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

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

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**.

KEYWORDS

Parallel Partition, EREW PRAM, in-place algorithms

ACM Reference Format:
William Kuszmaul. 2019. Simple and Space-Efficient Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19).* ACM, New York, NY, USA, Article 4, 12 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, 25].

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 [26]. 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 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, 25], or the loss of theoretical guarantees on

*Supported by a Hertz Fellowship and a NSF GRFP Fellowship

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SPAA '19, July 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s).
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.

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, 25], 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.

[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.

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]:

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

[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.

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

However, despite its theoretical appeal, the algorithm described above does not perform well in practice, as we show in the experiments section. The reason for the Strided Algorithm's poor performance is its lack of cache efficiency. In this section we provide a theoretical description of cache behavior, and use this model of cache behavior to design a new algorithm for parallel partition that is cache efficient and fast in practice.

Cache Behavior Preliminaries. In our model the ***cache*** is composed of a set of polylog n cache lines. A ***cache line*** is a set of polylog n values. Let ***block size*** denote the size of a cache line. A ***cache miss*** occurs when a program tries to access a value that is not loaded into cache at the time of the program's request. It takes time to read a value into cache, so the number of cache misses that an algorithm incurs plays a role in determining the running time of an algorithm in practice. We use the Least Recently Used (LRU) cache replacement algorithm to determine which cache line to discard when the cache is full. When the cache is full, the LRU cache replacement algorithm decides to evict the cache line that was least recently accessed of all cache lines currently in the cache. The LRU cache replacement algorithm is not the best cache replacement algorithm; the optimal cache replacement algorithm is known as OPT. OPT evicts the element that will be needed farthest into the future. The ability to predict

the future makes OPT impossible to implement in practice. The impossibility of implementing OPT initially seems to be an obstacle preventing analysis of cache behavior because for some access sequences OPT performs much better than LRU when both cache replacement algorithms operate on caches of the same size. However, if LRU is given a larger sized cache than OPT then LRU can achieve performance comparable to that of OPT. Tim Roughgarden's Resource Augmentation Theorem formalizes this idea,

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

We make $K = \operatorname{polylog} n$, so we can assume that LRU operating on cache of size $K \cdot M$ achieves nearly as good cache behavior as OPT operating on cache of size M. The specific behavior that we want is the ability to pin values to the cache. If there is a set of values that are accessed consistently throughout the program, then we would like to keep the values in cache all the time rather than periodically reloading them into cache. OPT would clearly accomplish this, because it could predict that these values would be needed in the future, and it would therefore not evict the values that we wished to be pinned to the cache. And, by the Resource Augmentation Theorem, LRU can be assumed to achieve this same cache behavior.

A Cache-Efficient Algorithm.

We now discuss an algorithm designed by Francis and Pannan [12], which is the only previously known algorithm of which we are aware that performs parallel-partition in-place using only exclusive-read-and-write memory. For certain inputs this algorithm achieves desirable cache behavior. The algorithm, which we call the ***Strided Algorithm*** consists of two steps:

- Partition the array A logically into t equal-size parts, denoted by $P_1, P_2, \ldots, P_t$ for some parameter t. Unlike the two-layer algorithm, the parts are interleaved, with each P_i consisting of array entries $A[i], A[i+t], A[i+2t], \ldots$. The first step of the algorithm is to perform a serial partition on each of the P_i's, rearranging the elements within the P_i so that the predecessors come first. This step has work $\Theta(n)$ and span $\Theta(n/t)$.
- For each P_i, define the ***splitting position*** v_i to be the position in A of the final predecessor in (the already partitioned) P_i. Define $v_{\min} = \min\{v_1, \ldots, v_t\}$ and define $v_{\max} = \max\{v_1, \ldots, v_t\}$. Then the second step of the algorithm is to perform a serial partition on the sub-array $A[v_{\min} : v_{\max}]$. This completes the full partition.

Note that Step 2 of the Strided Algorithm has no parallelism, with 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. For example, if A is randomly ordered, then one can use Chernoff bounds to prove that with high probability $v_{\max} - v_{\min} \leq O(\sqrt{n \cdot t \cdot \log n})$. The full span of the algorithm is then $\tilde{O}(n/t + \sqrt{n \cdot t})$, which optimizes at $t = n^{1/3}$ to $\tilde{O}(n^{2/3})$.

