We have four implementations (`RP-Naïve`, `RP-Flat`, `RP-OneRes`, and `RP-Final`) and compare it to the best non-in-place counterpart (`RP-PBBS` in the PBBS). We test their performance on inputs of 10 million to 1 billion 32-bit integers. The actual and relative running time is shown in Figure 5.

`RP-Naïve`. The framework of deterministic reservations allows us to work on a prefix of all active iterates. We note such prefix-based executing patterns are what we need for the PIP random permutation algorithm discussed in Section 4. Hence, we can directly use parallel hash tables to replace the auxiliary arrays R and H, and we refer to this implementation as this algorithm `RP-Naïve`. Unfortunately, compared to `RP-PBBS`, `RP-Naïve` has poor performance (2.9–3.5x overhead shown in Figure 5). We now discuss some of our preliminary attempts to improve the performance.

`RP-Flat`. Our first attempt is to use an array to replace the hash table to store the H array. We note that in Algorithm 1, the access of $H[i]$ is always associated i (so as the R array in Algorithm 2), which corresponds to each iterate (swap) in this round. We modify the code of deterministic reservations such that it also provides the index of each iterate in the overall list of active iterates. In this way we can store the value of $H[i]$ in a consecutive array. We refer to this implementation as `RP-Flat`, and it reduces the runtime overhead to about 2–2.5x over `RP-PBBS`.

`RP-OneRes`. `RP-PBBS` uses two reservations (Line 5 and Line 6 in Algorithm 1). Our second attempt is to use just one reservation. The reason that `RP-PBBS` does so is that in the commit phase, the algorithm can immediately reset $R[i]$ and $R[H[i]]$ if their value is i. We note that such an advantage for two reservations does not hold when using hash tables to store R. When using a hash table for R, it is too costly to directly reset $R[i]$ and $R[H[i]]$ in the commit phase. This would correspond to deletions in the hash table and deletions in concurrent hash tables are very slow, so it might be more efficient to erase the entire hash table at the end of each round. If so, we can instead apply just one reservation in Line 6, and modify the if-condition in Line 8 to $(R[i] = \perp \parallel R[i] = i)$ && $R[H[i]] = i$. Here $\perp$ indicates the initialized value of hash table, meaning that key i is not found in the hash table. We refer to this implementation as `RP-OneRes`. `RP-OneRes`'s runtime overhead is about 1.5–2x over `RP-PBBS`.

`RP-Final`. Our third attempt is to use a consecutive array for $R[i]$, similar to the approach for $H[i]$. Unfortunately, this is not applicable since we do not have the inverse mapping from iterate (swap) i to the index of its swap in the active list of all swaps in this round. However, we know that a large portion of the index is consecutive—in particular, all newly-added swaps in a round will be consecutive. Hence, we modify the code for deterministic reservations such that the framework provides the range of the consecutive indexes and the range of the keys. For each access to R, we first check if the key falls in the range. If so we map it to the associated index and reserve R in the array. Otherwise, we insert it to the hash table. We refer to this implementation as `RP-Final`, and it is about 30–40% slower than `RP-PBBS`.

Performance and analysis The actual and relative running time of all five implementations is shown in Figure 5. All of them have similar and consistent scalability, and the relative performance, shown in the right part of the figure, indicates the speedup of each optimization. By applying three optimizations to `RP-Naïve`, `RP-Final` only has a modest overhead of 30–40% over `RP-PBBS` while only using 10% auxiliary space.

All of the above implementations take $O(n)$ work, and the actual performance is largely decided by the number of random memory accesses of the program. To better understand the performance, we list the approximate number of serial and random accesses per swap per round in Table 3. We can see the performance curves almost match the number of random accesses per swap per round, and the overhead of `RP-Final` is in computing hash functions, linear probing, and cleaning the hash tables. The overall additional space for `RP-Final` is about 10% when input is 4-byte integers (smaller for larger items). In Table 4, we show the tradeoff between running time and the additional space.

