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

An In-Place Algorithm With Provably Optimal Cache Behavior

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 that has 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 that has linear work 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 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. As an immediate consequence, we also get an in-place EREW quicksort algorithm with work $O(n \log n)$, 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 avoiding the memory-bandwidth bottleneck. The 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.

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, 2]. Moreover, the algorithm uses only exclusive read/write shared memory variables (i.e., it is an **_EREW_** algorithm). The parallel-partition algorithm suffers from using a large amount of auxiliary memory, however. An algorithm's memory efficiency can be critical on large inputs.

2 PRELIMINARIES

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

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., [1–3]); 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 $\text{polylog} n$ 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.

However, we may assume for the sake of analysis that each cache is managed by the optimal off-line eviction strategy OPT

[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.

(i.e. Furthest in the Future). This is because, up to resource augmentation, LRU eviction is $(1+1/\text{polylog} n)$-competitive with OPT. Formally this is due to the following theorem by Sleator and Tarjan:

Theorem 2.1 (Resource Augmentation Theorem [7]). 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.

Because each processor's cache is managed by OPT (without loss of generality), we can assume that each processor *pins* certain small arrays to cache (i.e., the elements of those arrays are never evicted).

3 A CACHE EFFICIENT IN-PLACE PARALLEL PARTITION

The Strided Algorithm [4]. The Smoothed Striding Algorithm borrows several structural ideas from a previous algorithm of Francis and Pannan [4], which we call the Strided Algorithm. The Strided Algorithm is designed to behave well on random arrays A, achieving span $\tilde{O}(n^{2/3})$ and exhibiting only $n/b + \tilde{O}(n^{2/3}/b)$ cache misses on such inputs. On worst-case inputs, however, the Strided Algorithm has span $\Omega(n)$ and incurs $n/b + \Omega(n/b)$ cache misses. Our algorithm, the Smoothed Striding Algorithm, builds on the Strided Algorithm by randomly perturbing the internal structure of the original algorithm; in doing so, we are able to provide provable performance guarantees for arbitrary inputs, and to add a recursion step that was previously impossible.

The 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.

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, $v_{\max} - v_{\min} \le O(\sqrt{n \cdot g \cdot \log n})$. 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.[2]

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

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

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

The main challenge in designing the Smoothed Striding Algorithm becomes the construction of $U_1, \ldots, U_g$ without violating the in-place nature of the algorithm. A natural approach might be to store for each U_i, C_j the index of the cache line in C_j that U_i contains. This would require the storage of $\Theta(n/b)$ numbers as metadata, however, preventing the algorithm from being in-place. To save space, the key insight is to

---

[2]The original algorithm of Francis and Pannan [4] does not consider the cache-line size b. Frias and Petit later introduced the parameter b [5], and showed that by setting b appropriately, one obtains an algorithm whose empirical performance is close to the state-of-the-art.

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 $O\left(\frac{n}{gb}\right)$ 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 Appendix ?? in order to partition $A[v_{\min}],\ldots,A[v_{\max}-1]$. This results in an improved span (since the algorithm from Appendix ?? has span only $O(\log n \log\log n)$), while still incurring only $n(1+o(1))$ cache misses (since the cache-inefficient algorithm from Appendix ?? is only used on a small subarray of A). We analyze both the recursive version of the Smoothed Striding Algorithm, and the version which uses as a final step the algorithm from Appendix ??; one significant advantage of the recursive version is that it is simple to implement in practice.

Formal Algorithm Description. Let $b < n$ be the size of a cache line, let A be an input array of size n, and let g be a parameter. (One should think of g as being relatively large, satisfying $\frac{n}{bg} \leq \text{polylog}(n)$.) We assume for simplicity that that n is divisible by gb, and we define $s = \frac{n}{gb}$. [3]

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

Formally, the Partial Partition Step performs the following procedure:

- Set each of $X[1],\ldots,X[s]$ to be uniformly random and independently selected elements of $\{1,2,\ldots,g\}$. For each $i \in \{1,2,\ldots,g\}, j \in \{1,2,\ldots,s\}$, define
  $$G_i(j) = (X[j]+i \pmod{g})+(j-1)g+1.$$
  Using this terminology, we define each U_i for $i \in \{1,\ldots,g\}$ to contain the $G_i(j)$-th cache line of A for each $j \in \{1,2,\ldots,s\}$. That is, $G_i(j)$ denotes the index of the j-th cache line from array A contained in U_i.
  Note that, to compute the index of the j-th cache line in U_i, one needs only the value of $X[j]$. Thus the only metadata needed by the algorithm to determine $U_1,\ldots,U_g$ is the array X. If $|X| = s = \frac{n}{gb} \leq \text{polylog}(n)$, then the algorithm is in place.
- The algorithm performs an in-place (serial) partition on each U_i (and performs these partitions in parallel

with one another). In doing so, the algorithm, also collects $v_{\min} = \min_i v_i$, $v_{\max} = \max_i v_i$, where each v_i with $i \in \{1,\ldots,g\}$ is defined to be the index of the first successor in A (or n if no such successor exists). [4]
The array A is now partially partitioned, i.e. $A[i]$ is a predecessor for all $i \leq v_{\min}$, and $A[i]$ is a successor for all $i > v_{\max}$.

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

Algorithm Analysis. We prove the following proposition regarding the Partial Partition Step of the Smoothed Striding Algorithm.

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

Then the Partial-Partition Algorithm achieves work $O(n)$; achieves span $O(b \cdot s)$; incurs $\frac{s+n}{b} + O(1)$ cache misses; and guarantees with probability $1-\epsilon$ that
$$v_{\max} - v_{\min} < 4n\delta.$$

Using this we can analyze the hybrid and recursive smoothed striding algorithms.

We will assume that both the hybrid and the recursive algorithms use $\epsilon = 1/n^c$ for c of our choice (i.e. with high probability in n). Moreover, the Recursive Smoothed Striding Algorithm continues to use the same value of ϵ within recursive subproblems (i.e., the ϵ is chosen based on the size of the first subproblem in the recursion), so that the entire algorithm succeeds with high probability in n.

For both algorithms, the choice of δ results in a tradeoff between cache misses and span. For the Recursive algorithm, we allow for δ to be chosen arbitrarily at the top level of recursion, and then fix $\delta = \Theta(1)$ to be a sufficiently small constant at all levels of recursion after the first; this guarantees that

---

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

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

we at least halve the size of the problem between recursive iterations[5]. Optimizing δ further (after the first level of recursion) would only affect the number of undesired cache misses by a constant factor.

Formally we prove the following performance results for these algorithms:

Corollary 3.2. Suppose $b \leq o(\log \log n)$. Then the Hybrid Smoothed Striding Algorithm using $\delta = \Theta\left(\sqrt{b/\log\log n}\right)$, achieves work $O(n)$, and with high probability in n, achieves span $O(\log n \log \log n)$ and incurs fewer than $(n + o(n))/b$ cache misses.

Corollary 3.3. 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.

4 PERFORMANCE COMPARISONS

Speedup Versus Number of Threads
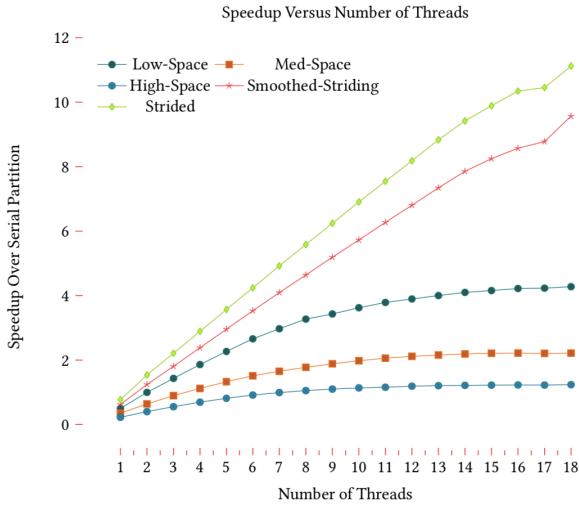
REFERENCES

[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] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.
[4] Rhys S. Francis and LJH Pannan. 1992. A parallel partition for enhanced parallel quicksort. *Parallel Comput.* 18, 5 (1992), 543–550.
[5] Leonor Frias and Jordi Petit. 2008. Parallel partition revisited. In *International Workshop on Experimental and Efficient Algorithms.* Springer, 142–153.
[6] Torben Hagerup and Christine Rüb. 1989. Optimal merging and sorting on the EREW PRAM. *Inform. Process. Lett.* 33, 4 (1989), 181–185.
[7] Daniel D Sleator and Robert E Tarjan. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM* 28, 2 (1985), 202–208.

---

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