A more cache-friendly version of the algorithm, in which each part P_i consists blocks of b elements separated from each other by runs of length $(t-1)b$ was considered by Frias and Petit [13]. With this optimization, one advantage of the Strided Algorithm is that when $v_{\max} - v_{\min}$ is small, the total number of cache misses by the algorithm is close to the same as for a single scan through the data.

Algorithm Concept. We now present an improved version of the Strided Algorithm that has optimal cache behavior and low span for arbitrary inputs which we call the ***Grouped Partition Algorithm***. The Strided Algorithm relied on inputs being randomly ordered, but the Grouped Partition Algorithm uses randomization in the algorithm to obviate the need for random input. The Grouped Partition Algorithm forms collections U_y, which are similar to the Strided Algorithm's P_js, and performs a serial partition of each U_y in parallel. We clearly cannot explicitly store each U_y, because that would involve storing n values (or n/b if we use indices of cache blocks) which makes the algorithm not be in place, and lose its desirable cache behavior. However, we can represent all of the collections U_y using very little space by making each collection individually random, but not independent of other collections. This construction makes it so that the fraction of successors in each group will cluster closely around the fraction of successors in A. The elements in each U_y are spread out in A such each section of a certain size in the array will contain the same number of elements from U_y. Define v_y to be the index of the first successor in U_y–this is similar to the definition of the splitting position in the Strided Algorithm. Because of this uniformity in how the elements of U_y are spread out, and the guarantee that the fraction of successors in each U_y will cluster closely around the fraction of successors in A, the index v_y of the first successor in collection U_y will, with high probability in n, be close to all other indices v_y. This means that after each U_y is partitioned in serial A will almost be fully partitioned with only a small subarray left to partition. The subarray to partition is determined by the difference $\max v_y - \min v_y$.

Algorithm Description. We now describe how the algorithm creates the collections U_y.

- Logically partition the array A into blocks P_j each of b adjacent elements, i.e.

  $$P_j = \{A[b \cdot j], A[b \cdot j + 1], \ldots, A[b \cdot (j+1) - 1]\}.$$

- Form an array X with $|X| = s$, where each element $X[i]$ is an integer chosen randomly at uniform from $[0, g-1]$.
- The values in X determine g groups G_y each of s indices to P_js. The index for block i of group G_y is determined by

  $$G_y[i] = (X[i] + y) \bmod g + i \cdot g.$$

Note that we do not need to store the indices of all the members of each group, because the constituent blocks of the group are determined by the group index and the array X. This means that as long as $|s|$ is made small, which it will be, we do not use significant extra space in creating this array.

- Define U_y to be the union of all blocks that belong to group G_y. That is,

$$U_y = \bigcup_{j \in G_y} P_j.$$

The algorithm performs a serial partition on each U_y in parallel.

- Define v_y to be the index of the first successor in U_y. Define $v_{min} = \min v_y$ and $v_{max} = \max v_y$.
- After recursively applying the Grouped Partition Algorithm to the subarray $A[v_{min}], \ldots, A[v_{max} - 1]$ the array is fully partitioned.

Algorithm Analysis. The Grouped Partition Algorithm achieves low span and high parallelism with high probability in n for appropriate choice of the algorithms parameters. Define μ_y to be the fraction of elements of U_y that are successors, and μ to be the fraction of elements of A that are successors. We can show using Hoeffding bounds (i.e. Chernoff Bounds for random variables on $[0, 1]$ rather than on $\{0, 1\}$) that by setting $s = \Theta(\frac{n}{\delta^2})$ we get, with high probability in n, for all y $|\mu - \mu_y| < \delta$. This makes $v_{max} - v_{min} < O(\delta \cdot n)$, where $v_{max} - v_{min}$ is the size of the recursive subproblem that the algorithm must solve. The algorithm uses $\delta = 1/\sqrt{\log n}$ on the top layer of recursion and $\delta = 1/2$ on lower layers. This results in span $O(b \log^2 n)$. Note that the algorithm has work $O(n)$ because serial partition has linear work. Also note that the algorithm incurs only $n(1 + o(1))$ cache misses. In addition to these desirable theoretical guarantees, the algorithm performs well in practice. The algorithm achieves better parallelism and performance than the algorithms discussed above. Also, because the Grouped Partition Algorithm achieves an optimal number of cache misses, its bandwidth bound is optimal.

