In-Place Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory

William Kuszmaul[*1] and Alek Westover[†1]

kuszmaul@mit.edu, alek.westover@gmail.com
[1]Massachusetts Institute of Technology

Abstract

We present a simple in-place algorithm for the parallel partition problem that has linear work and polylogarithmic span. The algorithm uses only exclusive read/write shared memory variables, and can be implemented using parallel-for-loops without any additional concurrency considerations. A key feature of the algorithm is that it exhibits provably optimal cache behavior, up to small-order factors.

We also present a second in-place EREW algorithm that has linear work and span $O(\log n \cdot \log\log n)$. This second algorithm has nearly optimal span but does not exhibit optimal cache behavior.

By using this low-span algorithm as a subroutine within the cache-friendly algorithm, we are able to obtain a single algorithm that combines their theoretical guarantees: the algorithm achieves span $O(\log n \cdot \log\log n)$ and optimal cache behavior. As an immediate consequence, we also get an in-place quicksort algorithm with work $O(n \log n)$, span $O(\log^2 n \cdot \log\log n)$.

[*]Supported by a Hertz Fellowship and a NSF GRFP Fellowship.
[†]Supported by Massachusetts Institute of Technology

1 Introduction

A ***partition*** operation rearranges the elements in an array so that the elements satisfying a particular ***pivot property*** appear first. In addition to playing a central role in quicksort, the partition operation is used (often implicitly) as a primitive throughout both sequential and parallel algorithms[1], and is arguably one of the simplest and most basic computational operations.

The partition operation is straightforward to implement efficiently (and in-place) in serial. Designing fast parallel algorithms is more difficult, however. Although there has been a great deal of empirical work on parallel partitioning (see, e.g., [41, 11, 60, 32, 33]), several fundamental theoretical questions remain unanswered.

In this paper we consider the following basic question: does there exist an *in-place* parallel algorithm for the partition operation? And, more generally, does there exist a parallel algorithm with optimal cache behavior?

The Classic Sum-and-Swap Algorithm. A parallel algorithm can be measured by its ***work***, the time needed to execute in serial, and its ***span***, the time to execute on infinitely many processors. A classic algorithm for parallel partitioning is the ***Sum-and-Swap Algorithm***, which achieves optimal bounds of $O(n)$ work and $O(\log n)$ span [16, 2].

The Sum-and-Swap Algorithm has several appealing properties. The algorithm can be implemented in the recursive fork-join parallel programming model [25, 34, 9, 24, 28, 30, 43, 52, 46], and uses parallel for loops as its only concurrency mechanism. This means that the algorithm is straightforward to implement in languages such as Cilk [34, 47, 42], Fortress [6], and OpenMP [53, 12], and that randomized work-stealing schedulers [25, 34, 9, 24] can execute the algorithm with provably good parallelism.

Additionally, the Sum-and-Swap Algorithm is ***race-free*** [30, 50], meaning that whenever a processor p is writing to a memory address, it is guaranteed that no other processor p' is trying to concurrently read or write the same address. In addition to being an appealing theoretical property, being race-free ensures that every program execution on a given input performs the same set of operations, regardless of scheduling, which makes test coverage, debugging, and reasoning about performance substantially easier [18].

The Sum-and-Swap Algorithm is *not* in-place, however, and requires large auxiliary arrays for both the Summing and Swapping phases of the algorithm. Additionally, the Sum-and-Swap Algorithm requires multiple passes over the input array, causing further cache inefficiency.

Why Cache Efficiency Matters. In addition to being of theoretical interest, the problem of designing in-place algorithms for parallel partitioning is also of practical importance.

One of the most widely used applications of the partition operation is in sorting. In library implementations of sorting, an in-place algorithm is needed to handle cases where there is not enough free memory in the system to allocate auxiliary arrays [49, 1]. One consequence of this is that, even sorting implementations that use mergesort as their baseline algorithm often resort to quicksort as a backup when memory allocation fails (see, e.g., [49]). But parallel quicksort requires the parallel partition operation, which until now has not yielded to in-place race-free parallel algorithms with theoretical guarantees on span.

The problem of designing cache-efficient algorithms, in general, is important for practical performance. This is especially true for parallel algorithms, since multiple processors may share a single memory bus which can cause parallel algorithms to become memory-bandwidth limited even when their sequential counterparts were not (see, e.g., discussion in [29, 51]).

[1] In several well-known textbooks and surveys on parallel algorithms [2, 16], for example, parallel partitions are implicitly used extensively to perform what are referred to as *filter* operations.

Our Results. We give an in-place parallel-partition algorithm called the ***Smoothed Striding Algorithm***, with linear work and polylogarithmic span. Additionally, the algorithm exhibits provably optimal cache behavior up to low-order terms. In particular, if the input consists of m cache lines, then the algorithm incurs at most $m(1 + o(1))$ cache misses, with high probability in m. The Smoothed Striding Algorithm is both simple and straightforward to implement, and has the potential to be useful in practice.

We also develop a suite of techniques for transforming the Sum-and-Swap Algorithm into an in-place algorithm. The new algorithm, which we call the ***In-Place Sum-and-Swap Algorithm***, has work $O(n)$ and span $O(\log n \log\log n)$, which is within a $\log\log n$ factor of optimal (and better than the polylogarithmic span of the Smoothed Striding Algorithm). The In-Place Sum-and-Swap Algorithm does not exhibit optimal cache behavior, however.

Finally, by considering a hybrid of two algorithms, we are able to achieve the best of both worlds: we obtain a single algorithm with work $O(n)$, span $O(\log n \log\log n)$ (assuming a cache-line size of $O(1)$) and nearly optimal cache behavior. As an immediate consequence, we also get an in-place and cache-efficient quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \log\log n)$.

Like the original Sum-and-Swap Algorithm, all of our algorithms are fork-join parallel, can be implemented using standard parallel for loops, and are race free.

Related Work. Our core algorithm, the Smoothed Striding Algorithm, borrows much of its basic structure from a previous algorithm, known as the ***Strided Algorithm*** [32, 33]. The Strided Algorithm has near optimal cache behavior in practice, but exhibits theoretical guarantees on span and cache behavior only on certain *random* input arrays. In contrast, the Smoothed Striding Algorithm achieves theoretical guarantees on all of work, span, and cache-optimality. This is achieved by randomly perturbing the internal structure of the Strided Algorithm, and adding a recursion step that was previously not possible. These random perturbations are reminiscent of smoothed analysis [31, 57, 14, 45, 58, 10], hence of the name of our algorithm. However, whereas classical smoothed analysis is performed by randomly perturbing the *input* to an algorithm, and then analyzing the performance on the perturbed input, our application instead randomly perturbs the *algorithm*, and analyzes the algorithm on a worst-case input.

In addition to theoretical work, there has been substantial empirical work on developing efficient implementations of parallel partitions [41, 11, 60, 32, 33]. Often, practical implementations of parallel partition leverage concurrency mechanisms such as locks or atomic operations (see, e.g., [41, 11, 60]). Such mechanisms have the benefit of offering substantial additional power to the programmer, but also have several drawbacks, making it difficult to reason about the scalability of an algorithm, the performance of the underlying scheduler, and the interactions between caches (e.g., elements of one processor's cache being invalidated due to writes in another).

The parallel partition problem is closely related to parallel sorting. Currently the only practical in-place parallel sorting algorithms either rely heavily on concurrency mechanisms such as atomic operations [41, 11, 60], or abstain from theoretical guarantees [32]. Parallel merge sort [38] was made in-place by Katajainen [44], but has proven too sophisticated for practical applications. Bitonic sort [23] is naturally in-place, and can be practical in certain applications on super computers, but suffers in general from requiring work $\Theta(n \log^2 n)$ rather than $O(n \log n)$. Parallel quicksort, on the other hand, despite the many efforts to optimize it [41, 11, 60, 32, 33], has eluded any in-place race-free algorithms due to its reliance on parallel partition. Our algorithms suggest one natural (and potentially practical) way to make parallel quicksort in-place.

The general problem of designing parallel algorithms with good cache behavior has been studied for many problems [21, 62, 20, 35, 27, 59, 15] besides parallel partition. The problem is also closely related to the study of of parallel algorithms on asymmetric and persistent memories (see, e.g.,

[19, 13, 17, 37, 22], or the survey by Gu [37]), and exploring parallel partition further in these contexts is an interesting direction for future work.

2 Preliminaries

We begin by describing the parallelism and memory model used in the paper, and by presenting background on the parallel partition problem.

Fork-Join Parallelism. In this paper we study parallel programs in the recursive fork-join programming model [25, 34, 9, 24, 28, 30, 43, 52, 46]. Following conventions similar to past work [16, 2, 28], we express the parallelism of our algorithms in terms of ***parallel-for-loops***; function calls within the inner loop then allow for more complicated recursive fork-join parallel structures. Our algorithms can also be implemented in the CREW PRAM model [16, 2].

Formally, a parallel-for-loop is given a range size $R \in \mathbb{N}$, a constant number of arguments $\arg_1, \arg_2, \ldots, \arg_c$, and a body of code. For each $i \in \{1, \ldots, R\}$, the loop launches a thread that is given loop-counter i and local copies of the arguments $\arg_1, \arg_2, \ldots, \arg_c$. The threads are then taken up by processors and the iterations of the loop are performed in parallel. Only after every iteration of the loop is complete can control flow continue past the loop.

