Engineering In-place (Shared-memory) Sorting Algorithms

MICHAEL AXTMANN, Karlsruhe Institute of Technology
SASCHA WITT, Karlsruhe Institute of Technology
DANIEL FERIZOVIC, Karlsruhe Institute of Technology
PETER SANDERS, Karlsruhe Institute of Technology

We present new sequential and parallel sorting algorithms that now represent the fastest known techniques for a wide range of input sizes, input distributions, data types, and machines. Somewhat surprisingly, part of the speed advantage is due to the additional feature of the algorithms to work in-place, i.e., they do not need a significant amount of space beyond the input array. Previously, the in-place feature often implied performance penalties. Our main algorithmic contribution is a blockwise approach to in-place data distribution that is provably cache-efficient. We also parallelize this approach taking dynamic load balancing and memory locality into account.

Our new comparison-based algorithm *In-place Superscalar Samplesort (IPS⁴o)*, combines this technique with branchless decision trees. By taking cases with many equal elements into account and by adapting the distribution degree dynamically, we obtain a highly robust algorithm that outperforms the best previous in-place parallel comparison-based sorting algorithms by almost a factor of three. That algorithm also outperforms the best comparison-based competitors regardless of whether we consider in-place or not in-place, parallel or sequential settings.

Another surprising result is that IPS⁴o even outperforms the best (in-place or not in-place) integer sorting algorithms in a wide range of situations. In many of the remaining cases (often involving near-uniform input distributions, small keys, or a sequential setting), our new *In-place Parallel Super Scalar Radix Sort (IPS²Ra)* turns out to be the best algorithm.

Claims to have the – in some sense – "best" sorting algorithm can be found in many papers which cannot all be true. Therefore, we base our conclusions on an extensive experimental study involving a large part of the cross product of 21 state-of-the-art sorting codes, 6 data types, 10 input distributions, 4 machines, 4 memory allocation strategies, and input sizes varying over 7 orders of magnitude. This confirms the claims made about the robust performance of our algorithms while revealing major performance problems in many competitors outside the concrete set of measurements reported in the associated publications. This is particularly true for integer sorting algorithms giving one reason to prefer comparison-based algorithms for robust general-purpose sorting.

Additional Key Words and Phrases: in-place algorithm, branch prediction

1 INTRODUCTION

Sorting an array of elements according to a total ordering of their keys is a fundamental subroutine used in many applications. Sorting is used for index construction, for bringing similar elements together, or for processing data in a "clever" order. Indeed, often sorting is the most expensive part of a program. Consequently, a huge amount of research on sorting has been done. In particular, algorithm engineering has studied how to make sorting practically fast in presence of complex features of modern hardware like multi-core (e.g., [9, 11, 28, 35, 37, 46, 51, 57, 61, 61, 63, 65, 66, 70, 70]) instruction parallelism (e.g., [17, 37, 61, 64]), branch prediction (e.g., [9, 21, 43, 64, 67, 76]), caches (e.g., [11, 14, 26, 46, 64]), or virtual memory (e.g., [42, 62, 71]). In contrast, the sorting algorithms used in the standard libraries of programming languages like Java or C++ still use variants of

quicksort – an algorithm that is more than 50 years old [36]. A reason seems to be that you have to outperform quicksort in every respect in order to replace it. This is less easy than it sounds since quicksort is a pretty good algorithm – a careful randomized implementation needs $O(n \log n)$ expected work independent of the input, it can be parallelized [66, 70], it can be implemented to avoid branch mispredictions [21], and it is reasonably cache-efficient. Furthermore, quicksort works in-place which is of crucial importance for very large inputs. These features rule out many contenders. Further algorithms are eliminated by the requirement to work for arbitrary data types and input distributions. This makes integer sorting algorithms like radix sort (e.g., [46]) or using specialized hardware (e.g., GPUs or SIMD instructions) less attractive since these algorithms are not sufficiently general for a reusable library that has to work for arbitrary data types. Another portability issue is that the algorithm should use no code specific to the processor architecture or the operating system like non-temporal writes or overallocation of virtual memory (e.g. [46, 71]). One aspect of making an algorithm in-place is that such "tricks" are not needed. Hence, this paper concentrates on portable algorithms with a particular focus on comparison-based algorithms and how they can be made robust for arbitrary inputs, e.g., with a large number of repeated keys or with skewed input distributions. That said, we also contribute to integer sorting algorithms and we extensively compare ourselves also to a number of non-portable and non-comparison-based sorting algorithms.

The main contribution of this paper is to propose a new algorithm – **In-place Parallel Super Scalar Samplesort** (IPS⁴o)[1] – that combines enough advantages to become an attractive replacement of quicksort. Our starting point is *Super Scalar Samplesort* (S⁴o) [64] which already provides a very good sequential non-in-place algorithm that is cache-efficient, allows considerable instruction parallelism, and avoids branch mispredictions. S⁴o is a variant of samplesort [29], which in turn is a generalization of quicksort to multiple pivots. The main operation is distributing elements of an input sequence to k output buckets of about equal size. Our two main innovations are that we make the algorithm in-place and parallel. The first phase of IPS⁴o distributes the elements to k *buffer blocks*. When a buffer block becomes full, it is emptied into a block of the input array that has already been distributed. Subsequently, the memory blocks are permuted into the globally correct order. A cleanup phase handles empty blocks and half-filled buffer blocks. The classification phase is parallelized by assigning disjoint pieces of the input array to different threads. The block permutation phase is parallelized using atomic fetch-and-add operations for each block move. Once subproblems become smaller, we adjust their parallelism until they can be solved independently in parallel. We also make IPS⁴o more robust by taking advantage of inputs with many identical keys.

It turns out that most of what is said above is not specific to samplesort. It also applies to integer sorting, specifically *most-significant-digit (MSD) radix sort* [30] where data distribution is based on extracting the most significant (distinguishing) bits of the input subproblem. We therefore also present *In-place Parallel Super Scalar Radix Sort (IPS²Ra)* – a proof-of-concept implementation of MSD radix sort using the in-place partitioning framework we developed for IPS⁴o.

After introducing basic tools in Section 2 and discussing related work in Section 3, we describe our new algorithm IPS⁴o in Section 4 and analyze our algorithm in Section 5. In Section 6 we give implementation details of IPS⁴o and IPS²Ra.

We then turn to an extensive experimental evaluation in Section 7. It turned out that there is a surprisingly large number of contenders for "the best" sorting algorithm, depending on which concrete setting is considered. We therefore consider a large part of the cross product of 21 state-of-the-art sorting codes, 6 data types, 10 input distributions, 4 machines, 4 memory allocation strategies, and input sizes varying over 7 orders of magnitude. Overall, more than 500 000 different configurations

---

[1] The Latin word "ipso" means "by itself", referring to the in-place feature of IPS⁴o.

where tried. Our algorithm IPS^{4}o can be called "winner" of this complicated comparison in the following sense: (1) It outperforms all competing implementations for most cases. (2) In many of these cases, the performance difference is large. (3) When IPS^{4}o is slower than a competitor this is only by a small percentage in the overwhelming majority of cases – the only exceptions are for easy instances that are handled very quickly by some contenders that however perform poorly for more difficult instances. Our radix sorter IPS^{2}Ra complements this by being even faster in some cases involving few processor cores and small, uniformly distributed keys. IPS^{2}Ra also outperforms other radix sorters in many cases. We believe that our approach to experimental evaluation is also a contribution independent from our new algorithms since it helps to better understand which algorithmic measures (e.g., exploiting various features of a processor architecture) have an impact under which circumstances. This approach may thus help to improve future studies of sorting algorithms.

An overall discussion and possible future work is given in Section 8. The appendix provides further experimental data and proofs. The codes and benchmarks are available at https://github.com/ips4o.

2 DEFINITIONS AND PRELIMINARIES

The input of a sorting algorithm is an array $A[0 .. n-1]$ of n elements, sorted by t threads. We expect that the output of a sorting algorithm is stored in the input array. We use the notation $[a .. b]$ as a shorthand for the ordered set $\{a, \ldots, b\}$ and $[a .. b)$ for $\{a, \ldots, b-1\}$. We also use $\log x$ for $\log_2 x$.

A machine has one or multiple *CPUs*. A CPU contains one or multiple *cores*, which in turn contain one, two, or more *hardware threads*. We denote a machine with multiple CPUs as a Non-Uniform Memory Access (*NUMA*) machine if the cores of the CPUs can access their "local main memory" faster than the memory attached to the other CPUs. We call the CPUs of NUMA machines *NUMA nodes*.

In algorithm theory, an algorithm works in-place if it uses only constant space in addition to its input. We use the term *strictly in-place* for this case. In algorithm engineering, one is sometimes satisfied if the additional space is logarithmic in the input size. In this case, we use the term *in-place*. In the context of in-place algorithms, we count machine words and equate the machine word size with the size of a data element to be sorted. Note that other space complexity measures may count the number of used bits.

The parallel external memory (*PEM*) model [1] is a cache-aware extension of the parallel random-access machine. This model is used to analyze parallel algorithms if the main issue is the number of accesses to the main memory. In the PEM model, each of the t threads has a private cache of size M and access to main memory happens in *memory blocks* of size B. The *I/O complexity* of an algorithm is the asymptotic number of parallel memory block transfers (I/Os) between the main memory and the private caches. An algorithm is denoted as *I/O-efficient* if its I/O complexity is optimal. In this work, we use the term *cache-efficient* as a synonym for I/O-efficient when we want to emphasize that we consider memory block transfers between the main memory and the private cache.

We adopt an asynchronous variant of the PEM model where we charge t I/Os if a thread accesses a variable which is shared with other threads. To make this a realistic assumption, our implementations avoid additional delays due to *false sharing* by allocating at most one shared variable to each memory block.

(Super Scalar) Samplesort. The k-way S^{4}o algorithm [64] starts with allocating two temporary arrays of size n – one data array to store the buckets, and one so-called *oracle array*. The partitioning routine contains three phases and is executed recursively. The sampling phase sorts $\alpha k - 1$ randomly sampled

input elements where the *oversampling factor* α is a tuning parameter. The splitters $S = [s_0 \, .. \, s_{k-2}]$ are then picked equidistantly from the sorted sample. The classification phase classifies each input element, stores its target bucket in a so-called *oracle array*, and increases the size of its bucket. Element e goes to bucket b_i if $s_{i-1} < e \leq s_i$ (with $s_{-1} = -\infty$ and $s_{k-1} = \infty$). Then, a prefix sum is used to calculate the bucket boundaries. The distribution phase uses the oracle array and the bucket boundaries to copy the elements from the input array into their buckets in the temporary data array.

The main contribution of S^4o to samplesort is to use a decision tree for element classification which eliminates branch mispredictions (*branchless decision tree*). Assuming k is a power of two, the splitters are stored in an array a representing a complete binary search tree: $a_1 = s_{k/2-1}$, $a_2 = s_{k/4-1}$, $a_3 = s_{3k/4-1}$, and so on. More generally, the left successor of a_i is a_{2i} and its right successor is a_{2i+1}. Thus, navigating through this tree is possible by performing a conditional instruction for incrementing an array index. S^4o completely unrolls the loop that traverses the decision tree to reduce the instruction count. Furthermore, the loop for classifying elements is unrolled several times to reduce data dependencies between instructions. This allows a higher degree of instruction parallelism.

Bingmann et al. [9] apply the branchless decision tree to parallel string sample sorting (StringPS^4o) and add additional buckets for elements identical to a splitter. After the decision tree of S^4o has assigned element e to bucket b_i, StringPS^4o updates the bucket to introduce additional equality buckets for elements corresponding to a splitter: Element e goes to bucket $b_{2i+\mathbb{1}_{e=s_i}}$ if $i < k - 1$, otherwise e goes to bucket b_{2i}.[2] The case distinction $i < k - 1$ is necessary as a_{k-1} is undefined.

For IPS^4o, we adopt (and refine) the approach of element classification but change the organization of buckets in order to make S^4o in-place and parallel. Our element classification works as follows: Beginning by the source node $i = 1$ of the decision tree, the next node is calculated by $i \leftarrow 2i + \mathbb{1}_{a_i < e}$. When the leafs of the tree are reached, we update i once more $i \leftarrow 2i + \mathbb{1}_{a_i < e} - k$. For now, we know for e that $s_{i-1} < e \leq s_i$ if we assume that $s_{-1} = -\infty$ and that $s_{k-1} = \infty$. Finally, the bucket of e is $2i + 1 - \mathbb{1}_{e<s_i}$. Note that we do not use the comparison $\mathbb{1}_{e=s_i}$, from StringPS^4o to calculate the final bucket. The reason is that our algorithm accepts a compare function $<$ and StringPS^4o compares radices. Instead, we use $1 - \mathbb{1}_{e<s_i}$ which is identical to $\mathbb{1}_{e=s_i}$ since we already know that $e \leq s_i$. Also, note that we avoid the case distinction $i < k - 1$ from the classification of StringPS^4o which may potentially cause a branch misprediction. Instead, we set $s_{k-1} = s_{k-2}$. Compared to S^4o and StringPS^4o, we support values of k which are no powers of two, i.e., when we had removed splitter duplicates in our algorithm. In these cases, we round up k to the next power of two and pad the splitter array S with the largest splitter. We note that this does not increase the depth of the decision tree. Fig. 1 depicts our refined decision tree and Algorithm 1 classifies elements using the refined decision tree. Algorithm 1 classifies a chunk of elements in one step. A single instruction of the decision tree traversal is executed on multiple elements before the next operation is executed. We use loops to execute each instruction on a constant number of elements. It turned out that recent compilers automatically unroll these loops and remove the instructions of the loops for code optimization.

3 RELATED WORK

Quicksort. Variants of Hoare's quicksort [36, 55] are generally considered some of the most efficient general-purpose sorting algorithms. Quicksort works by selecting a *pivot* element and partitioning the array such that all elements smaller than the pivot are in the left part and all elements larger than

---

[2]We use $\mathbb{1}_c$ to express a conversion of a comparison result to an integer. When c is true, $\mathbb{1}_c$ is equal to 1. Otherwise, it is equal to 0.

Sorted Splitters:

| s_0 | s_1 | s_2 | s_3 | s_4 | s_5 | s_6 |

Decision Tree a:

| $\perp$ | s_3 | s_1 | s_5 | s_0 | s_2 | s_4 | s_6 |

$a_1\ (s_3)$

$\leq$ $\quad$ $>$

$a_2\ (s_1)$ $\qquad\qquad\qquad\qquad$ $a_3\ (s_5)$

$\leq\quad>$ $\qquad\qquad\qquad\qquad$ $\leq\quad>$

$a_4\ (s_0)$ $\qquad$ $a_5\ (s_2)$ $\qquad\qquad$ $a_6\ (s_4)$ $\qquad$ $a_7\ (s_6)$

$\leq\qquad>$ $\quad$ $\leq\qquad>$ $\qquad$ $\leq\qquad>$ $\quad$ $\leq\qquad>$

| $< s_0$ | $= s_0$ | $< s_1$ | $= s_1$ | $< s_2$ | $= s_2$ | $< s_3$ | $= s_3$ | $< s_4$ | $= s_4$ | $< s_5$ | $= s_5$ | $< s_6$ | $= s_6$ | $> s_6$ | $\perp$ |

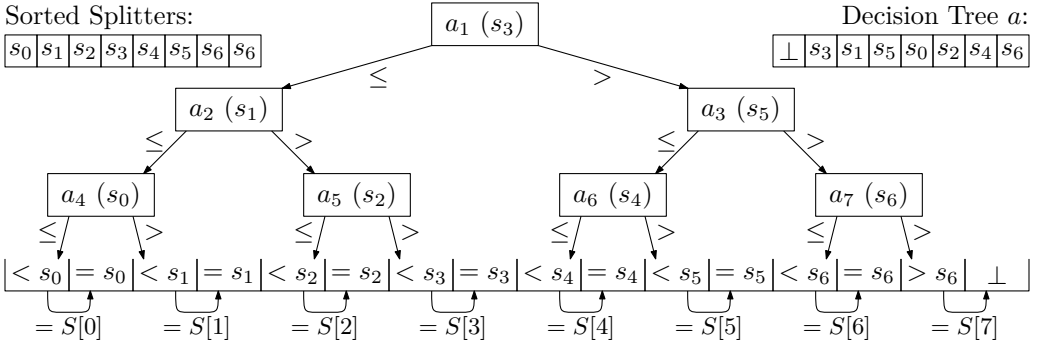$= S[0]\quad = S[1]\quad = S[2]\quad = S[3]\quad = S[4]\quad = S[5]\quad = S[6]\quad = S[7]$

Fig. 1. Branchless decision tree with 7 splitters and 15 buckets, including 7 equality buckets. The first entry of the decision tree array stores a dummy to allow tree navigation. The last splitter in the sorted splitter array is duplicated to avoid a case distinction.

