Glidesort: Efficient In-Memory Adaptive Stable Sorting

Orson R. L. Peters
Centrum Wiskunde & Informatica

Adaptive sorting is an important concept where an algorithm is able to adapt to pre-existing patterns in the input to speed up (or prevent a slowdown of) sorting. We posit two major categories of input patterns that are relevant and promising for comparison-based sorting:

1. **Ascending/descending runs.** Our data might have been (partially) sorted prior in our computer program, or our source inherently generates data that is (partially) sorted, such as sea level measurements due to tides.
2. **Low-cardinality data.** Datasets often contain many duplicates, especially under projection of a comparison operator. For example, a database of car models might have thousands of distinct rows but only dozens of distinct car brands.

In comparison-based sorting there are two fundamentally different divide-and-conquer approaches, each adept at handling one such pattern:

1. **Bottom-up.** Mergesort is the canonical bottom-up algorithm, merging small sorted subarrays into ever larger ones until the entire input is sorted. This property allows it to take advantage of pre-existing runs in the data, notably described by Donald Knuth as 'natural sorting' and used by Tim Peters in Timsort [4].
2. **Top-down.** Quicksort is the canonical top-down algorithm, partitioning elements into ever smaller subarrays until they have length one, which is trivially sorted. Naively implemented low-cardinality data is a worst-case input for Quicksort, as all duplicates end up in the same partition. The authors show in an earlier work [3] that this can be detected with minimal overhead, making it a best-case input instead.

In this talk we introduce glidesort, a novel **stable** comparison-based sorting algorithm that combines mergesort and quicksort to be fully adaptive to both runs in the data as well as low-cardinality data. In addition to this algorithmic novelty, glidesort is implemented with state-of-the-art and novel techniques that eliminate overhead and maximize hardware utilization.

In particular, Edelkamp and Weiß [1] identified branch mispredictions as not just overhead but in fact a primary runtime component of quicksort for sorting elements with cheap comparisons, similar to earlier work [2] by Elmasry et. al. for mergesort. We describe techniques for branchless stable partitioning and merging and identify a new source of overhead in the form of data dependencies. By interleaving parallel partitioning and merging loops, glidesort is able to reduce this overhead and use instruction-level parallelism found in modern superscalar CPUs more effectively.

Implemented in Rust, glidesort not only scales smoothly with pre-existing runs in the data, it is fast for random data as well, giving a 4.6x speed-up over `std::stable_sort` for sorting 2^{24} uniformly random 32-bit integers, while using a quarter of the memory. Sorting the same numbers $\bmod 4096$ gives a further 2.2x speed-up, increasing to 3.6x when comparing $\bmod 16$, reaping the benefits of low-cardinality data.

References

[1] S. Edelkamp and A. Weiß. BlockQuicksort: Avoiding branch mispredictions in quicksort. *ACM J. Exp. Algorithmics*, 24, jan 2019.

[2] A. Elmasry, J. Katajainen, and M. Stenmark. Branch mispredictions don't affect mergesort. In *Experimental Algorithms*, pages 160–171. Springer Berlin Heidelberg, 2012.

[3] O. R. L. Peters. Pattern-defeating quicksort, 2021.

[4] T. Peters. Timsort. `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`, 2002.