Simple and Space-Efficient Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory

William Kuszmaul*
Massachusetts Institute of Technology
kuszmaul@mit.edu

Alek Westover†
alek.westover@gmail.com

Abstract

We present a simple in-place algorithm for parallel partition that has work $O(n)$ and span $O(\log n \cdot \log\log n)$. The algorithm uses only exclusive read/write shared variables, and can be implemented using parallel-for-loops without any additional concurrency considerations (i.e., the algorithm is in the EREW PRAM model). As an immediate consequence, we also get a simple in-place quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \log\log n)$.

Using our algorithmic techniques, we implement an (almost) in-place parallel partition. In addition to achieving much better memory utilization, the algorithm leverages its improved cache behavior to achieve a speedup over its out-of-place counterpart.

We also present an alternative in-place parallel-partition algorithm with a larger span of $O(\sqrt{n \log n})$, but which is designed to have very small overhead. We show that when the algorithm is tuned appropriately, and given a large enough input to achieve high parallelism, the algorithm can be made to outperform is lower-span peers.

*.CCS Concepts• **Theory of computation → Shared memory algorithms**.

*.KeywordsParallel Partition, EREW PRAM, in-place algorithms

ACM Reference Format:
William Kuszmaul and Alek Westover. 2019. Simple and Space-Efficient Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory. In *Proceedings of (APOCS '20).* ACM, New York, NY, USA, Article 4, 15 pages. https://doi.org/10.475/123_4

1 Introduction

A ***parallel 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 parallel quicksort, the parallel partition operation is used as a primitive throughout parallel algorithms.[1]

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. There is a well-known algorithm for parallel partition on arrays of size n with work $O(n)$ and span $O(\log n)$ [1, 6]. Moreover, the algorithm uses only exclusive read/write shared memory variables (i.e., it is an ***EREW*** algorithm). This eliminates the need for concurrency mechanisms such as locks and atomic variables, and ensures good behavior even if the time to access a location is a function of the number of threads trying to access it (or its cache line) concurrently. EREW algorithms also have the advantage that their behavior is internally deterministic, meaning that the behavior of the algorithm will not differ from run to run, which makes test coverage, debugging, and reasoning about performance substantially easier [7].

The parallel-partition algorithm suffers from using a large amount of auxiliary memory, however. Whereas the serial algorithm is typically implemented in place, the parallel algorithm relies on the use of two auxiliary arrays of size n. To the best of our knowledge, the only known linear-work and polylog(n)-span algorithms for parallel partition that are in-place require the use of atomic operations (e.g, fetch-and-add) [5, 19, 26].

An algorithm's memory efficiency can be critical on large inputs. The memory consumption of an algorithm determines the largest problem size that can be executed in memory. Many external memory algorithms (i.e., algorithms for problems too large to fit in memory) perform large subproblems in memory; the size of these subproblems is again bottlenecked by the algorithm's memory-overhead [27]. In multi-user systems, memory efficiency is also important on small inputs, since processes with larger memory-footprints can hog the cache, slowing down other processes.

For sorting algorithms, in particular, special attention to memory efficiency is often given. This is because (a) a user calling the sort function may already be using almost all of the memory in the system; and (b) sorting algorithms, and

Note that strided algorithm has been shown by past work to perform well in practice

*Supported by a Hertz Fellowship and a NSF GRFP Fellowship
†Supported by MIT PRIMES.

APOCS '20, January 2020, Salt Lake City, UT, USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

[1] In several well-known textbooks and surveys on parallel algorithms [1, 6], for example, parallel partitions are implicitly used extensively to perform what are referred to as *filter* operations.

especially parallel sorting algorithms, are often bottlenecked by memory bandwidth.

In the context of parallel algorithms, however, the most practically efficient sorting algorithms fail to run in place, at least without the additional use of atomic-fetch-and-add variables [5, 19, 26], or the loss of theoretical guarantees on parallelism [12]. Parallel merge sort [16] was made in-place by Katajainen [20], but has proven too sophisticated for practical applications. Bitonic sort [8] 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 [5, 12, 13, 19, 26], has eluded any in-place EREW (or even CREW) algorithms due to its reliance on parallel partition.[2]

Results. We present a simple in-place algorithm for parallel partition that has work $O(n)$ and span $O(\log n \cdot \log\log n)$. The algorithm uses only exclusive read/write shared variables, and can be implemented using parallel-for-loops without any additional concurrency considerations. As an immediate consequence, we also get a simple in-place quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \log\log n)$.

Using our algorithmic techniques, we implement and optimize a space-efficient parallel partition. Because the in-place algorithm eliminates the use of large auxiliary arrays, the algorithm is able to achieve a significant reduction in cache misses over its out-of-place counterpart, resulting in performance improvements both in serial and in parallel.

We also present an alternative in-place parallel-partition algorithm with a larger span of $O(\sqrt{n \log n})$, but which is designed to have very small engineering overhead. On a single core, the algorithm performs within 25% of the serial GNU Libc quicksort partition algorithm. We show that when the algorithm is tuned appropriately, and given a large enough input to achieve high parallelism, the algorithm can be made to outperform is lower-span peers. The low-span algorithms, on the other hand, ensure good scaling with less sensitivity to the number of cores and the input size.

2 Preliminaries

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

Workflow Model. We consider a language-based model of parallelism in which algorithms may achieve parallelism through the use of ***parallel-for-loops***, though our algorithm also works in the less restrictive PRAM model [1, 6]. A parallel-for-loop is given a range $R \in \mathbb{N}$, a constant number of arguments $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_c$, 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 $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_c$. The threads then perform the body of the loop.[3]

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 the body of a parallel-for-loop has span s, then the full parallel loop has span $s + O(1)$ [1, 6].

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 [11] states that for any p,

$$T_1/p \leq T_p \leq T_1/p + T_\infty.$$

The work-stealing algorithms used in the Cilk extension of C/C++ realize the guarantee offered by Brent's Theorem within a constant factor [9, 10], with the added caveat that parallel-for-loops typically induce an additional additive overhead of $O(\log R)$.

Memory Model. Memory is ***exclusive-read*** and ***exclusive-write***. That is, no two threads are ever permitted to attempt to read or write to the same variable concurrently. The exclusive-read exclusive-write memory model is sometime referred to as the ***EREW model*** (see, e.g., [16]).

In an ***in-place*** algorithm, each thread is given $O(\log n)$ memory upon creation that is deallocated when the thread dies. This memory can be shared with the thread's children. The depth of the parent-child tree is not permitted to exceed $O(\log n)$.[4]

The Parallel Partition Problem. The parallel partition problem takes an input array A 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.

