

# **Cache-Efficient Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory**

An In-Place Algorithm With Provably Optimal Cache Behavior

**Alek Westover**

Belmont High School  
Massachusetts, USA

Mentor: William Kuszmaul (MIT)

MIT PRIMES Computer Science Research Program

# Cache-Efficient Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory

November 12, 2019

## Abstract

The parallel-partition problem, which is essential to Parallel Quicksort and appears in many other algorithms, is given an array  $A$  of length  $n$ , and must partition the array based on some pivot property. The standard solution to the parallel-partition problem is out-of-place. Having an in-place algorithm is desirable because it makes the algorithm faster in practice and because sorting problems are often memory intensive so extra space may be undesirable or high cost. Kuszmaul developed an in-place algorithm for the parallel-partition problem, but the algorithm performs multiple passes over the array and thus its performance is bottlenecked by memory-bandwidth. The Blocked Strided Algorithm of Francis, Pannan, Frias, and Petit is in-place and under certain conditions performs little more than a single pass over the array. Because of this, for certain inputs the Blocked Strided Algorithm incurs very few cache misses and thus performs well. However, in general this algorithm has no theoretical guarantees on span and cache-behavior.

We present an in-place EREW algorithm with polylogarithmic span and provably optimal cache behavior, up to small-order factors. The resulting algorithm achieves near-ideal scaling in practice by avoiding the memory-bandwidth bottleneck. The algorithm's performance is comparable to that of the Blocked Strided Algorithm, the previous state-of-the art for parallel EREW sorting algorithms.

## 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.<sup>1</sup>

A parallel algorithm's performance can be measured by its *work*, the time needed to execute in serial, and its *span*, the time needed 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

---

<sup>1</sup>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.

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].

This 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 multiple auxiliary arrays of size  $n$ . To the best of our knowledge, the only known linear-work and  $\text{polylog}(n)$ -span algorithms for parallel partition that are in-place require the use of atomic operations (e.g, fetch-and-add) [5, 20, 29].

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 [30]. In multi-user systems, processes with larger memory-footprints can hog the cache and the memory bandwidth, 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. The latter property, in particular, means that any parallel sorting algorithm that wishes to achieve state-of-the art performance on a large multi-processor machine must be (at least close to) in place.

Currently the only practical in-place parallel sorting algorithms either rely heavily on concurrency mechanisms such as atomic operations [5, 20, 29], or eschew theoretical guarantees [13]. Parallel merge sort [17] was made in-place by Katajainen [21], 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, 13, 14, 20, 29], has eluded any in-place EREW (or CREW) algorithms due to its reliance on parallel partition.<sup>2</sup>

**Results.** We consider the problem of designing a theoretically efficient parallel-partition algorithm that also performs well in practice. All of the algorithms considered in this paper use only exclusive read/write shared variables, and can be implemented using parallel-for-loops without any additional concurrency considerations.

We give a simple parallel-partition algorithm that simultaneously achieves good theoretical guarantees and near state-of-the-art empirical performance. In addition to incurring linear work

---

<sup>2</sup>In a **CREW** algorithm, reads may be concurrent, but writes may not. CREW stands for *concurrent-read exclusive-write*.

and polylogarithmic span, the algorithm exhibits provably optimal cache behavior up to low-order terms. In particular, if the input consists of  $m$  cache lines, then the algorithm incurs at most  $m(1 + o(1))$  cache misses, with high probability in  $m$ .

Concurrent work to ours has also developed a suite of techniques for transforming the standard linear-space parallel partition algorithm into an in-place algorithm [25]. The new algorithm, which we call Kuszmaul’s algorithm [25], is able to achieve a speedup over the linear-space algorithm due to its increased cache efficiency. Kuszmaul’s algorithm does not exhibit optimal cache behavior, however, and as we show in our experimental evaluation, the algorithm remains bottlenecked by memory throughput. In contrast, the cache-optimality of our algorithm eliminates the memory-throughput bottleneck, allowing for nearly perfect scaling on many processors.

The memory-bandwidth bottleneck has led past researchers [13, 14] to introduce the so-called **Strided Algorithm**, which has near optimal cache behavior in practice, but which exhibits theoretical guarantees only on certain random input arrays. The Strided Algorithm demonstrates the empirical importance of cache-behavior in the parallel-partition problem.

Our algorithm, which we call the **Smoothed Striding Algorithm** is designed to have similar empirical performance to the Strided Algorithm, while achieving both theoretical guarantees on work/span and on cache-optimality. This is achieved by randomly perturbing the internal structure of the Strided Algorithm, and adding a recursion step that was previously not possible. The Smoothed Striding Algorithm is in-place, has polylogarithmic span, and exhibits provably optimal cache behavior up to small-order factors. In practice, the Smoothed Striding Algorithm performs within 15% of the Strided Algorithm on a large number of threads.