7.5 List and Tree Contraction

Similar to random permutation, for list and tree contraction, we can base our PIP implementation on the highly-optimized non-in-place implementations in PBBS [67], which is the implementation of the algorithms in [68]. In this case, all accesses in Algorithm 2 and tree contraction is $R[i]$. Instead of using a hash table to maintain R, we note that for a prefix of size p, we can allocate a boolean array of size p and modify the framework for deterministic reservation and pass the index of this iteration in the prefix. In the commit and reserve phase, we can directly modify the value in this boolean array, so that the auxiliary space is limited to p and can be tuned. To achieve the best performance, p is set to be $n/50$ in our experiment, which is small compared to the input size. In fact, this implementation is more efficient than the PBBS version since overall it decreases the memory footprint.

Similar to the other algorithms, we test the performance from 10 million to 200 million entries that each contains two 64-bit pointers. Such input does not contain the data to be computed, but adding that will increase the same running time for all implementations. The running time is shown on the left side in Figure 6. We also fix the input size to be 100 million and test the relative speedup when varying the number of processors. The in-place versions are about 5% faster in a consistent way in almost the entire parameter space. The speedup is from the save of memory footprint, but the cost of

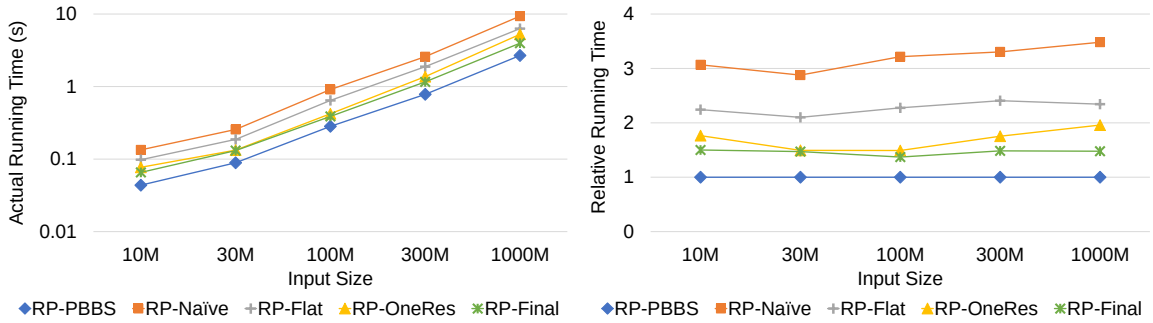

Figure 5: The actual and relative running time of different implementations for random permutation. The input are integers with size vary from 10 million to 1 billion. The running times in the right figure is normalized to RP-PBBS.

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 2: Wall-clock running time of the five implementations, shown in milliseconds.

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 3: Number of serial/random access per swap per round (approximately).

Additional space	1%	2%	4%	10%
Running time (ms)	537	425	411	388

Table 4: Running time with different restrictions on additional space. The input is 100 million integers.

accessing the array R in both versions is small since the majority of the cost is in checking the previous and next element, which incurs random access and is more costly.

8 CONCLUSION

In this paper we study the theory and practice of parallel in-place (PIP) algorithms, which is an area that was not paid attention previously but is getting emphasized recently. Inspired by recent work, we defined two models for designing PIP algorithms—the strong PIP model and the relaxed PIP model, and both are based on nested-parallelism. In addition to restricting auxiliary space, PIP algorithms designed based on the new models have scheduling guarantees such as the number of steals, parallel running time, I/O-efficiency.

The key algorithmic contribution in this paper is the Decomposable Property that bridges the PIP setting and the classic non-in-place setting. Based on the Decomposable Property, we have designed a list of simple and efficient PIP algorithms based on the existing parallel algorithms. Meanwhile, we have also shown many other new PIP algorithms in both of our models. Due to the simplicity of the new PIP algorithms, we implemented five of them and they all have competitive or even better performance than existing implementations. We have also mentioned many new open problems in this paper yet to be explored, especially for the strong PIP model.

