A SIMPLE IN-PLACE PARALLEL PARTITION USING EXCLUSIVE-READ-AND-WRITE MEMORY

WILLIAM KUSZMAUL

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

Using our algorithmic techniques, we implement an (almost) in-place parallel-prefix. In addition to achieving much better memory utilization, the algorithm achieves a modest speedup over its out-of-place counterpart.

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 $O(\log n)$ [**?**, **?**]. 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 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.

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

The parallel algorithm differs from its serial counterpart in the use of extra memory, however. Whereas the serial algorithm is typically implemented in place, the parallel algorithm relies on the use of two auxiliary arrays of size n. To the best of our knowledge, the only linear-work and polylog(n)-span algorithms for parallel partition that *are* in-place require the use of atomic operations (e.g, fetch-and-add) [**?**, **?**, **?**].

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 [**?**]. In multi-user systems, memory efficiency is also important on small inputs, since processes with larger memory-footprints can hog the cache, slowing down other processes.

For sorting algorithms, in particular, special attention to memory efficiency is often given. In the context of parallel algorithms, however, the most practically efficient algorithms fail to run in place, at least without the additional use of atomic-fetch-and-add variables [**?**, **?**, **?**], or the loss of theoretical guarantees on parallelism [**?**]. Parallel merge sort [**?**] was made in-place by Katajainen [**?**], but has proven too sophisticated for practical applications. Bitonic sort [**?**] 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 [**?**, **?**, **?**, **?**, **?**], has eluded any in-place EREW algorithms due to its reliance on parallel partition.

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

The simplicity of our algorithmic techniques makes them condusive to practical implementations. We implement and optimize an (almost) in-place implementation. Because the in-place algorithm eliminates the use of large auxilary arrays, the algorithm is able to achieve a significant reduction in cache misses over its out-of-place counterpart. In addition to significantly

reducing the memory overhead, the space-efficient algorithm is able to achieve a modest performance improvement, and on a single thread performs within a factor of two of the in-place serial algorithm.

2. Preliminaries

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

Workflow Model. We consider a language-based model of parallelism in which algorithms may achieve parallelism through the use of ***parallel-for-loops***, though our algorithm also works in the less restrictive EREW PRAM model [**?**, **?**]. A parallel-for-loop is given a range $R \in \mathbb{N}$, a constant number of arguments $\arg_1, \arg_2, \ldots, \arg_c$, and a body of code. For each $i \in \{1, \ldots, R\}$, the loop launches a thread that is given loop-counter i and local copies of the arguments $\arg_1, \arg_2, \ldots, \arg_c$. The threads then perform the body of the loop.[2]

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

The ***work*** T_1 of an algorithm is the time that the algorithm would require to execute on a single processor. The ***span*** T_∞ of an algorithm is the time to execute on infinitely many processors. The scheduler is assumed to contribute no overhead to the span. In particular, if the body of a parallel-for-loop has span s, then the full parallel loop has span $s + O(1)$ [**?**, **?**].

The work T_1 and span T_∞ can be used to quantify the time T_p that an algorithm requires to execute on p processors using a greedy online scheduler. If the scheduler is assumed to contribute no overhead, then Brent's Theorem [**?**] 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 [**?**, **?**], 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., [**?**]).

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

In an ***in-place*** algorithm, each thread is given $O(\log n)$ memory upon creation that is deallocated when the thread dies. This memory can be shared with the thread's children. The depth of the parent-child tree is not permitted to exceed $O(\log n)$.[3]

The Parallel Partition Problem. The parallel partition problem takes an input array A of size n, and a ***decider function*** dec that determines for each element $a[i] \in A$ whether or not $A[i]$ is a ***predecessor*** or a ***successor***. That is, $\text{dec}(A[i]) = 1$ if $A[i]$ is a predecessor, and $\text{dec}(A[i]) = 0$ if $A[i]$ is a successor. The behavior of the parallel partition is to reorder the elements in the array A so that the predecessors appear before the successors.