A parallel algorithm may be run on an arbitrary number p of processors. The algorithm itself is oblivious to p, however, leaving the assignment of threads to processors up to a scheduler.

The ***work*** T_1 of an algorithm is the time that the algorithm would require to execute on a single processor. The ***span*** T_∞ of an algorithm is the time to execute on infinitely many processors. The scheduler is assumed to contribute no overhead to the span. In particular, if each iteration of a parallel-for-loop has span s, then the full parallel loop has span $s + O(1)$ [16, 2].

The work T_1 and span T_∞ can be used to quantify the time T_p that an algorithm requires to execute on p processors using a greedy online scheduler. If the scheduler is assumed to contribute no overhead, then Brent's Theorem [26] states that for any p,

$$\max(T_1/p, T_\infty) \leq T_p \leq T_1/p + T_\infty.$$

The work-stealing algorithms used in the Cilk extension of C/C++ realize the guarantee offered by Brent's Theorem within a constant factor [24, 25], with the added caveat that parallel-for-loops typically induce an additional additive overhead of $O(\log R)$.

Race-Free Algorithms. In this paper, we require algorithms to be ***race free***, meaning that no processor p ever reads/writes a memory address being concurrently written by another processor p'. Note that processors are not in lockstep (i.e., they may progress at arbitrary different speeds), and the property of being race-free must hold for any possible parallel execution.

In-Place Algorithms. In order for an algorithm to be in-place, it must not rely on large auxiliary arrays. Formally, in an in-place algorithm, each thread is given $O(\text{polylog}\, n)$ memory upon creation that is deallocated when the thread dies.[2] This memory can be shared with other threads via pointers.

Modeling Cache Behavior. For sequential algorithms, cache behavior can be modeled using the ***External Memory Model*** [61, 4]. The External Memory Model (sometimes also called the Disk-Access Model) treats memory as consisting of M fixed-size cache lines, each of some size b. A cache miss occurs in a cache whenever the line being accessed is not currently in cache, in

[2]Note that $\Omega(\log n)$ memory is necessary just for a thread to store its stack in a divide-and-conquer recursion.

which case some other line is evicted from cache to make room for the new entry[3]. In this paper, we allow for the program to control the eviction strategy. This assumption is justified by the fact that standard eviction strategies such as LRU, coupled with $\omega(1)$ resource augmentation, are $(1 + o(1))$-competitive with the optimal eviction strategy [56].

A natural approach to analyzing the cache behavior of a *parallel* algorithm (see, e.g., [20]) is to analyze the behavior of a *serial execution* of the algorithm in the External Memory Model (see, e.g., [20]). In this paper, we take a slightly more fine-grained approach to measuring the cache performance. In particular, we explicitly model the behavior of each processor's cache.

We assume that threads are scheduled using work stealing [3, 25, 34, 9, 24], and that the work-stealing itself has no cache-miss overhead. We further assume that *each* processor has a cache of size $M = \text{polylog}\, n$ (for a polylog of our choice) that is analyzed using the External Memory Model.

In order to keep our analysis of cache misses independent of the number of processors, we will ignore the cost of warming up each processor's cache (i.e., the first M cache misses in each cache are free). All subsequent cache misses are counted, however, including any cache misses necessary to warm up the cache after a steal.[4] Finally, if a processor p writes to a cache line, then the cache line is automatically invalidated in (i.e., removed from) all other processors' caches.

The Parallel Partition Problem. The parallel partition problem takes an input array $A = (A[1], A[2], \ldots, A[n])$ of size n, and a ***decider function*** dec that determines for each element $A[i] \in A$ whether or not $A[i]$ is a ***predecessor*** or a ***successor***. That is, $\text{dec}(A[i]) = 1$ if $A[i]$ is a predecessor, and $\text{dec}(A[i]) = 0$ if $A[i]$ is a successor. The behavior of the parallel partition is to reorder the elements in the array A so that the predecessors appear before the successors. Note that, in this paper, we will always treat arrays as 1-indexed.

The Sum-and-Swap Algorithm. The Sum-and-Swap Algorithm for parallel partitioning consists of two phases [16, 2]:
The Sum Phase: In this phase, the algorithm first creates an array D whose i-th element $D[i] = \text{dec}(A[i])$. Then the algorithm constructs an array S whose i-th element $S[i] = \sum_{j=1}^{i} D[i]$ is the number of predecessors in the first i elements of A. The transformation from D to S is called a ***parallel prefix sum*** and can be performed with $O(n)$ work and $O(\log n)$ span using a simple recursive algorithm: (1) First construct an array D' of size $n/2$ with $D'[i] = D[2i-1] + D[2i]$; (2) Recursively construct a parallel prefix sum S' of D'; (3) Build S by setting each $S[i] = S'[\lfloor i/2 \rfloor] + A[i]$ for odd $i > 1$ and $S[i] = S'[i/2]$ for even i, and $S[1] = A[1]$.
The Swap Phase: In this phase, the algorithm constructs an output-array C by placing each predecessor $A[i] \in A$ in position $S[i]$ of C. If there are t predecessors in A, then the first t elements of C will now contain those t predecessors in the same order that they appear in A. The algorithm then places each successor $A[i] \in A$ in position $t + i - S[i]$. Since $i - S[i]$ is the number of successors in the first i elements of A, this places the successors in C in the same order that they appear in A. Finally, the algorithm copies C into A, completing the parallel partition.

Both phases can be implemented with $O(n)$ work and $O(\log n)$ span. Like its serial out-of-place counterpart, the algorithm is stable but not in place. The algorithm uses multiple auxiliary arrays of size n (one in each phase). Kiu, Knowles, and Davis [48] were able to reduce the extra space consumption to $n + p$ under the assumption that the number of processors p is hard-coded; their algorithm breaks the array A into p parts and assigns one part to each thread. Reducing the extra

[3] As is standard, we will make the simplifying assumption that all fixed-size (i.e., non-array) local variables are automatically in cache. Under this simplifying assumption, cache misses can only occur when accessing addresses not in the current stack frame.

[4] We continue to follow the convention that local variables, such as those passed to a thread via the arguments of a parallel for loop or function, do not incur cache misses, however.

space below $o(n)$ has remained open until now, even when the number of threads is fixed.

3 A Cache-Efficient Parallel Partition

In this section we present the ***Smoothed Striding Algorithm***, which exhibits provably optimal cache behavior (up to small-order factors). The Smoothed Striding Algorithm is fully in-place and has polylogarithmic span. In particular, this means that the total amount of auxiliary memory allocated at a given moment in the execution never exceeds $\text{polylog}\, n$ per active worker. In this section we also analyze a version of the Smoothed Striding Algorithm, called the Recursive Smoothed Striding Algorithm, that is remarkably simple.

The Strided Algorithm [32]. The Smoothed Striding Algorithm borrows several structural ideas from a previous algorithm of Francis and Pannan [32], which we call the Strided Algorithm. The Strided Algorithm is designed to behave well on random arrays A, achieving span $\tilde{O}(n^{2/3})$ and exhibiting only $n/b + \tilde{O}(n^{2/3}/b)$ cache misses on such inputs. On worst-case inputs, however, the Strided Algorithm has span $\Omega(n)$ and incurs $n/b + \Omega(n/b)$ cache misses. Our algorithm, the Smoothed Striding Algorithm, builds on the Strided Algorithm by randomly perturbing the internal structure of the original algorithm; in doing so, we are able to provide provable performance guarantees for arbitrary inputs, and to add a recursion step that was previously impossible.
The ***Strided Algorithm*** consists of two steps:
The Partial Partition Step: Let $g \in \mathbb{N}$ be a parameter, and assume for simplicity that $gb \mid n$. Logically partition the array A into $\frac{n}{gb}$ chunks $C_1, \ldots, C_{n/gb}$, each consisting of g cache lines of size b. For $i \in \{1, 2, \ldots, g\}$, define P_i to consist of the i-th cache line from each of the chunks $C_1, \ldots, C_{n/gb}$. The P_i's form a strided partition of array A, since consecutive cache lines in P_i are always separated by a fixed stride of $g-1$ other cache lines. The first step of the Strided Algorithm is to perform an in-place serial partition on each of the P_i's, rearranging the elements within the P_i so that the predecessors come first. This step requires work $\Theta(n)$ and span $\Theta(n/g)$.
The Serial Cleanup Step: For each P_i, define the ***splitting position*** v_i to be the position in A of the first successor in (the already partitioned) P_i. Define $v_{\min} = \min\{v_1, \ldots, v_g\}$ and define $v_{\max} = \max\{v_1, \ldots, v_g\}$. The second step of the Strided Algorithm is to partition the sub-array $A[v_{\min}], \ldots, A[v_{\max} - 1]$ in serial. This step has no parallelism, and thus has work and span $\Theta(v_{\max} - v_{\min})$.