The (Standard) Linear-Space Parallel Partition. The linear-space implementation of parallel partition consists of two phases [1, 6]:

[2] In a ***CREW*** algorithm, reads may be concurrent, but writes may not. CREW stands for *concurrent-read exclusive-write*.

[3] Note that parallel-for-loops also implicitly allow for the implementation of parallel recursion by placing recursive function calls in the body of the parallel-for-loop.

[4] The algorithm in this paper satisfies a slightly stronger property that the total memory being used is never more than $O(\log n) \cdot p$, where p is an upper-bound on the number of worker threads.

The Parallel-Prefix Phase: In this phase, the algorithm constructs an array B whose i-th element $B[i] = \sum_{j=1}^{i} \text{dec}(A[i])$ is the number of predecessors in the first i elements of A. The transformation from A to B 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 A' of size $n/2$ with $A'[i] = A[2i-1] + A[2i]$; (2) Recursively construct a parallel prefix sum B' of A'; (3) Build B by setting each $B[i] = B'[\lfloor i/2 \rfloor] + A[i]$ for odd i and $B[i] = A'[i/2]$ for even i.

The Reordering Phase: In this phase, the algorithm constructs an output-array C by placing each predecessor $A[i] \in A$ in position $B[i]$ 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 $t + i - B[i]$. Since $i - B[i]$ 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 two auxiliary arrays of size n. Kiu, Knowles, and Davis [22] 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 space below $o(n)$ has remained open until now, even when the number of threads is fixed.

3 A Simple In-Place Algorithm

In this section, we present a simple in-place algorithm for parallel partition.

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 $j = B[i]$ of A, then the swaps will almost certainly conflict. Indeed, $A[j]$ may also be a predecessor that needs to be swapped with $A[B[j]]$. 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 B, 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 $A[i]$ with the corresponding element $B[A[i]]$. The fact that the prefix P is successor-heavy ensures that the final $\frac{1}{5} \cdot n$ elements of the reordered P will consist of successors. This implies that for each of the swaps between predecessors $A[i]$ and earlier positions $B[A[i]]$, the latter element will be a successor. In other words, the swaps are now conflict free.

Next consider how to remove the array B from the Parallel-Prefix Phase. At face value, this would seem quite difficult since the reordering phase relies heavily on B. Our solution is to *implicitly* store the value of every $O(\log n)$-th element of B 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 B. (If the elements are not all distinct, then a slightly more sophisticated encoding is necessary.) Moreover, we modify the algorithm for building B to only construct every $O(\log n)$-th element and to perform the construction also using implicitly storing values. 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. It proceeds in three phases.

A Preprocessing Phase. 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 goal of the preprocessing phase is to rearrange A so that every prefix is successor heavy.

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.

This ensures that at least half the successors in A reside in the first $\lceil n/2 \rceil$ positions. Thus 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.

Since $\lceil n/4 \rceil \geq \frac{\lceil n/2 \rceil}{2}$, the first $\lceil n/2 \rceil$ positions of A now contain at least as many successors as predecessors. Thus we can recursively repeat the same process on the subarray $[A[1], \ldots, A[\lceil n/2 \rceil]]$ in order to make each of its prefixes successor-heavy.

Each recursive step 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 $O(\log n)$.

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. In particular, X_i can be broken into (at least) $\lceil \log(n+1) \rceil$ disjoint pairs of adjacent elements, 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_i with work $O(b) = O(\log n)$ and span $O(\log b) = O(\log \log n)$ using a simple divide-and-conquer recursive approach.

After the preprocessing phase, our algorithm performs a parallel-for loop through the blocks, and stores 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 $O(\log \log n)$.

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

At the end of this phase of the algorithm, the array A satisfies two important properties: (1) Every block X_i encodes a value v_i counting the number of predecessors in the prefix $X_1 \circ X_2 \circ \cdots \circ X_i$; and (2) Each prefix $X_1 \circ X_2 \circ \cdots \circ X_i$ is 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 contain 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.

After P has been reordered, 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 is successor-heavy, we have that $|P_2| \geq |P|/4$, and thus that $|P_2| \geq |X_{t+1} \circ \cdots \circ X_n|$.

To complete the reordering of A, we perform a parallel-for-loop through each of the blocks $X_{t+1}, \ldots, X_n$. For each block X_i, we first extract v_i (with work $O(\log n)$ and span $O(\log \log n)$). We then create an auxiliary array Y_i of size $|X_i|$, using $O(\log n)$ memory from the thread in charge of Y_i in the parallel-for-loop. 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. Critically, because $|P_2| \geq |X_{t+1} \circ \cdots \circ X_n|$, we are guaranteed that the element with which $X_i[j]$ is being swapped is a successor in P_2. After the swaps have been performed, all of the elements of X_i are now successors.

Once the outer for-loop through the X_i's is complete, so will be the parallel partition of A. 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 3.1. There exists an in-place algorithm using exclusive-read-write variables that performs parallel-partition with work $O(n)$ and span $O(\log n \cdot \log \log n)$.

Allowing for Repeated Elements. In proving Theorem 3.1 we assumed for simplicity that the elements of A are distinct. This plays an important role in how we store the values v_i in the blocks X_i. To eliminate this requirement without changing the work and span of the algorithm, we can require that $b \geq 4\lceil \log(n+1) \rceil + 2$, and use the following slightly more complex encoding of the v_i's.

Consider the first b letters of X_i as a sequence of pairs, given by $(x_1, x_2), \ldots, (x_{b-1}, x_b)$. If at least half of the pairs consist of distinct elements, then we can reorder those pairs to appear at the front of X_i, and use them to encode values v_i. (For each X_i this reordering can be done once before the Implicit-Parallel-Prefix-Sum phase, adding only linear work and logarithmic span to the full algorithm.) If, on the other hand, at least half the pairs consist of equal-value elements, then we can reorder the pairs so that the first $\lceil \log(n+1) \rceil + 1$ of them satisfy this property. That is, if we reindex the reordered elements as $x_0, x_1, \ldots$, then $x_{2j+1} = x_{2j+2}$ for each $j = 0, 1, \ldots, \lceil \log(n+1) \rceil$. To encode a value v_i, we explicitly overwrite the second element in each of the pairs $(x_3, x_4), (x_5, x_6), \ldots$ with the bits of v_i, overwriting each element with one bit.

To read the value v_i, we check whether $x_1 = x_2$ in order to determine which encoding is being used and then unencode the bits appropriately. In the Reordering phase of the algorithm, once the blocks X_i are no longer required to encode

values, we can replace each overwritten x_i with its correct value (given by the value of the preceding element).