## 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 simple language-based model of parallelism in which algorithms achieve parallelism through the use of **parallel-for-loops** (see, e.g., [1, 6, 12]). Function calls within parallel-for-loops then allow for more complicated parallel structures (e.g., recursion). Our algorithms can also be implemented in the less restrictive PRAM model [1, 6].

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

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

The **work**  $T_1$  of an algorithm is the time that the algorithm would require to execute on a single

processor. The *span*  $T_\infty$  of an algorithm is the time to execute on infinitely many processors. The scheduler is assumed to contribute no overhead to the span. In particular, if each iteration of a parallel-for-loop has span  $s$ , then the full parallel loop has span  $s + O(1)$  [1, 6].

The work  $T_1$  and span  $T_\infty$  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., [17]).

Note that threads are not in lockstep (i.e., they may progress at arbitrary different speeds), and thus the EREW model requires algorithms to be data-race free in order to avoid the possibility of non-exclusive data accesses.

In an *in-place* algorithm, each thread is given  $O(\text{polylog}(n))$  memory upon creation that is deallocated when the thread dies. This memory can be shared with the thread’s children. However, the depth of the parent-child tree is not permitted to exceed  $O(\text{polylog}(n))$ .

Whereas the EREW memory model prohibits concurrent accesses to memory, on the other side of the spectrum are CRCW (concurrent-read-concurrent-write) models, which allow for both reads and writes to be performed concurrently (and in some variants even allow for atomic operations) [1, 6, 24]. One approach to designing efficient EREW algorithms is to simulate efficient CRCW algorithms in the EREW model [24]. The known simulation techniques require substantial space overhead, however, preventing the design of in-place algorithms [24].<sup>3</sup>

**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  $\text{polylog}(n)$  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

---

<sup>3</sup>The known simulation techniques also increase the total work in the original algorithm, although this can be acceptable if only a small number of atomic operations need to be simulated.

eviction strategy OPT (i.e. Furthest in the Future). Formally this is due to the following theorem by Sleator and Tarjan:

*Theorem 2.1* (Resource Augmentation Theorem [28]). 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  $\text{polylog}(n)$  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 Partition Problem.** The partition problem takes an input array  $A$  of size  $n$ , and a *decider function*  $\text{dec}$  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 partition algorithm is to reorder the elements in the array  $A$  so that the predecessors appear before the successors.

**The Serial Partition Algorithm.** In our parallel algorithms for the partition problem we perform serial partitions on disjoint subarrays of the array. The serial partition algorithm runs fully in-place and incurs work linear in the input size i.e.  $O(n)$ . Also, crucially, the serial partition algorithm makes a single pass over the array, so it incurs only  $\lceil n/b \rceil$  cache misses where  $b$  is the size of a cache line.

**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 first creates an array  $D$  whose  $i$ -th element  $D[i] = \text{dec}(A[i])$ . Then the algorithm constructs an array  $S$  whose  $i$ -th element  $S[i] = \sum_{j=1}^i D[j]$  is the number of predecessors in the first  $i$  elements of  $A$ . The transformation from  $D$  to  $S$  is called a *parallel prefix sum* and can be performed with  $O(n)$  work and  $O(\log n)$  span using a simple recursive algorithm:

1. First construct an array  $D'$  of size  $n/2$  with  $D'[i] = D[2i-1] + D[2i]$
2. Recursively construct a parallel prefix sum  $S'$  of  $D'$
3. Build  $S$  by setting each  $S[i] = S'[\lceil i/2 \rceil] + A[i]$  for odd  $i$  and  $S[i] = S'[i/2]$  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  $S[i]$  of  $C$ . If there are  $t$  predecessors in  $A$ , then the first  $t$  elements of  $C$  will now contain those  $t$  predecessors in the same order that they appear in  $A$ . The algorithm then places each successor  $A[i] \in A$  in position  $t + i - S[i]$ . Since  $i - S[i]$  is the number of successors

in the first  $i$  elements of  $A$ , this places the successors in  $C$  in the same order that they appear in  $A$ . Finally, the algorithm copies  $C$  into  $A$ , completing the parallel partition.

Both phases can be implemented with  $O(n)$  work and  $O(\log n)$  span. Like its serial out-of-place counterpart, the algorithm is stable but not in place. The algorithm uses multiple auxiliary arrays of size  $n$ . Kiu, Knowles, and Davis [23] 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.

**An In-Place Parallel Partition.** In recent work, Kuszmaul [25] introduced a collection of theoretical techniques that could be used to make the standard parallel partition algorithm in place. The resulting algorithm achieves span  $O(\log n \log \log n)$ , which is only slightly larger than the  $O(\log n)$  span of the standard algorithm.

By being in place, Kuszmaul’s algorithm achieves fewer total cache misses than the standard algorithm. As a result, Kuszmaul’s algorithm substantially outperforms the standard linear-space parallel partition algorithm in practice.

On large numbers of processors, however, Kuszmaul’s algorithm becomes bottlenecked by memory bandwidth constraints, preventing linear scaling in performance. This is because, even though the algorithm is in-place, the multiple passes that it makes over the input cause poor cache behavior. In contrast, parallel algorithms that are designed to optimize cache behavior in practice, at the cost of lacking theoretical guarantees on span, can perform much faster [13, 14].