In general, the lack of parallelism in the Serial Cleanup step results in an algorithm with linear-span (i.e., no parallelism guarantee). When the number of predecessors in each of the P_i's is close to equal, however, the quantity $v_{\max} - v_{\min}$ can be much smaller than $\Theta(n)$. For example, if $b = 1$, and if each element of A is selected independently from some distribution, then one can use Chernoff bounds to prove that with high probability in n, $v_{\max} - v_{\min} \leq O(\sqrt{n \cdot g \cdot \log n})$. The full span of the algorithm is then $\tilde{O}(n/g + \sqrt{n \cdot g})$, which optimizes at $g = n^{1/3}$ to $\tilde{O}(n^{2/3})$. Since the Partial Partition Step incurs only n/b cache misses, the full algorithm incurs $n + \tilde{O}(n^{2/3})$ cache misses on a random array A.

Using Hoeffding's Inequality in place of Chernoff bounds, one can obtain analogous bounds for larger values of b; in particular for $b \in \text{polylog}(n)$, the optimal span remains $\tilde{O}(n^{2/3})$ and the number of cache misses becomes $n/b + \tilde{O}(n^{2/3}/b)$ on an array A consisting of randomly sampled elements.[5]

[5]The original algorithm of Francis and Pannan [32] does not consider the cache-line size b. Frias and Petit later introduced the parameter b [33], and showed that by setting b appropriately, one obtains an algorithm whose empirical performance is close to the state-of-the-art.

The Smoothed Striding Algorithm. To obtain an algorithm with provable guarantees for all inputs A, we randomly perturb the internal structure of each of the P_i's. Define $U_1, \ldots, U_g$ (which play a role analogous to $P_1, \ldots, P_g$ in the Strided Algorithm) so that each U_i contains one randomly selected cache line from each of $C_1, \ldots, C_{n/gb}$ (rather than containing the i-th cache line of each C_j). This ensures that the number of predecessors in each U_i is a sum of independent random variables with values in $\{0, 1, \ldots, n/g\}$.

By Hoeffding's Inequality, with high probability in n, the number of predecessors in each U_i is tightly concentrated around $\frac{\mu n}{g}$, where μ is the fraction of elements in A that are predecessors. It follows that, if we perform in-place partitions of each U_i in parallel, and then define v_i to be the position in A of the first successor in (the already partitioned) U_i, then the difference between $v_{\min} = \min_i v_i$ and $v_{\max} = \max_i v_i$ will be small (regardless of the input array A!).

Rather than partitioning $A[v_{\min}], \ldots, A[v_{\max} - 1]$ in serial, the Smoothed Striding Algorithm simply recurses on the subarray. Such a recursion would not have been productive for the original Strided Algorithm because the strided partition $P'_1, \ldots, P'_g$ used in the recursive subproblem would satisfy $P'_1 \subseteq P_1, \ldots, P'_g \subseteq P_g$ and thus each P'_i is already partitioned. That is, in the original Strided Algorithm, the problem that we would recurse on is a worst-case input for the algorithm in the sense that the partial partition step makes no progress.

The main challenge in designing the Smoothed Striding Algorithm becomes the construction of $U_1, \ldots, U_g$ without violating the in-place nature of the algorithm. A natural approach might be to store for each U_i, C_j the index of the cache line in C_j that U_i contains. This would require the storage of $\Theta(n/b)$ numbers as metadata, however, preventing the algorithm from being in-place. To save space, the key insight is to select a random offset $X_j \in \{1, 2, \ldots, g\}$ within each C_j, and then to assign the $(X_j + i \pmod g) + 1$-th cache line of C_j to U_i for $i \in \{1, 2, \ldots, g\}$. This allows for us to construct the U_i's using only $O(n/(gb))$ machine words storing the metadata $X_1, \ldots, X_{n/(gb)}$. By setting g to be relatively large, so that $n/(gb) \leq \text{polylog}(n)$, we can obtain an in-place algorithm that incurs $(1 + o(1))n/b$ cache misses.

We remark that the recursive structure of the Smoothed Striding Algorithm allows for the algorithm to achieve polylogarithmic span, and makes the algorithm very simple to implement in practice.

Formal Algorithm Description. Let $b < n$ be the size of a cache line, let A be an input array of size n, and let g be a parameter. (One should think of g as being relatively large, satisfying $n/(gb) \leq \text{polylog}(n)$.) We assume for simplicity that that n is divisible by gb, and we define $s = n/(gb)$. [6]

In the ***Partial Partition Step*** the algorithm divides the cache lines of A into g sets $U_1, \ldots, U_g$ where each U_i contains s cache lines, and then performs a serial partition on each U_i in parallel over the U_i's. To determine the sets $U_1, \ldots, U_g$, the algorithm uses as metadata an array $X = X[1], \ldots, X[s]$, where each $X[i] \in \{1, \ldots, g\}$; in particular each of $X[1], \ldots, X[s]$ is a uniformly random and independently selected element of $\{1, 2, \ldots, g\}$. For each $i \in \{1, 2, \ldots, g\}$, $j \in \{1, 2, \ldots, s\}$, define

$$G_i(j) = (X[j] + i \pmod g) + (j - 1)g + 1.$$

Using this terminology, we define each U_i for $i \in \{1, \ldots, g\}$ to contain the $G_i(j)$-th cache line of A for each $j \in \{1, 2, \ldots, s\}$. That is, $G_i(j)$ denotes the index of the j-th cache line from array A

[6] This assumption can be made without loss of generality by treating A as an array of size $n' = n + (gb - n \pmod{gb})$, and then treating the final $gb - n \pmod{gb}$ elements of the array as being successors (which consequently the algorithm needs not explicitly access). Note that the extra $n' - n$ elements are completely virtual, meaning they do not physically exist or reside in memory.

contained in U_i.

Note that, to compute the index of the j-th cache line in U_i, one needs only the value of $X[j]$. Thus the only metadata needed by the algorithm to determine $U_1, \ldots, U_g$ is the array X. If $|X| = s = \frac{n}{gb} \leq \text{polylog}(n)$, then the algorithm is in place.

The algorithm performs an in-place (serial) partition on each U_i (and performs these partitions in parallel with one another). In doing so, the algorithm, also collects $v_{\min} = \min_i v_i$, $v_{\max} = \max_i v_i$, where each v_i with $i \in \{1, \ldots, g\}$ is defined to be the index of the first successor in A (or n if no such successor exists).[7]

The array A is now "partially partitioned", i.e. $A[i]$ is a predecessor for all $i \leq v_{\min}$, and $A[i]$ is a successor for all $i > v_{\max}$.

The second step of the Smoothed Striding Algorithm is to complete the partitioning of $A[v_{\min} + 1], \ldots, A[v_{\max}]$. The ***Recursive Smoothed Striding Algorithm*** partitions $A[v_{\min}+1], \ldots, A[v_{\max}]$ recursively using the same algorithm (and resorts to a serial base case when the subproblem is small enough that $g \leq O(1)$). In Section 5 we discuss a different variant of the Smoothed Striding Algorithm, called the ***Hybrid Smoothed Striding Algorithm***, which partitions $A[v_{\min} + 1], \ldots, A[v_{\max}]$ using the in-place algorithm given in Theorem 4.2 instead of recursing. In general, the Hybrid algorithm yields better theoretical guarantees on span than the recursive version; on the other hand, the recursive version has the advantage that it is simple to implement as fully in-place, and still achieves polylogarithmic span. Detailed pseudocode for the Recursive Smoothed Striding Algorithm can be found in Appendix A.

Algorithm Analysis. Our first proposition analyzes the Partial Partition Step of the Smoothed Striding Algorithm.

Proposition 3.1. Let $\epsilon \in (0, 1/2)$ and $\delta \in (0, 1/2)$ such that $\epsilon \geq 1/\text{poly}(n)$ and $\delta \geq 1/\text{polylog}(n)$. Suppose $s > \frac{\ln(n/\epsilon)}{\delta^2}$, and that each processor has a cache of size at least $s + c$ for a sufficiently large constant c.

Then the Partial-Partition Step achieves work $O(n)$; achieves span $O(b \cdot s)$; incurs $\frac{s+n}{b} + O(1)$ cache misses; and guarantees that with probability at least $1 - \epsilon$,

$$v_{\max} - v_{\min} < 4n\delta.$$

Proof. Since $\sum_i |U_i| = n$, and since the serial partitioning of each U_i takes time $O(|U_i|)$, the total work performed by the algorithm is $O(n)$.

To analyze cache misses, we assume without loss of generality that array X is pinned in each processor's cache (note, in particular, that $|X| = s \leq \text{polylog}(n)$, and so X fits in cache), and that X is loaded into each processor's cache as the cache is warmed up. Thus we can ignore the cost of accesses to X, and treat all other accesses as being in a $\text{polylog}\, n$ cache managed using LRU.

Note that each U_i consists of $s = \text{polylog}\, n$ cache lines, meaning that each U_i fits entirely in cache. Thus the number of cache misses needed for a thread to partition a given U_i is just s. Since there are g of the $U_i's$, the total number of cache misses incurred in partitioning all of the U_i's is $gs = n/b$. Besides these, there are s/b cache misses for instantiating the array X; and $O(1)$ cache misses for other instantiating costs. This sums to

$$\frac{n+s}{b} + O(1).$$

[7] One can calculate $v_{\min}$ and $v_{\max}$ without explicitly storing each of $v_1, \ldots, v_g$ as follows. Rather than using a standard g-way parallel for-loop to partition each of $U_1, \ldots, U_g$, one can manually implement the parallel for-loop using a recursive divide-and-conquer approach. Each recursive call in the divide-and-conquer can then simply collect the maximum and minimum v_i for the U_i's that are partitioned within that recursive call. This adds $O(\log n)$ to the total span of the Partial Partition Step, which does not affect the overall span asymptotically.