4 A Cache Efficient Parallel Partition

In Section 5, we will see that, although the techniques introduced in Section 3 achieve speedups over the classic parallel-prefix-based partition algorithm, they nonetheless continue to be bottlenecked by cache misses. In this section, we introduce a simple alternative algorithm, which we call 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.

Modeling Cache Misses. We treat memory as consisting of fixed-size cache lines of some size b. Each processor is assumed to have a small cache of polylog n cache lines. A cache miss occurs on a processor when the line being accessed is not currently in cache, in which case some other line is evicted from cache to make room for the new entry. Each cache is managed with a LRU (Least Recently Used) eviction policy; when child threads are created, they inherit their cache contents from their parent.

We will also assume that the algorithm can choose for certain small arrays to be pinned in cache (i.e. their entries are never evicted from cache). This assumption is without loss of generality in the sense that LRU eviction is competitive (up to resource augmentation) with the optimal off-line eviction strategy OPT (i.e. Furthest in the Future). Formally this is due to the following theorem by Sleator and Tarjan:

Theorem 4.1 (Resource Augmentation Theorem [25]). LRU operating on a cache of size $K \cdot M$ for some $K > 1$ will incur at most $1 + \frac{1}{K-1}$ times the number of times cache misses of OPT operating on a cache of size M, for the same series of memory accesses.

Recall that each processor has a cache of size $\log^c n$ for c a constant of our choice. Up to changes in c LRU incurs no more than a $1 + \frac{1}{\text{polylog}\, n}$ factor more cache misses than OPT incurs. Thus, up to a $1 + \frac{1}{\text{polylog}(n)}$ multiplicative change in cache misses, and a polylog(n) change in cache size, we may assume without loss of generality that cache eviction is performed by OPT. Such an assumption will not be necessary for our algorithm analyses, however; instead it will suffice to assume that certain small arrays are pinned in cache and that other evictions are performed via LRU.

The Strided Algorithm [12]. The Smoothed Striding Algorithm borrows several structural ideas from a previous algorithm of Francis and Pannan [12], 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, will build on the Strided Algorithm by randomly perturbing the internal structure of the original algorithm; in doing so, we are able to provide provable guarantees on arbitrary inputs, and to add a recursion step that was previously impossible.

The original ***Strided Algorithm*** consists of two steps:

- **The Partial Partition Step.** Let $g \in \mathbb{N}$ be a parameter, and assume for simplicity that $gb \mid n$. Partition the array A into $\frac{n}{gb}$ chunks $C_1, \ldots, C_{n/gb}$, each consisting of g cache lines of size b. what if we make everything, *even the entries in X* be indexed from 1? making the entries of X be indexed from 1 actually seems surprisingly straightforward: because X is a bunch of shifts in mod g, it doesn't matter whether $X[i] \in \{0, 1, \ldots, g-1\}$ or if we say $X[i] \in \{1, 2, \ldots, g\} \equiv \{1, 2, \ldots, g-1, 0\}$ mod g because $X[i]$ never occurs outside of a mod g expression! If we don't do this, I think that we should definitely put in a footnote saying that the entries $X[i]$ are the only 0 indexed quantities that the reader has to worry about For $i \in \{1, 2, \ldots, t\}$, define this should be For $i \in \{1, 2, \ldots, g\}$, because t no longer exists P_i to consist of the i-th cache line from each of the chunks $C_1, \ldots, C_{n/gb}$. One can think of the P_i's as forming a strided partition of array A, in which which consecutive cache lines in P_i are always separated by a fixed stride of $g-1$ other cache lines.
  The first step of the algorithm is to perform an in-place serial partition on each of the P_is, 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 $v_{\max} = \max\{v_1, \ldots, v_g\}$. Then the second step of the algorithm is to perform a serial partition on the sub-array $A[v_{\min}], \ldots, A[v_{\max}-1]$. everywhere else that you take a subarray of A you use the notation $A[v_{\min}, v_{\max}-1]$. Also, if the colon notation that you used to have here is common (anyone who programs in python knows it as slicing) and doesn't include the upper index (like in python array slicing), then I like it a lot better than the comma notation, and it is somewhat nicer because we don't have to have the -1 all over the place, i.e. Suggestion is to replace $A[v_{\min}, v_{\max}-1]$ everywhere with $A[v_{\min} : v_{\max}]$ This completes the full partition.

Note that the Cleanup Step of the Strided Algorithm has no parallelism, and thus has span $\Theta(v_{\max} - v_{\min})$. In general, this 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 $O(n)$. 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]

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 $U_1, U_1, \ldots, U_g$ (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 $C_1, C_2, \ldots, C_{n/gb}$ (rather than containing the i-th cache line of each C_j). repeated "that" sounds bad. suggested replacement: "This ensures that the number of predecessors in each U_i is a sum of independent random variables with values in $\{0, 1, \ldots, b\}$, the key property required to apply Hoeffding's Inequality." The key property that this ensures is that the number of predecessors in each U_i is a sum of independent random variables with values in $\{0, 1, \ldots, b\}$.

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 (even if the input array A is worst-case!).

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 $U_1, U_2, \ldots, U_g$ without violating the in-place nature of the algorithm. A natural approach might be to store for each U_i and each 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 kind of a general note, I think it would look better if we allways gave the same number of numbers in an array before doing ldots. i.e. $X_j \in \{0, 1, \ldots, g-1\}$ (the 2 is not really needed to understand the pattern) and then do the same thing every time (sometimes we have just 0,...sometimes 0,1,...sometimes 0,1,2,...,) $X_j \in \{0, 1, 2, \ldots, g-1\}$ 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 $O\left(\frac{n}{gb}\right)$ machine words storing the metadata $X_1, X_2, \ldots, X_{n/gb}$. By setting g to be relatively large, so that $\frac{n}{gb} \leq \text{polylog}(n)$, we can obtain an in-place algorithm that incurs $n(1 + o(1))$ cache misses.

The recursive structure of the Smoothed Striding Algorithm allows for the algorithm to achieve polylogarithmic span. As an alternative to recursing, one can also use the in-place algorithm from Section 3 in order to sort I think you meant partition... $A[v_{\min}, v_{\max} - 1]$. This results in an improved span (since the algorithm from Section 3 has span only $O(\log n \log\log n)$), while still incurring only $n(1 + o(1))$ cache misses (since the cache-inefficient algorithm from Section 3 is only used on a small subarray of A). We analyze both the recursive version of the Smoothed Striding Algorithm, and the version which uses as a final step the algorithm from Section 3; one significant advantage of the recursive version is that it is 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 $\frac{n}{bg} \leq \text{polylog}(n)$.) We assume for simplicity that that n is divisible by gb, and we define $s = \frac{n}{gb}$.[6]

The ***Partial Partition Step*** if the algorithm partitions the cache lines of A into g sets $U_1, \ldots, U_g$ of size $s = \frac{n}{gb}$ and then performs a serial partition on each of those sets U_i in parallel. To determine the sets $U_1, \ldots, U_g$, the algorithm uses as metadata, an array $X = X[1], \ldots, X[s]$, where each $X[i] \in \{1, \ldots, g\}$.