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

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

The Reordering Phase: In this phase, the algorithm constructs an output-array C by placing each predecessor $A[i] \in A$ in position $B[i]$ of C. If there are t predecessors in A, then the first t elements of C will now contain those t predecessors in the same order that they appear in A. The algorithm then places each successor $A[i] \in A$ in position $t + i - B[i]$. Since $i - B[i]$ is the number of successors in the first i elements of A, this places the successors in C in the same order that they appear in A. Finally, the algorithm copies C into A, completing the parallel partition.

Both phases can be implemented with $O(n)$ work and $O(\log n)$ span. Like its serial out-of-place counterpart, the algorithm is stable but not in place. The algorithm uses two auxilary arrays of size n. Kiu, Knowles, and Davis [**?**] were able to reduce the extra space consumption to $n+p$ under the assumption that the number of processors p is hard-coded; their algorithm breaks the array A into p parts and assigns one part to each thread. Reducing the space below

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

$o(n)$ has remained open until now, even when the number of threads is fixed.

3. A Simple In-Place Algorithm

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

We assume without loss of generality that the total number of successors in A exceeds the number of predecessors, since otherwise their roles can simply be swapped in the algorithm. Further, we assume for simplicity that the elements of A are distinct; this assumption is removed at the end of the section.

Algorithm Outline. We begin by presenting an overview of the key algorithmic ideas needed to construct an in-place algorithm.

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

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

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

In the remainder of the section, we present the algorithm in detail. It proceeds in three phases.

A Preprocessing Phase. Recall that for each $t \in 1, \ldots, n$, we call the t-prefix $A[1], \ldots, A[t]$ of A successor-heavy if it contains at least $\frac{t}{4}$ successors. The goal of the preprocessing phase is to rearrange A so that every prefix is successor heavy.

We begin with a parallel-for-loop: For each $i = 1, \ldots, \lfloor n/2 \rfloor$, if $A[i]$ is a predecessor and $A[n-i+1]$ is a successor, then we swap their positions in A.

This ensures that at least half the successors in A reside in the first $\lceil n/2 \rceil$ positions. Thus the first $\lceil n/2 \rceil$ positions contain at least $\lceil n/4 \rceil$ successors, making every t-prefix with $t \geq \lceil n/2 \rceil$ successor-heavy.

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

Each recursive step has constant span and performs work proportional to the size of the subarray being considered. The preprocessing phase therefore has total work $O(n)$ and span $O(\log n)$.

An Implicit Parallel Prefix Sum. Pick a ***block-size*** $b \in \Theta(\log n)$ satisfying $b \geq 2\lceil \log(n+1) \rceil$. Consider A as a series of $\lfloor n/b \rfloor$ blocks of size b, with the final block of size between b and $2b-1$. Denote the blocks by $X_1, \ldots, X_{\lfloor n/b \rfloor}$.

Within each block X_i, we can implicitly store a value in the range $0, \ldots, n$ through the ordering of the elements. In particular, X_i can be broken into (at least) $\lceil \log(n+1) \rceil$ disjoint pairs of adjacent elements, and by rearranging the order in which a given pair (x_j, x_{j+1}) occurs, the lexicographic comparison of whether $x_j < x_{j+1}$ can be used to encode one bit of information. Values $v \in [0, n]$ can therefore be read and written to X_i with work $O(b) = O(\log n)$ and span $O(\log b) = O(\log \log n)$ using a simple divide-and-conquer recursive approach.

After the preprocessing phase, our algorithm performs a parallel-for loop through the blocks, and stores in each block X_i a value v_i equal to the number of predecessors in the block. This can be done in place with work $O(n)$ and span $O(\log \log n)$.