The span of the algorithm is $O(n/g + s) = O(b \cdot s)$, since the each U_i is of size $O(n/g)$, and because the initialization of array X can be performed in time $O(|X|) = O(s)$.

It remains to show that with probability $1 - \epsilon$, $v_{\max} - v_{\min} < 4n\delta$. Let μ denote the fraction of elements in A that are predecessors. For $i \in \{1, 2, \ldots, g\}$, let μ_i denote the fraction of elements in U_i that are predecessors. Note that each μ_i is the average of s independent random variables $Y_i(1), \ldots, Y_i(s) \in [0, 1]$, where $Y_i(j)$ is the fraction of elements in the $G_i(j)$-th cache line of A that are predecessors. By construction, $G_i(j)$ has the same probability distribution for all i, since $(X[j]+i) \pmod g$ is uniformly random in $\mathbb{Z}_g$ for all i. It follows that $Y_i(j)$ has the same distribution for all i, and thus that $\mathbb{E}[\mu_i]$ is independent of i. Since the average of the μ_is is μ, it follows that $\mathbb{E}[\mu_i] = \mu$ for all $i \in \{1, 2, \ldots, g\}$.

Since each μ_i is the average of s independent $[0, 1]$-random variables, we can apply Hoeffding's inequality (i.e. a Chernoff Bound for a random variable on $[0, 1]$ rather than on $\{0, 1\}$) to each μ_i to show that it is tightly concentrated around its expected value μ, i.e.,

$$\Pr[|\mu_i - \mu| \geq \delta] < 2\exp(-2s\delta^2).$$

Since $s > \frac{\ln(n/\epsilon)}{\delta^2} \geq \frac{\ln(2n/(b\epsilon))}{2\delta^2}$, we find that for all $i \in \{1, \ldots, g\}$,

$$\Pr[|\mu_i - \mu| \geq \delta] < 2\exp\left(-2\frac{\ln(2n/(b\epsilon))}{2\delta^2}\delta^2\right) = \frac{\epsilon}{n/b} < \frac{\epsilon}{g}.$$

By the union bound, it follows that with probability at least $1 - \epsilon$, all of $\mu_1, \ldots, \mu_g$ are within δ of μ.

To complete the proof we will show that the occurrence of the event that all μ_y simultaneously satisfy $|\mu - \mu_y| < \delta$ implies that $v_{\max} - v_{\min} \leq 4n\delta$.

Recall that $G_i(j)$ denotes the index within A of the j th cache-line contained in U_i. By the definition of $G_i(j)$,

$$(j - 1)g + 1 \leq G_i(j) \leq jg.$$

Note that $A[v_i]$ will occur in the $\lceil s\mu_i \rceil$-th cache-line of U_i because U_i is composed of s cache lines. Hence

$$(\lceil s\mu_i \rceil - 1)gb + 1 \leq v_i \leq \lceil s\mu_i \rceil gb,$$

which means that

$$s\mu_i gb - gb + 1 \leq v_i \leq s\mu_i gb + gb.$$

Since $sgb = n$, it follows that $|v_i - n\mu_i| \leq gb$. Therefore,

$$|v_i - n\mu| < gb + n\delta.$$

This implies that the maximum of $|v_i - v_j|$ for any i and j is at most, $2bg + 2\delta n$. Thus,

$$v_{\max} - v_{\min} \leq 2n\left(\delta + \frac{bg}{n}\right) = 2n\left(\delta + 1/s\right) \leq 2n\left(\delta + \frac{\delta^2}{\ln(n/\epsilon)}\right) < 4n \cdot \delta.$$

□

We use Proposition 3.1 as a tool to analyze the Recursive Smoothed Striding Algorithm. Rather than parameterizing the Partial Partition step in each algorithm by s, Proposition 3.1 suggests that it is more natural to parameterize by ϵ and δ, which then determine s.

We choose $\epsilon = 1/n^c$ for c of our choice (i.e. we want to guarantee success with high probability in n). Moreover, the Recursive Smoothed Striding Algorithm continues to use the same value of ϵ

within recursive subproblems (i.e., the ϵ is chosen based on the size of the first subproblem in the recursion), so that the entire algorithm succeeds with high probability in n.

The choice of δ results in a trade-off between cache misses and span. For the Recursive Smoothed Striding Algorithm we allow δ to be chosen arbitrarily at the top level of recursion, and then fix $\delta = \Theta(1)$ to be a sufficiently small constant at all levels of recursion after the first; this guarantees that we at least halve the size of the problem between recursive iterations[8]. Optimizing δ further (after the first level of recursion) would only affect the number of undesired cache misses by a constant factor.

Now we analyze the Recursive Smoothed Striding Algorithm. We prove the following theorem:

Theorem 3.1. With high probability in n, the Recursive Smoothed Striding algorithm using parameter $\delta \in (0, 1/2)$ satisfying $\delta \geq 1/\operatorname{polylog}(n)$: achieves work $O(n)$, attains span

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta^2}\right)\right),$$

and incurs $(n + O(n\delta))/b$ cache misses.

A particularly natural parameter setting for the Recursive Smoothed Striding algorithm occurs at $\delta = 1/\sqrt{\log n}$.

Corollary 3.2 (Corollary of Theorem 3.1). With high probability in n, the Recursive Smoothed Striding Algorithm using parameter $\delta = 1/\sqrt{\log n}$: achieves work $O(n)$, attains span $O(b \log^2 n)$, and incurs $(1 + o(1))n/b$ cache misses.

Proof of Theorem 3.1. To avoid confusion, we use δ', rather than δ, to denote the constant value of δ used at levels of recursion after the first.

By Proposition 3.1, the top level of the algorithm has work $O(n)$, span $O\left(b\frac{\log n}{\delta^2}\right)$, and incurs $\frac{s+n}{b} + O(1)$ cache misses. The recursion reduces the problem size by at least a factor of 4δ, with high probability in n.

At lower layers of recursion, with high probability in n, the algorithm reduces the problem size by a factor of at least $1/2$ (since δ is set to be a sufficiently small constant). For each $i > 1$, it follows that the size of the problem at the i-th level of recursion is at most $O(n\delta/2^i)$.

The sum of the sizes of the problems after the first level of recursion is therefore bounded above by a geometric series summing to at most $O(n\delta)$. This means that the total work of the algorithm is at most $O(n\delta) + O(n) \leq O(n)$.

Recall that each level $i > 1$ uses $s = \frac{\ln(2^{-i}n\delta'/b)}{\delta'^2}$, where $\delta' = \Theta(1)$. It follows that level i uses $s \leq O(\log n)$. Thus, by Proposition 3.1, level i contributes $O(b \cdot s) = O(b \log n)$ to the span. Since there are at most $O(\log n)$ levels of recursion, the total span in the lower levels of recursion is at most $O(b \log^2 n)$, and the total span for the algorithm is at most,

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta^2}\right)\right).$$

To compute the total number of cache misses of the algorithm, we add together $(n+s)/b+O(1)$ for the top level, and then, by Proposition 3.1, at most

$$\sum_{0 \leq i < O(\log n)} \frac{1}{b} O\left(2^{2-i}n\delta + \log n\right) \leq O\left(\frac{1}{b}(n\delta + \log^2 n)\right).$$

[8]In general, setting $\delta = 1/8$ will result in the problem size being halved. However, this relies on the assumption that $gb \mid n$, which is only without loss of generality by allowing for the size of subproblems to be sometimes artificially increased by a small amount (i.e., a factor of $1 + gb/n = 1 + 1/s$). One can handle this issue by decreasing δ to, say, $1/16$.

for lower levels. Thus the total number of cache misses for the algorithm is,

$$\frac{1}{b}\left(n + \frac{\log n}{\delta^2}\right) + O(n\delta + \log^2 n)/b = (n + O(n\delta))/b.$$

□

Proof of Corollary 3.2. By Theorem 3.1, with high probability in n, the algorithm has work $O(n)$, the algorithm has span

$$O\left(b\left(\log^2 n + \frac{\log n}{\delta^2}\right)\right) = O(b \log^2 n),$$

and the algorithm incurs

$$(n + O(n\delta))/b = (n + O(n/\sqrt{\log n}))/b = (n + o(n))/b$$

cache misses. □

4 An $O(\log n \log\log n)$-Span Parallel Partition

In this section, we present an in-place algorithm, called the ***In-Place Sum-and-Swap Algorithm***, for parallel partition with span $O(\log n \log\log n)$. Each thread in the algorithm requires memory at most $O(\log n)$.

In addition to being theoretically interesting on its own, the In-Place Sum-and-Swap Algorithm is useful as a subroutine in the Smoothed Striding Algorithm. The In-Place Sum-and-Swap Algorithm has poor cache behavior, and so performs somewhat poorly in practice. However, using it as a subroutine on a small subproblem in the Smoothed Striding Algorithm allows the Smoothed Striding Algorithm to achieve span $O(\log n \log\log n)$ without sacrificing its cache behavior; we discuss this in detail in Section 5. We now describe the In-Place Sum-and-Swap Algorithm.

