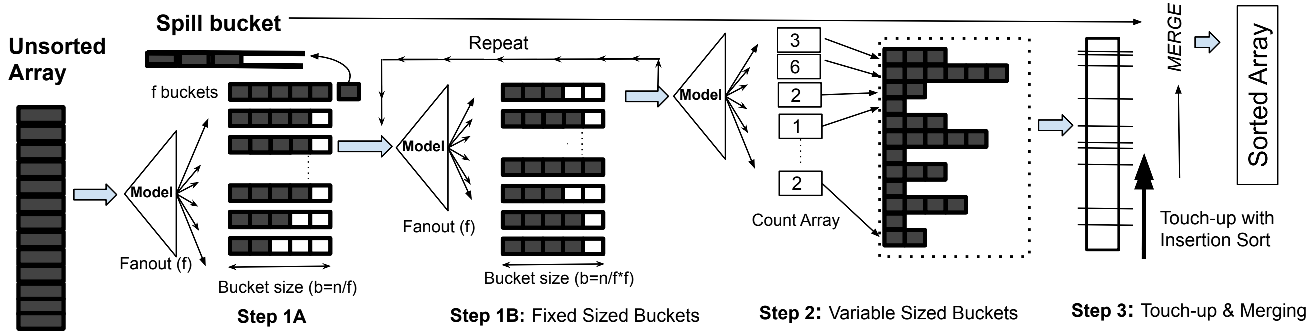

Figure 1: Algorithm Overview

The Case for a Learned Sorting Algorithm

Ani Kristo‡ *, Kapil Vaidya† *, Uğur Çetintemel‡, Tim Kraska†,
‡ Brown University, † Massachusetts Institute of Technology

1. Introduction

Sorting is one of the most well-studied problems in Computer Science and still one of the most fundamental algorithms used in any system. The fastest algorithms, such as Radix Sort, are able to achieve $O(w\dot{N})$ sorting time with w being the number of bits required to store each key and N the data size. However, Radix Sort alone cannot be easily generalized because of its sensitivity to the key size and data skew. Instead, most systems either implement variants of comparison-based sorting algorithms, such as Quicksort or Heapsort, which have a time complexity of $O(N \log N)$, or use hybrid algorithms, which combine various comparative and distributive sorting techniques with the goal of achieving better hardware utilization and overcome some of Radix Sort's limitations.

In this paper, we propose a new ML-enhanced sorting algorithm based on Radix Sort that has a near-linear sorting time, while being less sensitive to the key size w and any data skew. The core idea of the algorithm simple: we train a CDF model F_K over a subset of keys and then use the model to predict the position of each key within the sorted output. If we would have a perfect model of the empirical CDF, we could use the predicted probability $P(k \leq K)$ for a key k, scaled to the number of keys N, to perfectly predict the final position for every key: $p = F_K(k) * N = P(k \leq K) * N$. Assuming the model already exists, this would allow us to sort the data with one pass over the input with less sensitivity to the size of the key.

Unfortunately, it is unlikely that we can build a perfect empirical model, especially if the time to build a model counts towards the sorting time. Furthermore, most traditional ways to model the CDF, especially NN architectures, would be very expensive to execute and render the algorithm very uncompetitive. Finally, and most surprisingly, even with a perfect empirical CDF model that is cheap to execute, the sorting performance might still be worse than the more traditional Radix Sort, given that the index prediction phase is more prone to random memory access and cache misses, whereas Radix Sort makes sequential passes over the data hence enabling a better cache utilization.

In this paper, we outline a first (single threaded) ML-enhanced algorithm, called *MER_sort*(Model-Enhanced Radix Sort), which overcomes these challenges. To our own surprise, initial experiments indicate that the approach can actually achieve better performance than the highly tuned distribution-based, comparison-based, and hybrid sorting algorithms, even while counting for the model building time. For example, our experiments show that *MER_sort* yields an average of 3.22× performance improvement over C++ STL sort[13], 4.92× improvement over Timsort[19], 1.25× over Radix sort[3], and 1.23× over IS^{4}o[6], which is a cache-efficient version of the Sample Sort algorithm.

In the remainder of this paper we proceed to describe the algorithm and evaluate its performance as compared to other common sorting algorithms. However, it should be noted that these are still preliminary results and that a lot of work still has to be done, particular for sorting strings.