The algorithm then performs an in-place parallel-prefix operation on the values $v_1, \ldots, v_{\lfloor n/b \rfloor}$ stored in the blocks. This is done by first resetting each even-indexed value v_{2i} to $v_{2i} + v_{2i-1}$; then recursively performing a parallel-prefix sum on the even-indexed values; and then replacing each odd-indexed v_{2i+1} with $v_{2i+1} + v_{2i}$, where v_0 is defined to be zero. If the v_i's could be read and written in constant time, then the prefix sum would take work $O(n)$ and span $O(\log n)$. Since each v_i actually requires work $O(\log n)$ and span $O(\log \log n)$ to read/write, the prefix sum takes work $O(n)$ and span $O(\log n \cdot \log \log n)$.

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

In-Place Reordering. In the final phase of the algorithm, we reorder A so that the predecessors appear before the successors. Let $P = X_1 \circ X_2 \circ \cdots \circ X_t$ be the smallest prefix of blocks that contain at least $4/5$ of the elements in A. We begin by recursively reordering the elements in P so that the predecessors appear before the successors; as a base case, when $|P| \leq 5b = O(\log n)$, we simply perform the reordering in serial.

After P has been reordered, it will be of the form $P_1 \circ P_2$ where P_1 contains only predecessors and P_2 contains only successors. Because P is successor-heavy, we have that $|P_2| \geq |P|/4$, and thus that $|P_2| \geq |X_{t+1} \circ \cdots \circ X_n|$.

To complete the reordering of A, we perform a parallel-for-loop through each of the blocks $X_{t+1}, \ldots, X_n$. For each block X_i, we first extract v_i (with work $O(\log n)$ and span $O(\log \log n)$). We then create an auxiliary array Y_i of size $|X_i|$, using $O(\log n)$ memory from the thread in charge of Y_i in the parallel-for-loop. Using a parallel-prefix sum (with work $O(\log n)$ and span $O(\log \log n)$), we set each $Y_i[j]$ equal to v_i plus the number of predecessors in $X_i[1], \ldots, X_i[j]$. In other words, $Y_i[j]$ equals the number of predecessors in A appearing at or before $X_i[j]$.

After creating Y_i, we then perform a parallel-for-loop through the elements $X_i[j]$ of X_i (note we are still within another parallel loop through the X_i's), and for each predecessor $X_i[j]$, we swap it with the element in position $Y_i[j]$ of the array A. Critically, because $|P_2| \geq |X_{t+1} \circ \cdots \circ X_n|$, we are guaranteed that the element with which $X_i[j]$ is being swapped is a successor in P_2. After the swaps have been performed, all of the elements of X_i are now successors.

Once the outer for-loop through the X_i's is complete, so will be the parallel partition of A. The total work in the reordering phase is $O(n)$ since each X_i appears in a parallel-for-loop at exactly one level of the recursion, and incurs $O(\log n)$ work. The total span of the reordering phase is $O(\log n \cdot \log \log n)$, since there are $O(\log n)$ levels of recursion, and within each level of recursion each X_i in the parallel-for-loop incurs span $O(\log \log n)$.

Combining the phases, the full algorithm has work $O(n)$ and span $O(\log \log n)$. Thus we have:

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

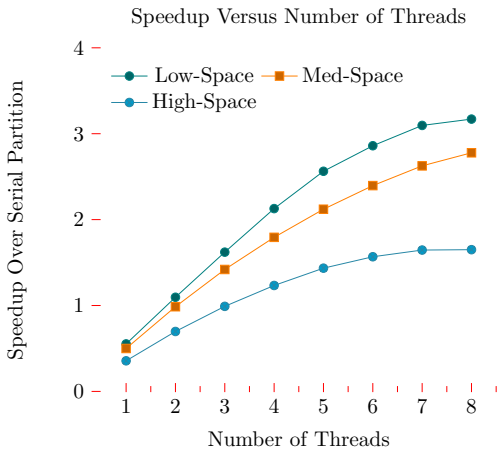
Allowing for Repeated Elements. In proving Theorem 3.1 we assumed for simplicity that the elements of A are distinct. This plays an important role in how we store the values v_i in the blocks X_i. To eliminate this requirement without changing the work and span of the algorithm, we can require that $b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