Prior to beginning the algorithm, the first implicit step of the algorithm is to count the number of predecessors in the array, in order to determine whether the majority of elements are either predecessors or successors. Throughout the rest of the section, we assume without loss of generality that the total number of successors in A exceeds the number of predecessors, since otherwise their roles can simply be swapped in the algorithm. Further, we assume for simplicity that the elements of A are distinct; this assumption is removed at the end of the section.

Algorithm Outline. We begin by presenting an overview of the key algorithmic ideas needed to construct an in-place algorithm.

Consider how to remove the auxiliary array C from the Reordering Phase. If one attempts to simply swap in parallel each predecessor $A[i]$ with the element in position $j = S[i]$ of A, then the swaps will almost certainly conflict. Indeed, $A[j]$ may also be a predecessor that needs to be swapped with $A[S[j]]$. Continuing like this, there may be an arbitrarily long list of dependencies on the swaps.

To combat this, we begin the algorithm with a Preprocessing Phase in which A is rearranged so that every prefix is ***successor-heavy***, meaning that for all t, the first t elements contain at least $\frac{t}{4}$ successors. Then we compute the prefix-sum array S, and begin the Reordering Phase. Using the fact that the prefixes of A are successor-heavy, the reordering can now be performed in place as follows: (1) We begin by recursively reordering the prefix P of A consisting of the first $4/5 \cdot n$ elements, so that the predecessors appear before the successors; (2) Then we simply swap each predecessor $A[i]$ in the final $1/5 \cdot n$ elements with the corresponding element $S[A[i]]$. The fact that

the prefix P is successor-heavy ensures that, after step (1), the final $\frac{1}{5} \cdot n$ elements of (the reordered) P are successors. This implies in step (2) that for each of the swaps between predecessors $A[i]$ in the final $1/5 \cdot n$ elements and earlier positions $S[A[i]]$, the latter element will be in the prefix P. In other words, the swaps are now conflict free.

Next consider how to remove the array S from the Parallel-Prefix Phase. At face value, this would seem quite difficult since the reordering phase relies heavily on S. Our solution is to *implicitly* store the value of every $O(\log n)$-th element of S in the ordering of the elements of A. That is, we break A into blocks of size $O(\log n)$, and use the order of the elements in each block to encode an entry of S. (If the elements are not all distinct, then a slightly more sophisticated encoding is necessary.) Moreover, we modify the algorithm for building S to only construct every $O(\log n)$-th element. The new parallel-prefix sum performs $O(n/\log n)$ arithmetic operations on values that are implicitly encoded in blocks; since each such operation requires $O(\log n)$ work, the total work remains linear.

In the remainder of the section, we present the algorithm in detail, and prove the key properties of each phase of the algorithm. We also provide detailed pseudocode in Appendix A. The algorithm proceeds in three phases.

A Preprocessing Phase. The goal of the Preprocessing phase is to make every prefix of A successor-heavy. To perform the Preprocessing phase on A, we begin with a parallel-for-loop: For each $i = 1, \ldots, \lfloor n/2 \rfloor$, if $A[i]$ is a predecessor and $A[n - i + 1]$ is a successor, then we swap their positions in A. To complete the Preprocessing phase on A, we then recursively perform a Preprocessing phase on $A[1], \ldots, A[\lceil n/2 \rceil]$.

Lemma 4.1. The Preprocessing Phase has work $O(n)$ and span $O(\log n)$. At the end of the Preprocessing Phase, every prefix of A is successor-heavy.

Proof. Recall that for each $t \in 1, \ldots, n$, we call the t-prefix $A[1], \ldots, A[t]$ of A successor-heavy if it contains at least $\frac{t}{4}$ successors.

The first parallel-for-loop ensures that at least half the successors in A reside in the first $\lceil n/2 \rceil$ positions, since for $i = 1, \ldots, \lfloor n/2 \rfloor$, $A[n - i + 1]$ will only be a successor if $A[i]$ is also a successor. Because at least half the elements in A are successors, it follows that the first $\lceil n/2 \rceil$ positions contain at least $\lceil n/4 \rceil$ successors, making every t-prefix with $t \geq \lceil n/2 \rceil$ successor-heavy.

After the parallel-for-loop, the first $\lceil n/2 \rceil$ positions of A contain at least as many successors as predecessors (since $\lceil n/4 \rceil \geq \frac{\lceil n/2 \rceil}{2}$). Thus we can recursively apply the argument above in order to conclude that the recursion on $A[1], \ldots, A[\lceil n/2 \rceil]$ makes every t-prefix with $t \leq \lceil n/2 \rceil$ successor-heavy. It follows that, after the recursion, every t-prefix of A is successor-heavy.

Each recursive level has constant span and performs work proportional to the size of the subarray being considered. The Preprocessing phase therefore has total work $O(n)$ and span $O(\log n)$. □

through each of the blocks $X_{t+1}, \ldots, X_{\lfloor n/b \rfloor}$. For each block X_i, we first extract v_i (with work $O(\log n)$ and span $O(\log \log n)$ using Lemma 4.3). We then create an auxiliary array Y_i of size $|X_i|$, using $O(\log n)$ thread-local memory. Using a parallel-prefix sum (with work $O(\log n)$ and span $O(\log \log n)$), we set each $Y_i[j]$ equal to v_i plus the number of predecessors in $X_i[1], \ldots, X_i[j]$. In other words, $Y_i[j]$ equals the number of predecessors in A appearing at or before $X_i[j]$.

After creating Y_i, we then perform a parallel-for-loop through the elements $X_i[j]$ of X_i (note we are still within another parallel loop through the X_i's), and for each predecessor $X_i[j]$, we swap it with the element in position $Y_i[j]$ of the array A. This completes the algorithm.

Lemma 4.2. The Reordering phase takes work $O(n)$ and span $O(\log n \log \log n)$. At the end of the phase, the array A is fully partitioned.

Proof. After P has been recursively partitioned, it will be of the form $P_1 \circ P_2$ where P_1 contains only predecessors and P_2 contains only successors. Because P was successor-heavy before the recursive partitioning (by Lemma 4.4), we have that $|P_2| \geq |P|/4$, and thus that $|P_2| \geq |X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}|$.

After the recursion, the swaps performed by the algorithm will swap the i-th predecessor in $X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}$ with the i-th element in P_2, for i from 1 to the number of predecessors in $X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}$. Because $|P_2| \geq |X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}|$ these swaps are guaranteed not to conflict with one-another; and since P_2 consists of successors, the final state of array A will be fully partitioned.

The total work in the reordering phase is $O(n)$ since each X_i appears in a parallel-for-loop at exactly one level of the recursion, and incurs $O(\log n)$ work. The total span of the reordering phase is $O(\log n \cdot \log \log n)$, since there are $O(\log n)$ levels of recursion, and within each level of recursion each X_i in the parallel-for-loop incurs span $O(\log \log n)$. □

Combining the phases, the full algorithm has work $O(n)$ and span $O(\log \log n)$. Thus we have:

Theorem 4.1. There exists an in-place algorithm using exclusive-read-write variables that performs parallel-partition with work $O(n)$ and span $O(\log n \cdot \log \log n)$.

An Implicit Parallel Prefix Sum. Pick a ***block-size*** $b \in \Theta(\log n)$ satisfying $b \geq 2\lceil \log(n+1) \rceil$. Consider A as a series of $\lfloor n/b \rfloor$ blocks of size b, with the final block of size between b and $2b-1$. Denote the blocks by $X_1, \ldots, X_{\lfloor n/b \rfloor}$.

Within each block X_i, we can implicitly store a value in the range $0, \ldots, n$ through the ordering of the elements:

Lemma 4.3. Given an array X of $2\lceil \log(n+1) \rceil$ distinct elements, and a value $v \in \{0, \ldots, n\}$, one can rearrange the elements of X to encode the bits of v using work $O(\log n)$ and span $O(\log \log n)$; and one can then later decode v from X using work $O(\log n)$ and span $O(\log \log n)$.

Proof. Observe that X can be broken into (at least) $\lceil \log(n+1) \rceil$ disjoint pairs of adjacent elements $(x_1, x_2), (x_3, x_4), \ldots$, and by rearranging the order in which a given pair (x_j, x_{j+1}) occurs, the lexicographic comparison of whether $x_j < x_{j+1}$ can be used to encode one bit of information. Values $v \in [0, n]$ can therefore be read and written to X with work $O(b) = O(\log n)$ and span $O(\log b) = O(\log \log n)$ using a simple divide-and-conquer recursive approach to encode and decode the bits of v. □

To perform the Parallel Prefix Sum phase, our algorithm begins by performing a parallel-for loop through the blocks, and storing in each block X_i a value v_i equal to the number of predecessors in the block. (This can be done in place with work $O(n)$ and span $O(\log \log n)$ by Lemma 4.3.)

The algorithm then performs an in-place parallel-prefix operation on the values $v_1, \ldots, v_{\lfloor n/b \rfloor}$ stored in the blocks. This is done by first resetting each even-indexed value v_{2i} to $v_{2i} + v_{2i-1}$; then recursively performing a parallel-prefix sum on the even-indexed values; and then replacing each odd-indexed v_{2i+1} with $v_{2i+1} + v_{2i}$, where v_0 is defined to be zero.