Formally, the Partial Partition Step performs the following procedure:

- Set each of $X[1], \ldots, X[s]$ to be uniformly random and independently selected elements of $\{0, 1, 2, \ldots, g-1\}$. For $i \in \{0, 1, 2, \ldots, g-1\}$, and for each $j \in \{0, 1, 2, \ldots, s-$

[5]The original algorithm of Francis and Pannan [12] does not consider the cache-line size b. Frias and Petit later introduced the parameter b [13], and showed that by setting b appropriately, one obtains an algorithm whose empirical performance is close to the state-of-the-art.

[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 successorswe are not treating the final "extra" elements as successors, we are ignoring them while all the serial partitions happen, and then swapping the final "extra" elements with the first $n - gb$ successors to add the extra elements to the recursive subproblem. (which consequently the algorithm needs not explicitly access).

$1\}$, define

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

because X is 1 indexed the jg should really be $(j-1)g$Using this terminology, we define each U_i for $i \in \{0, \ldots, g-1\}$ should be $\{1, \ldots, g\}$ to contain the $G_i(j)$-th cache line of A for each $j \in \{0, 1, 2, \ldots, s-1\}$X is indexed by 1, i.e. this should be $j \in \{1, 2, \ldots s\}$. That is, $G_i(j)$ denotes the index of the j-th cache line from array A to be 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 the this is wrong, should be $U_1, \ldots, U_g$ $U_0, \ldots, U_s$ 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 $i \in \{1, \ldots, s\}$ this is wrong, $i \in \{1, \ldots, g\}$ (g is big, s is small, if there were only s v_ys then it wouldn't be a problem to store all the v_ys (see footnote) because we wouldn't incur asymptotically more cache misses than we do for storign X) is defined to be the index of the first successor in A (or $n+1$ if no such successor exists).[7]

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

The second step of the Smoothed Striding Algorithm is to complete the partitioning of $A[v_{\min}, v_{\max} - 1]$. This can be done in one of two ways: The ***Recursive Smoothed Striding Algorithm*** partitions $A[v_{\min}, v_{\max} - 1]$ recursively using the same algorithm (and resorts to a serial base case when the subproblem is small enough that $g \leq O(1)$); the ***Hybrid Smoothed Striding Algorithm*** partitions $A[v_{\min}, v_{\max} - 1]$ using the in-place algorithm given in Theorem 3.1 with span $O(\log n \log\log n)$. 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 is simple to implement as fully in place, and still achieves polylogarithmic span. We analyze both algorithms in this section.

[7] One can calculate $v_{\min}$ and $v_{\max}$ without explicitly storing each of $v_1, \ldots, v_s$ this should be $v_1, \ldots, v_g$ as follows. Rather than using a standard s-wayg-way parallel for-loop to partition each of $U_1, \ldots, U_s$, $U_1, \ldots, U_g$ 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 is does not affect the overall span asymptotically.

Algorithm Analysis. Our first proposition analyzes the Partial Partition Step.

Proposition 4.1. Let $\epsilon \in (0, 1/2)$ and $\delta \in (0, 1/2)$ such that $\epsilon \geq \frac{1}{\text{poly}(n)}$ and $\delta \geq \frac{1}{\text{polylog}(n)}$. Suppose $s > \frac{\ln(n/\epsilon)}{\delta^2}$. Finally, suppose that each processor has a cache of size at least $s + c$ for a sufficiently large constant c.

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

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

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

Assuming that array X is pinned in cache (note, in particular, that $|X| = s \leq \text{polylog}(n)$, and so we are permitted to pin X in cache), algorithm's cache misses consist of: n/b misses from accessing each cache line of A; s/b for instantiating the array X; and $O(1)$ for other instantiating costs. This sums to

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

Note, in particular, that when each cache line in A is accessed, that line continues to be among the $O(1)$ most recently accessed cache lines until the final time that it is accessed, and thus does not get evicted from cache.

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. Since, by construction, $G_i(j)$ has the same probability distribution 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_i's is there supposed to be an apostrophe here (μ_i isn't possessing anything...)? 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 found, 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 y simultaneously satisfy $|\mu - \mu_y| < \delta$ implies that $v_{\max} - v_{\min} \leq 4n\delta$.

Let $U_i(j)$ denote the index in A of the i-th cache line in U_i should be Let... denote the index in A of the j-th cache line in U_i. i'm kind of confused by this derivation. I think you should add some explanation also, I think that the bounds that you get on $U_i(j)$ shouldn't be symmetric in fact, here is what I think the derivation should look like: Recall that $G_i(j)$ denotes the j th cache-line contained in U_i from A. Because of how the algorithm spreads out the cache-lines that it takes from the array we can bound $G_i(j)$ by

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

Note that v_i will occur in the $s\mu_i$-th cache-line of U_i because U_i is composed of s cache lines. Thus, we can bound v_i as

$$(s\mu_i g + 1)b + 1 \leq v_i \leq (s\mu_i g + g + 1)b.$$

Note that $sgb = n$, so this means that

$$b + 1 \leq v_i - n\mu_i \leq (g+1)b.$$

Then we can use our bound on $|\mu - \mu_i|$ to attain that for all i,

$$b + 1 - n\delta \leq v_i - n\mu \leq (g+1)b + n\delta.$$

Thus

$$v_{\max} - v_{\min} \leq 2n + bg < 4n\delta$$

Note that $U_i(j)$ satisfies

$$gj - bg + 1 \leq U_i(j) \leq gj + bg - 1$$

for all j. It follows that $v_i = U_i(\mu_i \cdot |U_i|)$ is within less than bg of $g\mu_j \cdot |U_i| = \mu_j n$. Therefore,

$$|v_i - \mu n| \leq bg + (\mu_j - \mu)n < bg + \delta n.$$

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

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

□

We will use Proposition 4.1 as a tool to analyze the Recursive and the Hybrid Smoothed Striding Algorithms.

Rather than parameterizing the Partial Partition step in each algorithm by s, Proposition 4.1 suggests that it is more natural to parameterize by ϵ and δ, which then determine s.

