Multiway Powersort

William Cawley Gelling[*]   Markus E. Nebel[†]   Benjamin Smith[‡]

Sebastian Wild[§]

January 17, 2023

Abstract

We present a stable mergesort variant, *Multiway Power-sort*, that exploits existing runs and finds nearly-optimal merging orders for *k-way* merges with negligible overhead. This builds on Powersort (Munro & Wild, ESA 2018), which has recently replaced Timsort's suboptimal merge policy in the CPython reference implementation of Python, as well as in PyPy and further libraries. Multiway Powersort reduces the number of memory transfers, which increasingly determine the cost of internal sorting (as observed with Multiway Quicksort (Kushagra et al., ALENEX 2014; Aumüller & Dietzfelbinger, TALG 2016; Wild, PhD thesis 2016) and the inclusion of Dual-Pivot Quicksort in the Java runtime library). We demonstrate that our 4-way Powersort implementation can achieve substantial speedups over standard (2-way) Powersort and other stable sorting methods without compromising the optimally run-adaptive performance of Powersort.

1. Introduction

Sorting a list of items is one of the foundational problems of computer science. Prominent sorting algorithms include Quicksort and Mergesort, for which highly engineered implementations exist. While Quicksort is typically faster, Mergesort has the optimal $\Theta(n \log(n))$ worst-case behavior and is stable, i.e., it preserves the relative order of elements that compare equal under the sorting criterion. Stability is a central requirement in many programming libraries, e.g., the Java runtime library requires the sorting method for objects to be stable. Indeed, the need of a fast and stable general-purpose sorting method for the CPython reference implementation of the Python programming language was the main motivation for Tim Peters to develop a new variant of

Mergesort, known as Timsort [13].

Apart from its stability, Mergesort is also particularly well-suited for *adaptive sorting*, i.e., for saving effort when the input is already partially sorted (see also Section 2). Again, Timsort did pioneering work in bringing such adaptive sorting to most modern standard libraries (including Python, Java, Android runtimes, the V8 Javascript engine, Rust, Swift, Apache Spark, Octave, and the NCBI C++ Toolkit), giving users linear complexity for sufficiently sorted inputs.

More specifically, Timsort takes advantage of existing *runs*, i.e., contiguous regions of the input that are already in sorted order. Timsort detects such runs on the fly and maintains a stack of runs yet to be merged. When the next run is found, a particular merge policy triggers a (potentially empty) sequence of merges before that new run is added to the stack; this merge policy determines which merges are executed. Since the runs in the input can vary wildly in length, the chosen merge order can have a huge impact on efficiency. Consider as an indicative example an input with one run of length $n/2$ and $n/4$ runs of size 2. Repeatedly merging the long run with one small run results would result in $\Theta(n^2)$ overall time, whereas first merging all small runs (using a balanced tree) into a single run and then combined this run with the original long run is much cheaper with $\Theta(n \log n)$ time.

Despite its vast success in practice, increased scrutiny revealed several issues with the original merge policy of Timsort: First, an algorithmic bug allowed the run stack to grow larger than anticipated, *twice* leaving widely deployed libraries vulnerable to memory access violations upon sorting an adversarial input [6, 1]. Second, Timsort incurs up to 50% overhead over an optimal merge policy for certain inputs [5, 1, 2] with noticeable effects on running times [11]. These deficiencies have sparked a flurry of work on suggesting possible alternatives [1, 11, 5, 8], some with much stronger efficiency guarantees. After proving efficient in running time studies [15], Powersort [11] has now replaced Timsort's original merge policy in CPython, PyPy, and AssemblyScript [14].

---

[*]University of Liverpool, UK,
  William.cawleygelling@gmail.com
[†]Bielefeld University, Germany, nebel@tf.unibi.de
[‡]University of Liverpool, UK, b.m.smith@liverpool.ac.uk
[§]University of Liverpool, UK,
  sebastian.wild@liverpool.ac.uk

Multiway algorithms have also seen recent adoption, with a surprising change in the Java runtime library's (unstable) method for sorting primitive-typed arrays. There, a state-of-the-art implementation of classic Quicksort – the standard sort for decades – was replaced in 2011 by a novel dual-pivot variant after the latter proved 10% faster in practice. Detailed analysis revealed that the observed speedup is due to memory transfers: dual-pivot Quicksort profits from savings in *"scanned elements"*, the number of touched memory cells not in cache [10, 3, 12]. Since improvements in CPU speed have outpaced improvements in memory bandwidth over several decades, it now pays to avoid memory transfers in internal sorting, even at the expense of slightly increasing effort inside the CPU or more complicated code [3, 16]. Multiway partitioning in Quicksort resp. multiway merging in Mergesort does exactly that: By getting more of the sorting task done in one pass over the data, the overall data volume transferred between memory and CPU is reduced. An abstract cost measure called *scanned elements* has proven appropriate to capture corresponding effects.

While multiway merging is an elementary trick of the trade to reduce the memory-transfer cost in Mergesort, applying it to state-of-the-art adaptive sorting methods is a delicate task: to not forsake the gains on partially sorted inputs, we have to generalize the policy for deciding which runs to merge. Indeed in the case of Timsort, it seems quite unclear how to utilize multiway merges; Timsort's local rules are tailored to 2-way merges. Apart from Peeksort and Powersort, the same is true for all other practical methods suggested to replace Timsort, in particular adaptive Shivers Sort [8] and α-MergeSort [5]. By contrast, Powersort is based on a principled construction, mimicking nearly-optimal binary search tree algorithms – which allows us to present a multiway generalization.

1.1. Contributions.   In this paper, we show how to use multiway merging in nearly-optimal Mergesort variants and that significant performance improvements result from that. More precisely, our contributions are as follows:

(1) We design k-way Powersort, a provably optimal generalization of Powersort which simultaniously merges k runs wherever this is possible. To our knowledge, this is the first run-adaptive multiway Mergesort variant presented in the literature. We prove that – just like standard 2-way Powersort – k-way Powersort uses $\leq n \cdot \mathcal{H} + 3n$ comparisons to sort a list of n elements, where $\mathcal{H}$ is the run-length entropy (defined in Section 2) when k is a power of two; this is optimal in the worst case. We furthermore show that the asymptotic worst-case merge cost for k-way Powersort is a factor $\log_2(k)$ smaller than

for 2-way Powersort. Here the merge cost counts the total number of output elements of all merges conducted by an algorithm and thus approximates the amount of memory transfers in Mergesort variants.

(2) We engineer an efficient implementation of 4-way Powersort in C++ and demonstrate in an extensive running time study that it realizes performance improvements between 15 % and 20% over standard 2-way Powersort. This observation is robust against changes of the size of elements to be sorted (we compare integer and record data) and different amounts of presortedness of the input. Besides wall-clock running times, we determine scanned elements and cache performance in valgrind/cachegrind to explain the observed speedup.

(3) We confirm that the results reported in [11] for Java implementations of the algorithms are reproducible in a low-level implementation.

1.2. Related Work.   Adaptive sorting refers to any algorithms that exploit structure in the input to improve their performance. A survey of adaptive sorting up to 2018 and the story behind Timsort was given in [11], so we focus on developments since then. After [5] pointed out the suboptimal worst-case of Timsort's merge policy, several papers suggested alternatives [1, 8, 11]. The most noteworthy of these is Adaptive Shivers Sort [8], which shares many favorable properties with Powersort and proves the existence of a merge policy that is optimal up to an $O(n)$ term in the theoretical model of Buss and Knop [5]. It is not clear whether Shivers Sort can easily be generalized to multiway merging.

A second related work studies Timsort's galloping merge method in detail. Ghasemi et al. [7] show that most Mergesort variants can sort in $\mathcal{H}^* n + O(n)$ comparisons when using a galloping merge, where $\mathcal{H}^*$ is the "dual-run-length entropy" (which is at most the classical entropy of multiplicities of equal elements). Note that for sorting arrays, galloping only saves on comparisons; one still has to move elements around. The improvements obtainable from galloping merges are thus largely orthogonal to our work.

2. Preliminaries

If considering a bottom-up approach for Mergesort, one can take advantage of continuous regions of the input that are already sorted. We call such a region a *run* and define both maximal weakly increasing (i.e., non-decreasing), or strictly decreasing regions as runs – the latter can easily be reversed without issues towards stability. In the sequel, we will always denote the number of runs in the input by r and by $L_0, \ldots, L_{r-1}$ their respective lengths such that $L_0 + \cdots + L_{r-1} = n$ holds. From an algorithmic point of view, the unique partitioning of

the input into sorted segments (runs) can be found by an initial scan of the input.