Lemma 4.4 analyzes the phase:

Lemma 4.4. The Parallel Prefix Sum phase uses work $O(n)$ and span $O(\log n \log \log n)$. At the end of the phase, each X_i encodes a value v_i counting the number of predecessors in the prefix $X_1 \circ X_2 \circ \cdots \circ X_i$; and each prefix of blocks (i.e., each prefix of the form $X_1 \circ X_2 \circ \cdots \circ X_i$) is successor-heavy.

Proof. If the v_i's could be read and written in constant time, then the prefix sum would take work $O(n/\log n)$ and span $O(\log n)$, since there are $O(n/\log n)$ v_i's. Because each v_i actually requires work $O(\log n)$ and span $O(\log \log n)$ to read/write (by Lemma 4.3), the prefix sum takes work $O(n)$ and span $O(\log n \cdot \log \log n)$.

Once the prefix-sum has been performed, every block X_i encodes a value v_i counting the number of predecessors in the prefix $X_1 \circ X_2 \circ \cdots \circ X_i$. Moreover, because the Parallel Prefix Sum phase only rearranges elements within each X_i, Lemma 4.1 ensures that each prefix of the form $X_1 \circ X_2 \circ \cdots \circ X_i$ remains successor-heavy. □

In-Place Reordering. In the final phase of the algorithm, we reorder A so that the predecessors appear before the successors. Let $P = X_1 \circ X_2 \circ \cdots \circ X_t$ be the smallest prefix of blocks that contains at least $4/5$ of the elements in A. We begin by recursively reordering the elements in P so that the predecessors appear before the successors; as a base case, when $|P| \leq 5b = O(\log n)$, we simply perform the reordering in serial.

To complete the reordering of A, we perform a parallel-for-loop through each of the blocks $X_{t+1}, \ldots, X_{\lfloor n/b \rfloor}$. For each block X_i, we first extract v_i (with work $O(\log n)$ and span $O(\log \log n)$ using Lemma 4.3). We then create an auxiliary array Y_i of size $|X_i|$, using $O(\log n)$ thread-local memory. Using a parallel-prefix sum (with work $O(\log n)$ and span $O(\log \log n)$), we set each $Y_i[j]$ equal to v_i plus the number of predecessors in $X_i[1], \ldots, X_i[j]$. In other words, $Y_i[j]$ equals the number of predecessors in A appearing at or before $X_i[j]$.

After creating Y_i, we then perform a parallel-for-loop through the elements $X_i[j]$ of X_i (note we are still within another parallel loop through the X_i's), and for each predecessor $X_i[j]$, we swap it with the element in position $Y_i[j]$ of the array A. This completes the algorithm.

Lemma 4.5. The Reordering phase takes work $O(n)$ and span $O(\log n \log \log n)$. At the end of the phase, the array A is fully partitioned.

Proof. After P has been recursively partitioned, it will be of the form $P_1 \circ P_2$ where P_1 contains only predecessors and P_2 contains only successors. Because P was successor-heavy before the recursive partitioning (by Lemma 4.4), we have that $|P_2| \geq |P|/4$, and thus that $|P_2| \geq |X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}|$.

After the recursion, the swaps performed by the algorithm will swap the i-th predecessor in $X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}$ with the i-th element in P_2, for i from 1 to the number of predecessors in $X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}$. Because $|P_2| \geq |X_{t+1} \circ \cdots \circ X_{\lfloor n/b \rfloor}|$ these swaps are guaranteed not to conflict with one-another; and since P_2 consists of successors, the final state of array A will be fully partitioned.

The total work in the reordering phase is $O(n)$ since each X_i appears in a parallel-for-loop at exactly one level of the recursion, and incurs $O(\log n)$ work. The total span of the reordering phase is $O(\log n \cdot \log \log n)$, since there are $O(\log n)$ levels of recursion, and within each level of recursion each X_i in the parallel-for-loop incurs span $O(\log \log n)$. □

Combining the phases, the full algorithm has work $O(n)$ and span $O(\log \log n)$. Thus we have:

Theorem 4.2. There exists an in-place algorithm using exclusive-read-write variables that performs parallel-partition with work $O(n)$ and span $O(\log n \cdot \log \log n)$.

Allowing for Repeated Elements. In proving Theorem 4.2 we assumed for simplicity that the elements of A are distinct. To remove this assumption, we conclude the section by proving a slightly more complex variant of Lemma 4.3, eliminating the requirement that the elements of the array X be distinct:

Lemma 4.6. Let X be an array of $b = 4\lceil \log(n+1) \rceil + 2$ elements. The there is an *encode* function, and a *decode* function such that:

- The encode function modifies the array X (possibly overwriting elements in addition to rearranging them) to store a value $v \in \{0, \ldots, n\}$. The first time the encode function is called on X it has work and span $O(\log n)$. Any later times the encode function is called on X, it has

work $O(\log n)$ and span $O(\log \log n)$. In addition to being given an argument v, the encode function is given a boolean argument indicating whether the function has been invoked on X before.

- The decode function recovers the value of v from the modified array X, and restores X to again be an array consisting of the same multiset of elements that it began with. The decode function has work $O(\log n)$ and span $O(\log \log n)$.

Proof. Consider the first b letters of X as a sequence of pairs, given by $(x_1, x_2), \ldots, (x_{b-1}, x_b)$. If at least half of the pairs $(x_i, x_{i+1}$ satisfy $x_i \neq x_{i+1}$, then the encode function can reorder those pairs to appear at the front of X, and then use them to encode v as in Lemma 4.3. Note that the reordering of the pairs will only be performed the first time that the encode function is invoked on X. Later calls to the encode function will have work $O(\log n)$ and span $O(\log \log n)$, as in Lemma 4.3.

If, on the other hand, at least half the pairs consist of equal-value elements $x_i = x_{i+1}$, then we can reorder the pairs so that the first $\lceil \log(n+1) \rceil + 1$ of them satisfy this property. (This is only done on the first call to encode.) To encode a value v, we simply explicitly overwrite the second element in each of the pairs $(x_3, x_4), (x_5, x_6), \ldots$ with the bits of v, overwriting each element with one bit. The reordering performed by the first call to encode has work and span $O(\log n)$; the writing of v's bits can then be performed in work $O(\log n)$ and span $O(\log \log n)$ using a simple divide-and-conquer approach.

To perform a decode and read the value v, we check whether $x_1 = x_2$ in order to determine which type of encoding is being used, and then we can unencode the bits of v using work $O(\log n)$ and span $O(\log \log n)$; if the encoding is the second type (i.e., $x_1 = x_2$), then the decode function also restores the elements $x_2, x_4, x_6, \ldots$ of the array X as it extracts the bits of v. Note that checking whether $x_1 = x_2$ is also used by the encode function each time after the first time it is called, in order determine which type of encoding is being used. □

The fact that the first call to the encode function on each X_i has span $O(\log n)$ (rather than $O(\log \log n)$) does not affect the total span of our parallel-partition algorithm, since this simply adds a step with $O(\log n)$-span to the beginning of the Parallel Prefix phase. Lemma 4.6 can therefore used in place of Lemma 4.3 in order to complete the proof of Theorem 4.2 for arrays A that contain duplicate elements.

5 A Cache-Efficient $O(\log n \log \log n)$-Span Parallel Partition

In this section we analyze the ***Hybrid Smoothed Striding Algorithm***, which, after the Partial Partition Step of the Smoothed Strided Algorithm, partitions the sub-array using the In-Place Sum-and-Swap Algorithm. We show that doing so results in an improved span (since the In-Place Sum-and-Swap Algorithm has span only $O(\log n \log \log n)$), while still incurring only $(1 + o(1))n/b$ cache misses (since the cache-inefficient In-Place Sum-and-Swap Algorithm is only used on a small subarray of A).

We now analyze the Hybrid Smoothed Striding Algorithm. We prove the following theorem:

Theorem 5.1. The Hybrid Smoothed Striding Algorithm using parameter $\delta \in (0, 1/2)$ satisfying $\delta \geq 1/\operatorname{polylog}(n)$: achieves work $O(n)$; achieves span

$$O\left(\log n \log \log n + \frac{b \log n}{\delta^2}\right),$$

with high probability in n; and incurs fewer than

$$(n + O(n\delta))/b$$

cache misses with high probability in n.

An interesting corollary of Theorem 5.1 concerns what happens when b is small (e.g., constant) and we choose δ to optimize span:

Corollary 5.1 (Corollary of Theorem 5.1). Suppose $b \leq o(\log \log n)$. Then the Hybrid Smoothed Striding using $\delta = \Theta\big(\sqrt{b/\log \log n}\big)$, achieves work $O(n)$, and with high probability in n, achieves span $O(\log n \log \log n)$ and incurs fewer than $(n + o(n))/b$ cache misses.

Proof of Theorem 5.1. We analyze the Partial Partition Step using Proposition 3.1. Note that by our choice of ϵ, $s = O\left(\frac{\log n}{\delta^2}\right)$. The Partial Partition Step therefore has work $O(n)$, span $O\left(\frac{b \log n}{\delta^2}\right)$, and incurs fewer than

$$\frac{n}{b} + O\left(\frac{\log n}{b\delta^2}\right) + O(1)$$

cache misses.