Algorithm 1 Element classification of the first $u\lfloor n/u \rfloor$ elements

---

Template parameters: s number of splitters, u unroll factor, *equalBuckets* boolean value which indicates the use of equality buckets
Input: $A[0 .. n-1]$ an array of n input elements
$\qquad$ *tree*$[1 .. s]$ decision tree, splitters stored in left-to-right breadth-first order
$\qquad$ *splitter*$[0 .. s]$ sorted array of s splitters, last splitter is duplicated
$\qquad$ COMPARE(e_l, e_r) a comparator function which returns 0 or 1
$\qquad$ OUTPUT(e, t) an output function which gets an element e and its target bucket t
$l \leftarrow \log_2(s+1)$ $\qquad\qquad\qquad$ ▷ Log. number of buckets (equality buckets are excluded)
$k \leftarrow 2^{l+1}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ Number of buckets
$b[0 .. u-1]$ $\qquad\qquad\qquad\qquad\qquad$ ▷ Array to store current position in the decision tree
for $j \leftarrow 0$ **in steps of** u **to** $n-u$ **do** $\qquad\qquad\qquad$ ▷ Loop over elements in blocks of u
$\quad$ **for** $i \leftarrow 0$ **to** $u-1$ **do**
$\qquad$ $b[i] \leftarrow 1$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ▷ Set position to the tree root
$\quad$ **for** $r \leftarrow 0$ **to** l **do** $\qquad\qquad$ ▷ Unrolled by most compilers as l and u are constants
$\qquad$ **for** $i \leftarrow 0$ **to** $u-1$ **do**
$\qquad\quad$ $b[i] \leftarrow 2 \cdot b[i] +$ COMPARE$(tree[b[i]], a[j+i])$ $\qquad$ ▷ Navigate through the tree
$\quad$ **if** *equalBuckets* **then**
$\qquad$ **for** $i \leftarrow 0$ **to** $u-1$ **do** $\quad$ ▷ Assign elements identical to the splitter to its equality bucket
$\qquad\quad$ $b[i] \leftarrow 2 \cdot b[i] + 1 -$ COMPARE$(a[j+i], splitter[b[i] - k/2])$
$\quad$ **for** $i \leftarrow 0$ **to** $u-1$ **do**
$\qquad$ OUTPUT$(b[i] - k, a[j+i])$

---

the pivot are in the right part. The subproblems are solved recursively. Quicksort (with recursion on the smaller subproblem first) needs logarithmic additional space for the recursion stack. Strictly in-place variants [7, 20, 72] of quicksort avoid recursion, process the array from left to right, and use a careful placement of the pivots to find the end of the leftmost partition. A variant of quicksort (with a fallback to heapsort to avoid worst-case scenarios) is currently used in the C++ standard library of GCC [55].

Some variants of quicksort use two or three pivots [49, 76] and achieve improvements of around 20 % in running time over the single-pivot case. The basic principle of quicksort remains, but elements are partitioned into three or four subproblems instead of two.

Quicksort can be parallelized in a scalable way by parallelizing both partitioning and recursion [28, 35, 51]. Tsigas and Zhang [70] show in practice how to do this in-place. Their algorithm scans the input from left to right and from right to left until the scanning positions meet – as in most sequential implementations. The crucial adaptation is to do this in a blockwise fashion such that each thread works at one block from each scanning direction at a time. When a thread finishes a block from one scanning direction, it acquires a new one using an atomic fetch-and-add operation on a shared pointer. This process terminates when all blocks are acquired. The remaining unfinished blocks are resolved in a sequential cleanup phase. Our IPS^{4}o algorithm can be considered as a generalization of this approach to k pivots. This saves a factor $\Theta(\log k)$ of passes through the data. We also parallelize the cleanup process.

Samplesort. Samplesort [11, 12, 29] can be considered as a generalization of quicksort which uses $k - 1$ splitters to partition the input into k subproblems (from now on called *buckets*) of about equal size. Unlike single- and dual-pivot quicksort, samplesort is usually not in-place, but it is well-suited for parallelization and more cache-efficient than quicksort.

S^{4}o [64] improves samplesort by avoiding inherently hard-to-predict conditional branches linked to element comparisons. Branch mispredictions are very expensive because they disrupt the pipelined and instruction-parallel operation of modern processors. Traditional quicksort variants suffer massively from branch mispredictions [43]. By replacing conditional branches with conditionally executed machine instructions, branch mispredictions can be largely avoided. This is done automatically by modern compilers if only a few instructions depend on a condition. As a result, S^{4}o is up to two times faster than quicksort (std::sort), at the cost of $O(n)$ additional space. BlockQuicksort [21] applies similar ideas to single-pivot quicksort, resulting in a very fast in-place sorting algorithm with performance similar to S^{4}o.

For IPS^{4}o, we used a refined version of the branchless decision tree from S^{4}o. As a starting point, we took the implementation of the branchless decision tree from S^{4}oS, an implementation of S^{4}o written by Lorenz Hübschle-Schneider. S^{4}o has also been adapted for efficient parallel string sorting [9]. We apply their approach of handling identical keys to our decision tree.

Radix Sort. As for samplesort, the core of radix sort is a k-way data partitioning routine which is recursively executed. In its simplest way, all elements are classified once to determine the bucket sizes and then a second time to distribute the elements. Most partitioning routines are applicable to samplesort as well as to radix sort. Samplesort classifies an element with $\Theta(\log k)$ invocations of the comparator function while radix sort just extracts a digit of the key in constant time. In-place k-way data partitioning is often done element by element, e.g., in the sequential in-place radix sorters American Flag [53] and SkaSort [67]. However, these approaches have two drawbacks. First, they perform the element classification twice. This is a particular problem when we apply this approach to samplesort as the comparator function is more expensive. Second, a naive parallelization where the threads use the same pointers and acquire single elements suffer from read/write dependencies.

In 2014, Orestis and Ross [61] outlined a parallel in-place radix sorter that moves blocks of elements in its k-way data partitioning routine. We use the same general approach for IPS^{4}o. However, the paper [61] leaves open how the basic idea can be turned into a correct in-place algorithm. The published prototypical implementation uses 20 % additional memory, and does not work for small inputs or a number of threads different from 64.

In 2015, Minsik et al. published PARADIS [15], a parallel in-place radix sorter. The partitioning routine of PARADIS classifies the elements to get bucket boundaries and each thread gets a

subsequence of unpartitioned elements from each bucket. The threads then try to move the elements within their subsequences so that the elements are placed in the subsequence of their target bucket. This takes time $O(n/t)$. Depending on the data distribution, elements may still be in the wrong bucket. In this case, the threads repeat the procedure on the unpartitioned elements. Depending on the key distribution, the load of the threads in the partitioning routine differs significantly. No bound better than $O(n)$ is known for this partitioning routine [57].

In 2019, Shun et al. [57] proposed an in-place k-way data partitioning routine for the radix sorter RegionSort. This algorithm builds a graph that models the relationships between element regions and their target buckets. Then, the algorithm performs multiple rounds where the threads swap regions into their buckets.

To the best of our knowledge, the initial version of IPS⁴o [6], published in 2017, is the first parallel k-way partitioning algorithm that moves elements in blocks, works fully in-place, and gives adequate performance guarantees. Our algorithm IPS⁴o is more general than RegionSort in the sense that it is comparison based. To demonstrate the advantages of our approach, we also propose the radix sorter IPS²Ra which adapts our in-place partitioning routine.

(Strictly) In-Place Mergesort. There is a considerable amount of theory work on strictly in-place sorting (e.g., [26, 27, 34]). However, there are few – mostly negative – results of transferring the theory work into practice. Implementations of non-stable in-place mergesort [22, 23, 44] are reported to be slower than quicksort from the C++ standard library. Katajainen and Teuhola report that their implementation [44] is even slower than heapsort, which is quite slow for big inputs due to its cache-inefficiency. The fastest non-stable in-place mergesort implementation we have found is QuickMergesort (QMSort) from Edelkamp et al. [22]. Relevant implementations of stable in-place mergesort are WikiSort (derived from [45]) and GrailSort (derived from [38]). However, Edelkamp et al. [22] report that WikiSort is a factor of more than 1.5 slower than QMSort for large inputs and that GrailSort performs similar to WikiSort. Edelkamp et al. also state that non-in-place mergesort is considerably faster than in-place mergesort. There is previous theoretical work on sequential (strictly) in-place multi-way merging [31]. However, this approach needs to allocate very large blocks to become efficient. In contrast, the block size of IPS⁴o does not depend on the input size. The best practical multi-core mergesort algorithm we found is the non-in-place multi-way mergesort algorithm (MCSTLmwm) from the MCSTL library [66]. We did not find any practical parallel in-place mergesort implementation.

4 IN-PLACE PARALLEL SUPER SCALAR SAMPLESORT (IPS⁴O)

IPS⁴o is a recursive algorithm. Each recursion level divides the input into k buckets (*partitioning step*), such that each element of bucket b_i is smaller than all elements of b_{i+1}. Partitioning steps operate on the input array in-place and are executed with one or more threads, depending on their size. If a bucket is smaller than a certain base case size, we invoke a base case algorithm on the bucket (*base case*) to sort small inputs fast. A scheduling algorithm determines at which time a base case or partitioning step is executed and which threads are involved. We describe the partitioning steps in Section 4.1 and the scheduling algorithm in Section 4.2.

4.1 Sequential and Parallel Partitioning

A partitioning step consists of four phases, executed sequentially or by a (sub)set of the input threads. **Sampling** determines the bucket boundaries. **Classification** groups the input into blocks such that all elements in a block belong to the same bucket. **(Block) permutation** brings the blocks into the globally correct order. Finally, we clean up blocks that cross bucket boundaries or remained
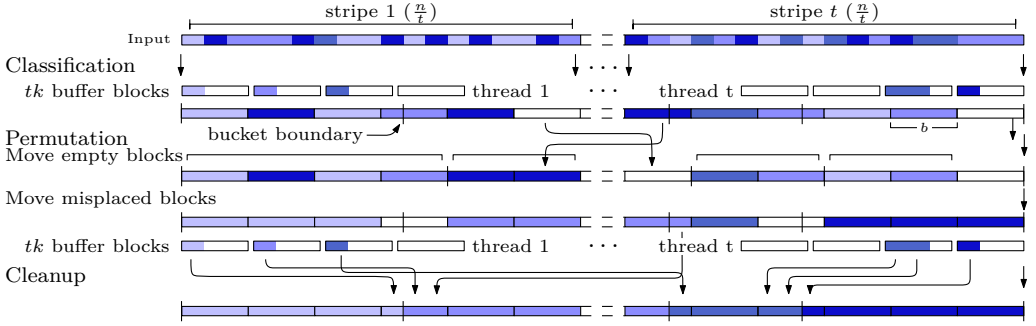
Fig. 2. Overview of a parallel k-way partitioning step ($k = 4$) with t threads and blocks of size three. Elements with the same color belong into the same bucket. The brighter the color, the smaller the bucket index. This figure depicts the first and last stripe of the input array, containing n/t elements each. In the classification phase, thread i classifies the elements of stripe i, moves elements of bucket j into its buffer block j, and flushes the buffer block back into its stripe in case of an overflow. In the permutation phase, the bucket boundaries are calculated and the blocks belonging into bucket j are placed in the blocks after bucket boundary j in two steps: First, the empty blocks are moved to the end of the bucket. Then, the misplaced blocks are moved into its bucket. The cleanup phase moves elements which remained in the buffer blocks and elements which overlap into the next bucket to their final positions.

partially filled in the **cleanup** phase. Figure 2 depicts an overview of a parallel partitioning step. The following paragraphs will explain each of these phases in more detail.

4.1.1 Sampling. Similar to the sampling in S^{4}o, the sampling phase of IPS^{4}o creates a branchless decision tree – the tree follows the description of the decision tree proposed by Sanders and Winkel, extended by equality buckets[3]. For a description of the decision tree used in S^{4}o including our refinements, we refer to Section 2. In IPS^{4}o, the decision tree is used in the classification phase to assign elements to buckets.

The sampling phase performs four steps. First, we sample $k\alpha$ elements of the input. We swap the samples to the front of the partitioning step to keep the in-place property even if the oversampling factor α depends on n. Second, $k - 1$ splitters are picked equidistantly from the sorted sample. Third, we check for and remove duplicates from the splitters. This allows us to decrease the number of buckets k if the input contains many duplicates. Finally, we create the decision tree. The strategy for handling identical keys is enabled conditionally: The decision tree only creates equality buckets when there are several identical splitters. Otherwise, we create a decision tree without equality buckets. Having inputs with many identical keys can be a problem for samplesort, since this might move large fractions of the keys through many recursion levels. The equality buckets turn inputs with many identical keys into "easy" instances as they introduce separate buckets for elements identical to splitters (keys occurring more than n/k times are likely to become splitters).

4.1.2 Classification. The input array A is viewed as an array of blocks each containing b elements (except possibly for the last one). For parallel processing, we divide the blocks of A into t stripes of equal size – one for each thread. Each thread works with a local array of k *buffer blocks* – one for each bucket. A thread then scans its stripe. Using the search tree created in the sampling phase, each element in the stripe is classified into one of the k buckets and then moved into the corresponding

---

[3]The authors describe a similar technique for handling duplicates, but have not implemented the approach for their experiments.
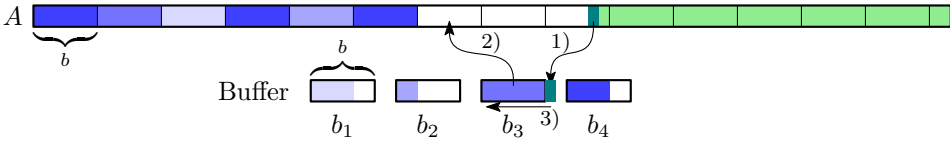
Fig. 3. Classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. Here, the next element (in dark green) has been determined to belong to bucket b_3. As that buffer block is already full, we first write it into the array A, then write the new element into the now empty buffer.
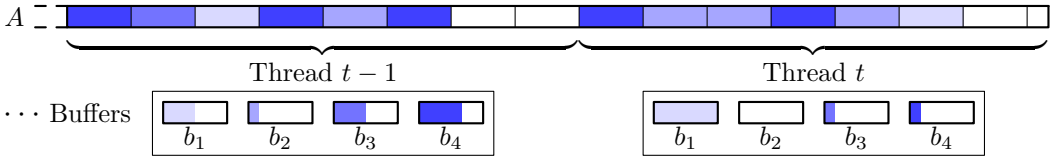

Fig. 4. Input array and block buffers of the last two threads after classification.

local buffer block. If this buffer block is already full, it is first written back into the local stripe, starting at the front. It is clear that there is enough space to write b elements into the local stripe, since at least b more elements have been scanned from the stripe than have been written back – otherwise, no full buffer could exist.

In this way, each thread creates blocks of b elements belonging to the same bucket. Figure 3 shows a typical situation during this phase. To achieve the in-place property, we do not track which bucket each block belongs to. However, we count how many elements are classified into each bucket, since we need this information in the following phases. This information can be obtained almost for free as a side effect of maintaining the buffer blocks. Figure 4 depicts the input array after classification. Each stripe contains full blocks, followed by empty blocks. The remaining elements are still contained in the buffer blocks.

4.1.3 Block Permutation. In this phase, the blocks in the input array are rearranged such that they appear in the correct order. From the classification phase we know, for each stripe, how many elements belong to each bucket. We first aggregate the per-thread bucket sizes and then compute a prefix sum over the total bucket sizes. This yields the exact boundaries of the buckets in the output. Roughly, the idea is then that each thread repeatedly looks for a misplaced block B in some bucket b_i, finds the correct destination bucket b_j for B, and swaps B with a misplaced block in b_j. If b_j does not contain a misplaced block, B is moved to an empty block in b_j. The threads are coordinated by maintaining atomic read and write pointers for each bucket. Costs for updating these pointers are amortized by making blocks sufficiently large.