2.1. Lower Bound.

In lower bounds for the cost of sorting, the binary Shannon entropy $\mathcal{H}$ often shows up: For $p_1, \ldots, p_m \in (0,1)$ with $p_1 + \cdots + p_m = 1$ we define $\mathcal{H}(p_1, \ldots, p_m) = \sum p_i \lg(1/p_i)$, where here and throughout $5\lg = \log_2$.

Theorem 2.1 ([4, Thm. 2]): *Sorting a list of n elements with initial runs of lengths $L_0, \ldots, L_{r-1}$ requires $\mathcal{H}(\frac{L_0}{n}, \ldots, \frac{L_{r-1}}{n})n - O(n)$ comparisons in the worst case.*◁

Where the run lengths are clear from the context, we will use $\mathcal{H}$ to abbreviate $\mathcal{H}(\frac{L_0}{n}, \ldots, \frac{L_{r-1}}{n})$.

2.2. Merging and its memory-transfer cost.

Merging describes the process of combining two or more sorted lists (i.e., runs) into a single, sorted list containing the same elements, by means of pairwise comparisons of elements. All our merge methods retain the input runs in a buffer area and produce the merged output in another area. (In-place methods exist, but their time overhead renders them uncompetitive for our purposes.) Important metrics for merging methods include the number of key comparisons and the number of *scanned elements,* i.e., memory accesses (read or write) that are not served from cache [12].

Merging can be achieved by iteratively finding the minimum among the smallest remaining elements of the runs and moving it to the output. When merging k input runs at once, maintaining a *tournament tree* [9] of the run minima (initialized using $k - 1$ comparisons) allows us to find the next output element after $\lceil \lg(k) \rceil$ or $\lfloor \lg(k) \rfloor$ comparisons. So we can merge k runs with n elements in total using at most $\lceil \lg(k) \rceil n + k - 1$ comparisons; unless the input runs are of very different lengths, this strategy is essentially optimal (for worst-case comparison cost). When the input runs already reside in the buffer area, we can assume that each element is "scanned" exactly twice (fetched once and output once); it can be read and compared many times, but these accesses can be assumed to be cached (for moderate k).

2.3. Run-adaptive Mergesort.

Mergesort operates by successively merging runs until only a single one remains. To get started, classic top-down Mergesort considers each individual element as a trivial run. An *adaptive* (i.e., *run-adaptive*) Mergesort instead finds (contiguous) segments of the input that are already in order.

Since both scanned-element count and comparison count are linear in the output size of a merge (for fixed k and sufficiently long runs), a convenient abstract measure to compare differentMergesort variants is the *merge cost* [5], defined as the total size of the outputs of all the merges.

Let $R_0, \ldots, R_{r-1}$ be the runs in the input; recall that L_i denotes the length of R_i. Any stable run-adaptive mergesort can be described as a *merge tree T*: an (ordered rooted) tree with leaves $R_0, \ldots, R_{r-1}$ (appearing in that order from left to right). Each internal node v of T corresponds to the result of merging its children; we write $M(v)$ for the total length of the result of that merge. The total merge cost of T is then $M = M(T) = \sum_{v \in T} M(v)$. T is a k-way merge tree if all its nodes have degree at most k. We remark here that it is in general not possible to have all merge-tree nodes combine exactly k runs, therefore a k-way merge tree is allowed to contain nodes of smaller degree. (Though we do not allow unary nodes.)

We further denote by B_i the boundary between the $(i-1)$st and ith run ($i = 1, \ldots, r-1$). In binary merge trees ($k = 2$), there is a one-to-one correspondence between the B_i and the (inner/non-leaf) merge-tree nodes; for multiway merging, this becomes many-to-one: each merge-tree node v of degree d is associated to exactly $d - 1$ (not necessarily adjacent) run boundaries $B(v, 1), \ldots, B(v, d-1)$; conversely each run boundary B_i is assigned exactly to one merge-tree node $m(B_i)$.

Let d_i be the *depth* of leaf R_i in T, where depth is the number of edges on the path to the root. Every element in run R_i is counted exactly d_i times in $\sum_v M(v)$, so we have $M = \sum_{i=0}^{r-1} d_i \cdot L_i$.

3. Multiway Powersort

We now introduce Powersort's merge policy for general k-way merging. We expressly include the special case $k = 2$ here; the resulting merge policy is equivalent to Powersort from [11].

3.1. Intuition and Merge Tree.

Let $k \geq 2$ be fixed, and first assume for ease of presentation that $n = k^m$, $m \in \mathbb{N}$. The intuition behind Powersort is to imagine a complete k-ary tree T_V sitting on top of the array, where each element $A[i]$ of the input corresponds to a leaf of T_V. T_V corresponds to the merge tree of a non-adaptive k-way Mergesort (starting with individual elements). Now each run boundary B_j also corresponds to an (inner) node w of this (virtual) k-ary tree T_V, namely the lowest common ancestor of the two leafs adjacent to B_j.

We cannot follow T_V if we want to retain all existing runs (most will usually straddle node boundaries); but we will use the depths d_w of the nodes of T_V as a guide: Any non-leaf node w in T_V splits (a range of) the array into k segments. At the $k - 1$ boundaries between these segments, we draw a vertical bar of height $m - d_w$ (see Figure 1). Then, we assign each boundary B_j its *mid-*
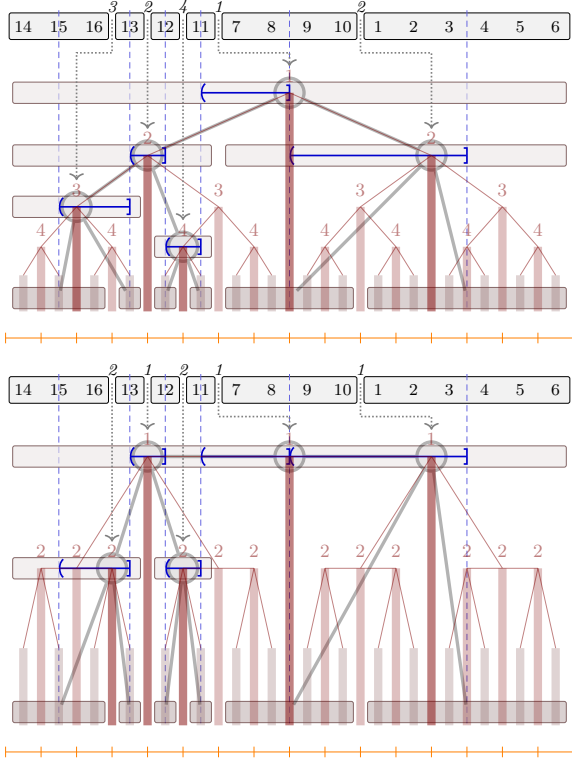
Figure 1: Example merge tree for Powersort (top) and 4-way Powersort (bottom) for an input of size $n = 16$: Run boundaries are mapped (gray arrows) to nodes of a (virtual) perfectly balanced binary resp. 4-ary tree T_V (shown in light red), which has the array elements as its leaves. The midpoints of two adjacent runs form the horizontal range in which this node is allowed to move; it then "snaps" to the highest pole in that range.

point interval: the horizontal interval from the middle of R_{j-1} (exclusive) up to the middle of R_j (inclusive). Midpoint intervals are shown in blue in Figure 1. Now, the *power* P_j of B_j is defined as the depth d_w of the node w contributing the tallest vertical bar inside B_j's midpoint interval (Definition 3.1).

Since the midpoint intervals are disjoint, each vertical bar is assigned at most one B_j, and so each node w in T_V is assigned at most $k - 1$ run boundaries. These $k - 1$ run boundaries define one merge-tree node $v(w)$; the collection of $v(w)$ forms the k-way merge tree T of Powersort. Intuitively, T is a good merge tree since it uses boundaries as close to the perfectly balanced T_V while respecting existing runs. Note that going from 2-way powers to 4-way powers as in Figure 1 corresponds to a simple transformation: We "squish" pairs of adjacent levels of 2-way powers into one layer of 4-way powers: $P_i^{(4)} = \lfloor (P_i^{(2)} - 1)/2 \rfloor + 1$. We thus obtain the merge tree of 4-way Powersort from the one for 2-way Powersort

by letting all nodes at even depth absorb their children (and adopt their grandchildren).

When n is not a power of k, the virtual tree T_V does not align with element boundaries, but we can treat the array and its runs as *continuous* intervals; the assignment and argument from above work the same way; see Figure 2 for an example. Formally, we map the array to the unit interval, partitioned by the runs to define the power of run boundaries:

Definition 3.1 (Run Boundary Power): *Let $k \geq 2$ be fixed. For run lengths $L_0, \ldots, L_{r-1}$, set $\ell_i = L_i/n$. For $1 \leq i < r$, let B_i be the run boundary between the $(i-1)$st and ith run. The (k-way) power of B_i is*

$$P_i^{(k)} = \min\Big\{p \in \mathbb{N} : \lfloor a_i \cdot k^p \rfloor < \lfloor b_i \cdot k^p \rfloor \Big\},$$
$$\text{where } a_i = \sum_{j=0}^{i-1} \ell_j - \tfrac{1}{2}\ell_{i-1}, \ \ b_i = \sum_{j=0}^{i-1} \ell_j + \tfrac{1}{2}\ell_i. \ \lhd$$

We drop the superscript when k is clear from context. Figure 3 illustrates Definition 3.1.