5 EXPERIMENTAL EVALUATION

In this section, we implement the techniques from Section 3 to build a space-efficient parallel-partition function. Our 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 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 step; 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 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.

In Subsection 5.2, we present a fourth parallel-partition algorithm which we call the ***two-layer algorithm***, and which runs fully in place but has a polynomial span of $\Theta(\sqrt{n \log n})$. The polynomial span of the algorithm makes it so that a naive implementation performs poorly. Nonetheless, we show that by tuning the algorithm to the number of worker threads, further speedup can often be achieved over the low-space algorithm.

The two-layer algorithm has the advantage that is very simple to implement, and runs in serial at almost the same speed as GNU Libc quicksort's serial algorithm. On the other hand the algorithm's performance is much more sensitive to input size and number of cores than is the low-space implementation. On a machine with sufficiently many cores (and sufficiently large memory bandwidth), the polylogarithmic span of the low-space implementation is desirable.

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/williamkuszmaul/Parallel-Partition.

5.1 Comparing Low-Span Algorithms

In this section, we compare four partition implementations:

- *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

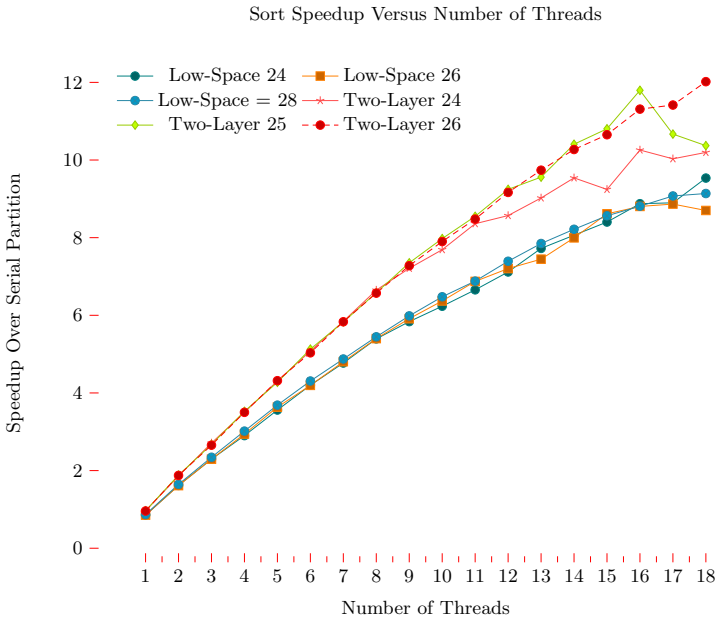

Figure 1: We compare the performance of the low-space and high-span sorting implementations running on varying numbers of threads and input sizes. The x-axis is the number of worker threads, the y-axis is the multiplicative speedup when compared to the serial baseline, and the log-base-two size of the input is indicated for each curve in the key. Each time (including each serial baseline) is averaged over five trials.