REFERENCES

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

[2] Laurent Alonso and Ren Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science (TCS)*, 159(1), 1996.

[3] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1990.

[4] Richard J Anderson and Gary L Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

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

[6] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (ipsssso). In *25th Annual European Symposium on Algorithms (ESA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[7] Sara Baase. Introduction to parallel connectivity, list ranking, and Euler tour techniques. In John Reif, editor, *Synthesis of Parallel Algorithms*, pages 61–114. Morgan Kaufmann, 1993.

[8] Paul Beame, Allan Borodin, Prabhakar Raghavan, Walter L Ruzzo, and Martin Tompa. A time-space tradeoff for undirected graph traversal by walking automata. *SIAM Journal on Computing*, 28(3):1051–1072, 1998.

[9] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[10] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IPDPS*, 2018.

[11] Michael A Bender, Erik D Demaine, Roozbeh Ebrahimi, Jeremy T Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 135–144, 2016.

[12] Michael A Bender, Roozbeh Ebrahimi, Jeremy T Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. Cache-adaptive algorithms. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 958–971. SIAM, 2014.

[13] Kyle Berney, Henri Casanova, Alyssa Higuchi, Ben Karsin, and Nodari Sitchinava. Beyond binary search: parallel in-place construction of implicit search tree layouts. pages 1070–1079. IEEE, 2018.