We will assume that both the hybrid and the recursive algorithms use $\epsilon = 1/n^c$ for c of our choice (i.e. 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), that way the entire algorithm succeeds with high probability in n.

For both algorithms, the choice of δ results in a tradeoff between cache misses and span. For the Recursive algorithm, we allow for 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 size 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.

Next we analyze the Hybrid Smoothed Striding Algorithm.

Theorem 4.2. The Hybrid Smoothed Striding Algorithm algorithm using parameter $\delta \in (0, 1/2)$ satisfying $\delta \geq 1/\text{polylog}(n)$: has 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 the above theorem concerns what happens when b is small (e.g., constant) and we choose δ to optimize span.

Corollary 4.2 (Corollary of Theorem 4.2). Suppose $b \leq o(\log\log n)$. Then the Cache-Efficient Full-Partition Algorithm algorithm using $\delta = \Theta\left(\sqrt{b/\log\log n}\right)$, 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 4.2. We analyze the Partial Partition Step using Proposition 4.1. 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 3.1, the subproblem of partitioning of $A[v_{\min}, v_{\max} - 1]$ 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.

[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, $1/16$.

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

Proof of Corollary 4.2. We use $\delta = \sqrt{b/\log\log n}$ in the result proved in Theorem 4.2.
First note that the assumptions of Theorem 4.2 are satisfied because $O(\sqrt{b/\log\log n}) > 1/\text{polylog}(n)$. 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. □

The next theorem analyzes the span of the Recursive Smoothed Striding Algorithm.

Theorem 4.3. With high probability in n, the Recursive Smoothed Striding algorithm using parameter $\delta \in (0, 1/2)$ satisfying $\delta \geq 1/\text{polylog}(n)$: 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 algorithm occurs at $\delta = 1/\sqrt{\log n}$.

Corollary 4.3 (Corollary of Theorem 4.3). With high probability in n, the its now called the Recursive Smoothed Striding Algorithm (not Grouped Partition)Grouped Partition Algorithm using parameter $\delta = 1/\sqrt{\log n}$: achieves work $O(n)$, attains span $O(b\log^2 n)$, and incurs $n/b \cdot (1 + O(1/\sqrt{\log n}))$ cache misses.

Proof of Theorem 4.3. 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 Theorem 4.1, 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 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 4.1, 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 4.1, at most

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

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 4.3. By Theorem 4.3, with high probability in n, the algorithm has work $O(n)$, the algorithm has span

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

and the algorithm incurs

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

cache misses. □

5 Performance Comparisons

In this section, we implement the techniques from Sections 3 and 4 to build space-efficient and in-place parallel-partition functions.

Each implementation considers an array of n 64-bit integers, and partitions them based on a pivot. The integers in the array are initially generated so that each is randomly either larger or smaller than the pivot.

In Subsection 5.1, we evaluate the techniques in Section 3 for transforming the standard parallel-prefix-based partition algorithm into an in-place algorithm. We compare the performance of three parallel-partition implementations: (1) The ***high-space*** implementation which follows the standard parallel-partition algorithm exactly; (2) a ***medium-space*** implementation which reduces the space used for the Parallel-Prefix Phase; and (3) a ***low-space*** implementation which further eliminates the auxiliary space used in the Reordering Phase of the algorithm. The low-space implementation still uses a small amount of auxiliary memory for the parallel-prefix, storing every $O(\log n)$-th element of the

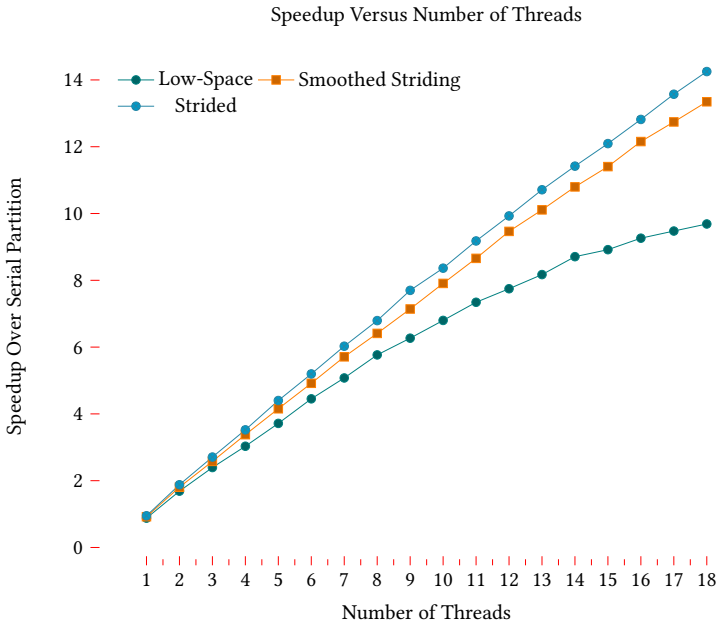

Figure 1. We compare the performance of the low-space and high-span sorting implementations running on varying numbers of threads and input sizes. The x-axis is the number of worker threads and the y-axis is the multiplicative speedup when compared to the Libc serial baseline. Each time (including each serial baseline) is averaged over five trials.

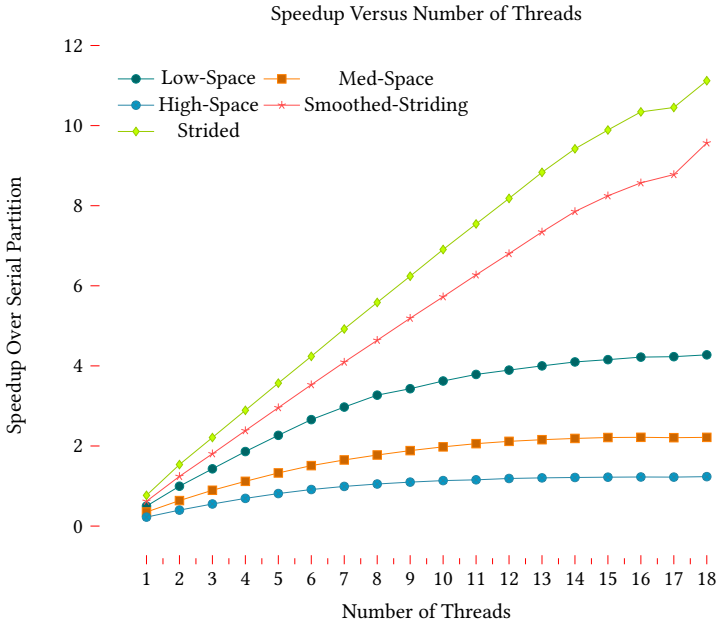

Figure 2. For a fixed table-size $n = 2^{30}$, we compare each implementation's runtime to the Libc serial baseline, which takes 3.9 seconds to complete (averaged over five trials). The x-axis plots the number of worker threads being used, and the y-axis plots the multiplicative speedup over the serial baseline. Each time (including the serial baseline) is averaged over five trials.