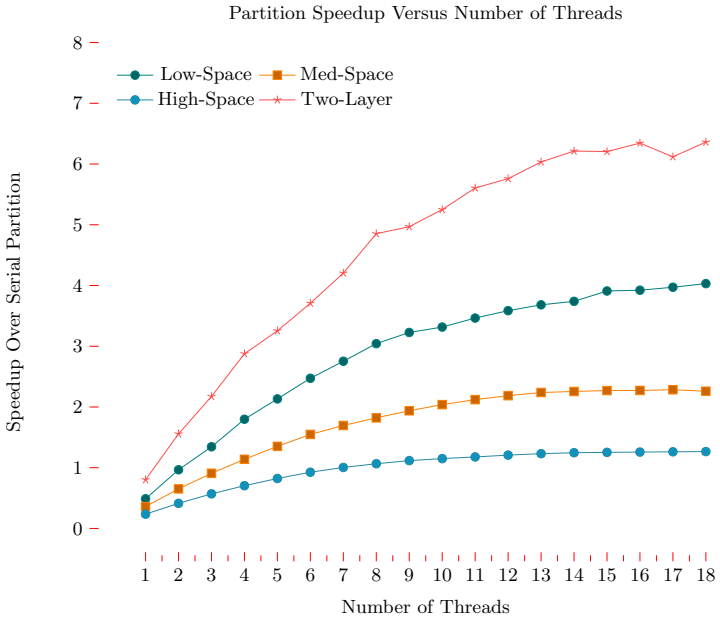

Figure 2: For a fixed table-size $n = 2^{28}$, we compare each implementation's runtime to a serial baseline, which takes 0.96 seconds to complete (averaged over five trials). The x-axis plots the number of worker threads being used, and the y-axis plots the multiplicative speedup over the serial baseline. Each time is averaged over five trials.

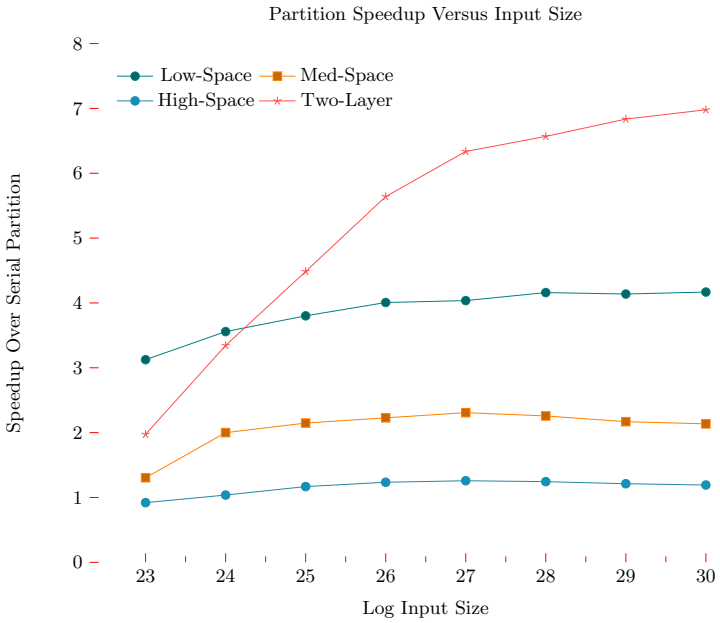

Figure 3: We compare the performance of the implementations running on eighteen worker threads on varying input sizes. The x-axis is the log-base-2 of the input size, and the y-axis is the multiplicative speedup when compared to the serial baseline. Each time (including each serial baseline) is averaged over five trials.

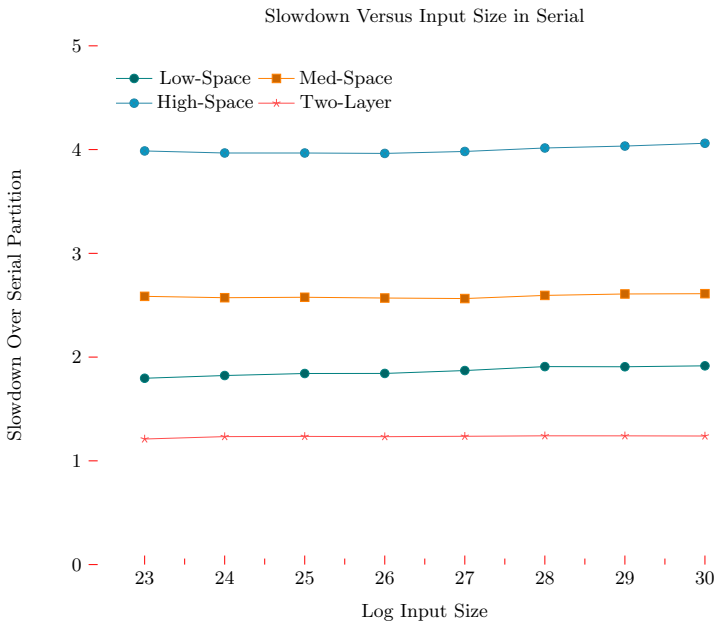

Figure 4: We compare the performance of the implementations in serial, with no scheduling overhead. The x-axis is the log-base-2 of the input size, and the y-axis is the multiplicative slowdown when compared to the serial baseline. Each time (including each serial baseline) is averaged over five trials.