Like Kuszmaul’s algorithm, the algorithms presented in this paper are in place and achieve polylogarithmic span. Additionally, the algorithms achieve *provably optimal* cache behavior, up to small-order factors. In Section 4, we show that these theoretical guarantees also translate to practical speedups, allowing us to avoid the performance bottlenecks exhibited by Kuszmaul’s algorithm [25].

### 3 A Cache Efficient In-Place Parallel Partition

In this section we introduce the *Smoothed Striding Algorithm* for solving the parallel partition problem. The Smoothed Striding Algorithm exhibits provably optimal cache behavior (up to small-order factors), is fully in-place, and has polylogarithmic span.

**The Strided Algorithm [13].** The Smoothed Striding Algorithm borrows several structural ideas from a previous algorithm of Francis and Pannan [13], which we call the Strided Algorithm. The Strided Algorithm is designed to behave well on random arrays  $A$ , achieving span  $\tilde{O}(n^{2/3})$  and exhibiting only  $n/b + \tilde{O}(n^{2/3}/b)$  cache misses on such inputs. On worst-case inputs, however, the Strided Algorithm has span  $\Omega(n)$  and incurs  $n/b + \Omega(n/b)$  cache misses. Our algorithm, the Smoothed Striding Algorithm, builds on the Strided Algorithm by randomly perturbing the internal structure of the original algorithm; in doing so, we are able to provide provable performance

guarantees 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  $C_1, \dots, C_{n/gb}$ , each consisting of  $g$  cache lines of size  $b$ . For  $i \in \{1, 2, \dots, g\}$ , define  $P_i$  to consist of the  $i$ -th cache line from each of the chunks  $C_1, \dots, C_{n/gb}$ . One can think of the  $P_i$ 's as forming a strided partition of array  $A$ , since consecutive cache lines in  $P_i$  are always separated by a fixed stride of  $g - 1$  other cache lines.

The first step of the algorithm is to perform an in-place serial partition on each of the  $P_i$ s, rearranging the elements within the  $P_i$  so that the predecessors come first. This step requires work  $\Theta(n)$  and span  $\Theta(n/g)$ .

- **The Serial Cleanup Step.** For each  $P_i$ , define the *splitting position*  $v_i$  to be the position in  $A$  of the final predecessor in (the already partitioned)  $P_i$ . Define  $v_{\min} = \min\{v_1, \dots, v_g\}$  and define  $v_{\max} = \max\{v_1, \dots, v_g\}$ . Then the second step of the algorithm is to perform a serial partition on the sub-array  $A[v_{\min} + 1, \dots, A[v_{\max}]]$ . This completes the partition of  $A$ .

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  $\Theta(n)$ . For example, if  $b = 1$ , and if each element of  $A$  is selected independently from some distribution, then one can use Chernoff bounds to prove that with high probability in  $n$ ,  $v_{\max} - v_{\min} \leq O(\sqrt{n \cdot g \cdot \log n})$ . The full span of the algorithm is then  $\tilde{O}(n/g + \sqrt{n \cdot g})$ , which optimizes at  $g = n^{1/3}$  to  $\tilde{O}(n^{2/3})$ . Since the Partial Partition Step incurs only  $n/b$  cache misses, the full algorithm incurs  $n + \tilde{O}(n^{2/3})$  cache misses on a random array  $A$ .

Using Hoeffding's Inequality in place of Chernoff bounds, one can obtain analogous bounds for larger values of  $b$ ; in particular for  $b \in \text{polylog}(n)$ , the optimal span remains  $\tilde{O}(n^{2/3})$  and the number of cache misses becomes  $n/b + \tilde{O}(n^{2/3}/b)$  on an array  $A$  consisting of randomly sampled elements.<sup>4</sup>

**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  $U_1, \dots, U_g$  (which play a role analogous to  $P_1, \dots, P_g$  in the Strided Algorithm) so that each  $U_i$  contains one randomly selected cache line from each of  $C_1, \dots, C_{n/gb}$  (rather than containing the  $i$ -th cache line of each  $C_j$ ). This ensures that the number of predecessors in each  $U_i$  is a sum of independent random variables with values in  $\{0, 1, \dots, n/g\}$ .

---

<sup>4</sup>The original algorithm of Francis and Pannan [13] does not consider the cache-line size  $b$ . Frias and Petit later introduced the parameter  $b$  [14], and showed that by setting  $b$  appropriately, one obtains an algorithm whose empirical performance is close to the state-of-the-art.



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  $\mu$  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 final predecessor in (the already partitioned)  $U_i$ , then the difference between  $v_{\min} = \min_i v_i$  and  $v_{\max} = \max_i v_i$  will be small (regardless of the input array  $A$ !).