We now describe this process in more detail beginning with the preparations needed before starting the actual block permutation. We mark the beginning of each bucket b_i with a delimiter pointer d_i, rounded up to the next block. We similarly mark the end of the last bucket b_k with a delimiter pointer d_{k+1}. Adjusting the boundaries may cause a bucket to "lose" up to $b - 1$ elements; this doesn't affect us, since this phase only deals with full blocks, and elements outside full blocks remain in the buffers. Additionally, if the input size is not a multiple of b, some of the d_is may end up outside the bounds of A. To avoid overflows, we allocate a single empty *overflow block* which the algorithm will use instead of writing to the final (partial) block.
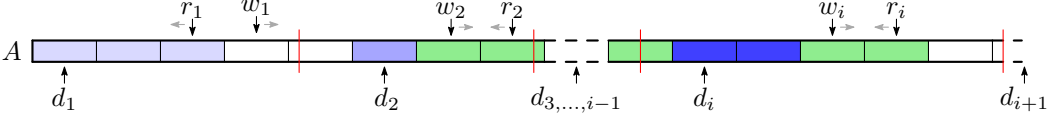
Fig. 5.   Invariant during block permutation. In each bucket b_i, blocks in $[d_i, w_i)$ are already correct (blue), blocks in $[w_i, r_i]$ are unprocessed (green), and blocks in $[\max(w_i, r_i + 1), d_{i+1})$ are empty (white).
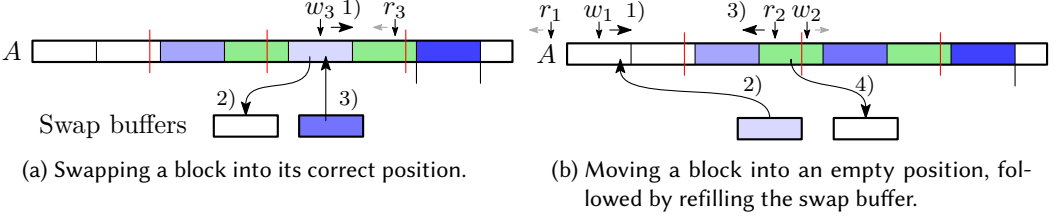

(a) Swapping a block into its correct position.

(b) Moving a block into an empty position, followed by refilling the swap buffer.

Fig. 6.   Block permutation examples. The numbers indicate the order of the operations.

For each b_i, a write pointer w_i and a read pointer r_i are introduced; these will be set such that all unprocessed blocks, i.e., blocks that still need to be moved into the correct bucket, are found between w_i and r_i. During the block permutation, we maintain the following invariant for each bucket b_i, visualized in Fig. 5:

- Blocks to the left of w_i (exclusive) are correctly placed, i.e., contain only elements belonging to b_i.
- Blocks between w_i and r_i (inclusive) are unprocessed, i.e., may need to be moved.
- Blocks to the right of $\max(w_i, r_i + 1)$ (inclusive) are empty.

In other words, each bucket follows the pattern of correct blocks followed by unprocessed blocks followed by empty blocks, with w_i and r_i determining the boundaries. In the sequential case, this invariant is already fulfilled from the beginning. In the parallel case, all full blocks are at the beginning of each stripe, followed by its empty blocks. This means that only the buckets crossing a stripe boundary need to be fixed.

To do so, each thread finds the bucket that starts before the end of its stripe but ends after it. It then finds the stripe in which that bucket ends (which will be the following stripe in most cases) and moves the last full block in the bucket into the first empty block in the bucket. It continues to do this until either all empty blocks in its stripe are filled or all full blocks in the bucket have been moved.

In rare cases, very large buckets exist that cross multiple stripes. In this case, each thread will first count how many blocks in the preceding stripes need to be filled. It will then skip that many blocks at the end of the bucket before starting to fill its own empty blocks.

The threads are then ready to start the block permutation. Each thread maintains two local swap buffers that can hold one block each. We define a *primary* bucket b_p for each thread; whenever both its buffers are empty, a thread tries to read an unprocessed block from its primary bucket. To do so, it decrements the read pointer r_p (atomically) and reads the block it pointed to into one of its swap buffers. If b_p contains no more unprocessed blocks (i.e., $r_p < w_p$), it switches its primary bucket to the next bucket (cyclically). If it completes a whole cycle and arrives back at its initial primary bucket, there are no more unprocessed blocks and the whole permutation phase ends. The starting points for the threads are distributed across that cycle to reduce contention.

Fig. 7. An example of a permutation phase with $k = 4$ buckets and $t = 3$ threads. The brackets above the buckets mark the unprocessed blocks. After five permutation steps, the blocks were moved into their target buckets. (1) The buffer blocks are filled with blocks. (2-3) Swap buffer block with the leftmost unprocessed block of the buffer block's buckets. (4) Thread 0 and 1 have a buffer block for the last bucket. They increase the write pointer of this bucket concurrently. Thread 0 executes the fetch-and-add operation first, the thread swaps its buffer block with the second block (unprocessed block) of the last bucket. Thread 1 writes its buffer block into the third block (empty block) of the last bucket. After step four, threads 1 and 2 finished a permutation chain, i.e., flushed their buffer block into an empty block. (5) Thread 0 flushes its buffer block into an empty block. Thread 1 classifies the last unprocessed block of the first bucket but this block is already in its target bucket.

Once it has a block, each thread classifies the first element of that block to find its destination bucket b_{dest}. There are now two possible cases, visualized in Fig. 6:

- As long as $w_{\text{dest}} \leq r_{\text{dest}}$, write pointer w_{dest} still points to an unprocessed block in bucket b_{dest}. In this case, the thread increases w_{dest}, reads the unprocessed block into its empty swap buffer, and writes the other one into its place.
- If $w_{\text{dest}} > r_{\text{dest}}$, no unprocessed block remains in bucket b_{dest} but w_{dest} now points to an empty block. In this case, the thread increases w_{dest}, writes its swap buffer to the empty block, and then reads a new unprocessed block from its primary bucket.

We repeat these steps until all blocks are processed. We can skip unprocessed blocks which are already correctly placed: We simply classify blocks *before* reading them into a swap buffer, and skip as needed.

It is possible that one thread wants to write to a block that another thread is currently reading from (when the reading thread has just decremented the read pointer but has not yet finished reading the block into its swap buffer). However, threads are only allowed to write to empty blocks if no other threads are currently reading from the bucket in question, otherwise, they must wait. Note that this situation occurs at most once for each bucket, namely when w_{dest} and r_{dest} cross

Fig. 8.   An example of the steps performed during cleanup.

each other. We avoid these data races by keeping track of how many threads are reading from each bucket.

When a thread fetches a new unprocessed block, it reads and modifies either w_i or r_i. The thread also needs to read the other pointer for the case distinctions. These operations are performed simultaneously to ensure a consistent view of both pointers for all threads. Figure 7 depicts an example of the permutation phase with three threads and four buckets.

4.1.4   Cleanup. After the block permutation, some elements may still be in incorrect positions since blocks may cross bucket boundaries. We call the partial block at the beginning of a bucket its *head* and the partial block at its end its *tail*.

Thread i performs the cleanup for buckets $[\lfloor ki/t \rfloor .. \lfloor k(i+1)/t \rfloor)$. Thread i first reads the head of the first bucket of thread $i+1$ into one of its swap buffers. Then, each thread processes its buckets from left to right, moving incorrectly placed elements into empty array entries The incorrectly placed elements of bucket b_i can be in four locations:

(1) Elements may be in the head of b_{i+1} if the last block belonging into bucket b_i overlaps into bucket b_{i+1}.
(2) Elements may be in the partially filled buffers from the classification phase.
(3) Elements of the last bucket (in this thread's area) may be in the swap buffer.
(4) Elements of one bucket may be in the overflow buffer.

Empty array entries consist of the head of b_i and any (empty) blocks to the right of w_i (inclusive). Although the concept is relatively straightforward, the implementation is somewhat involved, due to the many parts that have to be brought together. Figure 8 shows an example of the steps performed during the cleanup phase. Afterwards, all elements are back in the input array and correctly partitioned, ready for recursion.

4.2   Task Scheduling

In this section, we describe the task scheduling of IPS^{4}o. We also establish basic properties of the task scheduler. The properties are used to understand how the task scheduler works. The properties are also used later on in Section 5 to analyze the parallel I/O complexity and the local work of IPS^{4}o.

In general, IPS^{4}o uses static scheduling to apply tasks to threads. When a thread becomes idle, we additionally perform a dynamic rescheduling of sequential tasks to utilize the computation resources of the idle thread. Unless stated otherwise, we exclude the dynamic rescheduling from the analysis of IPS^{4}o and only consider the static load balancing. We state that dynamic load balancing – when implemented correctly – cannot make things worse asymptotically.

Before we describe the scheduling algorithm, we introduce some definitions and derive some properties from these definitions to understand how the task scheduler works. A task $T[l, r)$ either partitions the (sub)array $A[l, r-1]$ with a partitioning step (*partitioning task*) or sorts the base case

$A[l, r − 1]$ with a base case sorting algorithm (*base case task*). A partitioning step performed by a group of threads (a *thread group*) is a *parallel partitioning task* and a partitioning step with one thread is a *sequential partitioning task*. Each thread has a *local stack* to store *sequential tasks*, i.e., sequential partitioning tasks and base cases. Additionally, each thread i stores a handle G_i to its current thread group and has access to the handles stored by the other threads of the thread group.

To decide whether a task $T[l, r]$ is a parallel partitioning task or a sequential task, we denote $\underline{t}$ as $\lfloor lt/n \rfloor$ and $\bar{t}$ as $\lfloor rt/n \rfloor$. The task $T[l, r]$ is a parallel partitioning task when $\bar{t} − \underline{t} > 1$. In this case, the task is executed by the thread group $[\underline{t} .. \bar{t})$. Otherwise, the task is a sequential task processed by thread $\min(\underline{t}, \bar{t} − 1)$. As a parallel partitioning task is the only task type executed in parallel, we use *parallel task* as a synonym for parallel partitioning task.

We use a base case threshold n_0 to determine whether a sequential task is a sequential partitioning task or a base case task: Buckets with at most $2n_0$ elements as well as buckets of a task with at most kn_0 elements are considered as base cases. Otherwise, it is a sequential partitioning task. We use an adaptive number of buckets k for partitioning steps with less than $k^2 n_0$ elements, such that the expected size of the base cases is between $0.5n_0$ and n_0 while the expected number of buckets remains equal or larger than $\sqrt{k}$. To this end, for partitioning steps with $kn_0 < n' < k^2 n_0$ elements ($n_0 < n' \leq kn_0$ elements), we adjust k to $2^{\lceil (\log_2(n'/n_0)+1)/2 \rceil}$ (to $2^{\lceil \log_2(n'/n_0) \rceil}$). This adaption of k is important for a robust running time of IPS⁴o. In the analysis of IPS⁴o, the adaption of k will allow us to amortize the sorting of samples. In practice, our experiments have shown that for fixed k, the running time per element oscillates with maxima around $k^i n_0$.

From these definitions, the Lemmas 4.1 to 4.4 follow. The lemmas allow a simple scheduling of parallel tasks and thread groups.

LEMMA 4.1. *The parallel task $T[l, r]$ covers position $(i + 1)n/t − 1, i \in [0 .. t)$ of the input array if and only if thread i executes the parallel task $T[l, r]$.*

PROOF. We first prove that thread i executes the parallel task $T[l, r]$ if $T[l, r]$ covers position $(i + 1)n/t − 1, i \in [0 .. t)$ of the input array. Let the parallel task $T[l, r]$ cover position $w = (i + 1)n/t − 1, i \in [0 .. t)$ of the input array. From the inequalities

$$i = \lfloor ((i + 1)n/t − 1)t/n \rfloor \geq \lfloor l_s t/n \rfloor = \underline{t}_s$$
$$i = \lfloor ((i + 1)n/t − 1)t/n \rfloor < \lfloor r_s t/n \rfloor = \bar{t}_s$$

follows that thread i executes the parallel task $T[l, r]$. For the "$\geq$" and the "$<$", we use that task $T[l, r]$ covers position w of the input array, i.e., $l \leq w < r$.

We now prove that a parallel task $T[l, r]$ must cover position $(i + 1)n/t − 1$ of the input array if it is executed by thread i. Let us assume that a parallel task $T[l, r]$ is executed by thread i. From the inequalities

$$(i + 1)n/t − 1 \geq (\underline{t} + 1)n/t − 1 \geq l_s$$
$$(i + 1)n/t − 1 < \bar{t}n/t \leq r_s$$

follows that task $T[l, r]$ covers position $(i + 1)n/t − 1$ of the input array. For the second "$\geq$" and for the "$\leq$", we use the definition for the thread group of $T[l, r]$, i.e., $[\underline{t} .. \bar{t}) = [\lfloor lt/n \rfloor .. \lfloor rt/n \rfloor)$. □

LEMMA 4.2. *Let the sequential task $T[l_s, r_s)$, processed by thread i be a bucket of a parallel task $T[l, r]$. Then, task $T[l, r]$ is processed by thread i and others.*

PROOF. Let the parallel task $T[l, r]$ be processed by threads $[\underline{t} .. \bar{t})$. Task $T[l_s, r_s)$ is processed by thread $i = \min(\lfloor l_s t/n \rfloor, \bar{t} − 1)$. We have to show that $\underline{t} \leq i < \bar{t}$ holds. Indeed, inequality $i = \min(\lfloor l_s t/n \rfloor, \bar{t}−1) < \bar{t}$ holds. The inequality $i \geq \underline{t}$ is only wrong if $\lfloor l_s t/n \rfloor < \underline{t}$ or if $\bar{t}−1 < \underline{t}$. However,

Fig. 9. Example schedule of a task execution in IPS^{4}o with 8 threads where partitioning steps split tasks into 8 buckets. Each rectangle represents a task in execution. The height of a task is defined by the size of the task divided by the number of threads assigned to this task. For parallel tasks (green), the threads processing that task are shown in the rectangles. The sequential partitioning tasks (blue) are covered by the local stack which stores the task until processing. Base case tasks are omitted for the sake of simplicity. The crosses at the bottom of a rectangle indicate bucket boundaries. The brackets pointing downwards are used to decide in which local stack the sequential subtasks are inserted. Tasks stored in local stack i are executed by thread i.

we have $l \leq l_s$ as task $T[l_s, r_s)$ is a bucket of its parent task $T[l, r)$. Thus, $\lfloor l_s t/n \rfloor \geq \lfloor lt/n \rfloor = \underline{t}$ holds as the first thread $\underline{t}$ of $T[l, r)$ is defined as $\lfloor lt/n \rfloor$. Also, we have $\bar{t} - 1 > \underline{t}$ as task $T[l, r)$ is a parallel task with at least two threads, i.e., $\bar{t} - \underline{t} > 1$.   □

LEMMA 4.3. *Let the parallel subtask $T[l_s, r_s)$, processed by thread i and others, be a bucket of a task $T[l, r)$. Then, task $T[l, r)$ is also a parallel task processed by thread i and others.*

PROOF. Task $T[l, r)$ is a parallel task if $\lfloor rt/n \rfloor - \lfloor lt/n \rfloor > 1$. This inequality is true as

$$\lfloor rt/n \rfloor - \lfloor lt/n \rfloor \geq \lfloor r_s t/n \rfloor - \lfloor l_s t/n \rfloor > 1 \ .$$

For the "$\geq$" we use that $T[l_l, r_l)$ is a bucket of $T[l, r)$ and for the "$>$" we use that $T[l_s, r_s)$ is a parallel task.

As the parallel task $T[l_s, r_s)$ is processed by thread i, $T[l_s, r_s)$ covers the position $(i + 1)n/t - 1$ of the input array (see Lemma 4.1). As task $T[l_s, r_s)$ is a bucket of $T[l, r)$, the parallel task $T[l, r)$ also covers the position $(i + 1)n/t - 1$. From Lemma 4.1 follows that the parallel task $T[l, r)$ is processed by thread i.   □

LEMMA 4.4. *On each recursion level, thread i works on at most one parallel task.*

PROOF. Let $S_i^j, i \in [0..t)$ be the set of parallel tasks on recursion level j which cover the position $(i + 1)n/t - 1$ of the input array. From Lemma 4.1 follows that thread i processes on recursion level j only the tasks S_i^j. The set S_i^j contains at most one task as tasks on the same recursion level are disjoint.   □

4.2.1 Static Scheduling. We start the description of the task scheduler by describing the static scheduling part. The idea behind the static scheduling is that each thread executes its tasks in depth-first search order tracing parallel tasks first. From Lemmas 4.3 and 4.4 follows, keeping the order of execution in mind, that each thread first executes all of its parallel tasks before it starts to execute its sequential tasks.

IPS^{4}o starts by processing a parallel task $T[0, n)$ with threads $[0 .. t)$. In general, when a parallel task $T[l, r)$ is processed by the thread group $G = [\underline{t} .. \overline{t})$, five steps are performed.

(1) A parallel partitioning step is invoked on $A[l, r - 1]$.
(2) The buckets of the partitioning step induce a set of subtasks S.
(3) If subtask $T[l_s, r_s) \in S$ is a sequential task, thread $i = \min(l_s t/n, \overline{t} - 1)$ adds the subtask to its local stack. From Lemma 4.2, we know that thread i is actually also processing the current task $T[l, r)$. This allows threads to add sequential tasks exclusively to their own local stack, so no concurrent stacks are required.
(4) Each thread $i \in [\underline{t} .. \overline{t})$ extracts the subtask $T_s = T[l_s, r_s)$ from S which covers position $(i + 1)n/t - 1$ of the input array A. Also, thread i calculates $\underline{t}_s = \lfloor l_s t/n \rfloor$ as well as $\overline{t}_s = \lfloor r_s t/n \rfloor$ and continues with the case distinction $\overline{t}_s - \underline{t}_s \leq 1$ and $\overline{t}_s - \underline{t}_s > 1$.