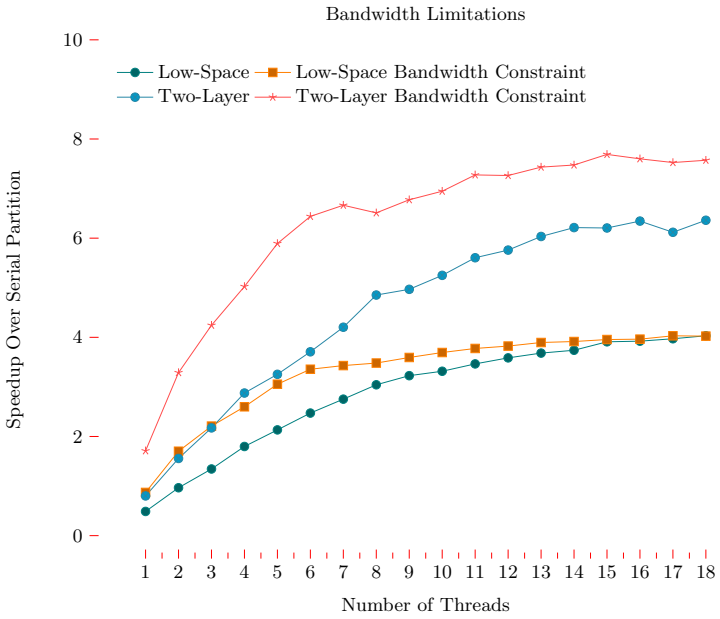

Figure 5: We compare the performances of the low-space and high-span parallel-partition algorithms to their ideal performance determined by memory-bandwidth constraints on inputs of size 2^{28}. The x-axis is the number of worker threads, and the y-axis is the multiplicative speedup when compared to the serial baseline (which is computed by an average over five trials). Each data-point is averaged over five trials.

$O(\log n)$-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. 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.[5] The algorithm has a space overhead of $\frac{n}{32} + n$ eight-byte words.[6]

- *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.[7] The only space overhead in this implementation is the $\frac{n}{32}$ additional 8-byte words used in the prefix sum.

All three parallel-implementations have work $O(n)$ The high- and medium- space implementations have span $O(\log n)$, while the low-space implementation has span $O(\log^2 n)$ (due to the fact that for convenience of implementation parallel-for-loops are broken into chunks of size $64 = O(\log n)$).

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 $n = 2^{28}$. Both space optimizations result in performance improvements, with the low-space implementation performing almost twice as well as the medium-space implementation on eighteen threads, and

[5] We suspect that an implementation in which the values are implicitly stored could also be made fast. In particular, the value 64 can be increased to compensate for whatever constant overhead is induced by the implicit storage of values. Nonetheless, the auxiliary array is already quite small relative to the input and is more practical to implement.

[6] In addition to the auxiliary array of size $n/64$, we use a series of smaller arrays of sizes $n/128, n/256, \ldots$ 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.

[7] 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.

almost four times as well as the high-space implementation. Similar speedups are achieved on smaller inputs; see Figure 3, which graphs speedup for input sizes starting at 2^{23}.

Figure 4 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 1.8 of the serial implementation. As in Tables 2 and 3, 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}, 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, as the number of worker threads grows, the performance of the low-space algorithm becomes 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***. Assuming that large scans through arrays are unaided by caching, the runtime of the low-space implementation is limited by the bandwidth constraint.[8]

Figure 5 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.

A Full Quicksort. In Figure 1, we graph the performance of a parallel quicksort implementation using our low-space algorithm. We compare the algorithm's performance to GNU Libc quicksort with varying numbers of worker threads and input sizes; 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 improving the scaling.

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; to amortize the cost of Cilk thread spawns.

5.2 An In-Place Algorithm with Polynomial Span

In this subsection, we consider a simple in-place parallel-partition algorithm with polynomial span. We evaluate the algorithm as a simple and even-lower-overhead alternative to the low-space algorithm in the previous subsection.