Rather than partitioning  $A[v_{\min} + 1], \dots, A[v_{\max}]$  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  $P'_1, \dots, P'_g$  used in the recursive subproblem would satisfy  $P'_1 \subseteq P_1, \dots, P'_g \subseteq P_g$  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  $U_1, \dots, U_g$  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  $X_j \in \{1, 2, \dots, g\}$  within each  $C_j$ , and then to assign the  $(X_j + i \pmod{g}) + 1$ -th cache line of  $C_j$  to  $U_i$  for  $i \in \{1, 2, \dots, g\}$ . This allows us to construct the  $U_i$ 's using only  $O\left(\frac{n}{gb}\right)$  machine words storing the metadata  $X_1, \dots, X_{n/gb}$ . 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 Kuszmaul's in-place algorithm [25] in order to partition  $A[v_{\min} + 1], \dots, A[v_{\max}]$ . This results in an improved span (since Kuszmaul's algorithm has span only  $O(\log n \log \log n)$ ), while still incurring only  $n(1 + o(1))$  cache misses (since Kuszmaul's cache-inefficient algorithm 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 Kuszmaul's algorithm; 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  $n$  is divisible by  $gb$ , and we define  $s = \frac{n}{gb}$ .<sup>5</sup>

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

---

<sup>5</sup>This assumption can be made without loss of generality by treating  $A$  as an array of size  $n' = n + (gb - n \pmod{gb})$ , and then treating the final  $gb - n \pmod{gb}$  elements of the array as being successors (which consequently the algorithm needs not explicitly access).

Formally, the Partial Partition Step performs the following procedure:

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

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

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

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

- The algorithm performs an in-place (serial) partition on each  $U_i$  (and performs these partitions in parallel with one another). In doing so, the algorithm, also collects  $v_{\min} = \min_i v_i$ ,  $v_{\max} = \max_i v_i$ , where each  $v_i$  with  $i \in \{1, \dots, g\}$  is defined to be the index of the final predecessor in  $A$  (or 0 if no such predecessor exists).<sup>6</sup>

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

The second step of the Smoothed Striding Algorithm is to complete the partitioning of  $A[v_{\min} + 1], \dots, A[v_{\max}]$ . This can be done in one of two ways: The ***Recursive Smoothed Striding Algorithm*** partitions  $A[v_{\min} + 1], \dots, A[v_{\max}]$  recursively using the same algorithm (and resorts to a serial base case when the subproblem is small enough that  $g \leq O(1)$ ); the ***Hybrid Smoothed Striding Algorithm*** partitions  $A[v_{\min} + 1], \dots, A[v_{\max}]$  using Kuszmaul's in-place algorithm with span  $O(\log n \log \log n)$  [25]. In general, the Hybrid algorithm yields better theoretical guarantees on span than the recursive version; on the other hand, the recursive version has the advantage that it is simple to implement as fully in-place, and still achieves polylogarithmic span. We analyze both algorithms in this section.

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

*Proposition 3.1.* 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  $c$ .

---

<sup>6</sup>One can calculate  $v_{\min}$  and  $v_{\max}$  without explicitly storing each of  $v_1, \dots, v_g$  as follows. Rather than using a standard  $g$ -way parallel for-loop to partition each of  $U_1, \dots, U_g$ , one can manually implement the parallel for-loop using a recursive divide-and-conquer approach. Each recursive call in the divide-and-conquer can then simply collect the maximum and minimum  $v_i$  for the  $U_i$ 's that are partitioned within that recursive call. This adds  $O(\log n)$  to the total span of the Partial Partition Step, which does not affect the overall span asymptotically.

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 serial 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), the 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  $\mu$  denote the fraction of elements in  $A$  that are predecessors. For  $i \in \{1, 2, \dots, g\}$ , let  $\mu_i$  denote the fraction of elements in  $U_i$  that are predecessors. Note that each  $\mu_i$  is the average of  $s$  independent random variables  $Y_i(1), \dots, Y_i(s) \in [0, 1]$ , where  $Y_i(j)$  is the fraction of elements in the  $G_i(j)$ -th cache line of  $A$  that are predecessors. By construction,  $G_i(j)$  has the same probability distribution for all  $i$ , since  $(X[j] + i) \pmod{g}$  is uniformly random in  $\mathbb{Z}_g$  for all  $i$ . It follows that  $Y_i(j)$  has the same distribution for all  $i$ , and thus that  $\mathbb{E}[\mu_i]$  is independent of  $i$ . Since the average of the  $\mu_i$ s is  $\mu$ , it follows that  $\mathbb{E}[\mu_i] = \mu$  for all  $i \in \{1, 2, \dots, g\}$ .