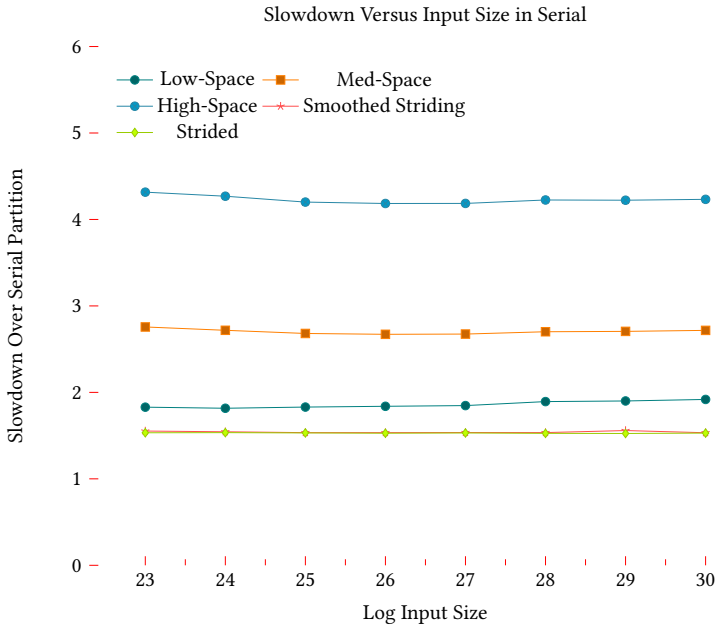

Figure 3. We compare the performance of the implementations in serial, with no scheduling overhead. The x-axis is the log-base-2 of the input size, and the y-axis is the multiplicative slowdown when compared to the Libc serial baseline. Each time (including the baseline) is averaged over five trials.

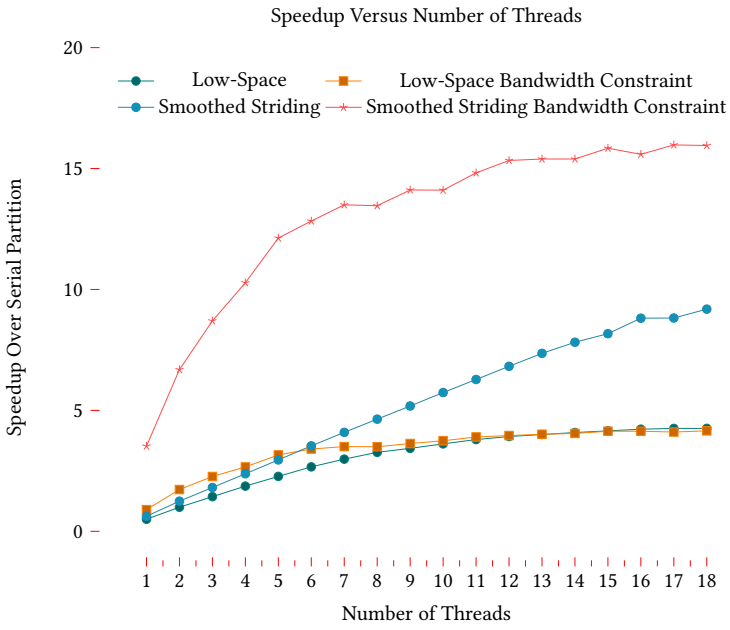

Figure 4. We compare the performances of the low-space and Smoothed Striding parallel-partition algorithms to their ideal performance determined by memory-bandwidth constraints on inputs of size 2^{30}. The x-axis is the number of worker threads, and the y-axis is the multiplicative speedup when compared to the Libc serial baseline (which is computed by an average over five trials). Each data-point is averaged over five trials.

parallel-prefix array explicitly rather than using the implicit-storage approach in Section 3. Nonetheless the space consumption is several orders of magnitude smaller than the original algorithm.

In addition to achieving a space-reduction, the better cache-behavior of the low-space implementation allows for it to achieve a speed advantage over its peers, in some cases completing roughly twice as fast as the medium-space implementation and four times as fast as the low-space implementation. We show that all three implementations are bottlenecked by memory throughput, however, suggesting that the cache-optimal Smoothed Striding Algorithm can do better.

In Subsection 5.2, we evaluate the performance of the Recursive Smoothed Striding Algorithm and the Strided Algorithms. The cache efficiency of the two algorithms allows for them to achieve substantially better scaling than their parallel-prefix-based counterparts. The Strided Algorithm tends to slightly outperform the Smoothed Striding Algorithm, though on 18 threads their performance is within 15% of one-another. We conclude that the Smoothed Striding Algorithm allows for one to obtain empirical performance comparable to that of the Strided Algorithm, while simultaneously achieving the provable guarantees on span and cache-efficiency missing from the original Strided Algorithm.

Machine Details. Our experiments are performed on a two-socket machine with eighteen 2.9 GHz Intel Xeon E5-2666 v3 processors. To maximize the memory bandwidth of the machine, we use a NUMA memory-placement policy in which memory allocation is spread out evenly across the nodes of the machine; this is achieved using the *interleave=all* option in the Linux *numactl* tool [21]. Worker threads in our experiments are each given their own core, with hyperthreading disabled.

Our algorithms are implemented using the CilkPlus task parallelism library in C++. The implementations avoid the use of concurrency mechanisms and atomic operations, but do allow for concurrent reads to be performed on shared values such as n and the pointer to the input array. Our code is compiled using g++ 7.3.0, with *march=native* and at optimization level three.

Our implementations are available at github.com/awestover/Parallel-Partition.

5.1 Comparing Parallel-Prefix-Based Algorithms

In this section, we compare four partition implementations, incorporating the techniques from Section 3 in order to achieve space efficiency:

- *A Serial Baseline:* This uses the serial in-place partition implementation from GNU Libc quicksort, with minor adaptions to optimize it for the case of sorting 64-bit integers (i.e., inlining the comparison function, etc.).
- *The High-Space Parallel Implementation:* This uses the standard parallel partition algorithm [1, 6], as described in Section 2. The space overhead is roughly $2n$ eight-byte words.
- *The Medium-Space Parallel Implementation:* Starting with the high-space implementation, we reduce the space used by the Parallel-Prefix Phase by only constructing every $O(\log n)$-th element of the prefix-sum array B, as in Section 3. (Here $O(\log n)$ is hard-coded as 64.) The array B is initialized to be of size $n/64$, with each component equal to a sum of 64 elements, and then a parallel prefix sum is computed on the array B. Rather than implicitly encoding the elements of B in A, we use an auxiliary array of size $n/64$ to explicitly store the prefix sums.
  The algorithm has a space overhead of $\frac{n}{32} + n$ eight-byte words.[9]
- *The Low-Space Parallel Implementation:* Starting with the medium-space implementation, we make the reordering phase completely in-place using the preprocessing technique in Section 3.[10] The only space overhead in this implementation is the $\frac{n}{32}$ additional 8-byte words used in the prefix sum.