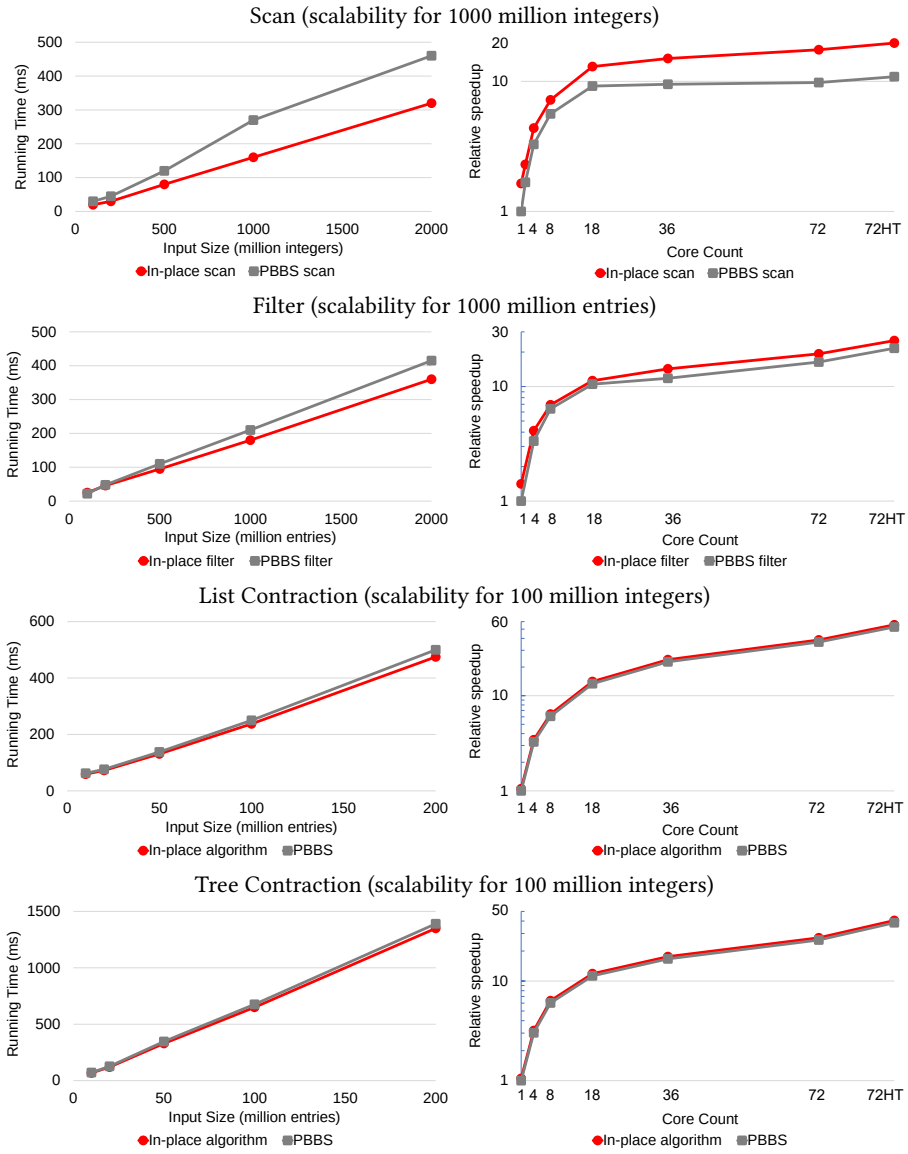

Figure 6: Running time and relative speedup for scan, filter, list and tree contraction. The left figures show the running time of the in-place version and the PBBS implementation for varying input sizes. The right figures show the relative speedup compared to sequential PBBS implementations, and the core count varies from 1 to 72 with hyperthreading.

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

[15] Guy E. Blelloch. Prefix sums and their applications. In John Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[17] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–12, 2015.

[18] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, 2016.

[19] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Notices*, 2012.

[20] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[21] Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. *arXiv preprint arXiv:1903.04650*, 2019.

[22] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[23] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *SPAA*, 2016.

[24] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1991.

[25] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1995.

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

[27] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), September 1999.

[28] Richard P Brent and Hsiang T Kung. A regular layout for parallel adders. *IEEE transactions on Computers*, (3):260–264, 1982.

[29] Andrei Z Broder, Anna R Karlin, Prabhakar Raghavan, and Eli Upfal. Trading space for time in undirected s-t connectivity. *SIAM Journal on Computing*, 23(2):324–334, 1994.

[30] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *International Colloquium on Automata, Languages, and Programming*, pages 226–237. Springer, 2010.

[31] Richard Cole and Vijaya Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 201–214. IEEE, 2012.

[32] Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *ACM Symposium on Theory of Computing (STOC)*, pages 206–219, 1986.

[33] Guojing Cong and David A. Bader. An empirical analysis of parallel random permutation algorithms on SMPs. In *Parallel and Distributed Computing and Systems (PDCS)*, 2005.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[35] Artur Czumaj, Przemyslawa Kanarek, Miroslaw Kutylowski, and Krzysztof Lorys. Fast generation of random permutations via networks simulation. *Algorithmica*, 1998.

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

[37] Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for st-connectivity on the nnjag model. *SIAM Journal on Computing*, 28(6):2257–2284, 1999.

[38] Jeff A Edmonds. Time–space tradeoffs for undirected st-connectivity on a graph automata. *SIAM Journal on Computing*, 27(5):1492–1513, 1998.

[39] Uriel Feige. A spectrum of time–space trade-offs for undirecteds-tconnectivity. *Journal of Computer and System Sciences*, 54(2):305–316, 1997.

[40] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3), 1996.

[41] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[42] Joseph Gil. Fast load balancing on a PRAM. In *Symposium on Parallel and Distributed Processing*, 1991.

[43] Xiaojun Guan and Michael A Langston. Parallel methods for solving fundamental file rearrangement problems. *Journal of Parallel and Distributed Computing*, 14(4):436–439, 1992.

[44] Xiaojun Guan and MS Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, (5):596–602, 1991.

[45] Jens Gustedt. Randomized permutations in a coarse grained parallel environment. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2003.

[46] Jens Gustedt. Engineering parallel in-place random generation of integer permutations. In *Workshop on Experimental Algorithmics*, 2008.