Since each  $\mu_i$  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  $\mu_i$  to show that it is tightly concentrated around its expected value  $\mu$ , 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  $i \in \{1, \dots, g\}$ ,

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

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

To complete the proof we will show that the occurrence of the event that all  $y$  simultaneously

satisfy  $|\mu - \mu_y| < \delta$  implies that  $v_{\max} - v_{\min} \leq 4n\delta$ . Recall that  $G_i(j)$  denotes the index within  $A$  of the  $j$ th cache-line contained in  $U_i$ . By the definition of  $G_i(j)$ ,

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

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

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

which means that

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

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

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

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

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

□

We will use Proposition 3.1 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 3.1 suggests that it is more natural to parameterize by  $\epsilon$  and  $\delta$ , 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  $\epsilon$  within recursive subproblems (i.e., the  $\epsilon$  is chosen based on the size of the first subproblem in the recursion), so that the entire algorithm succeeds with high probability in  $n$ .

For both algorithms, the choice of  $\delta$  results in a tradeoff between cache misses and span. For the Recursive algorithm, we allow for  $\delta$  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<sup>7</sup>. Optimizing  $\delta$

<sup>7</sup>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  $\delta$  to, say,  $1/16$ .

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 3.1.* 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  $\delta$  to optimize span.

*Corollary 3.2* (Corollary of Theorem 3.1). Suppose  $b \leq o(\log \log n)$ . Then the Cache-Efficient Full-Partition Algorithm algorithm using  $\delta = \Theta(\sqrt{b/\log \log n})$ , 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 3.1.* We analyze the Partial Partition Step using Proposition 3.1. Note that by our choice of  $\epsilon$ ,  $s = O\left(\frac{\log n}{\delta^2}\right)$ . 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.

Using Kuszmaul's in-place algorithm, the subproblem of partitioning  $A[v_{\min} + 1], \dots, A[v_{\max}]$  takes work  $O(n)$ . With high probability in  $n$ , the subproblem has size less than  $4n\delta$ , which means that the subproblem achieves span

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

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

The total number of cache misses is therefore,

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

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

*Proof of Corollary 3.2.* We use  $\delta = \sqrt{b/\log \log n}$  in the result proved in Theorem 3.1.

First note that the assumptions of Theorem 3.1 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.  $\square$

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

*Theorem 3.2.* 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 3.3* (Corollary of Theorem 3.2). With high probability in  $n$ , the Recursive Smoothed Striding Algorithm using parameter  $\delta = 1/\sqrt{\log n}$ : achieves work  $O(n)$ , attains span  $O(b \log^2 n)$ , and incurs  $n/b \cdot (1 + O(1/\sqrt{\log n}))$  cache misses.

*Proof of Theorem 3.2.* To avoid confusion, we use  $\delta'$ , rather than  $\delta$ , to denote the constant value of  $\delta$  used at levels of recursion after the first.

By Proposition 3.1, the top level of the algorithm has work  $O(n)$ , span  $O\left(b \frac{\log n}{\delta^2}\right)$ , and incurs  $\frac{s+n}{b} + O(1)$  cache misses. The recursion reduces the problem size by at least a factor of  $4\delta$ , 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  $\delta$  is set to be a sufficiently small constant). For each  $i > 1$ , it follows that the size of the problem at the  $i$ -th level of recursion is at most  $O(n\delta/2^i)$ .

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

Recall that each level  $i > 1$  uses  $s = \frac{\ln(2^{-i}n\delta'/b)}{\delta'^2}$ , where  $\delta' = \Theta(1)$ . It follows that level  $i$  uses  $s \leq O(\log n)$ . Thus, by Proposition 3.1, level  $i$  contributes  $O(b \cdot s) = O(b \log n)$  to the span. Since

there are at most  $O(\log n)$  levels of recursion, the total span in the lower levels of recursion is at most  $O(b \log^2 n)$ , and the total span for the algorithm is at most,

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

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

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

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 3.3.* By Theorem 3.2, 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.

□

## 4 Performance Comparisons

In this section, we implement the Smoothed-Striding algorithm and compare its performance to that of previously developed algorithms. The cache behavior of the Smoothed Striding algorithm allows it to outperform all past EREW algorithms with provable guarantees, and to perform competitively with the (Blocked) Strided algorithm (which does not exhibit provable guarantees). This is significant because, in addition to the Blocked Strided Argument representing the practical state-of-the-art for EREW algorithms, past work has also found Blocked Strided Algorithm to have 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) [14].

Each of the parallel-partition implementations that we test 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. We compare optimized implementations

of the following 5 partition algorithms:

- ***Serial Algorithm:*** A single-threaded in-place partition algorithm. This algorithm serves as a baseline to compare the parallel algorithms against. We use the implementation from GNU Libc quicksort, with minor adaptations to optimize it for the case of sorting 64-bit integers (i.e., inlining the comparison function, etc.).
- ***Standard Linear-Space Algorithm:*** The standard parallel-prefix-sum based algorithm. This algorithm has theoretically optimal span but uses linear auxiliary space.
- ***Kuszmaul’s In-Place Algorithm:*** Kuszmaul’s in-place parallel-prefix-sum based algorithm. Although Kuszmaul’s algorithm could, in principle, be implemented fully in-place, a more natural implementation approach in practice uses a small amount of auxiliary memory for the parallel-prefix sum, resulting in  $O(n/\log n)$  memory overhead. Nonetheless the space consumption is several orders of magnitude smaller than that of the Standard Linear-Space algorithm.
- ***Strided Algorithm:*** The (Blocked) Strided algorithm, which trades theoretical guarantees for practical performance. Note that, in our experiments, we make no effort to generate worst-case inputs for the Strided algorithm (in fact, a random array is one of the few inputs on which the Strided Algorithm does have theoretical guarantees!).
- ***Smoothed Striding Algorithm:*** The algorithm developed in this paper, which adds randomization to the Blocked Strided algorithm’s internal structure in order to achieve theoretical guarantees on span and cache-behavior for arbitrary inputs.