* denotes equal contribution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

AI Sys Workshop @ SOSP 2019, October 27, 2019, Huntsville, ON, Canada.
Copyright © 2019 ACM . . . $15.00.
http://dx.doi.org/10.1145/

2. The algorithm

Assuming that we already have a CDF model, our ML-enhanced sorting algorithm is similar to Radix Sort in that it splits the input array into finer buckets until a sorted array is obtained. However, instead of doing it based on the key-bits, *MER_sort* uses the CDF model to predict the position. As shown in Figure 3 (Step 1a). Here

we start with the unsorted array and predict for every key k its final position $p = F_K(k) \cdot N$. Based on this prediction, we position the key in one of M fixed-size buckets b. The number of buckets depend on the size of the CPU cache (at least one cache-line per bucket should be in the cache). If one of the fixed size buckets overflows (e.g., more keys than slots are available), the keys are placed into a spill bucket.

After every key was mapped to a bucket, we repeat that mapping for every bucket (Step 1b) (except for the spill bucket). This process repeats until the number of keys per bucket are less than a threshold (t); usually around 100 keys. After every bucket is small enough, the buckets are further split but now in variable sized buckets. This is necessary as the model is likely to get less accurate as smaller the buckets become. However, using a variable size data structure is expensive. Instead, we use a two step approach for the partitioning into variable sized buckets. First the model is used to determine how many keys would go in each bucket (one pass over the data), which are stored in a count array. This count array allows us now to store the variable buckets directly into the output array as we know the exact size per bucket (Step 2). Finally, the spill bucket is merged with the output array and insertion sort is used to correct any keys that might be out of order.

3. CDF modeling

So far we assumed that the CDF model is given. However, for sorting it is key to have a model with very fast inference time, which ideally is also fast to train. For this purpose, we use a set of linear models that are arranged in a hierarchical fashion that enables progressively finer approximations and is able to capture the input data's non-linearities by emulating a piecewise-linear function. A typical such CDF model is a 2-layer structure with one root and multiple leaf linear models. Given the key, the root model selects which subsequent linear model in the next layer to use for the final CDF approximation, which is then used by the sorting procedure. We train the individual models top-down as shown in [17] using a small sample (1%) of the data. However, in contrast to [17] we use simple linear interpoloation for the second stage. This procedure takes only a small portion of the entire sorting time.

4. Related work

Many sorting algorithms and hybrid variations of them have been proposed over the years [1, 7, 9–11, 14, 16, 20]. Here our focus is on single-thread sorting algorithms. For example, Timsort combines Insertion Sort's ability to benefit from the input's presortedness with the stability of Merge Sort, and is used as the default sorting algorithm in Java[15] and Python[19]. Whereas, std::sort uses Introsort[13], which is a hybrid implementation of Quicksort, Heapsort, and Insertion Sort. Finally, there have been several attempts to create more hardware-efficient implementions, such as the very recent IPS^{4}o algorithm [2], which is a highly cache-optimized Sample Sort algorithm designed especially for the GCC compiler (i.e., we saw significant slow-downs using other compilers).

5. Evaluation

We evaluated our approach with both synthetic and real datasets and compared it against C++ implementations of Radix Sort[3, 4], Introsort(`std::sort`), Timsort[5], Histogram Sort[8], and a sequential version of the IPS^{4}o algorithm (IS^{4}o)[2] which is the most competitive sample sort algorithm that we are aware of. All the experiments are measured use an Intel® Xeon® Gold 6150 CPU @ 2.70GHz with 376GB of memory, and compiled with the GCC compiler with the `-O3` flag for full optimizations. The model we used for training is of 2-layers and contains 1000 leaf models, trained with a uniformly selected 1% sample of the array.

Figure 2 shows the sorting throughput for 1 million up to 1 billion double-precision keys following a standard normal distribution. As it can be seen *MER_sort* achieves close to 30% higher throughput than the next best algorithm. However, the benefits are larger as soon as the data does not fit in the L3 cache.