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

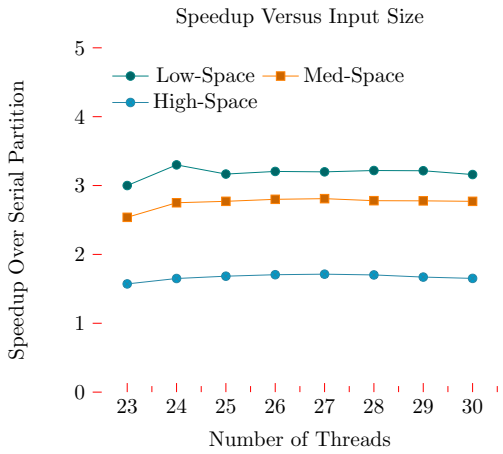
To read the value v_i, we check whether $x_1 = x_2$ in order to determine which encoding is being used and then unencode the bits appropriately. In the Reordering phase of the algorithm, once the blocks X_i are no longer required to encode values, we can replace each overwritten x_i with its correct value (given by the value of the preceding element).

4. Experimental Evaluation

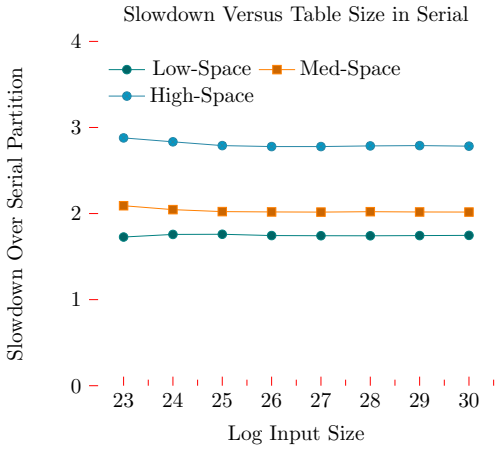
In this section, we implement the techniques from Section 3 to build a space-efficient parallel partition function. Our implementation considers an array of n



(A) For a fixed table-size $n = 2^{30}$, we compare each implementation's runtime to a serial baseline, which takes 4.27 seconds to complete. The x-axis plot the number of worker threads being used, and the y-axis plots the multiplicative speedup over the serial baseline.



(B) We compare the performance of the implementations running on eight worker threads on varying input sizes. The x-axis is the log-base-2 of the input size, and the y-axis is the multiplicative speedup when compared to the serial baseline.



(C) We compare the performance of the implementations in serial, with no scheduling overhead. The x-axis is the log-base-2 of the input size, and the y-axis is the multiplicative slowdown when compared to the serial baseline.

FIGURE 1

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 four implementations:

- *A Serial Baseline:* This uses the standard serial in-place partition algorithm.
- *The High-Space Parallel Implementation:* This uses the standard parallel partition algorithm [**?**, **?**], as described in Section 2. The implementation has a space overhead of roughly $2n$ eight-byte words.
- *The Medium-Space Parallel Implementation:* Starting with the high-space implementation, we reduce the space used by the Parallel-Prefix Phase by only constructing every $O(\log n)$-th element of the prefix-sum array B, as in Section 3. (Here $O(\log n)$ is hard-coded as 64.) The array B is initialized to be of size $n/64$, with each component

equal to a sum of 64 elements, and then a parallel prefix sum is computed on the array. Rather than implicitly encoding the elements of B in A, we use an auxilary array of size $n/64$ to explicitly store the prefix sums.[4] The algorithm has a space overhead of $\frac{n}{32} + n$ eight-byte words.[5]

- *The Low-Space Parallel Implementation:* Starting with the medium-space implementation, we make the reordering phase completely in-place using the preprocessing technique in Section 3.[6] The only space overhead in this implementation is the $\frac{n}{32}$ additional 8-byte words used in the prefix sum.