**Comparing running times in parallel and serial.** Figure 1 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  $n = 2^{30}$ . The optimal cache-behavior of Smoothed Striding Algorithm gives it a significant advantage over Kuszmaul’s In-Place Algorithm and the Standard Linear-Space Algorithm. On eighteen threads, the Smoothed Striding Algorithm runs roughly twice as fast as Kuszmaul’s In-Place Algorithm, and eight times as fast as the Standard Linear-Space Algorithm.

Both the Smoothed Striding and Strided algorithms scale close to linearly in the number of threads. On 18 threads, the Smoothed Striding Algorithm achieves a  $9.6\times$  speedup over the Libc-based Serial Baseline, and the Strided Algorithm achieves an  $11.1\times$  speedup over the same baseline.

Figure 2 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. By being (almost) in-place, Kuszmaul’s In-Place Algorithm is able to perform significantly better than the Standard Linear-Space Algorithm, running roughly



1.9 times slower than the serial implementation. The Smoothed Striding and Strided Algorithms further improve on this, running roughly 1.5-1.6 times slower than the serial implementation.

**Understanding the bottleneck: cache-misses.** In order to understand the importance of cache-misses in the performance of these algorithms, it is helpful to compare Kuszmaul’s In-Place Algorithm to the Standard Linear-Space Algorithm in more detail.

Using Cachegrind to profile the number of instructions performed in a (serial) execution on an input of size  $2^{28}$ , the Standard Linear-Space Algorithm actually performs *slightly fewer* instructions than its in-place counterpart, using 4.4 billion instructions instead of 4.6 billion.

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  $2^{28}$ , Kuszmaul’s algorithm incurs only 124 million cache misses, in comparison with the 305 million incurred by the Standard Linear-Space Algorithm.

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. This is why, by eliminating the use of large auxiliary arrays, Kuszmaul’s algorithm has the opportunity to achieve a reduction in the number of such scans; and why, by achieving optimal cache behavior, the Smoothed Striding Algorithm is able to do even better.

**Measuring the memory-bandwidth limitation.** The comparison of cache misses suggests that performance is bottlenecked by memory bandwidth for the non-cache-optimal algorithms. To evaluate whether this is the case, we measure for each  $t \in \{1, \dots, 18\}$  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 Kuszmaul’s In-Place 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 algorithm is, the bandwidth constraint serves as a hard lower bound for the running time.<sup>8</sup>

Figure 3 compares the time taken by Kuszmaul’s 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. This causes sub-linear scaling, especially on the second socket of the machine.

Whereas the parallel-prefix-based algorithms were bottlenecked by memory bandwidth, Figure 3 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 simply to read

---

<sup>8</sup>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.

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.

**Example Application: A Full Quicksort.** In Figure 4, we graph the performance of a parallel quicksort implementation using Kuszmaul’s In-Place 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.

**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 [22]. 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 on the author’s github.

**Further Implementation details.** Both the Smoothed Striding and the Strided algorithms use  $b = 512$ . The Smoothed Striding Algorithm uses slightly tuned  $\epsilon, \delta$  parameters similar to those outlined in Corollary 3.3. Although  $v_{\min}$  and  $v_{\max}$  could be computed using CilkPlus Reducers [15], 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.

## 5 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 con-

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

In this paper, we presented a new algorithm for the parallel partition problem. The Smoothed Striding Algorithm introduces randomization techniques to the previous (blocked) Striding Algorithm of [13, 14], which was known to perform well in practice but which has 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, 27], and integer sorting [2, 16, 18, 19, 26]? Such algorithms are of both theoretical and practical interest, and might be able to utilize some of the techniques introduced in this paper.

Another important direction of work is the design of in-place parallel algorithms for sample-sort, the variant of quicksort in which multiple pivots are used simultaneously in each partition. Sample-sort can be implemented to exhibit fewer cache misses than quicksort, which is especially important when the computation is memory-bandwidth bound. The known in-place parallel algorithms for sample-sort rely heavily on atomic instructions [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.

