Summary

Alek Westover

The parallel-partition problem, which is the first step of 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. When partitioning an array relative to some pivot value p, $A[i]$ is labelled a predecessor if $A[i] \leq p$ and a successor otherwise.

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], [3]. Moreover, the algorithm uses only exclusive read/write shared memory variables (i.e., it is an ***EREW*** algorithm). This eliminates the need for concurrency mechanisms such as locks and atomic variables, and ensures good behavior even if the time to access a location is a function of the number of threads trying to access it (or its cache line) concurrently. EREW algorithms also have the advantage that their behavior is internally deterministic, meaning that the behavior of the algorithm will not differ from run to run, which makes test coverage, debugging, and reasoning about performance substantially easier [4].

The parallel-partition algorithm suffers from using a large amount of auxiliary memory, however. Whereas the serial algorithm is typically implemented in place, the parallel algorithm relies on the use of two auxiliary arrays of size n. To the best of our knowledge, the only known linear-work and $\mathrm{polylog}(n)$-span algorithms for parallel partition that are in-place require the use of atomic operations (e.g, fetch-and-add) [2], [9], [11].

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 [12]. 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 atomic operations or other concurrency mechanisms [2], [9], [11], or eschew theoretical guarantees [6]. Parallel merge sort [8] was made in-place by Katajainen [10], but has proven too sophisticated for practical applications. Bitonic sort [5] 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 [2], [6], [7], [9], [11], has eluded any in-place EREW (or CREW i.e. *concurrent-read exclusive-write*) algorithms due to its reliance on parallel partition.

We show that parallel partition can be implemented in place, and without the use of concurrency mechanisms. 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.

Our first result is a set of techniques that allows us to adapt the standard parallel partition algorithm to be fully in place. The new algorithm has work $O(n)$ and span $O(\log n \cdot \log\log n)$. As an immediate consequence, we also get an in-place quicksort algorithm with work $O(n \log n)$ and span $O(\log^2 n \log\log n)$.

Using our algorithmic techniques, we implement a space-efficient parallel partition. Because the in-place algorithm eliminates the use of large auxiliary arrays, the algorithm is able to achieve a reduction in cache misses over its out-of-place counterpart, resulting in performance improvements over the standard parallel partition algorithm.

The in-place algorithm remains bottlenecked by memory bandwidth, however, due to the fact that multiple passes over the input array are required. The memory-bandwidth bottleneck has led past researchers [6], [7] 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 main result of this paper is an algorithm that we call the ***Smoothed Striding Algorithm***, which achieves theoretical guarantees not only on span, work, and memory usage, but also on cache behavior. By randomly perturbing the internal structure of the Strided Algorithm, and adding a recursion step that was previously not possible, we arrive at a new algorithm with provably good span and cache behavior. The Smoothed Striding Algorithm is in-place, has polylogarithmic span, and exhibits provably optimal cache behavior up to small-order factors.

Using the Smoothed Striding Algorithm, we implement and optimize a fully in-place parallel partition. The implementation of the Smoothed Striding Algorithm performs within 15% of the Strided Algorithm on a large number of threads, while achieving theoretical guarantees that were previously unattainable.

REFERENCES

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

[2] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort. *arXiv preprint arXiv:1705.02257*, 2017.

[3] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] 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.

[5] 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.

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

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

[8] Torben Hagerup and Christine Rüb. Optimal merging and sorting on the erew pram. *Information Processing Letters*, 33(4):181–185, 1989.

[9] Philip Heidelberger, Alan Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.

[10] Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. Space-efficient parallel merging. *RAIRO-Theoretical Informatics and Applications*, 27(4):295–310, 1993.

[11] 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.

[12] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.