12.2 Paper Review

12.2.1 Engineering in-place shared-memory sorting algorithms

As this paper makes very clear at the beginning, it is very easy (and apparently quite common) to claim you have the "best" sorting algorithm but difficult to definitively prove that this is the case. The authors of this paper not only claim that their in-place shared-memory sorting algorithm, which is based on superscalar (out-of-place) samplesort, but perform 500,000 tests on different configurations of processor architecture and operating system to establish its performance on a wide range of problems compared to other, established sorting methods. The solution is also remarkable for being an implementation of a strictly in-place sorting algorithm, to which the authors point out many theoretical solutions, but of which most are quite challenging to transfer into practice.

The algorithm's antecedents include quicksort, samplesort, and superscalar samplesort, all of which fall in the same lineage and have been incrementally improved over time. Samplesort is an improvement over quicksort because, instead of relying on a single partition and recursing, it randomly samples $k-1$ elements from the list to be sorted and uses these as pivots for k buckets, which are then recursed over until the bucket size is small enough to use insertion or heap sort. This has better guarantees for consistent bucket size than quicksort, but can be inefficient when it mispredicts branches (for each element, the algorithm has to find the appropriate bucket, which is done with binary search and can lead to breaks in parallel processing if incorrect), has a cache miss when writing to buckets, or if the input array has many duplicates (which can be problematic for bucket size). Superscalar samplesort is, in turn, an improvement for samplesort because it implements a binary tree for the splitter array (branchless decision tree, i.e. we can perform arithmetic to manage control flow so that are no conditional jumps. Since k is always a power of two, we can unroll the entire decision tree traversal into fast, straight-line code *and* parallelize it. Pretty cool). The author's algorithm further improves on superscalar samplesort by sorting the buckets in-place.

In-place sorting is achieved by treating the entire input array as a contiguous array of buckets. A set of buffers, whose size can be controlled, writes blocks of elements that belong to distinct buckets to The rule for navigating the branchless decision tree in the authors' algorithm is $i \leftarrow 2i + \mathbb{1}_{a_i<e}$. If e is larger than the splitter a_i then the index is incremented by 1, else we walk left. They remove the equality bucket from StringPS^{4}o and just use ¡, which is made possible by the use of the compare function and the leaf-level computation $2i + 1 - \mathbb{1}_{e<s_i}$ (except in the critical case where there are many duplicate values, in which case equality buckets allow bucket sizes to remain approximately the same). Additionally, to augment parallelization, each level of the decision tree is applied to a block of elements before moving the shared pointers to the next block of elements.

Given this setup, and the dependencies between serial and parallel processing for individual threads based on whether an equality bucket or not (interestingly, the conference portion of the article did not contain the sophisticated scheduling algorithm presented here, and just executed tasks with $> n/t$ elements with all t threads), the actual implementation is critical to the success of this algorithm. The parameters tuned by the authors include $k = 256$, base case size $n_0 = 16$, among others. They report that the buffer blocks and swap buffers (required for permuting the locations of the blocks within individual buckets) take up the majority of additional memory. Since both of these sizes can be modulated by the programmer and are largely independent of size n, this algorithm is considered to be in-place. On a secondary point, I appreciate that this algorithm works with the C++ standard library threads invocation. It is also notable that, for the purpose of comparison, the authors reimplement superscalar samplesort and make their own optimizations.

The experimentation portion of the paper evaluates a number of algorithms, input distributions, and hardware. The evaluation is performed as a grid search so that all permutations are explored, and comparisons between algorithms reported as average slowdown with respect to the set of all algorithms evaluated ($A \in \mathcal{A}$). They conclude that the in-place parallel superscalar samplesort is extremely competitive, if not the fastest, among peer algorithms both on sequential as well as parallel machines. Despite this success, the authors still recommend several potential avenues for future work. These include improving the smaller sorting algorithms for the base case, and using vector instructions for the branchless decision tree.

Honestly, compared to many of the previous papers we have read for this class, I thought there were no glaring omissions (certainly it seems to cover most of the pet peeves listed out by "A Theoretician's Guide to the Experimental Analysis of Algorithms"). The theory is laid out and followed by experiments that are clearly explained and replicable. I do wonder, since the authors emphasize that this is a prototypical

codebase, what the conversion to production code (slash integration into standard libraries) might look like and whether this "practical" implementaiton is also practical when deployed as an alternative to std::sort.

While this is less of an accomplishment, I want to note that the authors also have a very pleasant writing style that very much enhanced the experience of reading this paper.