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

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

Alek Westover†
alek.westover@gmail.com

ABSTRACT

We present an in-place algorithm for the parallel-partition problem with linear work and polylogarithmic span. 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 EREW). A key feature of the algorithm is that it exhibits provably optimal cache behavior up to small-order factors.

We also present a second in-place EREW algorithm with work $O(n)$ and span $O(\log n \cdot \log\log n)$, which is within an $O(\log\log n)$ factor of the optimal span. By using this low-span algorithm as a subroutine within the cache-friendly algorithm, we obtain a single EREW algorithm that combines their theoretical guarantees: the algorithm achieves span $O(\log n \cdot \log\log n)$ and exhibits optimal cache behavior. As an immediate consequence, we also get an in-place EREW Quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \cdot \log\log n)$.

Whereas the standard EREW algorithm for parallel-partition is memory-bandwidth bound on large numbers of cores, our cache-friendly algorithm is able to achieve near-ideal scaling in practice by avoiding the memory-bandwidth bottleneck. Our algorithm's performance is comparable to that of the Blocked Strided Algorithm of Francis, Pannan, Frias, and Petit, which is the previous state-of-the-art for parallel EREW sorting algorithms, but which lacks theoretical guarantees on its span and cache behavior.

CCS CONCEPTS

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

KEYWORDS

Parallel Partition, EREW, in-place algorithms, cache-efficient

ACM Reference Format:
William Kuszmaul and Alek Westover. 2020. Brief Announcement In-Place Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400234

*Supported by a Hertz Fellowship, a NSF GRFP Fellowship, and NSF grant 1533644.
†Supported by MIT PRIMES.

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400234

1 INTRODUCTION

Given an array A of length n and a ***decider function*** $\text{dec} : A \to \{0, 1\}$ labelling the elements of A as ***predecessors*** (if $\text{dec}(A[i]) = 0$) and ***successors*** (if $\text{dec}(A[i]) = 1$), a ***partition*** operation rearranges the elements in an array so that predecessors occur before successors. 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, 2]. Moreover, the algorithm uses only exclusive read/write shared memory variables (i.e., it is an ***EREW*** algorithm). This parallel-partition algorithm suffers from using a large amount of auxiliary memory, however, and thus performs poorly in practice.

Results. We present an in-place EREW algorithm, that we call the ***Smoothed Striding Algorithm***, for the parallel partition problem with work $O(n)$ and span polylog(n) (Section 4). A key feature of this algorithm is that it exhibits provably optimal cache behavior up to small-order factors.

We also present a second in-place EREW algorithm with work $O(n)$ and span $O(\log n \cdot \log\log n)$, which is within an $O(\log\log n)$ factor of the optimal span (Section 3). By using this low-span algorithm as a subroutine within the Smoothed Striding algorithm, we are able to obtain a single EREW algorithm that combines their theoretical guarantees: the algorithm achieves span $O(\log n \cdot \log\log n)$ and optimal cache behavior (Theorem 4.1). As an immediate consequence, we also get an in-place EREW Quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \cdot \log\log n)$.

Whereas the standard EREW algorithm for parallel partitioning is memory-bandwidth bound on large numbers of cores, our cache-friendly algorithm is able to achieve near-ideal scaling in practice by being in-place.

The full analysis of the algorithms, along with the empirical evaluations are available in the extended paper [5].

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

2 PRELIMINARIES

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, 2]); function calls within the inner loop then allow for more complicated parallel structures (e.g., recursion). Our algorithms can also be implemented in the less restrictive PRAM model [1, 2].

Memory Model. Memory is ***exclusive-read*** and ***exclusive-write*** (i.e. we are in the EREW model). That is, no two threads are ever permitted to attempt to read or write to the same variable concurrently. 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 $O(\text{polylog } n)$.

Modeling Cache Misses. We treat memory as consisting of fixed-size cache lines, each of some size b. Each processor is assumed to have a small cache of polylog n cache lines. A cache miss occurs on a processor when the line being accessed is not currently in cache, in which case some other line is evicted from cache to make room for the new entry. Each cache is managed with the optimal off-line policy (furthest-in-the-future).