Based on (just) $P_1, \ldots, P_{r-1}$, we can recover the merge tree T by recursively finding all occurrences of the smallest occurring power and using these run boundaries as the splitting points for recursion. The pseudocode in Algorithm 1 makes this concrete.


KwayTree$_k$($A[b..e]$)

1   Let $s_0 = b, s_1, \ldots, s_{r-1}, s_r = e$ be
        the start indices of runs in $A[b..e]$
2   Let $P_1, \ldots, P_{r-1}$ be the boundary powers
3   $\{i_1, \ldots, i_m\} = \arg\min\{P_i : i \in [1..r]\}$
4   $i_0 = s_0; \ i_{m+1} = s_r$
5   **for** $j = 0, \ldots, m$
6   KwayTree$_k$($A[i_j..i_{j+1}]$) **// recurse**
7   **end for**
8   Merge($A[i_0..i_1], \ldots, A[i_m..i_{m+1}]$)

Algorithm 1.: Conceptual algorithm to compute the merge tree of k-way Powersort.

This is just a conceptual algorithm; Multiway Powersort will produce the same merge tree as KwayTree$_k$($A[0..n]$), but without having to search for minimal powers explicitly or even knowing all runs and powers at any point in time (as described below). KwayTree$_k$ makes it obvious that P_j is the "intended" depth of $m(B_j)$ since we combine nodes recursively by increasing power. Nodes can end up *higher* in the merge tree if not all power values occur. Since we count edges for the depth, but powers start at 1, this yields following useful inequality.

Fact 3.2: depth($m(B_j)) \leq P_j - 1$. $\hfill \lhd$
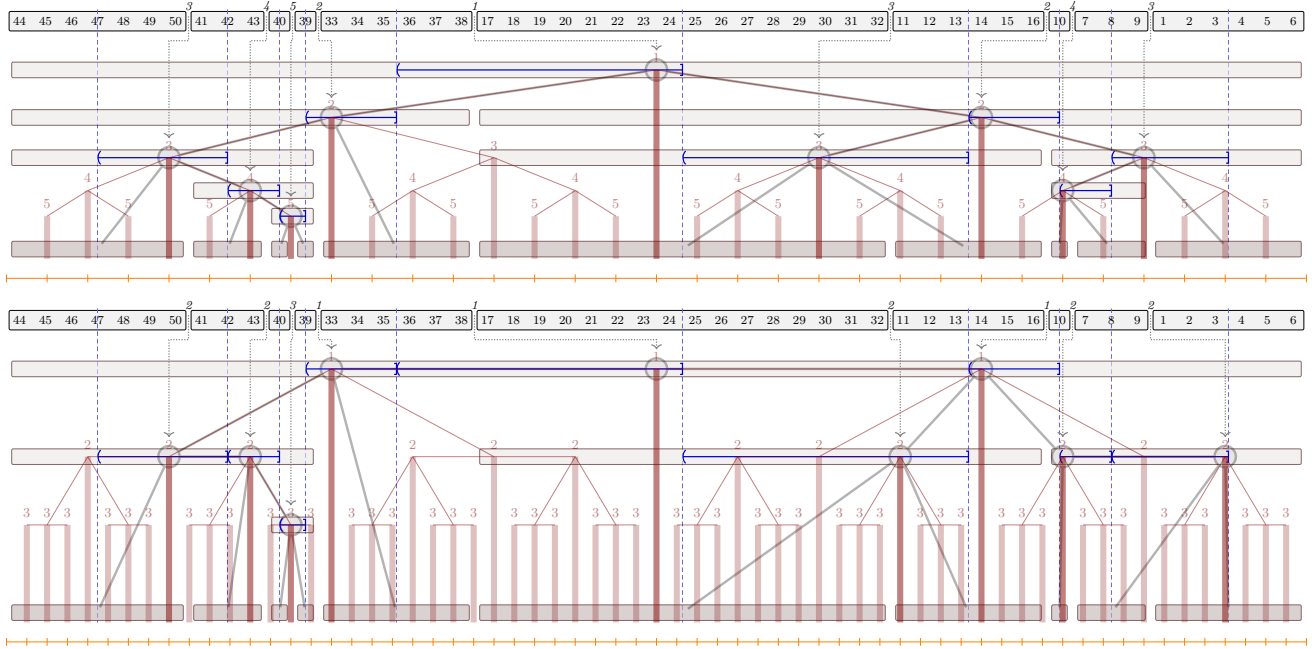
Figure 2: Example merge tree for Powersort (top) and 4-way Powersort (bottom) for an input of size $n = 50$: Here, merges are "rounded" to nodes in a (virtual) perfectly balanced binary resp. 4-ary tree T_V (shown in light red); drawing style as in Figure 1.
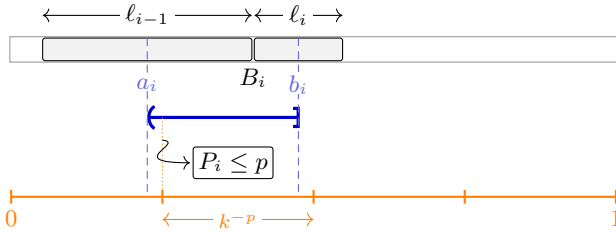
Figure 3: Illustration of the power of a run boundary. Since there is an orange tick inside the midpoint interval, we obtain $P_i^{(k)} \le p$; in the example, the midpoint interval is $(a_i, b_i] = (0.225, 0.475]$, $k = 4$ and $p = 1$, so $P_i^{(4)} = 1$.

A simple proof by induction shows that in each recursive call of $\text{KWAYTREE}_k(A[0..n))$, we have $m \le k - 1$, so we are merging at most k runs at once.

3.2. k-way Powersort. Like Timsort, Powersort processes the merge tree left to right. We maintain a stack S of runs and powers, subject to the invariant that powers are weakly increasing in S (from bottom to top). When the next run boundary B_i has a power P_i larger or equal than $S.top()$, we push (R_{i-1}, P_i) onto S and continue. Otherwise, we merge R_i with *all runs* on S with power *equal* to $S.top()$; this is a key difference to standard Powersort, where we always just merge with $S.top()$.

We repeatedly perform such merges until we can push (R_{i-1}, P_i) onto S without violating the invariant.

The definition of k-way powers ensures that there are at most $k - 1$ equal powers on S at any time, so all merges executed in the above process combine at most k runs. Once all run boundaries have been treated that way, the remaining stack is merged top to bottom (similar to Timsort)

3.3. Analysis. We state a few properties about Multiway Powersort, most importantly a bound on its merge cost. The case $k = 2$ has been covered in [11], but the following self-contained proof is arguably more direct.

Theorem 3.3: *The merge cost of k-WAY POWERSORT is $M \le \frac{1}{\lg(k)} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n}) n + 2n$ and the number of comparisons $C \le \frac{\lceil \lg k \rceil}{\lg k} \mathcal{H}(\frac{L_0}{n}, \dots, \frac{L_{r-1}}{n}) n + (1 + 2\frac{\lceil \lg k \rceil}{\lg k}) n + (k - 1) r.$* ◁

Proof: We actually show $M/n \le \sum_{i=0}^{r-1} \ell_i \lceil \log_k(1/\ell_i) + 1 \rceil$, which implies the claim. We start with the expression from Section 2.3 for the merge cost: $M = \sum_{i=0}^{r-1} d_i \cdot L_i$ where d_i is the depth of leaf R_i in T. Since R_i is a leaf, it must be the (direct) child of either $m(B_i)$ or $m(B_{i+1})$ in T and hence $d_i = 1 + \max\{\text{depth}(m(B_i)), \text{depth}(m(B_{i+1}))\}$; by formally setting $\text{depth}(m(B_0)) = \text{depth}(m(B_r)) = 0$, this equality holds for all $i = 0, \dots, r-1$. Now, Fact 3.2 yields $d_i \le \hat{P}_i$,

```
KwayPowerSort_k(A[0..n))
 1   S := empty stack     // capacity (k − 1)⌈log_k(n) + 1⌉
 2   b_1 := 0;  e_1 = FirstRunOf(A[b_1..n))
 3   while e_1 < n
 4         b_2 := e_1;  e_2 := FirstRunOf(A[b_2..n))
 5         P := Power_k(n, b_1, e_1, b_2, e_2)
 6         while S.top().power > P
 7               P' := S.top().power
 8               L := empty list;  L.append(S.pop())
 9               while S.top().power == P'
10                   L.append(S.pop())
11               end while
                  // merge runs in L with A[b_1..e_1)
12               (b_1, e_1) := Merge(L, A[b_1..e_1))
13         end while
14         S.push(A[b_1, e_1), P)
15         b_1 := b_2;  e_1 := e_2
16   end while // Now A[b_1..e_1) is the rightmost run
17   while ¬S.empty()
                  // pop (up to) k − 1 runs, merge with A[b_1..e_1)
18         (b_1, e_1) := Merge(S.pop(k − 1), A[b_1..e_1))
19   end while


Power_k(n, b_1, e_1, b_2, e_2)
 1   n_1 := e_1 − b_1;  n_2 := e_2 − b_2;  p := 0
 2   a := (b_1 + n_1/2)/n;  b := (b_2 + n_2/2)/n
 3   while ⌊a · k^p⌋ == ⌊b · k^p⌋ do p := p + 1 end while
 4   return p
```

Algorithm 2.: Multiway Powersort pseudocode. The function FirstRunOf finds the leftmost run in an array and returns its endpoint.