[47] Torben Hagerup. Fast parallel generation of random permutations. In *International Colloquium on Automata, Languages and Programming*. Springer, 1991.

[48] Bing-Chao Huang and Michael A Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, 1988.

[49] Bing Chao Huang and Michael A Langston. Stable duplicate-key extraction with optimal time and space bounds. *Acta Informatica*, 26(5):473–484, 1989.

[50] Riko Jacob, Tobias Lieber, and Nodari Sitchinava. On the complexity of list ranking in the parallel external memory model. In *International Symposium on Mathematical Foundations of Computer Science*, pages 384–395. Springer, 2014.

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

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

[53] Don Knuth. Notes on "open" addressing. 1963.

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

[55] Adrian Kosowski. Faster walks in graphs: a (n^2) time-space trade-off for undirected st connectivity. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1873–1883. SIAM, 2013.

[56] Michael A Langston. Time-space optimal parallel computation. In *Parallel Algorithm Derivation and Program Transformation*, pages 207–223. Springer, 1993.

[57] Andrea Lincoln, Quanquan C. Liu, Jayson Lynch, and Helen Xu. Cache-adaptive exploration: Experimental results and scan-hiding for adaptivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 213àĂŞ222, 2018.

[58] G.L. Miller and J.H. Reif. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1985.

[59] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 213–224. ACM, 2019.

[60] Andrea Pietracaprina, Geppino Pucci, Francesco Silvestri, and Fabio Vandin. Space-efficient parallel algorithms for combinatorial search problems. *Journal of Parallel and Distributed Computing*, 76:58–65, 2015.

[61] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing*, 18(3), 1989.

[62] A. Ranade. A simple optimal list ranking algorithm. In *International Conference on High Performance Computing (ICHP)*, 1998.

[63] Margaret Reid-Miller, Gary L. Miller, and Francesmary Modugno. List ranking and parallel tree contraction. In John Reif, editor, *Synthesis of Parallel Algorithms*, pages 115–194. Morgan Kaufmann, 1993.

[64] John H. Reif. *Synthesis of Paralell Algorithms*. Morgan Kaufmann, 1993.

[65] Raimund Seidel. Backwards analysis of randomized geometric algorithms. In *New trends in discrete and computational geometry*, pages 37–67. Springer, 1993.

[66] Julian Shun and Guy E Blelloch. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, 2014.

[67] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012.

[68] Julian Shun, Yan Gu, Guy E Blelloch, Jeremy T Fineman, and Phillip B Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.

[69] Harsha Vardhan Simhadri. *Program-centric cost models for locality and parallelism*. PhD thesis, IBM, 2013.

[70] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[71] Uzi Vishkin. Randomized speed-ups in parallel computation. In *ACM Symposium on Theory of Computing (STOC)*, pages 230–239, 1984.

[72] Uzi Vishkin. Advanced parallel prefix-sums, list ranking and connectivity. In John Reif, editor, *Synthesis of Parallel Algorithms*, pages 215–257. Morgan Kaufmann, 1993.

[73] James C. Wyllie. The complexity of parallel computations. Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, August 1979.

[74] SQ Zheng, Balaji Calidas, and Yanjun Zhang. An efficient general in-place parallel sorting scheme. *The Journal of Supercomputing*, 14(1):5–17, 1999.

A DETAILS FOR RANDOM PERMUTATION

We compare our implementation of the new relaxed PIP algorithm to the code in PBBS library, and here we refer to it as `RP-PBBS`. `RP-PBBS` runs in rounds that each processes 2% of all swaps, since empirically this provides the best overall performance. We note that this naturally fits into our PIP random permutation algorithm. We also process 2% of all swaps that gives the best performance for all our algorithms. We will analyze the space cost that in our final version, in which only less than 10% additional space is used. We also show the work-space tradeoff when the auxiliary space is very limited.