The algorithm takes two steps:

- **Step 1:** The algorithm breaks the input array A into t equal-sized parts, $P_1, \ldots, P_t$, for some parameter t. A serial partition is performed on each of the P_i's in parallel. This step takes work $\Theta(n)$ and span $\Theta(n/t)$.

[8] Empirically, 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.

- **Step 2:** The algorithm loops in serial through each of the t parts $P_1, \ldots, P_t$. Upon visiting P_i, the algorithm has already performed a complete partition on the subarray $P_1 \circ \cdots \circ P_{i-1}$. Let j denote the number of predecessors in $P_1, \ldots, P_{i-1}$, and k denote the number of predecessors in P_i. The algorithm computes k through a simple binary search in P_i. The algorithm then moves the k elements at the start of P_i to take the place of the k elements $A[j+1], \ldots, A[j+k]$. If the two sets of k elements are disjoint, then they are swapped with one-another in a parallel-for-loop; otherwise, the non-overlapping portions of the two sets of k elements are swapped in parallel, while the overlapping portion is left untouched. This completes the partitioning of the parts $P_1 \circ \cdots \circ P_i$. Performing this step for $i = 1, \ldots, t$ requires work $O(t \log n + n)$ and span $\Theta(t \log n)$.

Setting $t = \sqrt{n}$, the algorithm runs in linear time with span $\sqrt{n} \log n$; refining t to an optimal value of $\sqrt{n/\log n}$ results a span of $\sqrt{n \log n}$. In practice, however, this leaves too little parallelism in the parallel-for-loops in Step 2, resulting in poor scaling.[9] To mitigate this, we tune our implementation of the algorithm to the number of processors p on which it is being run, setting $t = 8 \cdot p$, in order to maximize the parallelism in the for-loops in Step 2, while still providing sufficient parallelism for Step 1.

Figures 2 and 3 compare the parallel performance of the algorithm, which is referred to as the ***two-layer algorithm***, to its lower-span peers. On 18 cores and on an input of size 2^{28}, the two-layer algorithm offers a speedup of roughly 50% over the low-space algorithm. The algorithm is more sensitive to input-size and number of cores, however, requiring a large enough ratio between the two to compensate for the algorithm's large span (See Figure 3).

Figure 4 compares the performance of the two-layer algorithm in serial to GNU Libc quicksort. The algorithm runs within a small fraction (less than 1/4) of the serial implementation.

Figure 5 evaluates the degree to which the algorithm is memory-bandwidth bound on an input of size 2^{28}. If the read/write bandwidth of the algorithm is w bytes/second, then the bandwidth constraint for the algorithm on an input of m bytes is given by $2m/w$. In particular, Step 1 of the algorithm makes one scan through the array, requiring time m/w; and then Step 2 rearranges the predecessors (which constitute half of the array and each must be moved to a new location), requiring additional time m/w. Figure 5 compares the time taken by the algorithm to the bandwidth constraint as the number of threads t varies from 1 to 18. Like the low-space algorithm, as the number of threads grows, the algorithm becomes close to bandwidth limited.

Figure 1 compares the performance of a quicksort implementation using the two-layer partition algorithm, to the performance of an implementation using the low-space algorithm. The implementation using the two-layer algorithm achieves a modest speedup over the low-space algorithm, but also demonstrates its larger span by suffering on smaller inputs as the number of cores grows.

[9] On 18 threads an on an input of size 2^{28}, for example, setting $t = \sqrt{n}$ results in a performance a factor of two slower than the low-space implementation, and setting $t = \sqrt{n/\log n}$ makes only partial progress towards closing that gap.

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 performance guarantees). In this paper, we have shown that, somewhat surprisingly, there exists a simple variant on the classic parallel algorithm that completely eliminates the use of auxiliary memory, while still using only exclusive read/write shared variables, and maintaining a polylogarithmic span. Moreover, our implementation is able to exploit the algorithm's superior cache-behavior in order to obtain practical speedups over its out-of-place counterpart. We have also presented an even simpler alternative, the two-layer algorithm, which has a larger polynomial span, but which can be tuned to perform well in practice.

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 author would like to thank Bradley C. Kuszmaul for several suggestions that helped simplify both the algorithm and its exposition. The author 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] 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.
[26] Jeffrey Scott Vitter. 2008. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science* 2, 4 (2008), 305–474.