$i = 0, \dots, r - 1$, where $\hat{P}_i := \max\{P_i, P_{i+1}\}$ and we similarly set $P_0 = P_r = 0$. So we have $M/n \leq \sum_{i=0}^{r-1} \ell_i \hat{P}_i$.

We will now show that $k^{-p} \leq \ell_i/2$ implies $P_i^{(k)} \leq p$. It is easy to see that with a_i and b_i as in Definition 3.1, we have $P_i^{(k)} = \min\{p : \exists c \in \mathbb{N} : c \cdot k^{-p} \in (a_i, b_i]\}$; (cf. Figure 3). It suffices to note that $|(a_i, b_i]| \geq \ell_i/2$ and whenever $k^{-p} \leq |(a_i, b_i]|$ is given, $(a_i, b_i]$ clearly contains a value ck^{-p} (an orange "tick" in Figure 3). Note that since $|(a_{i+1}, b_{i+1}]| \geq \ell_i/2$ similarly holds, the above argument also shows that $k^{-p} \leq \ell_i/2$ implies $P_{i+1}^{(k)} \leq p$.

Now, $k^{-p} \leq \ell_i/2$ iff $p \geq \log_k(1/\ell_i) + 1$, so setting $p = \lceil \log_k(1/\ell_i) + 1 \rceil$ implies that condition and hence $P_i^{(k)} \leq \lceil \log_k(1/\ell_i) \rceil + 1$ as well as $P_{i+1}^{(k)} \leq \lceil \log_k(1/\ell_i) \rceil + 1$; hence $\hat{P}_i \leq \lceil \log_k(1/\ell_i) \rceil + 1$. Inserting shows $M/n \leq \sum_{i=0}^{r-1} \ell_i \lceil \log_k(1/\ell_i) + 1 \rceil$ as claimed.

For the number of comparisons, recall that the tournament-tree merge uses $\leq \lceil \lg(k) \rceil$ comparisons per output element, except for the first output element of each merge, where we initialize the tournament using $k - 1$ comparisons. Let μ be the number of merges; the exact number depends on the merge policy, but $\lceil \frac{r-1}{k-1} \rceil \leq$

$\mu \leq r - 1$. Including the $n - 1$ comparisons to find runs, we overall spend $C \leq M \cdot \lceil \lg k \rceil + \mu \cdot (k - 1 - \lceil \lg k \rceil) + n - 1$. Inserting the bound for M, we obtain the claimed $C \leq \frac{\lceil \lg k \rceil}{\lg k} \cdot (\mathcal{H}n + 2n) + (k - 1)r + n$ comparisons. □

Remark 3.4: (a) We note that the analysis of C could be slightly sharpened, but we are mostly interested in the case where k is a small power of two; there, the above bound is tight up to lower order terms.

(b) For $k = 2$, the bounds simplify to Thm. 5 of [11], for which we above give an alternative proof that is entirely self-contained and elementary.   ◁

Proposition 3.5: *The maximal stack height for k-way Powersort is* $(k - 1)\lceil \log_k(n) + 1 \rceil$.   ◁

Proof: We use that $P_i \leq \lceil \log_k(1/\ell_i) + 1 \rceil$ (see above); since $\ell_i \geq 1/n$, we obtain $1 \leq P_i \leq \lceil \log_k(n) + 1 \rceil$. So there are $\lceil \log_k(n) + 1 \rceil$ different possible values for P_i. The run stack S is weakly increasing and can contain at most $k - 1$ entries with equal power, so $|S| \leq \lceil \log_k(n) + 1 \rceil (k - 1)$.   □

Remark 3.6: We point out that Peeksort, the other algorithm from [11], can also be generalized to k-way merges. In essence, instead of finding the (single) run boundary closest to the array midpoint, we now use $k - 1$ equidistant points and pick the run boundaries closest to those as boundaries of recursive calls. There are a few corner cases that require care, though. We defer our corresponding results to a full version of this paper due to space constraints and since Powersort is the preferred method in practice.

4. Results

This section describes our empirical results around Multiway Powersort.

4.1. Implementations and setup.   We have implemented 2-way and 4-way Powersort in C++ and engineered both for performance under the GNU Compiler Collection. We also compare them to a few standard algorithms including `std::sort` and `std::stable_sort` from the GNU implementation of the C++ Standard Library. Key parts of the code are shown in Appendix A. The full implementations including instructions to reproduce our running-time study are available on GitHub: *https://github.com/sebawild/powersort*.

We comment on a few implementation details that are relevant for the running-time comparisons.

Tournament tree.   For 4-way merging, we use a tournament tree with 3 internal nodes. We represent

it as a winner tree, i.e., each node stores the run contributing the smaller element among its two children; more specifically, each node store a pointer to the first element of a run that has not been output yet. The root additionally stores whether the minimum there came from the left or right subtree. That way each element moved to the output entails 2 tournaments (node recomputations), and we know which ones these are. Since our tournament tree is small and will entirely reside in registers, directly accessing the two children of a node for playing a match is very inexpensive, so there seems to be little benefit from using a loser tree instead of the simpler winner tree. Whenever an element of a run enters the tournament tree (because it has won against its sibling run), we already advance the pointer for the leaves of the tournament tree; this decouples the advancement of the pointer from writing the output.

Sentinel values. We implemented merging methods that assume the availability of a $+\infty$ value that is strictly larger than any element in the list to be sorted. Such values are often available; floating-point types directly support them, integer types may be restricted to not using the largest representable value; strings and characters can use the null character (with a symmetric implementation that uses values smaller than any value to sort). We can place such a sentinel value at the end of runs to avoid a pointer comparison in the inner loop of the merging methods, which saves a few instructions and branches per iteration. Below, we report on results for algorithms with and without sentinels.

Sentinel-free merging by stages. When a type does not (efficiently) support a $+\infty$ value, it can be beneficial to have a merge method that does not use sentinels. For 2-way Powersort, a trivial such merge implementation is only slightly slower overall than using sentinel-based 2-way merge. For 4-way merging using tournament trees, there are many more cases to distinguish (which runs are empty), making a trivial implementation inefficient. We combine two tricks to improve upon that: First, we merge in "stages" where a stage ends whenever one run is exhausted. This allows us to reduce the number of cases to distinguish in the code (at the expense of longer code). Second, if s is the length of the shortest remaining (nonempty) run, we can clearly do at least s iterations without checking boundaries, after which we recompute s. When s reaches 0, we know that the current stage is over. "4-way Powersort (no sentinel)" uses merging by stages.

Buffer for runs. All our merge methods have an in-place interface, i.e., they produce the merged result in the array initially occupied by the input runs (and assume those to be adjacent), but they use a linear-size buffer for efficient merging. Most methods copy all runs to the buffer (which tends to give the fastest code for the merging phase), but we also include a 2-way method that only copies the smaller run to the buffer and merges "into the gap" (a standard trick used, e.g., in Timsort). This method does not use sentinels.

Minimal run length. All our Mergesort variants use Insertionsort on subproblems of size ≤ 24; for Timsort-based methods like Powersort, this means that when a new run is found with fewer elements, it is extended to 24 elements.

4.2. Experimental Setup. The experiments were run on a PC running Ubuntu 20.04.4 LTS. The CPU has 4 cores running 2 hyperthreads each and 16 GB of main memory. The model of the CPU is Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz. Each core has a private level-1 cache with 32KB for data and 32KB for instructions and a unified level-2 cache of 256KB. Moreover, there's a shared level-3 cache of 8MB. All caches are organized into 64B cache lines and are 8-way associative; the level-3 cache is 16-way associative.