The three parallel algorithms all have work $O(n)$ and span $O(\log n)$.

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

Performance Comparison. Figure 1a graphs the speedup of the each of the parallel algorithms over the serial algorithm, using varying numbers of worker threads on an 8-core machine with a fixed input size of $n = 2^{30}$. Both space optimizations result in performance improvements, with the low-space implementation performing almost twice as well as the high-space implementation on eight threads. Similar speedups on eight threads are achieved on smaller inputs; see Figure 1b, which graphs speedup for input sizes starting at 2^{23}, the smallest size for which consistant time measurements can be taken without averaging over many trials.

Figure 1c compares the performances of the implementations in serial. Parallel for-loops are replaced with serial for-loops to eliminate scheduler overhead. As the input-size varies, the ratios of the runtimes vary only slightly. The low-space implementation performs within a factor of two of the serial implementation. As in Tables 1a and 1b, both space optimizations result in performance improvements.

The Source of the Speedup. If we compare the number of instructions performed by the three parallel implementations, then the medium-space algorithm would seem to be the clear winner. Using Cachegrind to profile the number of instructions performed in a (serial) execution on an input of size 2^{25}, the high-space, medium-space, and low-space implementations perform 897 million, 593 million, and 871 million instructions, respectively.

Cache misses tell a different story. Using Cachegrind to profile the number of top-level cache misses in a (serial) execution on an input of size 2^{25}, the high-space, medium-space, and low-space implementations incur 38.1 million, 21.4 million, and 17.1 million cache misses, respectively.

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

Machine and Implementation Details. Our experiments are performed on a machine with eight 2.9 GHz Intel Xeon E5-2666 v3 processors, and 30 GB of RAM.

Our algorithms are implemented using the Cilk-Plus 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. The parallelism is

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

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

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

achieved through simple parallel-for-loops, with one exception at the beginning of the low-space implementation, when the number of predecessors in the input array is computed. Although CilkPlus Reducers (or OpenMP Reductions) could be used to perform this parallel summation within a parallel-for-loop, we found a slightly more ad-hoc approach to be faster: Using a simple recursive structure, we manually implemented a parallel-for-loop with CILK Spawns and Syncs, allowing for the summation to be performed within the recursion; to amortize the cost of Cilk thread spawns, the ad-hoc parallel-for-loop reverts to a serial computation when the subarray size becomes 256 or smaller.

Our implementations are available at ??.

5. Conclusion and Open Questions

Parallel partition is a fundamental primitive in parallel algorithms [**?**, **?**]. Achieving faster and more space-efficient implementations, even by constant factors, is therefore of high practical importance. Until now, the only space-efficient algorithms for parallel partition have relied extensively on concurrency mechanisms or atomic operations. In this paper, we have shown that, somewhat surprisingly, there exists a simple variant on the classic parallel algorithm that completely eliminates the use of auxilary memory, while still using only exclusive read/write shared variables. Moreover, our implementation of (a variant of) the algorithm is able to exploit the algorithm's superior cache-behavior in order to obtain practical speedups over its out-of-place counterpart.

We suspect that even the implicit storage of values in blocks could be made practical in actual implementations (though in practice it is likely simpler to simply use a small auxiliary array of size $O(n/\log n)$). In particular, by setting b to be a constant factor larger, one can reduce the number of reads and writes of values v_i by the same constant factor, making the fraction of the time spent by the algorithm on such reads and writes an arbitrarily small constant.

Our work also prompts several theoretical questions. Can fast space-efficient algorithms with polylogarithmic span be found for other classic problems such as randomly permuting an array [**?**, **?**, **?**], and integer sorting [**?**, **?**, **?**, **?**, **?**]? Such algorithms are of both theoretical and practical interest, and might be able to utilize some of the techniques introduced in this paper.

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