`RP-Naïve`. The framework of deterministic reservations allows us to work on a prefix of all active iterates. We note such prefix-based executing patterns are what we need for the PIP random permutation algorithm discussed in Section 4. Hence, we can directly use parallel hash tables to replace the auxiliary arrays R and H, and we refer to this implementation as this algorithm `RP-Naïve`. Unfortunately, compared to `RP-PBBS`, `RP-Naïve` has poor performance in practice. When varying input size for 10 million to 1 billion, `RP-Naïve` requires 2.9–3.5x running time of the `RP-PBBS` algorithm as shown in Figure 5, which can be a significant overhead. We

now propose three implementation optimizations to reduce such overhead to only 35–50%.

RP-Flat: packing $H[i]$ as an array. In RP-PBBS, the value of $H[i]$ is computed by a hash function. Since generating a good hash function in practice is expensive and this value is used in a variety of places, we store it in an array H to avoid recomputation. We note that in Algorithm 1, the access of $H[i]$ is always associated i (so as the R array in Algorithm 2), which corresponds to each swap in this round. We modify the code of deterministic reservations such that it also provides the index of each swap in the overall list of active iterates. In this way we can store the value of $H[i]$ in a consecutive array. By doing so, we reduce the hash table insert/query of each swap from three to two. We refer to this implementation as RP-Flat.

RP-OneRes: using only one reservation. RP-PBBS uses two priority updates (line 5 and line 6 in Algorithm 1). The reason to do so is that in the commit phase, the algorithm can immediately reset $R[i]$ and $R[H[i]]$ if their value is i. The algorithm in total uses two serial accesses (to $R[i]$) and two random accesses (to $R[H[i]]$) and such implementation achieves the best performance.

In our in-place version, we note that such advantage for two priority updates does not hold. The reason is that, when using a hash table for R, it is too costly to directly reset R in the commit phase—such operation corresponds to a delete in the hash table. Deletions in concurrent hash tables are very slow, and we observe that it is much more efficient to erase the entire hash table at the end of each round. Since we can hold values of R to the end of the round, we can instead apply just one priority update in line 6 and save the other one line 5. Then if-condition in line 8 is modified to

$$(R[i] = \perp \,||\, R[i] = i) \;\&\&\; R[H[i]] = i$$

Here $\perp$ indicates the initialized value of hash table, meaning that key i is not found in the hash table. One can check that these two implementations are equivalent, and by doing so, we reduce the hash table insert of each swap from two to one, which consequently reduce the hash table size by a half and reduce the overall memory footprint. We refer to this implementation as RP-OneRes.

RP-Final: packing the consecutive part of $R[i]$ in an array. In the new if-condition, for all swaps from i we check the value of $R[i]$, so ideally we can use a similar approach for $H[i]$, so that when accessing $R[i]$ we need a sequential access to an array rather than a random access to a hash table. Unfortunately, this is not applicable since we do not have the back mapping from swap i to the index of this swap in the active list of all swaps in this round. This will cause problem when the destination of a swap is the source of another swap in this round.

Although we do not have the back mapping from each swap to the index in the active list, we do know that a large potion of the index is consecutive—for all newly added swaps in a round. Hence, we modify the code for deterministic reservation such that it provides the range of the consecutive indexes and the range of the keys. With these, we use an array with the size of the active list. For reserving $R[H[i]]$, we first check if the key falls in the range, and if so we map it to the associated index and update in the array. Otherwise we insert it to the hash table. This is similar for checking the values in the commit phase, and in this way the access of $R[i]$ is mostly serial while accessing $R[H[i]]$ is mostly random. By doing so, we can save about one random access per swap per round. The estimate numbers of serial and random accesses per swap per round is shown in Table 3. When cleaning the array, we first check if the value is changed, and only reset those changed once. Since this array is mostly empty, this will save many writes that are costly. We refer to this implementation as RP-Final.