The experiments use `g++` from the GNU Compiler Collection 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.1) with optimization flags `-Ofast -march=native -mtune=native`. (Results did not change noticeably when using `-O3` instead.)

4.3. Hypothesis 1: 4-way Powersort can yield significant performance improvements. To test this hypothesis, we conducted a running-time study varying several dimensions to investigate the speedup obtainable from 4-way merging in Powersort.

As an indicative first scenario, Figure 4 shows normalized running times for sorting a mildly presorted list of `int`s (as in [11]): across four orders of magnitude of input sizes, 4-way Powersort is consistently around 20% faster. Figure 5 shows that the distribution of running times is concentrated around the mean, showing a robust difference. Figure 4 also shows that adaptive mergesort methods outperform the Quicksort-based `std::sort`.

The random-runs inputs have substantial sorted parts; in expectation $\sqrt{n}$ runs with an expected $\sqrt{n}$ elements each, albeit with substantial variation. However, the lower bound $n\mathcal{H}$ approaches $\frac{1}{2}n\lg(n)$, so they are still far from fully sorted; in particular, sorting these still has linearithmic complexity (as clearly visible in Figure 4).

A first variation considers a more realistic data type than integers: Stable sorting is most relevant for sorting objects that have various properties / instance variables and which are (partially) sorted according to one of its properties. We model this by sorting records containing a `long` key and a pointer. Figure 6 shows the respective
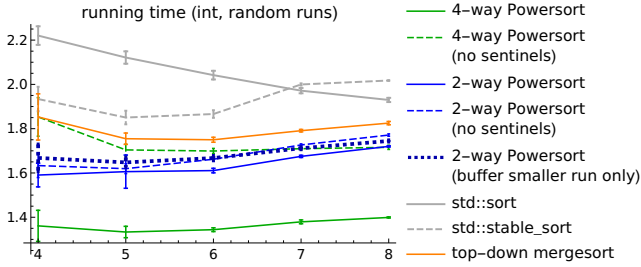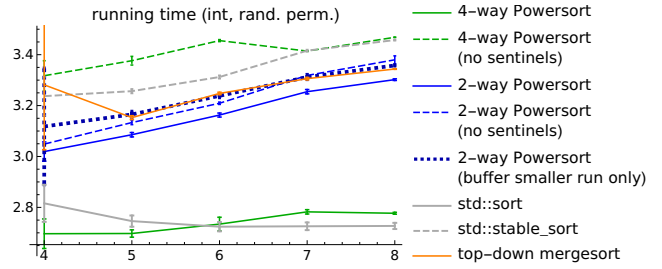
Figure 4: Normalized **running times** for sorting `int`s (4 byte signed integers) where the input has **random** initial **runs** with a geometric distribution and expected length $\sqrt{n}$. The x-axis shows $\log_{10}(n)$, the y-axis shows average running time in ms multiplied by $10^6/n \lg n$; error bars show one standard deviation. We used 1000 repetitions up to 10^6 elements and 100 repetitions for 10^7 and 10^8 elements.
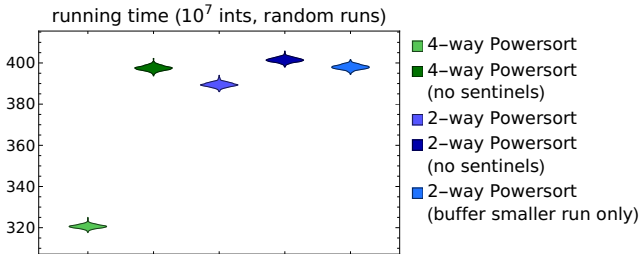
Figure 5: Distribution of running times for $n = 10^7$ from Figure 4 (sorting `int`s with random runs).
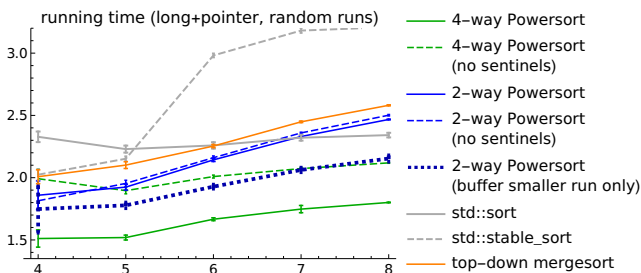
Figure 6: Normalized **running times** for sorting **records** of a `long` key and a pointer (16 byte in total) where the input has **random** initial **runs** with a geometric distribution and expected length $\sqrt{n}$. Axes as in Figure 4.

Figure 7: Normalized **running times** for sorting `int`s where the input is a **random permutation** of n. The x-axis shows $\log_{10}(n)$, the y-axis shows average running time in ms multiplied by $10^6/n \lg n$; error bars show one standard deviation.

running times.

Since elements are now 4 times larger (16 vs. 4 bytes), it is not surprising that all methods are slower than for `int`s, but they are not uniformly so. 4-way Powersort with the sentinel-free merging by stages was roughly as fast as 2-way Powersort, which in turn had essentially equal performance under all three considered 2-way merge methods. For records, the 4-way method and the 2-way method that copies only the smaller run to a buffer, substantially outperform the other 2-way Powersort variants. 4-way Powersort with sentinel-based merge remains substantially faster still (between 15 and 20% across the studied input-size range).

Finally, we vary the input distribution. In particular, any adaptive sorting method should remain competitive with the best general purpose methods when no significant presortedness exists to be exploited. Figure 7 shows running times for random permutations. Somewhat surprisingly, even where effectively no presortedness exists, 4-way Powersort is still 15–20% faster than the fastest 2-way method and almost as fast as `std::sort`.

We have taken care to implement the merge methods (2-way and multiway) as efficiently as possible. The presented methods are chosen from a much larger collection of implementation alternatives we have experimented with. As Figures 4–7 show, a $+\infty$ value, that can be used as a sentinel value for stopping loops (where otherwise a range check for the iterator would be needed), can yield a substantial speedup in multiway merging.

4.4. Hypothesis 2: Scanned elements explain the speedups. After documenting the speedup realized by 4-way Powersort, we now explore likely explanations for these observations. To that end, we ran our code through the cachegrind/callgrind memory hierarchy simulator for valgrind; Table 1 shows the results. While there is an 8% saving in the plain number of executed

Algorithm	L1 rm	L1 wm	instructions	cycle estimate	merge cost	buffer cost	comparisons
nop	17	6 250 001	400 000 361	1 087 503 111	—	—	—
4P	90 998 034	90 941 221	14 386 631 421	26 834 656 351	678 233 797	678 244 262	1 401 885 151
4P w/o sent	91 104 862	90 991 531	21 788 074 703	34 238 304 273	678 233 797	678 233 797	1 401 804 460
2P	168 472 841	168 392 912	15 593 213 610	36 636 357 390	1 298 329 585	1 298 349 759	1 398 341 256
2P w/o sent	124 242 042	118 893 488	17 678 030 619	32 327 153 489	1 298 329 585	603 947 459	1 398 280 296

Table 1: Cachegrind results for sorting one input of $n = 10^8$ `ints` with random runs of expected length $\sqrt{n} = 10^4$. The table shows level 1 cache read misses (L1 rm), level 1 cache write misses (L1 wm), the total number of executed instructions, and cachegrind's total-time estimate, defined as cycles $= \text{instr} + 10 \cdot \text{L1-misses} + 100 \cdot \text{LL-misses}$, where LL $=$ level 3 cache. The algorithms are "nop" $=$ no sorting at all, "4P" $=$ 4-way Powersort, "2P" $=$ 2-way Powersort, "4P w/o sent" $=$ 4-way Powersort without sentinels, "2P w/o sent" $=$ 2-way Powersort without sentinels and copying only the smaller run to the buffer. Here, "nop" is used to get a baseline; the reads come from checking that the input is correctly sorted as part of our running time setup. The reported numbers do not include the cost of generating the random input. The L1 cache lines can hold 16 elements (for our machine); read and write misses are determined by cachegrind. The last three columns were determined directly from the code using counters.
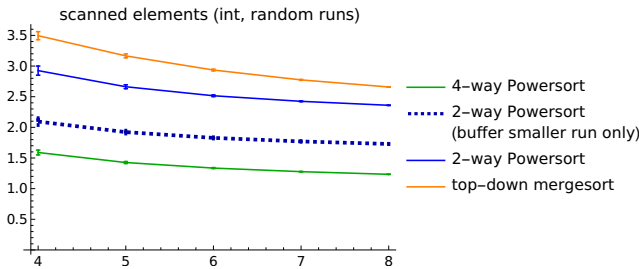
Figure 8: Normalized **scanned elements** when the input has random initial runs with a geometric distribution and expected length $\sqrt{n}$. The x-axis shows $\log_{10}(n)$, the y-axis shows average merge cost divided by $n \lg(n/24)$. Error bars show one standard deviation.
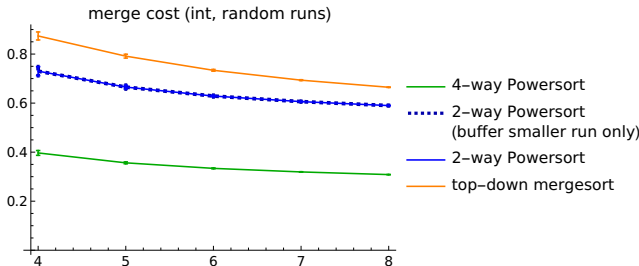
Figure 9: Normalized **merge cost** when the input has random initial runs with a geometric distribution and expected length $\sqrt{n}$. The x-axis shows $\log_{10}(n)$, the y-axis shows average merge cost divided by $n \lg(n/24)$. Error bars show one standard deviation.

instructions – owing to the extremely lean inner loops in our sentinel-based 4-way method – the much more dramatic improvement lies in the number of cache misses: these are reduced to 54% of the 2-way method that copies both runs to buffer resp. 73% when copying the smaller only. Our data shows that 4-way Powersort simultaneously reduces both the number of instruction and the number of cache misses.