   If $\overline{t}_s - \underline{t}_s \leq 1$, thread i once synchronizes with G and starts processing the sequential tasks on its private stack.

   Otherwise, T_s is actually a parallel task that has to be processed by the threads $[\underline{t}_s .. \overline{t}_s)$. From Lemmas 4.1 and 4.3 follows that the threads $[\underline{t}_s .. \overline{t}_s)$ are currently all processing $T[l, s)$ and exactly these threads selected the same task T_s. This allows setting up the threads $[\underline{t}_s .. \overline{t}_s)$ for the next parallel task T_s without keeping the threads waiting: The first thread $\underline{t}_s$ of task T_s creates the data structure representing the task's new thread group $G' = [\underline{t}_s .. \overline{t}_s)$ and updates the thread group handles $[G_{\underline{t}_s} .. G_{\overline{t}_s})$ of the threads $[\underline{t}_s .. \overline{t}_s)$ to the new data structure. Afterwards, all threads of $[\underline{t}_s .. \overline{t}_s)$ synchronize with thread group G and access their new thread group G' using the updated thread group handles. Finally, the threads $[\underline{t}_s .. \overline{t}_s)$ start processing task T_s with thread group G'.

If a thread no longer processes another parallel task, it starts processing the sequential tasks of its stack until the stack is empty. Base cases are sorted right away. When the next task $T[l, r)$ is a sequential partitioning task, three steps are performed. First, a sequential partitioning step is executed on $A[l, r - 1]$. Second, a new sequential subtask is created for each bucket. Finally, the thread adds these subtasks to its local stack in sorted order. Algorithm 2 shows the steps of the task scheduling algorithm in detail. The scheduling algorithm is executed by all threads simultaneously. Figure 9 shows an example schedule of a task execution in IPS^{4}o.

From Lemmas 4.5 and 4.6 follows that the workload of sequential tasks and parallel tasks is evenly divided between the threads. This property is used in Section 5 to analyze the parallel I/O complexity and the local work.

LEMMA 4.5. *Let $T[l, r)$ be a parallel task with thread group $[\underline{t} .. \overline{t})$ and $t' = \overline{t} - \underline{t}$ threads. Then, $T[l, r)$ processes a consecutive sequence of elements which starts at position $l \in [\underline{t}n/t .. (\underline{t} + 1)n/t - 1]$ and which ends at position $r \in [\overline{t}n/t - 1 .. (\overline{t} + 1)n/t - 1]$ of the input array. This sums up to $\Theta(t'n/t)$ elements in total.*

Thus, the size of a parallel task is proportional to the size of its thread group.

PROOF OF LEMMA 4.5. From Lemma 4.1 follows that $T[l, r)$ covers position $(\underline{t} + 1)n/t - 1$ but not position $\underline{t}n/t - 1$ of the input array. It also follows, that $T[l, r)$ covers position $\overline{t}n/t - 1$ but not position $(\overline{t} + 1)n/t - 1$ of the input array. □

Algorithm 2 Task Scheduler

Input: $A[0 \mathbin{..} n-1]$ array of n input elements, t number of threads, i current thread
$T[l,r] \leftarrow T[0,n]$ ▷ Current task, initialized with $A[1 \mathbin{..} n]$
$G_i[\underline{t}, \overline{t}] = G[0,t]$ ▷ Initialize thread group containing thread $\underline{t} = 0$ to $\overline{t} = t$ (excl.)
$D \leftarrow \emptyset$ ▷ Empty local stack
if $\overline{t} - \underline{t} = 1$ **then** D.PUSHFRONT($T[l,r]$)   ▷ Initial task is a sequential, go to sequential phase
else
    while true do   ▷ Execute current parallel task
        $[b_0 \mathbin{..} b_{k-1}] \leftarrow$ PARTITIONPARALLEL($A[l, r-1], G_i$)   ▷ Partitioning step; returns buckets
        for $[l_s \mathbin{..} r_s) := b_{k-1}$ **to** b_0 **do**   ▷ Handle the buckets
            if $(i+1)n/t - 1 \in [l_s, r_s)$ **then**   ▷ Update current task
                $T[l,r] \leftarrow T[l_s, r_s)$ ▷ It might be i's next parallel task
            if $\lceil r_s t/n \rceil - \lceil l_s t/n \rceil \leq 1$ **and** $i = \min(l_s t/n, \overline{t} - 1)$ **then**
                D.PUSH($\{T[l_s, r_s), l, r\}$)   ▷ Thread i adds sequential task to its local stack
        $\underline{t} \leftarrow l \cdot t/n; \overline{t} \leftarrow r \cdot t/n$ ▷ Range of threads used by current task
        if $\overline{t} - \underline{t} \leq 1$ **then break**   ▷ Go to sequential phase as $T[l,r]$ is not a parallel task
        if $i = \underline{t}$ **then**   ▷ Thread i creates the thread subgroup as it is the first thread
            $G_i \leftarrow$ CREATETHREADGROUP($[\underline{t} \mathbin{..} \overline{t}]$)
            for $j := \underline{t}$ **to** $\overline{t} - 1$ **do**   ▷ Set subgroup for all subgroup threads
                $G_j \leftarrow$ REFERENCEOF(G_i)
        WAITFOR($\underline{t}$)   ▷ Wait until thread subgroup is created
        JOINTHREADGROUP(G_i)   ▷ Join shared data structures
while NOTEMPTY(D) **do**   ▷ Execute sequential tasks
    $\{T[l_s, r_s), l, r\} \leftarrow$ POP(D)
    if $r_s - l_s \leq 2n_0$ **or** $r - l \leq kn_0$ **then**
        PROCESSBASECASE($A[l_s, r_s - 1]$)
    else
        $[b_0 \mathbin{..} b_{k-1}] \leftarrow$ PARTITIONSEQUENTIAL($A[l_s, r_s - 1]$) ▷ Partitioning step – returns buckets
        for $b := b_{k-1}$ **to** b_0 **do**
            D.PUSH($\{T[\text{BEGIN(b)}, \text{END(b)}), l_s, r_s\}$)   ▷ Add seq. subtasks

---

LEMMA 4.6. *Thread i processes sequential tasks only containing elements from $A[in/t, (i+2)n/t-1]$. This sums up to $O(n/t)$ elements in total.*

This lemma shows that the load of sequential tasks is evenly distributed among the threads.

PROOF OF LEMMA 4.6. We prove the following proposition: When a thread i starts processing sequential tasks, the tasks only contain elements from $A[in/t, (i+2)n/t - 1]$. From this proposition, Lemma 4.6 follows directly as thread i only processes sequential subtasks of these tasks.

Let the sequential subtask $T[l_s, r_s)$ be a bucket of a parallel task $T[l, r]$ with threads $[\underline{t} \mathbin{..} \overline{t}]$. Assume that $T[l_s, r_s)$ was assigned to the stack of thread i. We show $in/t \leq l_s < r_s \leq (i+2)n/t$ with the case distinction $i < \overline{t} - 1$ and $i \geq \overline{t} - 1$.

Assume $i < \overline{t} - 1$. From the calculation of i, we know that

$$i = \min(\lfloor l_s t/n \rfloor, \overline{t} - 1) = \lfloor l_s t/n \rfloor \leq l_s t/n$$
$$\implies l_s \geq in/t \ .$$

We show that $r_s \leq (i+2)n/t$ with a proof by contradiction. For the proof, we need the inequality $l_s < (i+1)n/t$ which is true as

$$i = \min(\lfloor l_s t/n \rfloor, \bar{t} - 1) = \lfloor l_s t/n \rfloor > lt/n - 1 \ .$$

Now, we assume that $r_s > (i+2)n/t$. As $T[l_s, r_s)$ is a sequential task, we have $\lfloor r_s t/n \rfloor - \lfloor l_s t/n \rfloor = 1$. However, this leads to the contradiction

$$1 = \lfloor r_s t/n \rfloor - \lfloor l_s t/n \rfloor \geq i + 2 - l_s t/n > (i+2) - (i+1) = 1 \ .$$

Thus, we limited the end of the sequential task to $r_s \leq (i+2)n/t$ and its start to $l_s \geq in/t$ for $i < \bar{t} - 1$.

Assume $i \geq \bar{t} - 1$. In this case, i is essentially equal to $\bar{t} - 1$ as Lemma 4.2 tells us that a sequential subtask of a parallel task is assigned to a thread of the parallel task. From the calculation of thread i, we know that

$$i = \min(\lfloor l_s t/n \rfloor, \bar{t} - 1) = \bar{t} - 1 \leq lt/n$$
$$\implies l_s \geq n/t(\bar{t} - 1) \ .$$

The end r of the parent task can be bounded by

$$\bar{t} = \lfloor rt/n \rfloor \geq rt/n - 1$$
$$\implies r \leq (\bar{t} + 1)n/t$$

We can use this inequality to bound the end r_s of the sequential subtask $T[l_s, r_s)$ to

$$r_s \leq r \leq (\bar{t} + 1)n/t$$

as the subtask does not end after the parent task's end r. Thus, we limited the end of the sequential task to $r_s \leq (i+2)n/t$ and its start to $l_s \geq in/t$ for $i \geq \bar{t} - 1$. □

4.2.2 Dynamic Rescheduling. The task scheduler is extended to utilize computing resources of threads that no longer have tasks. We implement a simplified version of *voluntary work sharing* proposed for parallel string sorting [9]. A global stack is used to transfer sequential tasks to idle threads. Threads without sequential tasks increase a global atomic counter which tracks the number of idle threads. Threads with sequential tasks check the counter after each partitioning step and move one task to the global stack if the counter is larger than zero. Then, an idle thread can consume the task from the global stack by decreasing the counter and processing the task. The algorithm terminates when the counter is equal to t which implies that no thread has a task left. We expect that we are able to amortize the additional cost in most cases or even reduce the work on the critical execution path: As long as no thread becomes idle, the counter remains valid in the thread's private cache and the threads only access their local stacks. When a thread becomes idle, the local counter-copies of the other threads are invalidated and the counter value is reloaded into their private cache. In most cases, we can amortize the counter reload by the previously processed task, as the task has typically more than $\Omega(kn_0)$ elements. When a thread adds an own task to the global stack, the task transfer is amortized by the workload reduction.

5 ANALYSIS OF IPS^{4}O

In this section, we analyze the additional memory requirement (Section 5.1), the I/O complexity (Section 5.2), and the local work (Section 5.3) of IPS^{4}o. The analysis in Sections 5.2 and 5.3 assumes the following constraints for IPS^{4}o:

ASSUMPTION 1. *Minimum size of a logical data block of IPS^{4}o: $b \in \Omega(tB)$*

ASSUMPTION 2. *Minimum number of elements per thread: $n/t \in \Omega(\max(M, bt))$.*

ASSUMPTION 3. *Restrict I/Os while sampling and buffers fit into private cache: $M \in \Omega(Bk \log k + bk)$.*

ASSUMPTION 4. *Oversampling factor: $\alpha \in \Theta(\log k')$ where k' is the current number of buckets.*

ASSUMPTION 5. *Restrict maximum size of base cases: $n_0 \in \Omega(\log k) \cap O(M/k)$.*

Without loss of generality, we assume that an element has the size of one machine word. In practice, we keep the block size b the same, i.e., the number of elements in a block is inverse proportional to the element size. In result, we can guarantee that the size of the buffer blocks does not exceed the private cache without adapting k.

5.1 Additional Memory Requirement

In this section, we show that IPS[4]o can be implemented either strictly in-place if the local task stack is implicitly represented or in-place if the tasks are stored on the recursion stack.

THEOREM 5.1. *IPS[4]o can be implemented with $O(kb)$ additional memory per thread.*

PROOF OF THEOREM 5.1. Each thread has a space overhead of two swap buffers and k buffer blocks of size b (in total $O(kb)$). This bound also covers smaller amounts of memory required for the partitioning steps. A partitioning step uses a search tree ($O(k)$), an overflow buffer ($O(b)$), read and write pointers ($O(kB)$ if we avoid false sharing), end pointers, and bucket boundary pointers ($\Theta(k)$ each). All of these data structures can be used for all recursion levels.

The classification phase stores elements only in the buffer blocks and the overflow buffer. As each thread reads its elements sequentially into its buffer blocks, there is always an empty block in the input array when a buffer block is flushed. When the size of the input array is not a multiple of the block size, a single overflow buffer may be required to store the overflow. The permutation phase only requires the swap buffers and the read and write pointers to move blocks into their target bucket. In the sampling phase, we do not need extra space as we swap the sample to the front of the input array. Nor do we need the local stacks (each of size $O(k \log_k \frac{n}{n_0})$) since we can use an implicit representation of the sequential tasks as described in Appendix B.   □

THEOREM 5.2. *With a local stack, IPS[4]o can be implemented with $O(tk \log_k \frac{n}{n_0})$ additional memory per thread.*

PROOF OF THEOREM 5.2. Each recursion level stores at most k tasks on the local stack. Only $O(\log_k \frac{n}{n_0})$ levels of parallel recursion are needed to get to the base cases with a probability of at least $1 - n_0/n$ (see Theorem A.1). In the rare case that the memory is exhausted, the algorithm is restarted.   □

5.2 I/O Complexity

Apart from the local work, the main issue of a sorting algorithm is the number of accesses to the main memory. In this section, we analyze this aspect in the PEM model. First, we show that IPS[4]o is I/O-efficient if the constraints we state at the beginning of this chapter apply. Then, we discuss how the I/O efficiency of IPS[4]o relates to practice.

THEOREM 5.3. *IPS[4]o has an I/O complexity of $O(\frac{n}{tB} \log_k \frac{n}{M})$ memory block transfers with a probability of at least $1 - M/n$.*

Before we prove Theorem 5.3, we prove that sequential partitioning steps exceeding the private cache are I/O-efficient (Lemma 5.4) and that parallel partitioning steps are I/O-efficient (Lemma 5.5).

LEMMA 5.4. *A sequential partitioning task with $n' \in \Omega(M)$ elements transfers $\Theta(n'/B)$ memory blocks.*

PROOF. A sequential partitioning task performs a partitioning step with one thread. The *sampling phase* of the partitioning step requires $\Theta(k \log k)$ I/Os for sorting the random sample (Assumption 4). We have $k \log k \in O(n'/B)$ as $M \in \Omega(Bk \log k)$ (Assumption 3). During the *classification phase*, the thread reads $O(n')$ consecutive elements, writes them to its local buffer blocks, and eventually moves them blockwise back to the main memory. This requires $O(n'/B)$ I/Os in total. As $M \in \Omega(kb)$, the local buffer blocks fit into the private cache. The same asymptotic cost occurs for moving blocks during the *permutation phase*. In the *cleanup phase*, the thread has to clean up k buckets. To clean up bucket i, the thread moves the elements from buffer block i and, if necessary, elements from a block which overlaps into bucket $i + 1$ to bucket boundary i. The elements from these two blocks are moved consecutively. We can amortize the transfer of full memory blocks with the I/Os from the classification phase as these blocks have been filled in the classification phase. We account $O(1)$ I/Os for potential truncated memory blocks at the ends of the consecutive sequences. For k bucket boundaries, this sums up to $O(k) \in O(n'/B)$ as $n' \in \Omega(M) \in \Omega(Bk)$ (Assumptions 1 and 3). □

LEMMA 5.5. *A parallel task with $\Theta(t'n/t)$ elements and t' threads transfers $\Theta\left(\frac{n}{tB}\right)$ memory blocks per thread.*

PROOF. A parallel task performs a partitioning step. The *sampling phase* of the partitioning step requires $O(k \log k)$ I/Os for loading the random sample (Assumption 4). We have $k \log k \in O\left(\frac{n}{tB}\right)$ as $n/t \in \Omega(Bk \log k)$ (Assumptions 2 and 3). During the *classification phase*, each thread reads $O(n/t)$ consecutive elements, writes them to its local buffer blocks, and eventually moves them blockwise back to the main memory. As $M \in \Omega(kb)$, the local buffer blocks fit into the private cache. In total, the classification phase transfers $O\left(\frac{n}{tb}\right)$ logical data blocks causing $O\left(\frac{n}{tB}\right)$ I/Os per thread.