By Theorem 4.2, the subproblem of partitioning of $A[v_{\min} + 1], \ldots, A[v_{\max}]$ takes work $O(n)$. With high probability in n, the subproblem has size less than $4n\delta$, which means that the subproblem achieves span

$$O(\log n\delta \log \log n\delta) = O(\log n \log \log n),$$

and incurs at most $O(n\delta/b)$ cache misses.

The total number of cache misses is therefore,

$$\frac{n}{b} + O\left(\frac{\log n}{b\delta^2} + \frac{n\delta}{b}\right) + O(1),$$

which since $\delta \geq 1/\operatorname{polylog}(n)$, is at most $(n + O(n\delta))/b + O(1) \leq (n + O(n\delta))/b$, as desired. □

Proof of Corollary 5.1. We use $\delta = \sqrt{b/\log \log n}$ in the result proved in Theorem 5.1.
First note that the assumptions of Theorem 5.1 are satisfied because $O(\sqrt{b/\log \log n}) > 1/\operatorname{polylog}(n)$.
The algorithm achieves work $O(n)$. With high probability in n the algorithm achieves span

$$O\left(\log n \log \log n + \frac{b \log n}{\delta^2}\right) = O(\log n \log \log n).$$

With high probability in n the algorithm incurs fewer than

$$(n + O(n\delta))/b = (n + O(n\sqrt{b/\log \log n}))/b$$

cache misses. By assumption $\sqrt{b/\log \log n} = o(1)$, so this reduces to $(n + o(n))/b$ cache misses, as desired. □

6 Conclusion and Open Questions

Parallel partition is a fundamental primitive in parallel algorithms [16, 2]. Achieving faster and more space-efficient implementations, even by constant factors, is therefore of high practical importance. Until now, the only space-efficient algorithms for parallel partition have relied extensively on concurrency mechanisms or atomic operations, or lacked provable performance guarantees. If a

parallel function is going to be invoked within a large variety of applications, then provable guarantees are highly desirable. Moreover, algorithms that avoid the use of concurrency mechanisms tend to scale more reliably (and with less dependency on the particulars of the underlying hardware).

In this paper, we have shown that, somewhat surprisingly, one can adapt the classic parallel algorithm to completely eliminate the use of auxiliary memory, while still using only exclusive read/write shared variables, and maintaining a polylogarithmic span. Although the superior cache performance of the low-space algorithm results in practical speedups over its out-of-place counterpart, both algorithms remain far from the state-of-the art due to memory bandwidth bottlenecks. To close this gap, we also presented a second in-place algorithm, the Smoothed Striding Algorithm, which achieves polylogarithmic span while guaranteeing provably optimal cache performance up to low-order factors. The Smoothed Striding Algorithm introduces randomization techniques to the previous (blocked) Striding Algorithm of [33, 32], which was known to perform well in practice but which previously exhibited poor theoretical guarantees. Our implementation of the Smoothed Striding Algorithm is fully in-place, exhibits polylogarithmic span, and has optimal cache performance.

Our work prompts several theoretical questions. Can fast space-efficient algorithms with polylogarithmic span be found for other classic problems such as randomly permuting an array [8, 7, 55], and integer sorting [54, 40, 5, 39, 36]? Such algorithms are of both theoretical and practical interest, and might be able to utilize some of the techniques introduced in this paper.

Another important direction of work is the design of in-place parallel algorithms for sample-sort, the variant of quicksort in which multiple pivots are used simultaneously in each partition. Sample-sort can be implemented to exhibit fewer cache misses than quicksort, which is especially important when the computation is memory-bandwidth bound. The known in-place parallel algorithms for sample-sort rely heavily on atomic instructions [11] (even requiring 128-bit compare-and-swap instructions). Finding fast algorithms that use only exclusive-read-write memory (or concurrent-read-exclusive-write memory) is an important direction of future work.

A Pseudocode

Figure 1: Smoothed Striding Algorithm

Recall:
A is the array to be partitioned, of length n.
We break A into chunks, each consisting of g cache lines of size b.
We create g groups $U_1, \ldots, U_g$ that each contain a single cache line from each chunk,
U_i's j-th cache line is the $(X[j] + i \bmod g + 1)$-th cache line in the j-th chunk of A.

```
procedure GET BLOCK START INDEX(X, g, b, i, j)   ▷ This procedure returns the index in A of the start of U_i's j-th block.
    return b · ((X[j] + i mod g) + (j − 1) · g) + 1
end procedure

procedure PARALLELPARTITION(A, n, g, b)
    if g < 2 then
        serial partition A
    else
        for j ∈ {1, 2, ..., n/(gb)} do
            X[j] ← a random integer from [1, g]
        end for
        for all i ∈ {1, 2, ..., g} in parallel do                    ▷ We perform a serial partition on all U_i's in parallel
            low ← GetBlockStartIndex(X,g,b,i,1)                      ▷ low ← index of the first element in U_i
            high ← GetBlockStartIndex(X,g,b,i,n/(gb)) + b − 1        ▷ high ← index of the last element in U_i
            while low < high do
                while A[low] ≤ pivotValue do
                    low ← low+1
                    if low mod b ≡ 0 then                            ▷ Perform a block increment once low reaches the end of a block
                        k ← number of block increments so far (including this one)
                        low ← GetBlockStartIndex(X,g,b,i,k)          ▷ Increase low to start of block k of G_i
                    end if
                end while
                while A[high] > pivotValue do
                    high ← high−1
                    if high mod b ≡ 1 then                           ▷ Perform a block decrement once high reaches the start of a block
                        k ← number of block decrements so far (including this one)
                        k' ← n/(gb) − k
                        high ← GetBlockStartIndex(X, g,b,i,k') +b − 1   ▷ Decrease high to end of block k' of G_i
                    end if
                end while
                Swap A[low] and A[high]
            end while
        end for
        Recurse on A[v_min], ..., A[v_max − 1]
    end if
end procedure
```

Figure 2: In-Place Sum-and-Swap Algorithm: Helper Functions

procedure WriteToBlock(A, b, i, v) ▷ Write value v to the i-th block X_i of A, where $A = X_1 \circ X_2 \circ \cdots \circ X_{\lfloor n/b \rfloor}$
  for all $j \in \{1, 2, \ldots, \lfloor b/2 \rfloor\}$ in parallel **do**
    if $\mathbb{1}_{X_i[2j] < X_i[2j+1]} \neq$ (the j-th binary digit of v) **then**
      Swap $X_i[2j]$ and $X_i[2j+1]$
    end if
  end for
end procedure

procedure ReadFromBlock(A, i, j) ▷ Reads the value v stored in $A[i], A[i+1], \ldots, A[j]$
  if $j - i = 2$ **then**
    return $\mathbb{1}_{A[i] < A[i+1]}$
  else
    Parallel-Spawn $v_0 \leftarrow$ ReadFromBlock(A, i, $i + (j-i)/2$)
    Parallel-Spawn $v_f \leftarrow$ ReadFromBlock(A, $i + (j-i)/2 + 1$, j)
    Parallel-Sync
    return $v_f \cdot 2^{\frac{j-i}{4}} + v_0$
  end if
end procedure

Require: A has more successors than predecessors
Ensure: Each prefix of A is "successor heavy"
procedure MakeSuccessorHeavy(A, n)
  for all $i \in \{1, 2, \ldots, \lfloor n/2 \rfloor\}$ in parallel **do**
    if $A[i]$ is a predecessor and $A[n-i+1]$ is a successor **then**
      Swap $A[i]$ and $A[n-i+1]$
    end if
  end for
  MakeSuccessorHeavy(A, $\lceil n/2 \rceil$) ▷ Recurse on $A[1], A[2], \ldots, A[\lceil n/2 \rceil]$
end procedure

Figure 3: In-Place Sum-and-Swap Algorithm: Main Functions

Require: Each prefix of A is "successor heavy"
Ensure: Each block X_i stores how many predecessors occur in $X_1 \circ X_2 \circ \cdots \circ X_i$
procedure ImplicitParallelPrefixSum(A, n)
  Pick $b \in \Theta(\log n)$ to be the "block size"
  Logically partition A into blocks, with $A = X_1 \circ X_2 \circ \cdots \circ X_{\lfloor n/b \rfloor}$
  for all $i \in \{1, 2, \ldots, \lfloor n/b \rfloor\}$ in parallel **do**
    $v_i \leftarrow 0$ ▷ v_i will store number of predecessors in X_i
    for all $a \in X_i$ in serial **do**
      if a is a predecessor **then**
        $v_i \leftarrow v_i + 1$
      end if
    end for
    WriteToBlock(A, b, i, v_i) ▷ Now we encode the value v_i in the block X_i
  end for
  Perform a parallel prefix sum on the values v_i stored in the X_i's
end procedure

Require: Each block X_i stores how many predecessors occur in $X_1 \circ X_2 \circ \cdots \circ X_i$
Ensure: A is partitioned
procedure Reorder(A, n)
  $t \leftarrow$ least integer such that $t \cdot b > n \cdot 4/5$
  Reorder(A, t) ▷ Recurse on $A[1], A[2], \ldots, A[t]$
  for all $i \in \{t+1, t+2, \ldots, \lfloor n/b \rfloor\}$ **do**
    $v_i \leftarrow$ ReadFromBlock(A, $b \cdot i + 1$, $b \cdot (i+1)$)
    Instantiate an array Y_i with $|Y_i| = |X_i| \in \Theta(\log n)$,
    In parallel, set $Y_i[j] \leftarrow 1$ if $X_i[j]$ is a predecessor, and $Y_i[j] \leftarrow 0$ otherwise.
    Perform a parallel prefix sum on Y_i, and add v_i to each $Y_i[j]$
    for all $j \in \{1, 2, \ldots, b\}$ **do**
      if $X_i[j]$ is a predecessor **then**
        Swap $X_i[j]$ and $A[Y_i[j]]$
      end if
    end for
  end for
end procedure

procedure ParallelPartition(A, n)
  $k \leftarrow$ count number of successors in A in parallel
  if $k < n/2$ **then**
    Swap the role of successors and predecessors in the algorithm (i.e. change the decider function)
    At the end we consider $A'[i] = A[n - i + 1]$, the logically reversed array, as output
  end if
  MakeSuccessorHeavy(A, n) ▷ *prepreocessing phase*
  ImplicitParallelPrefixSum(A, n) ▷ *Implicit Parallel Prefix Sum*
  Reorder(A, n) ▷ *In-Place Reordering Phase*
end procedure

References

[1] *OpenBSD C qsort, version 1.18*, 1992.