We point out that all memory accesses in our sorting methods happen in sequential scans, so the memory access patterns are already as cache- and prefetcher friendly as can be; instead of cache misses, rather the sheer data volume transferred from memory is the issue. Since the memory scans in merging are either read-only or write-only, the total number of scanned elements can be computed very accurately: On the used machine, a cache line has 64 bytes, i.e., 16 `ints`, so we obtain scanned element counts as $16 \cdot (\text{L1rm} + \text{L1wm})$ (Figure 8). As a sanity check, we can compare numbers as follows: Each merge with merge cost m entails $2m$ read and $2m$ writes, moreover, we do (in total) one scan over the input for run detection (n reads) and initialize the buffer (n writes); this yields 100.034% of the reported L1 read misses and 100.097% of L1 write misses for 4P (slightly more since a few reads are cached).

We use cachegrind's simple linear-combination model as a first-order approximation for explaining running times from scanned elements and instruction counts; see column cycle estimate in Table 1. Comparing Figure 4 (at $n = 10^8$) to the cycle estimate, we see that despite its simplicity, this model is able to correctly identify 4P as the most efficient variant and it even gives a usable approximation of the relative improvement over the 2-way methods. While it does not rank the remaining algorithms (whose running time is very similar) correctly, we conclude that the large reduction in scanned elements is

the most likely reason for the speedup in 4-way Powersort. However, reducing scanned elements at the expense of a substantial increase in executed instructions is not fruitful.

4.5. Hypothesis 3: 4-way Powersort halves merge cost.   Last, we tie the reduction in scanned elements to the key algorithmic innovation. Figure 9 shows the merge cost over different input sizes.

We note that for worst-case inputs and as $\mathcal{H} \to \infty$, Theorem 3.3 shows that the merge cost of 4-way Powersort is asymptotically half of that for 2-way Powersort. Whether that is possible to realize for practical values of n is not clear a priori, though. As Figure 9 shows, 4-way Powersort reduces the merge cost to around 52% of that of Powersort, very close to halving costs.

5. Conclusions

Powersort, introduced in 2018 by Munro & Wild, is a stable (binary) Mergesort variant that exploits existing runs and finds nearly-optimal merging orders with negligible overhead. In this paper we presented Multiway Powersort, a generalization of Powersort, that uses k-way merges in order to speedup the algorithm. While the extension of classic 2-way Mergesort to k-way merges is straightforward, doing the same with optimal run-adaptive performance is an algorithmic challenge. In solving that, we also generalize the underlying algorithm for nearly-optimal *binary* search trees to nearly-optimal k-way search trees, which might be of independent interest.

The motivation for our generalization are expected advantages of multiway merges for the memory hierarchy of modern computers. We conducted extensive experiments based on our highly engineered implementation of 4-way Powersort around following hypotheses: Compared to 2-way Powersort

1. 4-way Powersort can yield significant performance improvements;

2. Scanned elements are a good predictor for this speedup;

3. 4-way Powersort halves the merge cost.

Our experiments fully confirmed the first hypothesis. Considering presorted integer inputs, 4-way Powersort was about 20% faster than the 2-way variant. This speedup was robust in terms of the size of the data items to be sorted (record data instead of integers) as well as the amount of presortedness. However, significant speedups are only realized when the data type allows sentinels to be used. If these are not available (because

there is no largest possible value), the running time advantages are made up for by additional range checks.

Regarding the second hypothesis, we find that compared to different versions of 2-way Powersort, 4-way Powersort incurs between 54% and 73% of the cache misses, and scanned-element counts very accurately reflect that. This is a plausible explanation for the observed speedup. A simple linear combination of the number of executed instructions and scanned elements explains much (but not all) of performance differences.

Last but not least, we find that the reduced flexibility of doing 4-way merges instead of 2-way merges hardly affects the effectiveness of Powersort's nearly-optimal merge policy: The merge cost is reduced to 52% for moderate-sized inputs, close to the theoretical optimum of $\frac{1}{2}$ approached in the limit.

In conclusion, 4-way Powersort provides notable advantages over both 2-way Powersort, and other known stable sorting algorithms. Thus, Multiway Powersort is a strong contender for the fastest general, stable sorting algorithm – especially in cases where presorted data can be expected. Both Mergesort variants use $\Theta(n)$ words of extra space for buffer areas; while binary merges can easily work with a buffer for $n/2$ elements, the same is not easily achieved with 4-way merging, where our current implementation uses space n elements.

Future work will study whether similar speedups are observed when sorting pointers to objects (as always happens in CPython). Moreover, when comparisons are expensive, combining multiway merging with the galloping merge strategy could be useful. Another interesting question is whether values of k larger than 4 offer some benefit. On one hand, a further reduction of memory transfers is likely beneficial for large inputs; on the other hand, the merging method becomes more complicated and thus implies more overheads. It is also conceivable that with larger k, the fraction of merge cost that arises from non-full merges, i.e., merging $< k$ runs, would grow.

References

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of TimSort. In Hannah Bast Yossi Azar and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), 2018. *doi:10.4230/LIPIcs.ESA.2018.4.*

[2] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort, 2018. URL: *https://arxiv.org/abs/1805.08612, doi:10.48550/ARXIV.1805.08612.*

[3] Martin Aumüller, Martin Dietzfelbinger, and Pascal Klaue. How good is multi-pivot quicksort? *ACM Transactions on Algorithms*, 13(1):1–47, dec 2016. *doi:10.1145/2963102*.

[4] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, November 2013. *doi:10.1016/j.tcs.2013.10.019*.

[5] Sam Buss and Alexander Knop. Strategies for stable merge sorting. In *Symposium on Discrete Algorithms (SODA 2019)*, pages 1272–1290. SIAM, jan 2019. *doi:10.1137/1.9781611975482.78*.

[6] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDK's sort method for generic collections. *Journal of Automated Reasoning*, August 2017. *doi:10.1007/s10817-017-9426-4*.

[7] Elahe Ghasemi, Vincent Jugé, and Ghazal Khalighinejad. Galloping in fast-growth natural merge sorts. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 68:1–68:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. *doi:10.4230/LIPIcs.ICALP.2022.68*.

[8] Vincent Jugé. Adaptive shivers sort: An alternative sorting algorithm. In *Symposium on Discrete Algorithms (SODA 2020)*, pages 1639–1654. SIAM, jan 2020. URL: *https://doi.org/10.1137%2F1.9781611975994.101*, *doi:10.1137/1.9781611975994.101*.

[9] Donald E. Knuth. *The Art Of Computer Programming: Searching and Sorting*. Addison Wesley, 2nd edition, 1998.

[10] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. Multi-pivot quicksort: Theory and experiments. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 47–60. SIAM, dec 2013. *doi:10.1137/1.9781611973198.6*.

[11] J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, volume 112 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl– Leibniz-Zentrum fuer Informatik, 2018. URL: *https://www.wild-inter.net/publications/munro-wild-2018*, *doi:10.4230/LIPIcs.ESA.2018.63*.

[12] Markus E. Nebel, Sebastian Wild, and Conrado Martínez. Analysis of pivot sampling in dual-pivot Quicksort – a holistic analysis of yaroslavskiy's partitioning scheme. *Algorithmica*, 75(4):632–683, August 2016. URL: *https://www.wild-inter.net/publications/nebel-wild-martinez-2016*, *arXiv:1412.0193*, *doi:10.1007/s00453-015-0041-7*.

[13] Tim Peters. [Python-Dev] Sorting, 2002. URL: *https://mail.python.org/pipermail/python-dev/2002-July/026837.html*.