The same asymptotic cost occurs for moving blocks during the *permutation phase*. Each thread performs $O\left(\frac{n}{tb}\right)$ successful acquisitions of the next block in a bucket. The successful acquisitions require $O\left(\frac{n}{tb}\right)$ reads and writes of the read pointers r_i and the write pointers w_i – for each read and write, we charge $O(t)$ I/Os for possible contention with other threads. Thus, the successful acquisitions sum up to $O\left(t \cdot \frac{n}{tb}\right) \in O\left(\frac{n}{tB}\right)$ I/Os (Assumption 1). For the block permutations of $O\left(\frac{n}{tb}\right)$ elements, we get the overall cost of $O\left(\frac{n}{tB}\right)$ I/Os. Furthermore, an additional block is loaded for each of the k buckets to recognize that the bucket does not contain any unprocessed blocks. Similar to the successful acquisitions we charge an overall cost of $O(kt)$ I/Os. Since $n/t \in \Omega(bk)$ (Assumptions 2 and 3), we have $k \in O\left(\frac{n}{t^2B}\right)$ and hence $O(kt) \in O\left(\frac{n}{tB}\right)$.

In the *cleanup phase*, t' threads have to clean up k buckets. To clean up a single bucket, elements from $t' + 2$ buffer blocks and bucket boundaries are moved. This takes $O(t'b/B)$ I/Os for cleaning a bucket. We consider a case distinction with respect to k and t'. If $k \leq t'$, then each thread cleans at most one bucket. This amounts to a cost of $O(t'b/B) \in O\left(\frac{n}{tB}\right)$ since $n/t \in \Omega(tb)$ (Assumption 2). If $k > t'$, then each thread cleans $O(k/t')$ buckets with a total cost of $O(k/t' \cdot t'b/B) \in O(kb/B)$ I/Os. We have $O(kb/B) \in O\left(\frac{n}{tB}\right)$ since $\Theta(n/t) \in \Omega(kb)$ (Assumptions 2 and 3). □

Now, we can prove that IPS^{4}o is I/O-efficient if the constraints we state at the beginning of this chapter apply.

PROOF OF THEOREM 5.3. In this proof, we can assume that IPS^{4}o performs $O\left(\log_k \frac{n}{M}\right)$ recursion levels until the tasks have at most M elements. According to Theorem A.1, this assumption holds with a probability of at least $1 - M/n$. We do not consider the situation of many identical keys since the elements with these identical keys will not be processed at later recursion levels anymore.

From Theorem 5.1 we know that IPS^{4}o uses additional data structures that require $O(kb)$ additional memory. In addition to the accesses to these data structures, a task $T[l, r)$ only accesses

$A[l .. r - 1]$. As $M \in \Omega(bk)$ (Assumption 3) we can keep the additional data structures in the private cache. Thus, we only have to count the memory transfers of tasks from and to the input array.

In this analysis, we consider a case distinction with respect to the task type, its size, and the size of its parent task. Each configuration requires at most $O\big(\frac{n}{tB} \log_k \frac{n}{M}\big)$ I/Os per thread.

Parallel tasks: IPS⁴o processes the parallel tasks first. Parallel tasks transfer $\Theta\big(\frac{n}{tB}\big)$ memory blocks per thread (see Lemma 5.5). As a thread performs at most one parallel task on each recursion level (see Lemma 4.4), parallel tasks on the first $O\big(\log_k \frac{n}{M}\big)$ recursion levels perform $O\big(\frac{n}{tB} \log_k \frac{n}{M}\big)$ I/Os per thread. On subsequent recursion levels, no parallel tasks are executed: After $O\big(\log_k \frac{n}{M}\big)$ recursion levels, the size of tasks is at most M. However, parallel tasks have more than M elements. This follows from Lemma 4.5 and Assumption 2.

Large tasks (Sequential partitioning task with $\omega(M)$ elements): A large task with n' elements takes $\Theta(n'/B)$ I/Os (see Lemma 5.4). A thread processes sequential tasks covering a continuous stripe of $O(n/t)$ elements of the input array (see Lemma 4.6). Thus, the large tasks of a thread transfer $O\big(\frac{n}{tB}\big)$ memory blocks on each recursion level. This sums up to $O\big(\frac{n}{tB} \log_k \frac{n}{M}\big)$ I/Os per thread for the first $\log_k \frac{n}{M}$ recursion levels. After $O\big(\log_k \frac{n}{M}\big)$ recursion levels, the size of tasks fits into the main memory, i.e., their size is $O(M)$.

Small tasks (Sequential tasks containing $O(M) \cap \Omega(B)$ elements with parent tasks having $\omega(M)$ elements or with parallel parent tasks having $\Omega(M)$ elements): In the first step of small tasks, the classification phase, the thread reads the elements of the task from left to right. As the task fits into the private cache, the task does not perform additional I/Os after the classification phase. For a small task of size n', we have $\lfloor n'/B \rfloor$ I/Os as $n' \in \Omega(B)$. Buckets of sequential partitioning tasks are sequential subtasks which again have $O(M)$ elements. Thus, each input element is only once part of a small task and small tasks cover disjoint parts of the input array. Additionally, we know from Lemma 4.6 that the sequential tasks of a thread contain $O(n/t)$ different elements. From this follows that a thread transfers $O\big(\frac{n}{tB}\big)$ memory blocks for small tasks.

Tiny tasks (Sequential tasks with $O(B)$ elements whose parent tasks have $\omega(M)$ elements or with parallel parent tasks having $\Omega(M)$ elements): A tiny task needs $O(1)$ I/Os. We account these I/Os to its parent task. A parent task gets at most $O(k)$ additional I/Os in the worst-case, $O(1)$ for each bucket. The parent task has $\Omega(tBk)$ elements: By definition, the parent task has $\Omega(M)$ elements and we have $M \in \Omega(tBk)$ (Assumptions 1 and 3). We have already accounted $\Omega(tBk/B)$ I/Os ($\Omega(Bk/B)$ I/Os) for this sequential (parallel) parent task previously (see I/Os of large tasks and parallel tasks). Thus, the parent task can amortize the I/Os of its tiny subtasks.

Middle tasks (Sequential tasks with sequential parent tasks containing $O(M)$ elements): Let the middle task $T[l_s, r_s)$ processed by thread i be a bucket of a sequential task $T[l, s)$ contained $O(M)$ elements. When thread i processed task $T[l, r)$, the subarray $A[l .. r - 1]$ was loaded into the private cache of thread i. As the thread processes the sequential tasks from its local stack in depth-first search order, $A[l .. r - 1]$ remains in the thread's private cache until the middle task $T[l_s, r_s)$ is executed. The middle task does not require any memory transfers – it only accesses $A[l_s .. r_s - 1]$ which is a subarray of $A[l .. r - 1]$.   □

In Appendix A.2 , we analyze the constant factors of the I/O volume (i.e., data flow between cache and main memory) for the sequential algorithms I1S⁴o (IPS⁴o with $t = 1$) and S⁴o. To simplify the discussion, we assume a single recursion level, $k = 256$ and 8-byte elements. We show that I1S⁴o needs about $48n$ bytes of I/O volume, whereas S⁴o needs between $67n$ and $84n$, depending on whether we use a conservative calculation or not. This is surprising since, at the first glance, the partitioning algorithm of I1S⁴o writes the data twice, whereas S⁴o does this only once. However, this is more than offset by "hidden" overheads of S⁴o like memory management, allocation misses, and associativity misses.

5.3 Branch Mispredictions and Local Work

Besides the latency of loading and writing data, which we analyze in the previous section, branch mispredictions and the (total) work of an algorithm can limit its performance. In the next paragraph, we address branch mispredictions of IPS^{4}o and afterwards, we analyze the total work of IPS^{4}o.

Our algorithm IPS^{4}o has virtually no branch mispredictions during element classification. The main exception is when the algorithm detects that a bucket has to be flushed. A bucket is flushed on average after b element classifications (after $b \log k$ element comparisons).

We now analyze the local work of IPS^{4}o. We neglect delays introduced by thread synchronizations and accesses to shared variables as we accounted for those in the I/O analysis in the previous section. For the analysis, we assume that the base case algorithm performs partitioning steps with $k = 2$ and a constant number of samples until at most one element remains. Thus, the local work of the base case algorithm is identical to the local work of quicksort. We also assume that the base case algorithm is used to sort the samples.

Actually, our implementation of IPS^{4}o sorts the base cases with insertion sort. The reason is that a base case with more than $2n_0$ elements can only occur if its parent task has between n_0 and kn_0 elements. In this case, the average base case size is between $0.5n_0$ and n_0. Our experiments have shown that insertion sort is more efficient than quicksort for these small inputs. Also, base cases with much more than $2n_0$ elements are very rare. To sort the samples, our implementation recursively invokes IPS^{4}o.

THEOREM 5.6. *When using quicksort to sort the base cases and the samples, IPS^{4}o has a local work of $O(n/t \log n)$ with a probability of at least $1 - n^{-4}$.*

For the proof of Theorem 5.6, we need Lemmas 5.7 to 5.12. These lemmas use the term *small task* for tasks with at most kn_0 elements and the term *large task* for tasks with at least kn_0 elements.

LEMMA 5.7. *A partitioning task with n' elements and t' threads has a local work of $O(n'/t' \log k)$ excluding the work for sorting the samples.*

PROOF. In the *classification phase* of the partitioning step, the comparisons in the branchless decision tree dominate. Each thread classifies n/t' elements with takes $O(\log k)$ comparisons each. This sums up to $O(n'/t' \log k)$. The element classification dominates the remaining work of this phase, e.g., the work for loading each element once, and every b elements, the work for flushing a local buffer.

In the *permutation phase*, each block in the input array is swapped into a buffer block once and swapped back into the input array once. As each thread swaps at most $\lfloor \frac{n'}{tb} \rfloor$ blocks of size b, the phase has $O(n'/t)$ local work.

When the *cleanup phase* is executed sequentially, the elements of the local buffers are flushed into blocks that overlap into the next bucket. This may displace elements stored in these blocks. The displaced elements are written into empty parts of blocks. Thus, each element is moved at most once which sums up to $O(n')$ work. For the cleanup phase of a parallel partitioning step, we conclude from the proof of IPS^{4}o's I/O complexity (see Theorem 5.3) that the local work is in $O(n'/t')$: The proof of the I/O complexity shows that a parallel cleanup phase is bounded by $O\left(\frac{n'}{t'B}\right)$ I/Os. Also, each element that is accessed in the cleanup phase is moved at most once and no additional work is performed. We account a local work of B for each memory block which a thread accesses. Thus, we can derive from $O\left(\frac{n'}{t'B}\right)$ I/Os a local work of $O(n'/t')$ for the parallel cleanup phase. □

LEMMA 5.8. *At most one parallel task which is processed by a thread is a small task.*

PROOF. Assume that a thread processes at least two small parallel tasks p_1 and p_2. According to Lemma 4.4, a thread processes at most one of these tasks per recursion level. A parallel subtask of thread i is a subtask of a parallel task of thread i and represents a bucket of this task (see Lemma 4.3). Thus, p_1 and p_2 must be on different recursion levels, and p_1 processes a subset of elements processed by p_2 or vice versa. However, this is a contradiction as buckets of small tasks are base cases.   □

LEMMA 5.9. *The local work for all small partitioning tasks is in total $O(n/t \log k)$ excluding the work for sorting the samples.*

PROOF. In this proof, we neglect the work for sorting the sample. Lemma 5.7 tells us that a partitioning task with n' elements and t' threads has a local work of $O(n'/t' \log k)$. Also, parallel tasks with t' threads have $\Theta(t'n/t)$ elements (see Lemma 4.5). Thus, we have $O(n/t \log k)$ local work for a small parallel task. Overall, we have $O(n/t \log k)$ local work for small parallel tasks as each thread processes at most one of these tasks (see Lemma 5.8).

We now consider small sequential partitioning tasks. small sequential partitioning tasks that are processed by a single thread cover in total at most $O(n/t)$ different elements (see Lemma 4.6). Each of these elements passes at most one of the small partitioning tasks since buckets of these tasks are base cases. Thus, a thread processes small partitioning tasks of size $O(n/t)$ in total. As small partitioning tasks with n' elements require $O(n' \log k)$ local work (see Lemma 5.7), the local work of all small partitioning tasks is $O(n/t \log k)$.   □

LEMMA 5.10. *The partitioning tasks of one recursion level require $O(n/t \log k)$ local work excluding the work for sorting the samples.*

PROOF. Lemma 5.7 tells us that a partitioning task with n' elements and t' threads has a local work of $O(n'/t' \log k)$ excluding the work for sorting the samples. Parallel tasks with t' threads have $\Theta(t'n/t)$ elements (see Lemma 4.5). Thus, we have $O(n/t \log k)$ local work on a recursion level for parallel tasks. A thread processes sequential partitioning tasks covering a continuous stripe of $O(n/t)$ elements of the input array (see Lemma 4.6). Thus, we also have $O(n/t \log k)$ local work on a recursion level for sequential partitioning tasks.   □

LEMMA 5.11. *All large partitioning tasks have in total $O(n/t \log n)$ local work with a probability of at least $1 - n^{-1}$ excluding the work for sorting the samples.*

PROOF. In this proof, we neglect the work for sorting the samples of a partitioning task. Large partitioning tasks create between $\sqrt{k}$ and k buckets (see Section 4.1.1). According to Theorem A.1, IPS⁴o performs at most $\Theta\left(\log_{\sqrt{k}} \frac{n}{kn_0}\right) = \Theta\left(\log_k \frac{n}{kn_0}\right)$ recursion levels with a probability of at least $1 - kn_0/n$ until all partitioning tasks have less than kn_0 elements. However, this probability is not tight enough. Instead, we can perform up to $O(\log_k n)$ recursion levels and still have $O(n/t \log n)$ local work as each recursion level requires $O(n/t \log k)$ local work (see 5.10). In this case, Theorem A.1 tells us that all partitioning tasks have at most one element with a probability of $1 - n^{-1}$. This probability also holds if we stop partitioning buckets with less than kn_0 elements.   □

LEMMA 5.12. *The local work of all base case tasks is in total $O(n/t \log n)$ with a probability of at least $1 - n^{-1}$.*

PROOF. Sorting $O(n)$ elements with the base case algorithm quicksort does not exceed $O(\log n)$ recursion levels with probability $1 - n^{-1}$ [41]. Thus, an execution of quicksort with $O(n/t)$ elements requires $O(n/t \log n)$ local work with a probability of at least $1 - n^{-1}$ as it would also not exceed $O(\log n)$ recursion levels with at least the same probability. Sorting all base case of a thread is

asymptotically at least as efficient as sorting $O(n/t)$ elements at once: The base cases have in total at most $O(n/t)$ elements (see Lemma 4.6) but the input is already prepartitioned. □

LEMMA 5.13. *The local work for sorting the samples of all small partitioning tasks is $O(n/t \log n)$ in total with a probability of at least $1 - n^{-1}$*

PROOF. Small partitioning tasks with n' elements have n'/n_0 buckets and a sample of size $O(n'/n_0 \log \frac{n'}{n_0})$ (see Assumption 4 for the oversampling factor). The size of a sample is in particular bounded by $O(n')$. For this, we use $n_0 \in \Omega(\log k)$ (Assumption 5) and $k \geq n'/n_0$ from which follows that $n_0 \in \Omega(\log \frac{n'}{n_0})$ holds. Furthermore, small sequential partitioning tasks which are processed by a single thread cover in total at most $O(n/t)$ different elements (see Lemma 4.6). Thus, the total size of all samples from small sequential partitioning tasks sorted by a thread is limited to $O(n/t)$. We count one additional sample from a potential small parallel task (see Lemma 5.8). We can also limit its size to $O(n/t)$ using Assumptions 2 and 5. We can prove that the work for sorting samples of a total size of $O(n/t)$ is in $O(n/t \log n)$ with a probability of at least $1 - n^{-1}$. We refer to the proof of Lemma 5.12 for details. □

LEMMA 5.14. *The local work for sorting the samples of all large partitioning tasks is $O(n/t \log n)$ in total with a probability of at least $1 - n^{-1}$*

PROOF. A sequential large partitioning task with $\Omega(kn_0)$ elements has $\Omega(k \log k)$ elements (see Assumption 5) and a parallel large partitioning task has $\Omega(t'n/t)$ elements (see Lemma 4.5) with $n/t \in \Omega(k \log k)$ which is a result from Assumptions 2 and 3. Thus, a large partitioning task with t' threads has $\Omega(t'k \log k)$ elements. From Lemma 5.7 follows that a large partitioning task excluding the work for sorting the samples has $\Omega(k \log^2 k)$ local work.

Each thread invokes at most $r = l \frac{n \log n}{tk \log^2 k}$ large partitioning tasks for a constant l: These tasks require $\Omega(k \log^2 k)$ local work each and all partitioning tasks performed by a single thread require $O(n/t \log n)$ local work in total (see Lemma 5.11) – excluding the work for sorting the samples. Thus, when we execute large partitioning tasks, each thread performs at most r sample sorting routines – one for each task.