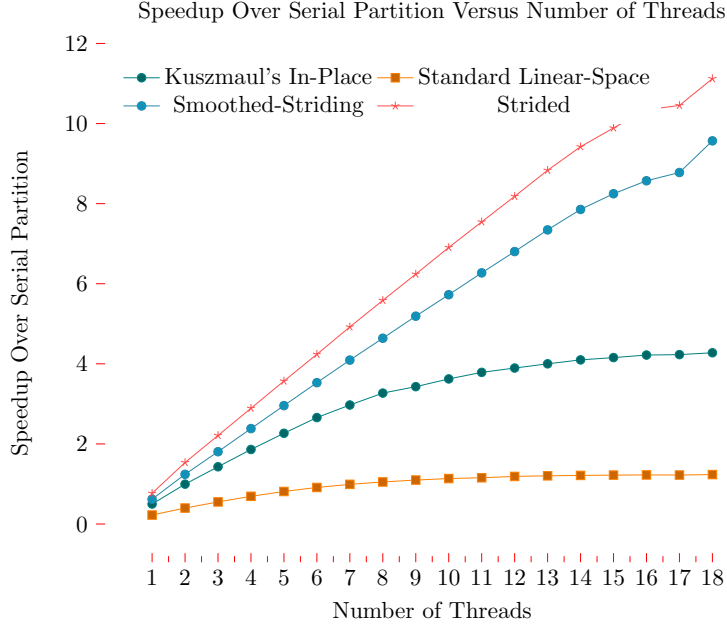


Figure 1: For a fixed table-size  $n = 2^{30}$ , we compare each implementation's runtime to the Libc serial baseline, which takes 3.9 seconds to complete (averaged over five trials). The  $x$ -axis plots the number of worker threads being used, and the  $y$ -axis plots the multiplicative speedup over the serial baseline. Each time (including the serial baseline) is averaged over five trials.

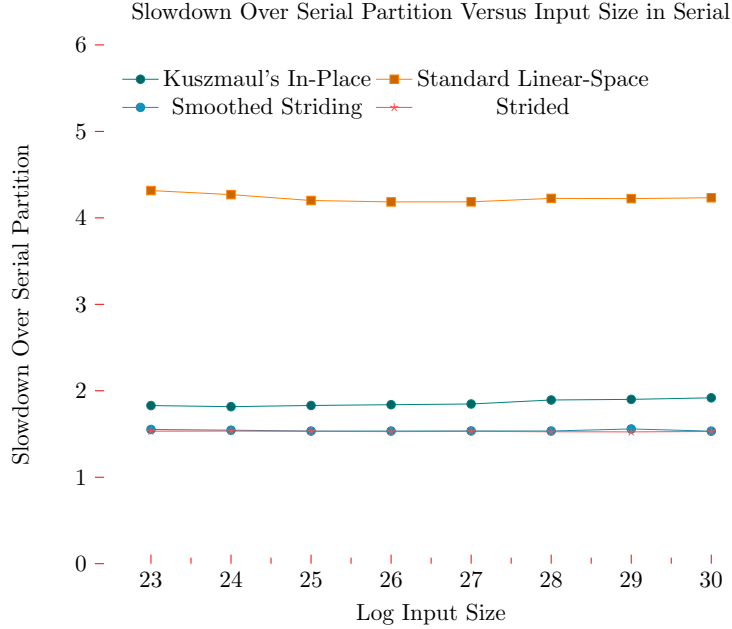


Figure 2: We compare the performance of the implementations in serial, with no scheduling overhead. The  $x$ -axis is the log-base-2 of the input size, and the  $y$ -axis is the multiplicative slowdown when compared to the Libc serial baseline. Each time (including the baseline) is averaged over five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

Speedup Over Serial Partition and Bandwidth Constraint Versus Number of Threads

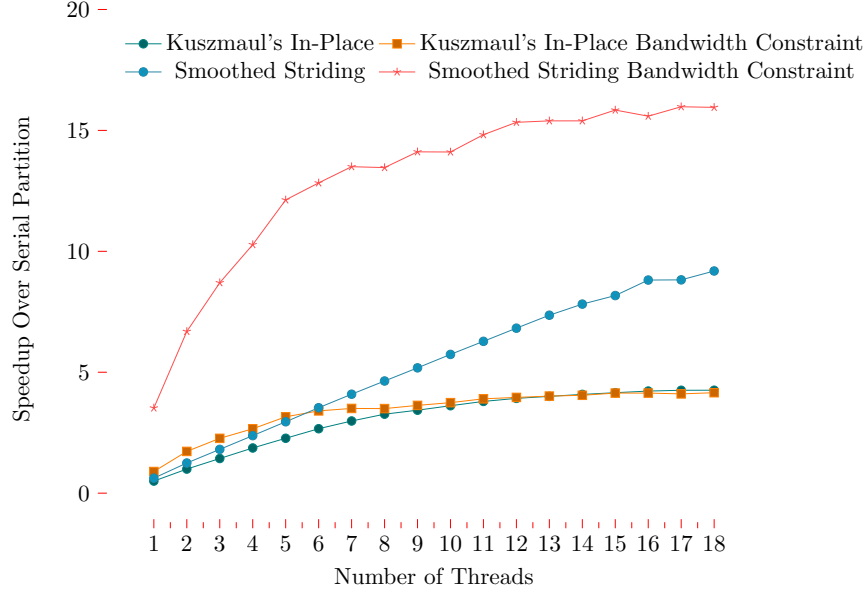


Figure 3: We compare the performances of the low-space and Smoothed Striding parallel-partition algorithms to their ideal performance determined by memory-bandwidth constraints on inputs of size  $2^{30}$ . The  $x$ -axis is the number of worker threads, and the  $y$ -axis is the multiplicative speedup when compared to the Libc serial baseline (which is computed by an average over five trials). Each data-point is averaged over five trials.