[14] Tim Peters. Timsort (listsort.txt), 2021. URL: *https://github.com/python/cpython/blob/main/Objects/listsort.txt#L343*.

[15] Tim Peters et al. Replace list sorting `merge_collapse()`?, 2018. URL: *https://github.com/python/cpython/issues/78742*.

[16] Sebastian Wild. *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*. Dissertation (Ph. D. thesis), 2016. URL: *https://www.wild-inter.net/publications/wild-2016*.

A. C++ Code

We provide some key parts of our C++ implementation (slightly redacted for readability).

A.1. 4-way merge with sentinels.   We first give the sentinel-based 4-way merge method; this is the method that gave the best performance.

```
1   /**
2    * Merge runs [l..g1) and [g1..g2) and [g2..g3) and [g3..r) in-place into [l..r)
3    * using a buffer at B of length at least r-l+4.
4    */
5   template<typename Iter, typename Iter2>
6   void merge_4runs(Iter l, Iter g1, Iter g2, Iter g3, Iter r, Iter2 B) {
7       auto n = r - l;
8       // Copy runs to B and append a sentinel value after each.
9       std::copy(l,  g1, B);                      *(B + (g1 - l))     = plus_inf_sentinel();
10      std::copy(g1, g2, B + (g1 - l) + 1);  *(B + (g2 - l) + 1) = plus_inf_sentinel();
11      std::copy(g2, g3, B + (g2 - l) + 2);  *(B + (g3 - l) + 2) = plus_inf_sentinel();
12      std::copy(g3, r,  B + (g3 - l) + 3);  *(B + (r  - l) + 3) = plus_inf_sentinel();
13      Iter2 c[4];  // pointers to runs in B.
14      c[0] = B, c[1] = B + (g1-l)+1, c[2] = B + (g2-l)+2, c[3] = B + (g3-l)+3;
15      // initialize tournament tree
16      Iter2 x, y;  std::pair<Iter2, bool> z;
17      //        z                   x,y,z store iterator
18      //      /    \                from winning run;
19      //    x        y              z also whether min
20      //   / \      / \             min came from left subtree
21      //  0   1    2   3
22      if (*c[0] <= *c[1]) x = c[0]++; else x = c[1]++;
23      if (*c[2] <= *c[3]) y = c[2]++; else y = c[3]++;
24      if (*x <= *y) z = {x, true}; else z = {y, false};

26      *l++ = *(z.first); // vacate root into output
27      for (auto i = 1; i < n; ++i) {
28          if (z.second) { // min came from c[0] or c[1], so recompute x.
29              if (*c[0] <= *c[1]) x = c[0]++; else x = c[1]++;
30          } else { // min came from c[2] or c[3], so recompute y.
31              if (*c[2] <= *c[3]) y = c[2]++; else y = c[3]++;
32          }
33          // always recompute z
34          if (*x <= *y) z = {x, true}; else z = {y, false};
35          *l++ = *(z.first);
36      }
37  }
```

A.2. 3-way merge and 2-way merge.   The method for merging 3 runs is similar, but doesn't use the right subtree; instead `y` is simply also set to `c[2]`. For 2-way merging many variations have been explored; when the type supports a sentinel value, the following method has a very efficient merging loop.

```
1   void merge_2runs(Iter l, Iter m, Iter r, Iter2 B) {
2       auto n1 = m-l, n2 = r-m;
3       std::copy(l, m, B);          *(B+(m-l))   = plus_inf_sentinel();
4       std::copy(m, r, B+(m-l+1));  *(B+(r-l)+1) = plus_inf_sentinel();
5       auto c1 = B,  c2 = B + (m - l + 1),  o = l;
6       while (o < r) *o++ = *c1 <= *c2 ? *c1++ : *c2++;
7   }
```

A.3. 4-way Powersort. Our implementation of 4-way Powersort uses a stack of records maintained in a fixed-size array. As in Timsort, we extend the natural runs in the input to a minimal length `minRunLen`; setting this parameter to 1 disables that optimization.

```
1  struct run {  Iter begin; Iter end;  };
2  struct run_n_power {  Iter begin; Iter end; int power = 0;  };
3  struct run_begin_n_power{  Iter begin;  int power;  };
4  run_begin_n_power NULL_RUN_N_POWER{};
5
6  /** sorts [begin,end) */
7  void powersort_4way(Iter begin, Iter end) {
8      auto n = end - begin;
9      auto maxStackHeight = 3*(ceil_log4(n)+1);
10     auto stack = new run_n_power[maxStackHeight];
11     run_begin_n_power *top = stack; // topmost valid stack element
12     *top = NULL_RUN_N_POWER; // power 0 as sentinel entry
13     run_begin_n_power * const endOfStack = stack + maxStackHeight;
14
15     run_n_power runA = {begin, extend_and_reverse_run(begin, end), 0};
16     // extend to minimal run length
17     if (auto lenA = runA.end - runA.begin < minRunLen) {
18         runA.end = std::min(end, runA.begin + minRunLen);
19         insertionsort(runA.begin, runA.end, lenA); // skips first lenA iterations
20     }
21     while (runA.end < end) {
22         run runB = {runA.end, extend_and_reverse_run(runA.end, end)};
23         if (size_t lenB = runB.end - runB.begin < minRunLen) {
24             runB.end = std::min(end, runB.begin + minRunLen);
25             insertionsort(runB.begin, runB.end, lenB);
26         }
27         runA.power = node_power(0, n, runA.begin - begin, runB.begin - begin, runB.end - begin);
28         // Invariant: powers on stack are weakly increasing from bottom to top
29         while (top->power > runA.power)
30             merge_loop(top, runA);
31         *(++top) = {runA.begin, runA.power}; // push new run onto stack
32         runA = {runB.begin, runB.end, 0};
33     }
34     merge_down(stack, top, runA);
35     delete[] stack;
36 }
```

To make the code more readable, the body of the main loop has been moved to a function `merge_loop`; g++ inlined the call as part of its compiler optimizations. `merge_loop` counts the number of (contiguous) entries on top of the stack with equal power and then merges these with `runA`.

```
1  void merge_loop(run_begin_n_power * &top_of_stack, run_n_power &runA) {
2      int nRunsSamePower = 1;
3      while((top_of_stack - nRunsSamePower)->power == top_of_stack->power)
4          ++nRunsSamePower;
5      if (nRunsSamePower == 1) { // 2way
6          Iter g[] = {top_of_stack->begin};
7          merge_2runs(g[0], runA.begin, runA.end, _buffer.begin());
8          runA.begin = g[0];
9      } else if (nRunsSamePower == 2) { // 3way
10         Iter g[] = {(top_of_stack-1)->begin, top_of_stack->begin};
11         merge_3runs(g[0], g[1], runA.begin, runA.end, _buffer.begin());
12         runA.begin = g[0];
13     } else { // 4way
14         Iter g[] = {(top_of_stack-2)->begin, (top_of_stack-1)->begin, top_of_stack->begin};
```

```
15          merge_4runs(g[0], g[1], g[2], runA.begin, runA.end, _buffer.begin());
16          runA.begin = g[0];
17      }
18      top_of_stack -= nRunsSamePower; // pop runs
19  }
```

The function `merge_down` successively merges the top 4 elements on the stack; since runs are typically exponentially increasing in size as we work our way through the stack, it is beneficial to first bring the number of runs to $3k + 1$ for a $k \in \mathbb{N}$ using a single 2-way or 3-way merge at the beginning. Then all remaining merges are 4-way merges.

```
1   void merge_down(run_begin_n_power *begin_of_stack, run_begin_n_power * &top_of_stack, run_n_power ↩
        &runA) {
2       // we have the entire stack of runs, so instead of following exactly the powersort rule, we can
3       // be slightly more clever and make sure we have 4way merges all the way through except the ↩
             first merge
4       auto nRuns = top_of_stack - begin_of_stack + 1; // stack size + runA
5       // We want 3k+1 runs, so that repeatedly merging 4 and putting the result back gives 4way merges ↩
             all the way through.
6       switch (nRuns % 3) {
7           case 0: // merge topmost 3 runs
8               merge_3runs<mergingMethod>((top_of_stack-1)->begin, top_of_stack->begin,
9                                         runA.begin, runA.end, _buffer.begin());
10              runA.begin = (top_of_stack-1)->begin;
11              top_of_stack -= 2;
12              break;
13          case 2: // merge topmost 2 runs
14              merge_2runs(top_of_stack->begin, runA.begin, runA.end, _buffer.begin());
15              runA.begin = top_of_stack->begin;
16              --top_of_stack;
17              break;
18          default:
19              break;
20      }
21      assert(((top_of_stack - begin_of_stack) % 3) == 0);
22      // merge remaining stack 4way each
23      while (top_of_stack > begin_of_stack) {
24          merge_4runs((top_of_stack-2)->begin, (top_of_stack-1)->begin,
25                                      top_of_stack->begin, runA.begin, runA.end, _buffer.begin());
26          runA.begin = (top_of_stack-2)->begin;
27          top_of_stack -= 3;
28      }
29  }
```

Note that this merge-down strategy can lead to a slightly different merge tree compared to the "pure Powersort" merge tree realized by `kway_tree`$_k$; however, it is easy to show this change can only reduce the resulting merge cost.