For the local work analysis, we consider a modified sample sorting algorithm. Instead of using quicksort, we use an adaption that restarts quicksort when it exceeds $O(k \log^2 k)$ local work until the sample is finally sorted. The bounds for the local work which we obtain from this variant also hold when IPS^{4}o executes quicksort until success instead of restarting the algorithm: A restart means to neglect the prepartitioned buckets which makes the problem unnecessarily difficult.

For the sample sorting routines of large partitioning tasks, each thread can spend $O(rk \log^2 k)$ local work in total. As we restart quicksort after $O(k \log^2 k)$ local work, we can amortize even xr (re)starts of quicksort for any constant x. We show that xr (re)starts are sufficient to successfully sort r samples with a probability of at least $1 - n^{-1}$.

We observe that one execution of quicksort unsuccessfully sorts a sample with a probability of at most $p = k^{-3} \log_2^{-3} k$ as the size of the samples is bounded by $O(k \log k)$. For this approximation, we use that sorting n elements with quicksort takes $O(n \log n)$ work with high probability [41]. Each execution of quicksort is a Bernoulli trial as we have exactly two possible outcomes, "successful sorting in time" and "unsuccessful sorting in time", and the probability of failure is bounded by p each time. When we consider all quicksort invocations of IPS^{4}o, we need r successes. We define a binomial experiment which repeatedly invokes quicksort on the sample of the first large partitioning task until success and then continues with the second large partitioning task, until the sample of each partitioning step of a thread is sorted. Asymptotically, we can spend xr (re)starts of quicksort for any constant $x \geq 1$ such that the binomial experiment does not exceed $O(n/t \log n)$ local work.

Let the random variable X be the number of unsuccessful sample sorting executions and assume that $x \geq 2$. Then, the probability I

$$I = \mathbb{P}[X > (x-1)r]$$

$$\leq \sum_{j>(x-1)r} \binom{xr}{j} p^j (1-p)^{xr-j} \leq \sum_{j>(x-1)r} \left(\frac{xre}{j}\right)^j p^j$$

$$\leq \sum_{j>(x-1)r} \left(\frac{xre}{(x-1)r}\right)^j p^j = \sum_{j>(x-1)r} \left(\frac{pex}{x-1}\right)^j$$

$$= \frac{\left(\frac{pex}{x-1}\right)^{(x-1)r+1}}{1 - \frac{pex}{x-1}} \tag{1}$$

$$\leq \left(\frac{pex}{x-1-pex}\right)\left(\frac{pex}{x-1}\right)^{\frac{(x-1)lk\log k\log n}{k\log^2 k}}$$

$$\leq \left(\frac{pex}{x-1-pex}\right)(n)^{\frac{(x-1)l\log\left(\frac{pex}{x-1}\right)}{\log k}}$$

$$\leq 2.13 n^{-\frac{1}{2}(x-1)}$$

defines an upper bound of the probability that xr (re)starts of the the sample sorting algorithm execute less than r successful runs. The second "$\leq$" uses $\binom{n}{k} \leq (en/k)^k$, the third "$=$" uses $\sum_{k=n}^{\infty} r^k = \frac{r^n}{1-r}$, derived from the geometric series, the second "$\leq$" and the third "$=$" use $\frac{pex}{x-1} < 1$, and the last "$\leq$" uses $\frac{\log\left(\frac{pex}{x-1}\right)}{\log k} < -1/2$ and $\frac{pex}{x-1-pex} < 2.13$.   □

PROOF OF THEOREM 5.6. According to Lemma 5.9, the small partitioning tasks excluding the work for sorting the samples require $O(n/t \log k)$ local work. For large partitioning tasks, we have $O(n/t \log n)$ local work with a probability of at least $1 - n^{-1}$ (see Lemma 5.11). The same holds for the base cases (see Lemma 5.12). Lemmas 5.13 and 5.14 bound the local work for sorting the samples of small and large partitioning tasks to $O(n/t \log n)$, each with a probability of at least $1 - n^{-1}$. This sums up to a total local work of $O(n/t \log n)$ with a probability of at least $1 - 4/n$.   □

6 IMPLEMENTATION DETAILS

IPS^{4}o has several parameters that can be used for tuning and adaptation. We performed our experiments using (up to) $k = 256$ buckets, an oversampling factor $\alpha = 0.2 \log n$, a base case size $n_0 = 16$ elements, and a block size of $b = \max(1, 2^{\lfloor 11 - \log_2 D \rfloor})$ elements, where D is the size of an element in bytes (i.e., about 2 KiB). In the sequential case, we avoid the use of atomic operations on pointers and we use the recursion stack to store the tasks. On the last level, we perform the base case sorting immediately after the bucket has been completely filled in the cleanup phase, before processing the other buckets. This is more cache-friendly, as it eliminates the need for another sweep over the data. Furthermore, we use insertion sort as the base case sorting algorithm.

IPS^{4}o (I1S^{4}o) detects sorted inputs. If these inputs are detected, our algorithm reverses the input in the case that the input was sorted in decreasing order, and returns afterward. Note that such heuristics for detecting "easy" inputs are quite common [57, 63].

For parallelization, we support OpenMP or std::thread transparently. If the application is compiled with OpenMP support, IPS^{4}o employs the existing OpenMP threads. Otherwise, IPS^{4}o uses C++ threads and determines t by invoking the function std::thread::hardware_concurrency.

Additionally, the application can use its own custom threads to execute IPS^4o. For that, the application creates a thread pool object provided by IPS^4o, adds its threads to the thread pool, and passes the thread pool to IPS^4o.

We store each read pointer r_i and its corresponding write pointer r_i in a single 128-bit word which we read and modify atomically. We use 128-bit atomic compare-and-swap operations from the GNU Atomic library `libatomic` if the CPU supports these operations. Otherwise, we guard the pointer pair with a mutex. We did not measure a difference in running time between these two approaches except for some very special corner cases. We align the thread-local data to 4 KiB which is typically the memory page size in systems. The alignment avoids false sharing and simplifies the migration of memory pages if a thread is moved to a different NUMA node.

We decided to implement our own version of the (non-in-place) algorithm S^4o from Sanders and Winkel [64]. This has two reasons. First, the initial implementation is only of explorative nature, e.g., the implementation does not handle duplicate keys, and, the implementation is highly tuned for outdated hardware architectures. Second, the reimplementation S^4oS [39] seemed to be unreasonably slow. We use IPS^4o as an algorithmic framework to implement PS^4o, our version of S^4o. For PS^4o, we had to implement the three main parts of S^4o: (1) the partitioning step, (2) the decision tree, and (3), the base case algorithm and additional parameters of the algorithm, e.g., for the maximum base case size, the number of buckets, and the oversampling factor. We replace the partitioning step of IPS^4o with the one described by Sanders and Winkel. For the element classification, we reuse the branchless decision tree of IPS^4o. We also reuse the base case algorithm and the parameters from IPS^4o which seem to work very well for PS^4o. As we use IPS^4o as an algorithmic framework, we can execute PS^4o in parallel or with only one thread. If we refer to PS^4o in its sequential form, we use the term $1S^4o$.

We also use IPS^4o as an algorithmic framework to implement *In-place Parallel Super Scalar Radix Sort* (IPS^2Ra). For IPS^2Ra, we replaced the branchless decision tree of IPS^4o with a simple radix extractor function that accepts unsigned integer keys. For tasks with less than 2^{12} elements, we use SkaSort as a base case sorting algorithm. For small inputs ($n \leq 2^7$), SkaSort then falls back to quicksort which again uses insertion sort for $n \leq 2^5$. If we refer to IPS^2Ra in its sequential form, we use the term $I1S^2Ra$. IPS^2Ra only sorts data types with unsigned integer keys. The author of SkaSort [67, 68] demonstrates that a radix sorter can be extended to sort inputs with floating-point keys and even compositions of primitive data types. We note that $I1S^2Ra$ can be extended to sort these data types as well.

Our algorithms IPS^4o, IPS^2Ra, and PS^4o are written in C++ and the implementations can be found on the official website https://github.com/ips4o. The website also contains the benchmark suite used for this publication and a description of how the experiments can be reproduced.

7 EXPERIMENTAL RESULTS

In this section, we present results from ten data distributions, generated for four different data types obtained on four different machines with one, two, and four processors and 21 different sorting algorithms. We extensively compare our in-place parallel sorting algorithms IPS^4o and IPS^2Ra as well as their sequential counterparts $I1S^4o$ and $I1S^2Ra$ to various competitors[4]:

- **Parallel in-place comparison-based sorting**
  - MCSTLbq (OpenMP): Two implementations (balanced and unbalanced) from the GCC STL library [66] based on quicksort proposed by Tsigas and Zhang [70].
  - TBB [75] (TBB): Quicksort from the Intel® TBB library [63].
- **Parallel non-in-place comparison-based sorting**

---

[4]Since several algorithms were implemented by third parties, we may cite their publication and implementation separately.

- – PBBS [65] (Cilk): $\sqrt{n}$-way samplesort [11] implemented in the so-called problem based benchmark suite.
  - – MCSTLmwm (OpenMP): Stable multiway mergesort from the GCC STL library [66].
  - – PS^{4}o [4] (OpenMP or `std::thread`): Our parallel and stable implementation of S^{4}o from Section 6.
  - – ASPaS [69] (POSIX Threads): Stable mergesort which vectorizes the merge function with AVX2 [37]. ASPaS only sorts `int`, `float`, and `double` inputs and uses the comparator function "$<$".
- **Parallel in-place radix sort**
  - – RegionSort [56] (Cilk): *Most Significant Digit* (MSD) radix sort [57] which only sorts keys of unsigned integers. RegionSort skips the most significant bits which are zero for all keys.
  - – IMSDradix [60] (POSIX Threads): MSD radix sort [61] from Orestis and Ross with blockwise redistribution. The implementation, published by Orestis and Ross, however, requires 20 % of additional memory in addition to the input array and is very explorative.
- **Parallel non-in-place radix sort**
  - – PBBR [65] (Cilk): A simple implementation of stable MSD radix sort from the so-called problem based benchmark suite.
  - – RADULS2 [33] (`std::thread`): MSD radix sort which uses non-temporal writes to avoid write allocate misses [46]. RADULS2 requires 256-bit array alignment. Both keys and objects have to be aligned at 8-byte boundaries. We partially compiled the code with the flag `-O1` as recommended by the authors. The algorithm does not compile with the Clang compiler.
- **Sequential in-place comparison-based sorting**
  - – BlockQ [73]: An implementation of BlockQuicksort [21] provided by the authors of the sorting algorithm publication.
  - – BlockPDQ [58]: Pattern-Defeating Quicksort which integrated the approach of Block-Quicksort in 2016. BlockPDQ has similar running times as BlockQuicksort using Lomuto's Partitioning [2], published in 2018.
  - – DualPivot [74]: A C++ port of Yaroslavskiy's Dual-Pivot Quicksort [76]. Yaroslavskiy's Dual-Pivot Quicksort is the default sorting routine for primitive data types in Oracle's Java runtime library since version 7.
  - – std::sort: Introsort from the GCC STL library.
  - – WikiSort [52]: An implementation of stable in-place mergesort [45].
- **Sequential non-in-place comparison-based sorting**
  - – Timsort [32]: A C++ port of Timsort [59]. Timsort is an implementation of stable mergesort which takes advantage of presorted sequences of the input. Timsort is part of Oracle's Java runtime library since version 7 to sort non-primitive data types.
  - – S^{4}oS [39]: A recent implementation of non-in-place S^{4}o [64] optimized for modern hardware.
  - – 1S^{4}o [4]: Our implementation of S^{4}o which we describe in Section 6.
- **Sequential in-place radix sort**
  - – SkaSort [68]: MSD radix sort [67] which accepts a key-extractor function returning primitive data types or pairs, tuples, vector, and arrays containing primitive data types. The latter ones are sorted lexicographically.
- **Sequential non-in-place radix sort**
  - – IppRadix [17]: Radix sort from the Intel® Integrated Performance Primitives library optimized with the AVX2 and AVX-512 instruction set.
- **Sequential non-in-place sorting with machine learning models**

Algorithm 3 Quartet comparison
function LESSTHAN(l, r)
if $l.a \neq r.a$ **then**
return $l.a < r.a$
else if $l.b \neq r.b$ **then**
return $l.b < r.b$
else return $l.c < r.c$

Algorithm 4 100B comparison
function LESSTHAN(l, r)
for $i \leftarrow 0$ **to** 9 **do**
if $l.k[i] \neq r.k[i]$ **then**
return $l.k[i] < r.k[i]$
return False;

- LearnedSort [47]: An algorithm [48] for sorting numeric data using a learned representation of the cumulative key distribution function as a piecewise linear function. This can be viewed as a generalization of radix sort.

We do not compare our algorithm to PARADIS as its source code is not publicly available. However, Omar et. al. [57] compare RegionSort to the numbers reported in the publication of PARADIS and conclude that RegionSort is faster than PARADIS. Additionally, RegionSort and our algorithm have stronger theoretical guarantees (see Section 3).

Most radix sorters, i.e., LearnedSort, IppRadix, IMSDradix, RADULS2, PBBR, and RegionSort, do not support all data types used in our experiments. Only the radix sorter SkaSort supports all data types as the types used here are either primitives or compositions of primitives, which are sorted lexicographically. All algorithms are written in C++ and compiled with version 7.5.0 of the GNU compiler collection, using the optimization flags "-march=native -O3". We have not found a more recent compiler that supports Cilk threads, needed for RegionSort, PBBS, and PBBR.

We ran benchmarks with 64-bit floating-point elements, 64-bit unsigned integers, 32-bit unsigned integers, and *Pair*, *Quartet*, and *100B* data types. Pair (Quartet) consists of one (three) 64-bit unsigned integers as key and one 64-bit unsigned integer of associated information. 100B consists of 10 bytes as key and 90 bytes of associated information. The keys of Quartet and 100B are compared lexicographically. Algorithms 3 and 4 show the lexicographical compare function which we used in our benchmarks. We want to point out that lexicographical comparisons can be implemented in different ways. We also tested std::lexicographical_compare for Quartet and std::memcmp for 100B. However, it turned out that these compare functions are (much) less efficient for all competitive algorithms. SkaSort is the only radix sorter that is able to sort keys lexicographically. For Quartet and 100B data types, we invoke SkaSort with a key-extractor function that returns the values of the key stored in a std::tuple object.

We ran benchmarks with ten input distributions: Uniformly distributed (*Uniform*), exponentially distributed (*Exponential*), and almost sorted (*AlmostSorted*), proposed by Shun et. al. [65]; *RootDup*, *TwoDup*, and *EightDup* from Edelkamp et. al. [21]; and *Zipf* (Zipf distributed input), *Sorted* (sorted Uniform input), *ReverseSorted*, and *Zero* (just zeros). The input distribution Exponential generates and hashes numbers selected uniformly at random from $[2^i, 2^{i+1})$ with $i \in \mathbb{N} \wedge i \leq \log n$, RootDup sets $A[i] = i \mod \lfloor\sqrt{n}\rfloor$, TwoDup sets $A[i] = i^2 + n/2 \mod n$, and EightDup sets $A[i] = i^8 + n/2 \mod n$. The input distribution Zipf generates the integer number $k \in [1, 10^2]$ with probability proportional to $1/k^{0.75}$. Figure 10 illustrates the nontrivial input distributions RootDup, Zipf, Exponential, TwoDup, EightDup, and AlmostSorted.

We ran our experiments on the following machines:

- Machine *A1x16* with one AMD Ryzen 9 3950X 16-core processor and 32 GiB of memory.
- Machine *A1x64* with one AMD EPYC Rome 7702P 64-core processor and 1024 GiB of memory.
- Machine *I2x16* with two Intel Xeon E5-2683 v4 16-core processors and 512 GiB of memory.

Fig. 10. Examples of nontrivial input distributions for 512 uint32 values.

• Machine *I4x20* with four Intel Xeon Gold 6138 20-core processors and 768 GiB of memory.

Each algorithm was executed on all machines with all input distributions and data types. The parallel (sequential) algorithms were executed for all input sizes with $n = 2^i, i \in \mathbb{N}^+$, until the input array exceeds 128 GiB (32 GiB). For $n < 2^{33}$ ($n < 2^{30}$), we perform each parallel (sequential) measurement 15 times and for $n \geq 2^{33}$ ($n \geq 2^{30}$), we perform each measurement twice. Unless stated otherwise, we report the average over all runs except the first one[5]. We note that non-in-place sorting algorithms which require an additional array of n elements will not be able to sort the largest inputs on *A1x16*, the machine with 32 GiB of memory. We also want to note that some algorithms did not support all data types because their interface rejects the key. In our figures, we emphasize algorithms that are "not general-purpose" with red lines, i.e., because they assume integer keys (I1S²Ra, IppRadix, RADULS2, RegionSort, and PBBR), make additional assumptions on the data type (RADULS2), or at least because they do not accept a comparator function (SkaSort). All non-in-place algorithms except RADULS2 return the sorted output in the input array. These algorithms copy the input back into the input array if the algorithm has executed an even number of recursion levels. Only RADULS2 returns the output in a second "temporary" array. To be fair, we copy the data back into the input array in parallel and include the time in the measurement.