Speedup Of Quicksort Over Serial Partition's Quicksort Versus Number of Threads

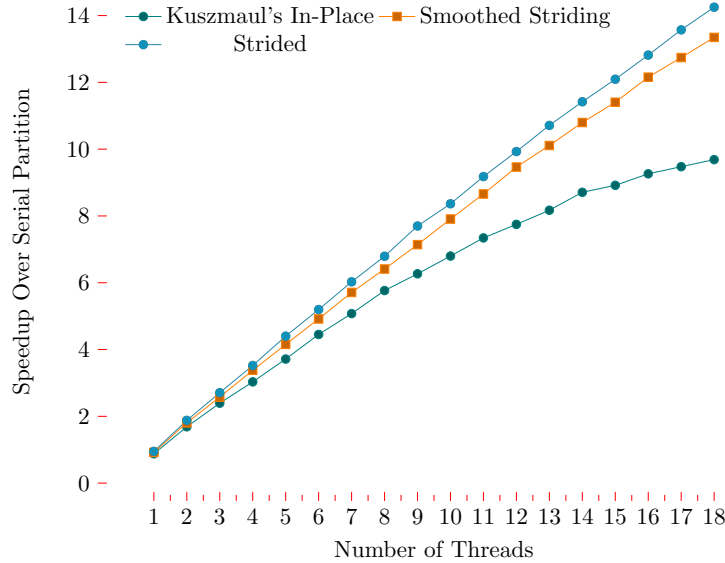


Figure 4: We compare the performance of the derived sorting implementations running on varying numbers of threads. The  $x$ -axis is the number of worker threads and the  $y$ -axis is the multiplicative speedup when compared to the Libc serial baseline. Each time (including each serial baseline) is averaged over five trials.

## References

- [1] Umut A Acar and Guy Blelloch. Algorithm design: Parallel and sequential, 2016.
- [2] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997.
- [3] Laurent Alonso and René Schott. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science*, 159(1):15–28, 1996.
- [4] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proceedings of the second annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–102. ACM, 1990.
- [5] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort. *arXiv preprint arXiv:1705.02257*, 2017.
- [6] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
- [7] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Notices*, volume 47, pages 181–192. ACM, 2012.
- [8] Guy E. Blelloch, Charles E. Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.
- [9] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.
- [10] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
- [11] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.
- [12] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- [13] Rhys S. Francis and LJH Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18(5):543–550, 1992.

- [14] Leonor Frias and Jordi Petit. Parallel partition revisited. In *International Workshop on Experimental and Efficient Algorithms*, pages 142–153. Springer, 2008.
- [15] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90. ACM, 2009.
- [16] Alexandros V Gerbessiotis and Constantinos J Siniolakis. Probabilistic integer sorting. *Information processing letters*, 90(4):187–193, 2004.
- [17] Torben Hagerup and Christine Rüb. Optimal merging and sorting on the erew pram. *Information Processing Letters*, 33(4):181–185, 1989.
- [18] Yijie Han. Improved fast integer sorting in linear space. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 793–796. Society for Industrial and Applied Mathematics, 2001.
- [19] Yijie Han and Xin He. More efficient parallel integer sorting. In *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management*, pages 279–290. Springer, 2012.
- [20] Philip Heidelberger, Alan Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.
- [21] Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. Space-efficient parallel merging. *RAIRO-Theoretical Informatics and Applications*, 27(4):295–310, 1993.
- [22] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.
- [23] Jie Liu, Clinton Knowles, and Adam Brian Davis. A cost optimal parallel quicksorting and its implementation on a shared memory parallel computer. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 491–502. Springer, 2005.
- [24] E Matias and Uzi Vishkin. A note on reducing parallel model simulations to integer sorting. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 208–212. IEEE, 1995.
- [25] Mentor and Student. In-place parallel-partition algorithms using exclusive-read-and-write memory. *APOCS*, 2019.
- [26] Sanguthevar Rajasekaran and Sandeep Sen. On parallel integer sorting. *Acta Informatica*, 29(1):1–15, 1992.
- [27] Julian Shun, Yan Gu, Guy E Blelloch, Jeremy T Fineman, and Phillip B Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proceedings*

- of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 431–448. Society for Industrial and Applied Mathematics, 2015.
- [28] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
  - [29] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 372. IEEE, 2003.
  - [30] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.