We remark that the ample parallelism of the low-space algorithm makes it so that for large inputs the value 64 can easily be increased substantially without negatively effecting algorithm performance. For example, on an input of size 2^{28}, increasing it to 4096 has essentially no effect on the empirical runtime while bringing the auxiliary space-consumption down to a $\frac{1}{2048}$-fraction of the input size. (In fact, the increase from 64 to 4096 results in roughly a 5% speedup.)

An Additional Optimization for The High-Space Implementation. The optimization of reducing the prefix-sum by a factor of $O(\log n)$ at the top level of recursion, rather than simply by a factor of two, can also be applied to the standard parallel-prefix algorithm when constructing a prefix-sum array of size n. Even without the space reduction, this reduces the (constant) overhead in the parallel prefix sum, while keeping the overall span of the parallel-prefix operation at $O(\log n)$. We perform this optimization in the high-space implementation.

Performance Comparison. Figure 2 graphs the speedup of the each of the parallel algorithms over the serial algorithm, using varying numbers of worker threads on an 18-core machine with a fixed input size of $n = 2^{30}$. Both space optimizations result in performance improvements, with the low-space implementation performing almost twice as well

[9] In addition to the auxiliary array of size $n/64$, we use a series of smaller arrays of sizes $n/128$, $n/256$, ... in the recursive computation of the prefix sum. The alternative of performing the parallel-prefix sum in place, as in Section 3, tends to be less cache-friendly in practice.

[10] Depending on whether the majority of elements are predecessors are successors, the algorithm goes down separate (but symmetric) code paths. In our timed experiments we test only with inputs containing more predecessors than successors, since this the slower of the two cases (by a very slight amount) for our implementation.

as the medium-space implementation on eighteen threads, and almost four times as well as the high-space implementation.

Figure 3 compares the performances of the implementations in serial. Parallel-for-loops are replaced with serial for-loops to eliminate scheduler overhead. As the input-size varies, the ratios of the runtimes vary only slightly. The low-space implementation performs within a factor of roughly 1.9 of the serial implementation. As in Figure 2, both space optimizations result in performance improvements.

The Source of the Speedup. If we compare the number of instructions performed by the three parallel implementations, then the medium-space algorithm would seem to be the clear winner. Using Cachegrind to profile the number of instructions performed in a (serial) execution on an input of size 2^{28},[11] the high-space, medium-space, and low-space implementations perform 4.4 billion, 2.9 billion, and 4.6 billion instructions, respectively.

Cache misses tell a different story, however. Using Cachegrind to profile the number of top-level cache misses in a (serial) execution on an input of size 2^{28}, the high-space, medium-space, and low-space implementations incur 305 million, 171 million, and 124 million cache misses, respectively.

To a first approximation, the number of cache misses by each algorithm is proportional to the number of times that the algorithm scans through a large array. By eliminating the use of large auxiliary arrays, the low-space implementation has the opportunity to achieve a reduction in the number of such scans. Additionally, the low-space algorithm allows for steps from adjacent phases of the algorithm to sometimes be performed in the same pass. For example, the enumeration of the number of predecessors and the top level of the Preprocessing Phase can be performed together in a single pass on the input array. Similarly, the later levels of the Preprocessing Phase (which focus on only one half of the input array) can be combined with the construction of (one half of) the auxiliary array used in the Parallel Prefix Sum Phase, saving another half of a pass.

The Memory-Bandwidth Limitation. The comparison of cache misses suggests that performance is bottlenecked by memory bandwidth. To evaluate whether this is the case, we measure for each $t \in \{1, \ldots, 18\}$ the memory throughput of t threads attempting to scan through disjoint portions of a large array in parallel. We measure two types of bandwidth, the ***read-bandwidth***, in which the threads are simply trying to read from the array, and the ***read/write bandwidth***, in which the threads are attempting to immediately overwrite entries to the array after reading them. Given read-bandwidth r bytes/second and read/write bandwidth w bytes/second, the time needed for the low-space algorithm to perform its memory operations on an input of m bytes will be roughly $3.5m/w + .5m/r$ seconds. We call this the ***bandwidth constraint***. No matter how optimized the implementation of the low-space algorithm is, the bandwidth constraint serves as a hard lower bound for the running time.[12]

Figure 4 compares the time taken by the low-space algorithm to the bandwidth constraint as the number of threads t varies from 1 to 18. As the number of threads grows, the algorithm becomes bandwidth limited, achieving its best possible parallel performance on the machine. The algorithm scales particularly well on the first socket of the machine, achieving a speedup on nine cores of roughly six times better than its performance on a single core, and then scales more poorly on the second socket as it becomes bottlenecked by memory bandwidth.

Implementation Details. In each implementation, the parallelism is achieved through simple parallel-for-loops, with one exception at the beginning of the low-space implementation, when the number of predecessors in the input array is computed. Although CilkPlus Reducers (or OpenMP Reductions) could be used to perform this parallel summation within a parallel-for-loop [14], we found a slightly more ad-hoc approach to be faster: Using a simple recursive structure, we manually implemented a parallel-for-loop with Cilk Spawns and Syncs, allowing for the summation to be performed within the recursion.

5.2 Comparing the Smoothed Striding and Strided Algorithms

In this section we consider the performance of the Strided Algorithm and the Recursive Smoothed Striding Algorithm. Past work [13] found that, on large numbers of threads, the Strided Algorithm has performance close to that of other non-EREW state-of-the art partition algorithms (i.e., within 20% of the best atomic-operation based algorithms). The Strided Algorithm does not offer provable guarantees on span and cache-efficiency, however; and indeed, the reason that the algorithm cannot recurse on the subarray $A[v_{\min}, v_{\max} - 1]$ is that the subarray has been implicitly constructed to be worst-case for the algorithm. In this subsection, we show that, with only a small loss in performance, the Smoothed Striding Algorithm can be used to achieve provable guarantees on arbitrary inputs. We remark that we do not make any attempt to generate worst-case inputs for the Strided Algorithm (in fact the random inputs that we use are among the only inputs for which the Strided Algorithm does exhibit provable guarantees!).