We tested all parallel algorithms on Uniform input with and without hyper-threading. Hyper-threading did not slow down any algorithm. Thus, we give results of all algorithms with hyper-threading. Overall, we executed more than 500 000 combinations of different algorithms, input

---

[5]The first run is excluded because we do not want to overemphasize time effects introduced by memory management, instruction cache warmup, side effects of different benchmark configurations, . . .

distributions, input sizes, data types, and machines. We now present an interesting selection of our measurements and discuss our results.

This chapter is divided as follows. Section 7.1 introduces and discusses the statistical measurement *average slowdown* which we use to compare aggregated measurements. We present the results of I1S^{4}o and I1S^{2}Ra and their sequential competitors in Section 7.2. In Section 7.3, we discuss the influence of different memory allocation policies on the running time of parallel algorithms. Section 7.4 compares our parallel algorithm IPS^{4}o to its implementation presented in the conference version of this article [6]. We compare the results of IPS^{4}o and IPS^{2}Ra to their parallel competitors in Section 7.5. Finally, Section 7.6 evaluates the subroutines of IPS^{4}o, IPS^{2}Ra, and their sequential counterparts.

7.1 Statistical Evaluation

Many methods are available to compare algorithms. In our case, the cross product of machines, input distributions, input sizes, data types, and array types describes the possible inputs of our benchmark. In this work we consider the result of a benchmark input always averaged over all executions of the input, using the arithmetic mean. A common approach of presenting benchmark results is to fix all but two variables of the benchmark set and show a plot for these two variables, e.g., plot the running time of different algorithms over the input size in a graph for a specific input distribution, data type, and array type, executed on a specific machine. Often, an interesting subset of all possible graphs is presented as the benchmark instances have too many parameters. However, in this case, a lot of data is not presented at all and general propositions require further interpretation and aggregation of the presented, and possibly incomplete, data. Besides running time graphs and speedup graphs, we use average slowdown factors (*average slowdowns*) and *performance profiles* to present our benchmark results.

Let $\mathcal{A}$ be a set of algorithms, let $\mathcal{I}$ be a set of inputs, let $S_A(\mathcal{I})$ be the inputs of $\mathcal{I}$ which algorithm A sorts successfully, and let $r(A, I)$ be the running time of algorithm A for input I. Furthermore, let $r(A, I, T)$ be the running time of an algorithm A for an input I with array type T. Note that A might not sort I successfully i.e., because its interface does not accept the data type or because A does not return sorted input. In this case, the running time of A is not defined.

To obtain **average slowdowns**, we first define the *slowdown factor* of an algorithm $A \in \mathcal{A}$ to the algorithms $\mathcal{A}$ for the input I

$$f_{\mathcal{A},I}(A) = \begin{cases} r(A, I)/\min(\{r(A', I) \mid A' \in \mathcal{A}\}) & I \in S_A(\mathcal{I}), \text{ i.e., } A \text{ successfully sorts } I \\ \infty & \text{otherwise.} \end{cases}$$

as the slowdown using algorithm A to sort input I instead of using the fastest algorithm for I from the set of algorithms $\mathcal{A}$. Then, the *average slowdown of algorithm $A \in \mathcal{A}$ to the algorithms $\mathcal{A}$ for the inputs $\mathcal{I}$*

$$s_{\mathcal{A},\mathcal{I}}(A) = \sqrt[|S_A(\mathcal{I})|]{\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{A},I}(A)}$$

is the geometric mean of the slowdown factors of algorithm A to the algorithms $\mathcal{A}$ for the inputs of $\mathcal{I}$ which A sorts successfully.

Besides the average slowdown of algorithms, we present average slowdowns of an input array type to compare its performance to a set $\mathcal{T}$ of array types. The slowdown factor of an array $T \in \mathcal{T}$ to the arrays $\mathcal{T}$ for the input I and an algorithm A

$$f_{\mathcal{T},A,I}(T) = \begin{cases} r(A, I, T)/\min(\{r(A, I, T') \mid T' \in \mathcal{T}\}) & I \in S_A(\mathcal{I}), \text{ i.e., } A \text{ successfully sorts } I \\ \infty & \text{otherwise.} \end{cases}$$

is defined as the slowdown of using array type T to sort input I with algorithm A instead of using the best array from the set of array types $\mathcal{T}$.

Then, the *average slowdown of an array $T \in \mathcal{T}$ to the array types $\mathcal{T}$ for the inputs $\mathcal{I}$ and the algorithm A*

$$s_{A,\mathcal{T},\mathcal{I}}(T) = \sqrt[|S_A(I)|]{\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{T},A,I}(T)}$$

is the geometric mean of the slowdown factors of T to the arrays $\mathcal{T}$ for the inputs $\mathcal{I}$ and algorithm A which A sorts successfully.

Average slowdown factors are heavily used by Timo Bingmann [8] to compare parallel string sorting algorithms. We want to note that the average slowdown could also be defined as the arithmetic mean of the slowdown factors, instead of using the geometric mean. In this case, the average slowdown would have a very strong meaning: The average slowdown of an algorithm A over a set of inputs is the expected average slowdown of A when an input of the benchmark set is picked at random to the fastest algorithm for this particular input. However, Timo Bingmann used in his work the geometric mean for the average slowdowns to "emphasize small relative differences of the fastest algorithms". Additionally, the geometric mean is more robust against outliers and skewed measurements than the arithmetic mean [50, p. 229]. Furthermore, the arithmetic mean of ratios is "meaningless" in the general case. For example, Fleming and Wallance [25] state that the arithmetic mean is meaningless when different machine instances are compared relative to a baseline machine. In this case, ratios smaller than one and larger than one can occur. However, combining those numbers is "meaningless" as ratios larger than one depend linearly on the measurements but ratios smaller than one do not. Note that the slowdown factors in this work will never be smaller than one.

A **performance profile** [18] is the cumulative distribution function of an algorithm for a specific performance metric. We use the slowdown factors to quantify the relative performance distribution of an algorithm to a set of competitors on a set of inputs. The *performance profile of algorithm $A \in \mathcal{A}$ to the algorithms $\mathcal{A}$ for the inputs $\mathcal{I}$*

$$p_{\mathcal{A},\mathcal{I},A}(\tau) = \frac{|\{I \in \mathcal{I} \mid f_{\mathcal{A}}(A, I) \leq \tau\}|}{|\mathcal{I}|}$$

is the probability that algorithm A sorts a random input $I \in \mathcal{I}$ at most a factor τ slower than the fastest algorithm for input I.

To avoid skewed measurements we only use input sizes above a certain threshold for the calculation of average slowdowns and performance profiles. This is an obvious decision as it is very common that algorithms switch to a base case algorithm for small inputs. By restricting the input size, the results are not affected by the choice of different thresholds and the choice of the algorithm for small inputs. Additionally, the algorithms generally sort "very small" inputs inefficiently (except algorithms with are designed for small inputs) and we do not want those measurements to dominate the average performance. For sequential algorithms, we use inputs with at least 2^{18} bytes and for parallel algorithms, we use inputs with at least $2^{21}t$ bytes.

When an algorithm uses a heuristic to detect easy inputs and quickly transforms these inputs into sorted output, the slowdown factors of algorithms that do not use such heuristics are usually orders of magnitude larger than the remaining ratios of our benchmark. When aggregating those large ratios with ratios of inputs that are not easy, the large ratios would dominate the result. Therefore, we exclude the inputs Zero, Sorted, and ReverseSorted when we average over all input distributions to obtain average slowdowns and performance profiles.

7.2 Sequential Algorithms

In this section, we compare sequential algorithms for different machines, input distributions, input sizes, and data types. We begin with a comparison of the average slowdowns of $I1S^2Ra$, $I1S^4o$, and their competitors for ten input distributions executed with six different data types (see Section 7.2.1). This gives a first general view of the performance of our algorithms as the presented results are aggregated across all machines. Afterwards, we compare our algorithms to their competitors on different machines by scaling with input sizes for input distribution Uniform and data type uint64 (see Section 7.2.2). Finally, we discuss the performance profiles of the algorithms in Section 7.2.3.

7.2.1 Comparison of Average Slowdowns. In this section, we discuss the average slowdowns of sequential algorithms for different data types and input distributions aggregated over all machines and input sizes with at least 2^{18} bytes shown in Table 1. The results indicate that a sorting algorithm performs similarly good for inputs with "similar" input distributions. Thus, we divide the inputs into four groups: The largest group, *Skewed inputs*, contains inputs with duplicated keys and skewed key occurrences, i.e., Exponential, Zipf, RootDup, TwoDup, and EightDup. The second group, *Uniform inputs*, contains Uniform distributed inputs. For these inputs, each bit of the key has maximum entropy. Thus, we can expect that radix sort performs the best for these inputs. The third group, *Almost Sorted inputs*, are AlmostSorted distributed inputs. The last group, *(Reverse) Sorted inputs*, contains "easy inputs", i.e., Sorted, ReverseSorted, and Zero. For the average slowdowns separated by machine, we refer to Appendix C, Tables 7 to 10.

In this section, an *instance* describes the inputs of a specific data type and input distribution. We say that "algorithm A is faster than algorithm B (by a factor of C) for some instances" if the average slowdown of B is larger than the average slowdown of A (by a factor of C) for these instances.

The subsequent paragraph summarizes the performance of our algorithms. Then, we compare our competitors to our radix sorter $I1S^2Ra$. Finally, we compare our competitors to our samplesort algorithm $I1S^4o$.

Overall, $I1S^2Ra$ is significantly faster than our fastest radix sort competitor SkaSort. For example, $I1S^2Ra$ is for all instances at least a factor of 1.10 faster than SkaSort and for even 63 % of the instances more than a factor of 1.40. The radix sorter IppRadix is faster than $I1S^2Ra$ in some special cases. However, IppRadix is even slower than our competitor SkaSort for the remaining inputs. Our algorithm $I1S^2Ra$ also outperforms the comparison-based sorting algorithms for Uniform inputs and Skewed inputs significantly. For example, $I1S^2Ra$ is faster for all of these instances and for 56 % of these instances even a factor of 1.20 or more. Only for Almost Sorted inputs and the "easy" (Reverse) Sorted inputs, the comparison-based algorithms BlockPDQ and Timsort are faster than $I1S^2Ra$. For the remaining inputs – Uniform inputs and Skewed inputs – not only our radix sorter $I1S^2Ra$ but also our samplesort algorithm $I1S^4o$ is faster than all comparison-based competitors (except for one instance). For example, $I1S^4o$ is faster than our fastest comparison-based competitor, BlockPDQ by a factor of 1.10 and 1.20 for 25 respectively 15 out of 26 instances with Uniform and Skewed input. $I1S^4o$ is on average also faster than the fastest radix sorter.

Type	Distribution	$\mathrm{I1S^4o}$	BlockPDQ	BlockQ	$\mathrm{1S^4o}$	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	LearnedSort	$\mathrm{IPS^2Ra}$
double	Sorted	1.05	1.70	25.24	**1.05**	12.90	20.49	1.09	62.42	2.81	21.83	62.61	56.19	
double	ReverseSorted	**1.04**	1.71	14.28	1.06	5.09	5.93	1.07	25.34	5.89	9.41	25.22	21.24	
double	Zero	**1.07**	1.77	21.20	1.10	1.20	14.98	1.08	2.72	3.58	16.36	24.23	15.08	
double	Exponential	**1.02**	1.13	1.28	1.27	2.30	2.57	4.23	4.04	4.20	1.29	1.38	2.08	
double	Zipf	**1.08**	1.25	1.42	1.37	2.66	2.87	4.63	4.21	4.72	1.17	1.28	2.30	
double	RootDup	**1.10**	1.50	1.83	1.65	1.44	2.30	1.32	6.01	3.12	1.90	2.69	3.18	
double	TwoDup	1.17	1.33	1.37	1.41	2.48	2.65	2.96	3.42	3.20	**1.07**	1.22	2.52	
double	EightDup	**1.01**	1.13	1.41	1.30	2.42	2.84	4.43	4.69	4.40	1.31	1.60	2.46	
double	AlmostSorted	2.33	1.15	2.21	2.99	1.68	1.80	**1.14**	6.76	2.57	2.39	4.53	4.88	
double	Uniform	1.08	1.21	1.22	1.28	2.35	2.43	3.59	2.98	3.58	**1.08**	1.29	2.07	
Total		**1.20**	1.24	1.50	1.54	2.15	2.47	2.81	4.42	3.61	1.40	1.77	3.00	
Rank		1	2	4	5	7	8	9	12	11	3	6	10	
uint64	Sorted	1.17	1.78	23.69	**1.02**	11.94	19.96	1.11	55.40	2.88	26.64	76.86	95.80	13.33
uint64	ReverseSorted	**1.03**	1.63	12.93	1.04	4.47	5.51	1.04	21.01	5.93	10.46	28.99	34.39	5.97
uint64	Zero	1.17	1.69	21.43	**1.06**	1.14	14.02	1.11	2.42	3.74	17.40	25.30	14.90	1.35
uint64	Exponential	1.06	1.22	1.37	1.37	2.28	2.64	4.52	3.82	4.51	1.21	1.74	2.09	**1.05**
uint64	Zipf	1.53	1.86	2.13	2.06	3.62	4.04	6.65	5.53	6.79	1.73	1.99	2.56	**1.01**
uint64	RootDup	1.25	1.73	2.19	2.07	1.60	2.60	1.70	6.34	3.91	2.08	2.88	3.60	**1.13**
uint64	TwoDup	1.73	2.07	2.11	2.17	3.56	3.88	4.54	4.65	4.93	1.58	2.66	3.32	**1.00**
uint64	EightDup	1.26	1.39	1.74	1.64	2.75	3.29	5.46	5.12	5.38	1.71	2.97	2.40	**1.02**
uint64	AlmostSorted	2.34	**1.11**	2.19	3.28	1.68	1.81	1.24	6.11	2.79	2.79	6.67	8.55	1.28
uint64	Uniform	1.35	1.60	1.60	1.71	2.85	3.02	4.62	3.49	4.63	1.20	2.19	4.46	**1.04**
Total		1.46	1.54	1.88	1.97	2.51	2.95	3.56	4.90	4.56	1.69	2.74	4.87	**1.07**
Rank		2	3	5	6	7	9	10	13	11	4	8	12	1
uint32	Sorted	2.44	3.89	57.73	2.42	28.63	53.53	**1.96**	139.13	6.34	46.41	44.93	275.54	29.91
uint32	ReverseSorted	1.40	2.06	17.70	1.47	6.09	8.37	**1.03**	29.37	5.57	10.08	20.92	53.61	7.28
uint32	Zero	2.30	3.71	59.44	2.28	2.28	37.19	**2.06**	6.19	8.98	24.29	14.03	40.53	3.05
uint32	Exponential	1.49	1.77	2.03	1.82	3.66	4.04	6.67	5.91	6.51	1.38	**1.08**	3.44	1.09
uint32	Zipf	1.82	2.33	2.75	2.37	4.97	5.46	8.68	7.55	8.81	1.41	1.27	3.93	**1.12**
uint32	RootDup	1.41	1.92	2.46	2.15	1.84	2.97	1.48	7.54	3.78	1.58	1.77	4.43	**1.18**
uint32	TwoDup	2.09	2.56	2.67	2.52	4.82	5.11	5.59	5.94	5.95	1.34	1.44	5.08	**1.09**
uint32	EightDup	1.40	1.68	2.09	1.76	3.67	4.19	6.47	6.23	6.45	1.35	1.77	3.05	**1.02**
uint32	AlmostSorted	3.07	1.45	2.79	4.24	2.15	2.58	**1.06**	8.24	2.97	2.66	5.45	12.04	1.51
uint32	Uniform	1.67	2.01	2.05	2.04	3.85	4.02	5.92	4.55	5.79	1.39	**1.08**	5.22	1.20
Total		1.78	1.93	2.39	2.32	3.37	3.93	4.09	6.47	5.45	1.54	1.67	6.71	**1.16**
Rank		4	5	7	6	8	9	10	12	11	2	3	13	1
Pair	Sorted	1.06	1.62	16.88	**1.04**	9.36	14.67	1.04	34.54	2.30	17.51			10.48
Pair	ReverseSorted	1.13	1.21	8.47	**1.08**	3.65	4.19	1.12	13.71	6.60	6.86			4.87
Pair	Zero	1.09	1.61	13.30	**1.03**	1.07	11.63	1.08	1.94	2.71	11.09			1.21
Pair	Exponential	1.10	1.92	1.20	1.36	1.84	2.12	3.87	3.11		1.16			**1.05**
Pair	Zipf	1.48	2.72	1.64	1.86	2.64	2.83	5.02	3.87	5.50	1.46			**1.01**
Pair	RootDup	1.27	1.44	1.78	1.84	1.42	2.16	1.83	4.70	4.05	1.69			**1.03**
Pair	TwoDup	1.63	2.81	1.69	1.92	2.71	2.84	3.62	3.45	4.35	1.41			**1.01**
Pair	EightDup	1.27	2.19	1.45	1.59	2.14	2.47	4.50	3.95	4.81	1.56			**1.00**
Pair	AlmostSorted	3.20	**1.01**	2.79	4.00	2.18	2.39	2.34	6.56	4.55	3.24			1.74
Pair	Uniform	1.37	2.46	1.45	1.66	2.40	2.45	3.89	2.88	4.26	1.17			**1.03**
Total		1.52	1.97	1.66	1.91	2.15	2.45	3.40	3.94	4.50	1.57			**1.10**
Rank		2	6	4	5	7	8	9	10	11	3			1
Quartet	Uniform	1.14	1.85	1.29	1.49	1.89	1.86	3.14	2.15	3.52	**1.02**			
Rank		2	5	3	4	7	6	9	8	10	1			
100B	Uniform	1.41	1.27	1.27	1.64	1.83	1.33	2.22	1.78	3.17	**1.06**			
Rank		5	2	3	6	8	4	9	7	10	1			

Table 1. Average slowdowns of sequential algorithms for different data types and input distributions. The slowdowns average over the machines and input sizes with at least 2^{18} bytes.