A.4. Sentinel-free 2-way merge. If $+\infty$ values are not available, we need to include boundary checks. This changes the code to the following method.

```
1   void merge_2runs_no_sentinel(Iter l, Iter m, Iter r, Iter2 B) {
2       auto n1 = m-l, n2 = r-m;
3       std::copy(l,r,B);
4       auto c1 = B, e1 = B + n1, c2 = e1, e2 = e1 + n2;  auto o = l;
5       while (c1 < e1 && c2 < e2)  *o++ = *c1 <= *c2 ? *c1++ : *c2++;
6       while (c1 < e1)  *o++ = *c1++;
7       while (c2 < e2)  *o++ = *c2++;
8   }
```

For 2-way merging, we also use the following merge method that only copies the smaller run to the buffer.

```
1  void merge_2runs_copy_smaller(Iter l, Iter m, Iter r, Iter2 B) {
2      auto n1 = m-l, n2 = r-m;
3      if (n1 <= n2) {
4          std::copy(l,m,B);
5          auto c1 = B, e1 = B + n1;  auto c2 = m, e2 = r, o = l;
6          while (c1 < e1 && c2 < e2)  *o++ = *c1 <= *c2 ? *c1++ : *c2++;
7          while (c1 < e1) *o++ = *c1++;
8      } else {
9          std::copy(m,r,B);
10         auto c1 = m-1, s1 = l, o = r-1;  auto c2 = B+n2-1, s2 = B;
11         while (c1 >= s1 && c2 >= s2)  *o-- = *c1 <= *c2 ? *c2-- : *c1--;
12         while (c2 >= s2) *o-- = *c2--;
13     }
14 }
```

A.5. Sentinel-free 4-way merge (merging by stages). We created a 4-way merging method that does not use sentinels and works in "stages" (as described in the main text). We use C++ templates to exploit similarities of the stages; during compilation, these templates are unfolded and code optimization runs on the deflated code.

```
1  struct tournament_tree_node {  Iter2 it; bool fromRun0Or1;  };
2  enum number_runs {  TWO = 2, THREE = 3, FOUR = 4  };

4  template<number_runs nRuns>
5  long compute_safe(std::vector<Iter2> & c, std::vector<Iter2> & e, std::vector<long> & nn) {
6      for (int i = 0; i < nRuns; ++i) nn[i] = e[i] - c[i];
7      long safe = *(std::min_element(nn.begin(), nn.end()));
8      return safe;
9  }

11 template<number_runs nRuns>
12 void initialize_tournament_tree(std::vector<Iter2> &c, std::vector<Iter2> &e,
13                                 std::array<tournament_tree_node<Iter2>, 3> &N) {
14     // tourament tree:
15     //       N[0]
16     //     /     \
17     //   N[1]    N[2]
18     //  / \     / \
19     // 0   1   2   3
20     if (*c[0] <= *c[1]) N[1] = {c[0]++, true}; else N[1] = {c[1]++, true};
21     if (nRuns == 4)
22         if (*c[2] <= *c[3]) N[2] = {c[2]++, false}; else N[2] = {c[3]++, false};
23     else
24         N[2] = {c[2]++, false};
25     N[0] = *(N[1].it) <= *(N[2].it) ? N[1] : N[2];
26 }

28 template<number_runs nRuns>
29 void update_tournament_tree(std::vector<Iter2> &c, std::vector<Iter2> &e,
30                             std::array<tournament_tree_node<Iter2>, 3> &N) {
31     if (N[0].fromRun0Or1) {
32         if (*c[0] <= *c[1]) N[1] = {c[0]++, true}; else N[1] = {c[1]++, true};
33     } else { // otherwise min came from c[2] or c[3], so recompute y.
34         if (nRuns == 4)
35             if (*c[2] <= *c[3]) N[2] = {c[2]++, false}; else N[2] = {c[3]++, false};
36         else
37             N[2] = {c[2]++, false};
```

```
38        }
39        // always recompute z
40        N[0] = *(N[1].it) <= *(N[2].it) ? N[1] : N[2];
41  }


43  template<number_runs nRuns>
44  bool rollback_tournament_tree(std::vector<Iter2> &c, std::vector<Iter2> &e,
45                                std::array<tournament_tree_node<Iter2>, 3> &N,
46                                std::vector<long> &nn) {
47      auto other = N[0].fromRun0Or1 ? N[2] : N[1];
48      // roll back into 'its' run
49      int rollbacks = 0;
50      for (auto i = 0; i < nRuns; ++i)
51          if (c[i] - 1 == other.it) {
52              --c[i], ++nn[i], ++rollbacks; break;
53          }
54      int i = std::find(nn.begin(), nn.end(), 0) - nn.begin();
55      if (i == nRuns) {
56          // rolled back into run that got empty; nasty special case.
57          // But we made progress in the root, so just continue one more round with same nRuns.
58          // need to rebuild the tree for that
59          initialize_tournament_tree<nRuns>(c, e, N);
60          return false;
61      } else {
62          c.erase(c.begin() + i);
63          e.erase(e.begin() + i);
64          return true;
65      }
66  }


68  template<number_runs nRuns>
69  bool do_merge_runs(Iter & l, Iter const r, std::vector<Iter2> &c, std::vector<Iter2> &e) {
70      if (nRuns == TWO) {
71          // simple twoway merge
72          while (c[0] < e[0] && c[1] < e[1])   *l++ = *c[0] <= *c[1] ? *c[0]++ : *c[1]++;
73          while (c[0] < e[0]) *l++ = *c[0]++;
74          while (c[1] < e[1]) *l++ = *c[1]++;
75          return true;
76      } else {
77          // use tournament tree
78          std::array<tournament_tree_node<Iter2>, 3> N;
79          initialize_tournament_tree<nRuns>(c, e, N);
80          std::vector<long> nn(nRuns); // run sizes
81          while (l < r) {
82              long safe = compute_safe<nRuns>(c, e, nn);
83              if (safe > 0) {
84                  for (; safe > 0; --safe) {
85                      *l++ = *(N[0].it); // output root
86                      update_tournament_tree<Iter2, nRuns>(c, e, N);
87                  }
88              } else {
89                  // one run is exhausted; need to handle elements in the tree
90                  *l++ = *(N[0].it); // easy for the root (guaranteed min)
91                  // rollback other element into its run
92                  if (rollback_tournament_tree<nRuns>(c, e, N, nn))
93                      // occasionally, we rollback into an empty run and have to keep going; ↵
                                otherwise, terminate loop.
94                      break;
```

```
 95            }
 96        }
 97        return false;
 98    }
 99  }

101  void detect_and_remove_empty_runs(std::vector<Iter2> & c, std::vector<Iter2> & e) {
102      int nRuns = c.size();  long safe;
103      while (true) {
104          std::vector<long> nn(nRuns);
105          for (int i = 0; i < nRuns; ++i) nn[i] = e[i] - c[i];
106          safe = *(std::min_element(nn.begin(), nn.end()));
107          if (safe > 0) return;
108          int i = std::find(nn.begin(), nn.end(), 0) - nn.begin();
109          c.erase(c.begin() + i);  e.erase(e.begin() + i);  --nRuns;
110      }
111  }

113  template<typename Iter, typename Iter2>
114  void merge_4runs_no_sentinels(Iter l0, Iter g1, Iter g2, Iter g3, Iter r, Iter2 B) {
115      Iter l = l0;  const auto n = r - l;
116      std::copy(l,  g1, B);
117      std::copy(g1, g2, B + (g1 - l));
118      std::copy(g2, g3, B + (g2 - l));
119      std::copy(g3,  r, B + (g3 - l));
120      *(B+n) = *(B+n-1); // sentinel value so that accesses to endpoints don't fail
121      std::vector<Iter2> c {B, B+(g1-l), B+(g2-l), B+(g3-l)     }; // current element
122      std::vector<Iter2> e {   B+(g1-l), B+(g2-l), B+(g3-l), B+n}; // endpoints
123      detect_and_remove_empty_runs(c, e);
124      while (l < r) {
125          switch (c.size()) {
126              case 4:  if (do_merge_runs<FOUR> (l, r, c, e)) break;
127              case 3:  if (do_merge_runs<THREE>(l, r, c, e)) break;
128              case 2:  if (do_merge_runs<TWO>  (l, r, c, e)) break;
129              case 1:  return;
130          };
131      }
132  }
```