[11] This smaller problem size is used to compensate for the fact that Cachegrind can be somewhat slow.

[12] Empirically, on an array of size $n = 2^{28}$, the total number of cache misses is within 8% of what this assumption would predict, suggesting that the bandwidth constraint is within a small amount of the true bandwidth-limited runtime.

Figures 3 and 2 evaluate the performance of the Smoothed Striding and Strided Algorithms in serial and in parallel. On a single thread, the Smoothed Striding and Strided Algorithms perform approximately 1.5 times slower than the Libc-based serial implementation baseline. When executed on multiple threads, the performances of the Smoothed Striding and Strided Algorithms scale close to linearly in the number of threads. On 18 threads, the Smoothed Striding Algorithm achieves a 9.6× speedup over the Libc-based Serial Baseline, and the Strided Algorithm achieves an 11.1× speedup over the same baseline.

The nearly-ideal scaling of the two algorithms can be explained by their cache behavior. Whereas the parallel-prefix-based algorithms were bottlenecked by memory bandwidth, Figure 4 shows that the same is no longer true for the Smoothed Striding Algorithm. The figure compares the performance of the Smoothed Striding Algorithm to the minimum time needed simple to read and overwrite each entry of the input array using 18 concurrent threads without any other computation (i.e., the memory bandwidth constraint). On 18 threads, the time required by the memory bandwidth constraint constitutes 58% of the algorithm's total running time.

NUMA Effects. We remark that the use of the Linux *numactl* tool [21] to spread memory allocation evenly across the nodes of the machine is necessary to prevent the Smoothed Striding Algorithm and the Strided Algorithm from being bandwidth limited. For example, if we replicate the 18-thread column of Figure 4 without using *numactl*, then the speedup of the Smoothed Striding Algorithm is 8.4, whereas the memory-bandwidth bound for maximum possible speedup is only slightly larger at 10.0.

Implementation Details. Both algorithms use $b = 512$. The Smoothed Striding Algorithm uses slightly tuned ϵ, δ parameters similar to those outlined in Corollary 4.3. Although $v_{\min}$ and $v_{\max}$ could be computed using CilkPlus Reducers [14], we found it advantageous to instead manually implement the parallel-for-loop in the Partial Partition step with Cilk Spawns and Syncs, and to compute $v_{\min}$ and $v_{\max}$ within the recursion.

Example Application: A Full Quicksort. In Figure 1, we graph the performance of a parallel quicksort implementation using the low-space parallel-prefix-based algorithm, the Smoothed Striding Algorithm, and the Strided Algorithm. We compare the algorithm performances with varying numbers of worker threads and input sizes to GNU Libc quicksort; the input array is initially in a random permutation.

Our parallel quicksort uses the parallel-partition algorithm at the top levels of recursion, and then swaps to the serial-partitioning algorithm once the input size has been reduced by at least a factor of $8p$, where p is the number of worker threads. By using the serial-partitioning algorithm on the small recursive subproblems we avoid the overhead of the parallel algorithm, while still achieving parallelism between subproblems. Small recursive problems also exhibit better cache behavior than larger ones, reducing the effects of memory-bandwidth limitations on the performance of the parallel quicksort, and further improving the scaling.

6 Conclusion and Open Questions

Parallel partition is a fundamental primitive in parallel algorithms [1, 6]. 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 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 bandwith 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 [12, 13], 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 [3, 4, 24], and integer sorting [2, 15, 17, 18, 23]? 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 be especially important when the computation is memory-bandwidth bound. The known in-place parallel algorithms for sample-sort rely

heavily on atomic instructions [5] (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.

Acknowledgments. The authors would like to thank Bradley C. Kuszmaul for several suggestions that helped simplify both the algorithm and its exposition. The authors would also like to thank Reza Zadeh for encouragement and advice.

This research was supported in part by NSF Grants 1314547 and 1533644.

References

[1] Umut A Acar and Guy Blelloch. 2016. Algorithm design: Parallel and sequential.

[2] Susanne Albers and Torben Hagerup. 1997. Improved parallel integer sorting without concurrent writing. *Information and Computation* 136, 1 (1997), 25–51.

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

[4] R_ Anderson. 1990. 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, 95–102.

[5] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-place Parallel Super Scalar Samplesort. *arXiv preprint arXiv:1705.02257* (2017).

[6] Guy E Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[7] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 181–192.

[8] Guy E. Blelloch, Charles E. Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. 1998. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems* 31, 2 (1998), 135–167.

[9] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.

[10] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[11] Richard P Brent. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)* 21, 2 (1974), 201–206.

[12] Rhys S. Francis and LJH Pannan. 1992. A parallel partition for enhanced parallel quicksort. *Parallel Comput.* 18, 5 (1992), 543–550.

[13] Leonor Frias and Jordi Petit. 2008. Parallel partition revisited. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 142–153.

[14] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 79–90.

[15] Alexandros V Gerbessiotis and Constantinos J Siniolakis. 2004. Probabilistic integer sorting. *Information processing letters* 90, 4 (2004), 187–193.

[16] Torben Hagerup and Christine Rüb. 1989. Optimal merging and sorting on the EREW PRAM. *Inform. Process. Lett.* 33, 4 (1989), 181–185.

[17] Yijie Han. 2001. 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, 793–796.

[18] Yijie Han and Xin He. 2012. More efficient parallel integer sorting. In *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management*. Springer, 279–290.

[19] Philip Heidelberger, Alan Norton, and John T. Robinson. 1990. Parallel quicksort using fetch-and-add. *IEEE Trans. Comput.* 39, 1 (1990), 133–138.

[20] Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. 1993. Space-efficient parallel merging. *RAIRO-Theoretical Informatics and Applications* 27, 4 (1993), 295–310.

[21] Andi Kleen. 2005. A numa api for linux. *Novel Inc* (2005).

[22] Jie Liu, Clinton Knowles, and Adam Brian Davis. 2005. 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, 491–502.

[23] Sanguthevar Rajasekaran and Sandeep Sen. 1992. On parallel integer sorting. *Acta Informatica* 29, 1 (1992), 1–15.

[24] Julian Shun, Yan Gu, Guy E Blelloch, Jeremy T Fineman, and Phillip B Gibbons. 2015. 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, 431–448.

[25] Daniel D Sleator and Robert E Tarjan. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM* 28, 2 (1985), 202–208.

[26] Philippas Tsigas and Yi Zhang. 2003. 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, 372.

[27] Jeffrey Scott Vitter. 2008. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science* 2, 4 (2008), 305–474.