The (Standard) Linear-Space Parallel Partition. The linear-space implementation of parallel partition consists of two phases [1, 2]:
The Parallel-Prefix Phase: In this phase, the algorithm first creates an array D whose i-th element $D[i] = \text{dec}(A[i])$ indicates whether $A[i]$ is a predecessor ($\text{dec}(A[i]) = 0$ if $A[i]$ is a predecessor and $\text{dec}(A[i]) = 1$ if $A[i]$ is a successor). Then the algorithm constructs an array S whose i-th element $S[i] = \sum_{j=1}^{i} D[i]$ is the number of predecessors in the first i elements of A. The transformation from D to S is called a ***parallel prefix sum*** and can be performed with $O(n)$ work and $O(\log n)$ span using a simple recursive algorithm: (1) First construct an array D' of size $n/2$ with $D'[i] = D[2i-1]+D[2i]$; (2) Recursively construct a parallel prefix sum S' of D'; (3) Build S by setting each $S[i] = S'[\lfloor i/2 \rfloor] + A[i]$ for odd i 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. However, the algorithm uses multiple auxiliary arrays of size n. Reducing the extra space below $o(n)$ has remained open until now, even when the number of threads is fixed.

3 AN IN-PLACE PARTITION ALGORITHM WITH SPAN $O(\log n \log\log n)$

We now outline the key algorithmic ideas needed to make the classic Parallel Partition algorithm in-place with span $O(\log n \log\log n)$. Although it is in-place, and hence more cache-efficient than the standard out-of-place parallel partition algorithm, the algorithm described here still exhibits poor cache behavior, and performs poorly in practice; however, using this algorithm as a subroutine on a small subproblem in the Smoothed Striding Algorithm will allow the Smoothed Striding Algorithm to achieve span $O(\log n \log\log n)$ without sacrificing its good cache behavior.

Algorithm Outline. Prior to beginning the algorithm, the first implicit step of the algorithm is to count the number of predecessors in the array in order to determine whether the majority of elements are either predecessors or successors. 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 can be removed.

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

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

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

Analysis of the algorithm yields the following theorem:

Theorem 3.1. There exists an in-place algorithm using exclusive-read-write variables that performs parallel-partition with work $O(n)$ and span $O(\log n \cdot \log\log n)$.

4 A CACHE EFFICIENT IN-PLACE PARALLEL PARTITION ALGORITHM

The Strided Algorithm [3]. Our Smoothed Striding Algorithm borrows several structural ideas from a previous algorithm of Francis and Pannan [3], which we call the ***Strided Algorithm***[2]. 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 Strided Algorithm; in doing so, we are able to provide provable performance guarantees for arbitrary inputs and add a previously impossible recursion step. The original Strided Algorithm consists of two steps:

The Partial Partition Step. Let $g \in \mathbb{N}$ be a parameter, and assume for simplicity that $gb \mid n$. Partition the array A into $\frac{n}{gb}$ chunks $C_1, \ldots, C_{n/gb}$, each consisting of g cache lines of size b. For $i \in \{1, 2, \ldots, g\}$, define P_i to consist of the i-th cache line from each of the chunks $C_1, \ldots, C_{n/gb}$. 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_is, rearranging the elements within the P_i so that the predecessors come first. This step requires work $\Theta(n)$ and span $\Theta(n/g)$. *The Serial Cleanup Step.* For each P_i, define the ***splitting position*** v_i to be the position in A of the first successor in (the already partitioned) P_i. Define $v_{\min} = \min\{v_1, \ldots, v_g\}$ and define $v_{\max} = \max\{v_1, \ldots, v_g\}$. Then the second step of the algorithm is to perform a serial partition on the sub-array $A[v_{\min}], \ldots, A[v_{\max} - 1]$. This completes the full partition of A.

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

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

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

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

The recursive structure of the Smoothed Striding Algorithm allows for the algorithm to achieve polylogarithmic span. As an alternative to recursing, one can also use the in-place algorithm from Section 3 in order to partition the subarray. Doing so results in the following theorem:

Theorem 4.1. There exists an in-place EREW parallel-partition algorithm with work $O(n)$ that, with high probability in n, has span $O(\log n \log\log n)$ and incurs fewer than $(n + o(n))/b$ cache misses.

[2]The original algorithm of Francis and Pannan [3] does not consider the cache-line size b. Frias and Petit later introduced the parameter b [4] and showed that by setting b appropriately, one obtains an algorithm whose empirical performance is close to the state-of-the-art.

REFERENCES

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

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

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

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

[5] William Kuszmaul and Alek Westover. 2020. In-Place Parallel-Partition Algorithms using Exclusive-Read-and-Write Memory: An In-Place Algorithm With Provably Optimal Cache Behavior. *arXiv preprint arXiv:2004.12532* (2020).