[2] U. A. Acar and G. Blelloch, *Algorithm design: Parallel and sequential*, 2016.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, *The data locality of work stealing*, in Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, 2000, pp. 1–12.

[4] A. Aggarwal and S. Vitter, Jeffrey, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31 (1988), pp. 1116–1127.

[5] S. Albers and T. Hagerup, *Improved parallel integer sorting without concurrent writing*, Information and Computation, 136 (1997), pp. 25–51.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt, *The Fortress Language Specification Version 1.0*, Sun Microsystems, Inc., Mar. 2008.

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

[8] R. Anderson, *Parallel algorithms for generating random permutations on a shared memory machine*, in Proceedings of the second annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 1990, pp. 95–102.

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, *Thread scheduling for multiprogrammed multiprocessors*, in SPAA, 1998, pp. 119–129.

[10] D. Arthur, B. Manthey, and H. Röglin, *Smoothed analysis of the k-means method*, Journal of the ACM (JACM), 58 (2011), pp. 1–31.

[11] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, *In-place parallel super scalar samplesort*, arXiv preprint arXiv:1705.02257, (2017).

[12] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, *The design of OpenMP tasks*, TPDS, 20 (2009), pp. 404–418.

[13] 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 Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, 2016, pp. 145–156.

[14] M. A. Bender, M. Farach-Colton, and W. Kuszmaul, *Achieving optimal backlog in multi-processor cup games*, in Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, 2019, pp. 1148–1157.

[15] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, *Concurrent cache-oblivious b-trees*, in Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, 2005, pp. 228–237.

[16] G. E. Blelloch, *Programming parallel algorithms*, Communications of the ACM, 39 (1996), pp. 85–97.

[17] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun, *Sorting with asymmetric read and write costs*, in Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures, 2015, pp. 1–12.

[18] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, *Internally deterministic parallel algorithms can be fast*, in ACM SIGPLAN Notices, vol. 47, ACM, 2012, pp. 181–192.

[19] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun, *The parallel persistent memory model*, in Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, 2018, pp. 247–258.

[20] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, *Low depth cache-oblivious algorithms*, in Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, 2010, pp. 189–199.

[21] G. E. Blelloch and Y. Gu, *Improved parallel cache-oblivious algorithms for dynamic programming [extend abstract]*, in Symposium on Algorithmic Principles of Computer Systems, SIAM, 2020, pp. 105–119.

[22] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun, *Parallel write-efficient algorithms and data structures for computational geometry*, in Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, 2018, pp. 235–246.

[23] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, *An experimental analysis of parallel sorting algorithms*, Theory of Computing Systems, 31 (1998), pp. 135–167.

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, Journal of parallel and distributed computing, 37 (1996), pp. 55–69.

[25] R. D. Blumofe and C. E. Leiserson, *Scheduling multithreaded computations by work stealing*, Journal of the ACM (JACM), 46 (1999), pp. 720–748.

[26] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, Journal of the ACM (JACM), 21 (1974), pp. 201–206.

[27] G. S. Brodal and R. Fagerberg, *Cache oblivious distribution sweeping*, in International Colloquium on Automata, Languages, and Programming, Springer, 2002, pp. 426–438.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT press, 2009.

[29] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, *Evaluating attainable memory bandwidth of parallel programming models via babelstream*, International Journal of Computational Science and Engineering, 17 (2018), pp. 247–262.

[30] M. Feng and C. E. Leiserson, *Efficient detection of determinacy races in Cilk programs*, Theory of Computing Systems, 32 (1999), pp. 301–326.

[31] M. Fouz, M. Kufleitner, B. Manthey, and N. Z. Jahromi, *On smoothed analysis of quicksort and hoare's find*, in International Computing and Combinatorics Conference, Springer, 2009, pp. 158–167.

[32] R. S. Francis and L. Pannan, *A parallel partition for enhanced parallel quicksort*, Parallel Computing, 18 (1992), pp. 543–550.

[33] L. Frias and J. Petit, *Parallel partition revisited*, in International Workshop on Experimental and Efficient Algorithms, Springer, 2008, pp. 142–153.

[34] M. Frigo, C. E. Leiserson, and K. H. Randall, *The implementation of the Cilk-5 multithreaded language*, in PLDI, 1998, pp. 212–223.

[35] M. Frigo and V. Strumpen, *The cache complexity of multithreaded cache oblivious algorithms*, Theory of Computing Systems, 45 (2009), pp. 203–233.

[36] A. V. Gerbessiotis and C. J. Siniolakis, *Probabilistic integer sorting*, Information processing letters, 90 (2004), pp. 187–193.

[37] Y. Gu, *Survey: computational models for asymmetric read and write costs*, in 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2018, pp. 733–743.

[38] T. Hagerup and C. Rüb, *Optimal merging and sorting on the erew pram*, Information Processing Letters, 33 (1989), pp. 181–185.

[39] Y. Han, *Improved fast integer sorting in linear space*, in Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2001, pp. 793–796.

[40] Y. Han and X. He, *More efficient parallel integer sorting*, in Frontiers in Algorithmics and Algorithmic Aspects in Information and Management, Springer, 2012, pp. 279–290.

[41] P. Heidelberger, A. Norton, and J. T. Robinson, *Parallel quicksort using fetch-and-add*, IEEE Transactions on Computers, 39 (1990), pp. 133–138.

[42] Intel Corporation, *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[43] T. Kaler, W. Kuszmaul, T. B. Schardl, and D. Vettorel, *Cilkmem: Algorithms for analyzing the memory high-water mark of fork-join parallel programs*, in Symposium on Algorithmic Principles of Computer Systems, SIAM, 2020, pp. 162–176.

[44] J. Katajainen, C. Levcopoulos, and O. Petersson, *Space-efficient parallel merging*, RAIRO-Theoretical Informatics and Applications, 27 (1993), pp. 295–310.

[45] W. Kuszmaul, *Achieving optimal backlog in the vanilla multi-processor cup game*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2020, pp. 1558–1577.

[46] D. Lea, *A java fork/join framework*, in Proceedings of the ACM 2000 conference on Java Grande, 2000, pp. 36–43.

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

[48] J. Liu, C. Knowles, and A. B. Davis, *A cost optimal parallel quicksorting and its implementation on a shared memory parallel computer*, in International Symposium on Parallel and Distributed Processing and Applications, Springer, 2005, pp. 491–502.

[49] D. C. S. Mike Haertel, *GNU C qsort, version 2.28.9*, 1991.

[50] R. H. B. Netzer and B. P. Miller, *What are race conditions?*, LOPLAS, 1 (1992), pp. 74–88.

[51] J. Ni and P. DME, *Enabling technology of multi-core computing for medical imaging*, Consortium of College of Computer Science & Technology, Harbin Engineering University, China, (2009).

[52] L. Nyman and M. Laakso, *Notes on the history of fork and join*, IEEE Annals of the History of Computing, 38 (2016), pp. 84–87.

[53] *OpenMP Application Program Interface, Version 3.0*, May 2008.

[54] S. Rajasekaran and S. Sen, *On parallel integer sorting*, Acta Informatica, 29 (1992), pp. 1–15.

[55] 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 Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2015, pp. 431–448.

[56] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM, 28 (1985), pp. 202–208.

[57] D. A. Spielman, *The smoothed analysis of algorithms*, in International Symposium on Fundamentals of Computation Theory, Springer, 2005, pp. 17–18.

[58] D. A. Spielman and S.-H. Teng, *Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time*, Journal of the ACM (JACM), 51 (2004), pp. 385–463.

[59] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, *Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency*, in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2015, pp. 205–214.

[60] P. Tsigas and Y. Zhang, *A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000*, in Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, IEEE, 2003, p. 372.

[61] J. S. Vitter, *Algorithms and data structures for external memory*, Foundations and Trends® in Theoretical Computer Science, 2 (2008), pp. 305–474.

[62] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager, *Introducing a parallel cache oblivious blocking approach for the lattice boltzmann method*, Progress in Computational Fluid Dynamics, an International Journal, 8 (2008), pp. 179–188.