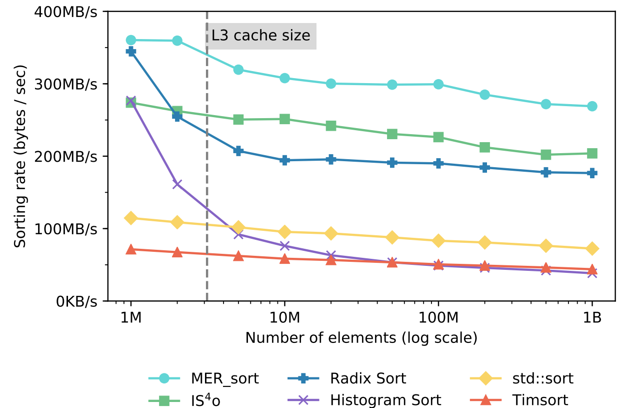

Figure 2: The sorting throughput for normally distributed double-precision keys.

Figure 3 compares *MER_sort* against the other baselines for 100M randomly-generated double-precision keys following Uniform, Exponential, and Multimodal distributions (mixture of 5 Gaussian distributions) and two real-world datasets, OpenStreetMap's longitude-latitude pairs (osm)[18] (100M elements), and the `mem_free` and `bytes_in` columns of the IoT dataset [12]. Similar to before *MER_sort* outperforms the other baselines by up to 30%. However, for the IoT/mem and IoT/bytes dataset, Radix sort is very competitive, which contain a lot of duplicate values. From other experiments we know, that the number of duplicates have a negative impact on the sorting performance; something we plan to address in the future.

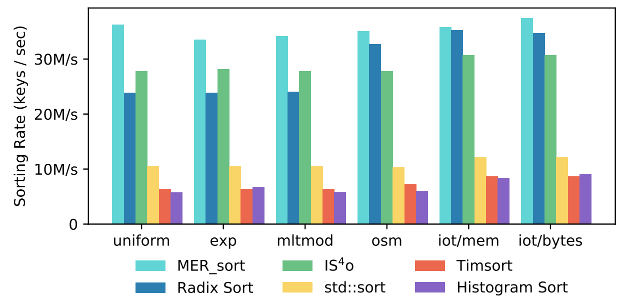

Figure 3: The sorting rate of *MER_sort* and other baselines.

6. Summary and conclusion

In this paper we presented a novel approach to accelerate sorting by leveraging models that approximate the empirical CDF of the input to quickly map elements into the sorted position within a small error margin. This approach results in significant performance improvements as compared to the most competitive and widely used sorting algorithms, and marks an important step into building ML-enhanced algorithms and data structures. A lot of future work remains open, most notably how to handle duplicate values and strings. We do have initial results on those question and given a presentation slot, would be happy to present them as well.

References

[1] paradis: an efficient parallel algorithm for in-place radix sort.

[2] Open-source C++ implementation of the IPS4o algorithm. https://github.com/SaschaWitt/ips4o.

[3] Open-source C++ implementation of Radix Sort for double-precision floating points, . https://bitbucket.org/ais/usort/src/474cc2a19224/usort/f8_sort.c.

[4] Open-source C++ implementation of Radix Sort for std::string types, . https://stackoverflow.com/a/32545991/4122148.

[5] Open-source C++ implementation of Timsort. https://github.com/gfx/cpp-TimSort.

[6] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. In-place parallel super scalar samplesort (IPS4o). In *25th Annual European Symposium on Algorithms (ESA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[7] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[8] P. E. Black. Histogram Sort, 2019. https://www.nist.gov/dads/HTML/histogramSort.html.

[9] B. Bramas. Fast sorting algorithms using AVX-512 on Intel Knights Landing. *arXiv preprint arXiv:1704.08579*, 305:315, 2017.

[10] T. Dachraoui and L. Narayanan. Fast deterministic sorting on large parallel machines. In *Proceedings of SPDP'96: 8th IEEE Symposium on Parallel and Distributed Processing*, pages 273–280. IEEE, 1996.

[11] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 348–357. ACM, 2007.

[12] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5643-5. . URL http://doi.acm.org/10.1145/3299869.3319860.

[13] GNU. C++: STL sort. https://gcc.gnu.org/onlinedocs/libstdc/libstdc-html-USERS-4.4/a01347.html.

[14] H. Inoue and K. Taura. SIMD-and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 8(11): 1274–1285, 2015.

[15] Java. Java 9: List.sort. https://docs.oracle.com/javase/9/docs/api/java/util/List.html#sort-java.util.Comparator-.

[16] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub. Tencent sort, 2017.

[17] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[18] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org , 2017. https://www.openstreetmap.org.

[19] T. Peters. Python: list.sort. https://github.com/python/cpython/blob/master/Objects/listsort.txt.

[20] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362. ACM, 2010.