Comparison to I1S^{2}Ra. I1S^{2}Ra outperforms **SkaSort** by a factor of 1.10, 1.20, 1.30, and 1.40 for respectively 100 %, 83 %, 73 %, and 63 % of the instances. Also, I1S^{2}Ra performs much better than **LearnedSort** for any data type and input distribution. **IppRadix** is the only non-comparison-based algorithm that is able to outperform I1S^{2}Ra for at least one instance, i.e., IppRadix is faster by a factor of 1.01 (of 1.11) for Exponential (Uniform) distributed inputs with the uint32 data type. However, IppRadix is (significantly) slower than I1S^{2}Ra for other Exponential (Uniform) instances, e.g., a factor of 1.66 (of 2.11) for the uint64 data type. For the remaining instances, IppRadix is (much) slower than I1S^{2}Ra.

For the comparison of I1S^{2}Ra to comparison-based algorithms, we first consider Uniform and Skewed inputs. For these instances, I1S^{2}Ra is significantly faster than all comparison-based algorithms (including I1S^{4}o). For example, I1S^{2}Ra is faster than any of these algorithms by a factor of more than 1.00, 1.10, 1.20, and 1.30 for respectively 100 %, 89 %, 78 %, and 56 % of the instances. We now consider Almost Sorted inputs which are sorted the fastest by **BlockPDQ**. For all 3 instances, I1S^{2}Ra is slower than BlockPDQ, i.e., by a factor of respectively 1.04, 1.16, and 1.72. The reason is that BlockPDQ heuristically detects and skips presorted input sequences. Even though BlockPDQ is our fastest comparison-based competitor, it is significantly slower than I1S^{2}Ra for many instances which are not (almost) sorted. For example, BlockPDQ is for 8 of these instances even more than a factor of 2.00 slower than I1S^{2}Ra. For (Reverse) Sorted inputs, I1S^{2}Ra is slower than at least one comparison-based sorting algorithm for all 9 instances. However, I1S^{2}Ra could easily detect these instances by scanning the input array once. We want to note that I1S^{2}Ra already scans the input array to detect the significant bits of the input keys.

Comparison to I1S^{4}o. Our algorithm I1S^{4}o is faster than any comparison-based competitor for 28 instances and slower for only 15 instances. However, when we exclude Almost Sorted inputs and (Reverse) Sorted inputs, I1S^{4}o is still faster for the same number of instances but the number of instances for which I1S^{4}o is slower drops to one instance. When we only exclude (Reverse) Sorted inputs, I1S^{4}o is still only slower for 5 instances.

BlockPDQ is a factor of 1.10, 1.15, 1.20, and 1.25 slower than I1S^{4}o for respectively 100 %, 71.43 %, 42.85 %, and 28.57 % out of 21 instances with Uniform input and Skewed input. BlockPDQ is also much slower for (Reverse) Sorted inputs. Only for Almost Sorted inputs, BlockPDQ is significantly faster than I1S^{4}o, e.g., by a factor of 2.03 to 3.17. Again, the reason is that BlockPDQ takes advantage of presorted sequences in the input.

BlockQ shows similar performance as BlockPDQ for Uniform inputs. The reason is that BlockPDQ reimplemented the partitioning routine proposed BlockQ [21]. However, BlockQ does not take advantage of presorted sequences and BlockQ handles duplicate keys less efficient. Thus, BlockQ is slower than BlockPDQ for (Reverse) Sorted inputs and Almost Sorted inputs.

I1S^{4}o outperforms **SkaSort** for Skewed inputs by a factor of at least 1.10 for 50 % out of 20 instances whereas SkaSort is faster by a factor of at least 1.10 for only 15 % of the instances. I1S^{4}o is also faster than SkaSort for all 12 (Reverse) Sorted inputs. For Almost Sorted inputs, both algorithms are for one instance at least a factor of 1.10 faster than the other algorithm (out of 4 instances). Only for Uniform inputs, SkaSort is the better algorithm. I.e., SkaSort is faster by a factor of at least 1.10 on 83 % out of 6 Uniform instances whereas I1S^{4}o is not faster on one of these instances.

As expected, **1S^{4}o** is slower than I1S^{4}o for all instances except (Reverse) Sorted inputs. For (Reverse) Sorted inputs, both algorithms execute the same heuristic to detect and sort "easy" inputs. Also, as 1S^{4}o is not in-place, 1S^{4}o can sort only about half the input size as I1S^{4}o can sort. The results strongly indicate that the I/O complexity of I1S^{4}o has smaller constant factors than the I/O complexity of 1S^{4}o as both algorithms share the same sorting framework including the same sampling routine, branchless decision tree, and base case sorting algorithm.

Fig. 11. Running times of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, WikiSort, and LearnedSort cannot be seen as their running times exceed the plot.

The algorithms **std::sort**, **DualPivot**, and BlockPDQ are adaptions of quicksort. However, std::sort and DualPivot do not avoid branch mispredictions by classifying and moving blocks of elements. The result is that these algorithms are always significantly slower than BlockPDQ.

We also compare $I1S^4o$ to the mergesort algorithms Timsort, QMSort, and WikiSort. The in-place versions of mergesort, **QMSort** and **WikiSort**, are significantly slower than $I1S^4o$ for all instances. **Timsort** is also much slower than $I1S^4o$ for almost all input distributions – in most cases even more than a factor of three. Only for Almost Sorted inputs, Timsort is faster than $I1S^4o$ and for (Reverse) Sorted inputs, Timsort has similar running times as $I1S^4o$.

We did not present the results of S^4oS, an implementation of Super Scalar Samplesort [64]. We made this decision as S^4oS is for all instances except 100B instances slower or significantly slower than $1S^4o$, our implementation of Super Scalar Samplesort. For further details, we refer to Table 11 in Appendix C which shows average slowdowns of $1S^4o$ and S^4oS for different data types and input distributions. We did not the present results of the sequential version of ASPaS for three reasons. First, ASPaS performs worse than $1S^4o$ for all instances. Second, ASPaS only sorts inputs with the data type double. Finally, ASPaS returns unsorted output for inputs with at least 2^{31} elements.

7.2.2 Running Times for Uniform Input. In this section, we compare $I1S^2Ra$ and $I1S^4o$ to their closest sequential competitors for Uniform distributed uint64 inputs. Figure 11 depicts the running

times of our algorithms I1S^{4}o and I1S^{2}Ra as well as their fastest competitors BlockPDQ and SkaSort separately for each machine. Additionally, we include measurements obtained from 1S^{4}o (which we used as a starting point to develop I1S^{4}o) and IppRadix (which is fast for uint32 data types with Uniform distribution). We decided to present results for uint64 inputs as our radix sorter does not support double inputs. We note that this decision is not a disadvantage for our fastest competitors as they show similar running times relative to our algorithms for both data types (see slowdowns in Table 1, Section 7.2.1).

Overall, I1S^{2}Ra outperforms its radix sort competitors on all but one machine, and I1S^{4}o significantly outperforms its comparison-based competitors. In particular, I1S^{2}Ra is a factor of up to 1.40 faster than SkaSort, and I1S^{4}o is a factor of up to 1.44 (1.60) faster than BlockPDQ (1S^{4}o) for the largest input size. As expected, I1S^{2}Ra is in almost all cases significantly faster than I1S^{4}o on all machines, e.g., a factor of 1.10 to 1.52 for the largest input size. I1S^{2}Ra shows the fastest running times on the two machines with the most recent CPUs, A1x16 and A1x64. On these two machines, the gap between I1S^{2}Ra and I1S^{4}o is larger than on the other machines. This indicates that sequential comparison-based algorithms are not memory bound in general, and, on recent CPUs, radix sorters may benefit even more from their reduced number of instructions (for uniformly distributed inputs). In the following, we compare our algorithms to their competitors in more detail.

Comparison to I1S^{2}Ra. Our algorithm I1S^{2}Ra outperforms **SkaSort** on two machines significantly (A1x16 and A1x64), on one machine slightly (I4x20), and on one machine (I2x16), I1S^{2}Ra is slightly slower than SkaSort. For example, I1S^{2}Ra is on average a factor of respectively 2.06, 2.13, 1.12, and 0.82 faster than SkaSort for $n \geq 2^{15}$ on A1x16, A1x64, I4x20, and I2x16. According to the performance measurements, obtained with the Linux tool perf, SkaSort performs more cache misses (factor 1.25) and significantly more branch mispredictions (factor 1.52 for $n = 2^{28}$ on I4x20).

On machine A1x16, I2x16, and A1x64, we see that the running times of SkaSort and I1S^{2}Ra vary – with peaks at 2^{15}, 2^{23} and 2^{31}. We assume that the running time peaks as these radix sorters perform an additional k-way partitioning step with $k = 256$. We have seen the same behavior with our algorithm I1S^{4}o when we do not adjust k at the last recursion levels. However, with our adjustments, the large running time peaks disappear for I1S^{4}o.

We also compare our algorithm against **IppRadix** which takes advantage of the *Advanced Vector Extensions* (AVX). All machines support the instruction extension AVX2. I4x20 additionally provides AVX-512 instructions. We expected that IppRadix is competitive, at least on I4x20. However, IppRadix is significantly slower than I1S^{2}Ra on all machines. For example, I1S^{2}Ra outperforms IppRadix by a factor of 1.76 to 1.88 for the largest input size on A1x64, A1x16, and I4x20. On I2x16, I1S^{2}Ra is even a factor of 3.00 faster. We want to note that IppRadix is surprisingly fast for (mostly small) Uniform distributed inputs with data type uint32 (see Fig. 18 in Appendix C). Unfortunately, IppRadix fails to sort uint32 inputs with more than 2^{28} elements. In conclusion, it seems that AVX instructions only help for inputs whose data type size is very small, i.e., 32-bit unsigned integers in our case.

Contrary to the experiments presented by Kristo et al. [48], the running times of LearnedSort are very large and would break the running time limits of Fig. 11. Our experiments have shown that the performance of LearnedSort degenerates by orders of magnitude for input sizes which are not a multiple of 10^6. This problem has been identified by others and was still an open issue [47] at the time when we finished our experiments.

Comparison to I1S^{4}o. For most medium and large input sizes, **BlockPDQ** and **1S^{4}o** are significantly slower than I1S^{4}o. For example, on A1x16, I1S^{4}o is a factor of 1.29 faster than 1S^{4}o and a factor of 1.44 faster than BlockPDQ for the largest input size ($n = 2^{32}$). On the other machines,

Fig. 12. Pairwise performance profiles of our algorithms I1S^{4}o and I1S^{2}Ra to BlockPDQ and SkaSort. The performance plots with I1S^{4}o use all data types. The performance plots with the radix sort algorithm I1S^{2}Ra use inputs with with unsigned integer keys (uint32, uint64, and Pair data types). The results were obtained on all machines for all input distributions with at least 2^{18} bytes except Sorted, ReverseSorted, and Zero.

BlockPDQ is our closest competitor: I1S^{4}o is a factor of 1.23 to 1.44 (of 1.29 to 1.60) faster than BlockPDQ (1S^{4}o) for $n = 2^{32}$. According to the performance measurements, obtained with the Linux tool perf, there may be several reasons why I1S^{4}o outperforms BlockPDQ and 1S^{4}o. Consider the machine I4x20 and $n = 2^{38}$: BlockPDQ performs significantly more instructions (factor 1.30), more cache misses (factor 2.05), and more branch mispredictions (factor 1.69) compared to I1S^{4}o. Also, 1S^{4}o performs significantly more total cache misses (factor 1.79), more L3-store operations (factor 2.68), and more L3-store misses (factor 9.71). We note that the comparison-based competitors Du-alPivot, std::sort, Timsort, QMSort, and WikiSort perform significantly more branch mispredictions than 1S^{4}o, BlockPDQ, and BlockQ. We think that this is the reason for their poor performance.

7.2.3 Comparison of Performance Profiles. In this section, we discuss the pairwise performance profiles, shown in Fig. 12, of our algorithms I1S^{4}o and I1S^{2}Ra to the fastest comparison-based competitor (BlockPDQ) and the fastest radix sort competitor (SkaSort).

Overall, I1S^{2}Ra has a significantly better profile than BlockPDQ and SkaSort. The performance profile of I1S^{4}o is slightly better than the profile of SkaSort and significantly better than the one of BlockPDQ. Exceptions are AlmostSorted inputs for which I1S^{4}o is much slower than BlockPDQ.

Comparison to I1S^{2}Ra. For the profiles containing I1S^{2}Ra, we used only inputs with unsigned integer keys. The performance profile of I1S^{2}Ra is significantly better than the profile of **SkaSort**. I1S^{2}Ra is much faster for most of the inputs and for the remaining inputs only slightly slower. For example, I1S^{2}Ra sorts 84 % of the inputs faster than SkaSort. Also, I1S^{2}Ra sorts 91 % of the inputs at least a factor of 1.25 faster than SkaSort. SkaSort on the other hand sort only 34 % of the inputs at most a factor of 1.25 faster. The performance profile of **BlockPDQ** is even worse than the profile of SkaSort. For example, I1S^{2}Ra sorts 97 % of the inputs at least a factor of 1.25 faster than BlockPDQ. BlockPDQ on the other hand sort only 26 % of the inputs at most a factor of 1.25 faster.

Comparison to I1S^{4}o. The performance profile of I1S^{4}o is in most ranges significantly better than the profile of **BlockPDQ**. For example, I1S^{4}o sorts 79 % of the inputs faster than BlockPDQ. Also, BlockPDQ sorts only 60 % of the inputs at least a factor of 1.25 faster than I1S^{4}o whereas I1S^{4}o sorts 86 % of the inputs at least a factor of 1.25 faster than BlockPDQ. We note that I1S^{4}o is significantly slower than BlockPDQ for some inputs. These inputs are AlmostSorted inputs. The performance profile of I1S^{4}o is slightly better than the profile of **SkaSort**. For example, I1S^{4}o sorts 54 % of the

inputs faster than SkaSort. Also, I1S^{4}o (SkaSort) sorts 83 % (77 %) of the inputs at least a factor of 1.25 faster.

7.3 Influence of the Memory Allocation Policy

On NUMA machines, access to memory attached to the local NUMA node is faster than memory access to other nodes. Thus, the memory access pattern of a shared-memory algorithm may highly influence its performance. For example, an algorithm can greatly benefit by minimizing the memory access of its threads to other NUMA nodes. However, we cannot avoid access to non-local NUMA nodes for shared-memory sorting algorithms: For example, when the input array is distributed among the NUMA nodes, the input- and output-position of elements may be on different nodes. In this case, it can be an advantage to distribute memory access evenly across the NUMA nodes to utilize the full memory bandwidth. Depending on the access patterns of an algorithm, a memory layout may suit a parallel algorithm better than another. If we do not consider different memory layouts of the input array in our benchmark, the results may wrongly indicate that one algorithm is better than another.

The memory layout of the input array depends on the *NUMA allocation policy* of the input array and former access to the array. The *local allocation policy* allocates memory pages at the thread's local NUMA node if memory is available. This memory policy is oftentimes the default policy. Note that after the user has allocated memory with this policy, the actual memory pages are not allocated until a thread accesses them the first time. A memory page is then allocated on the NUMA node of the accessing thread. This principle is called *first touch*. The *interleaved allocation policy* pins memory pages round-robin to (a defined set of) NUMA nodes. The *bind allocation policy* binds memory to a defined set of NUMA nodes and *preferred allocation* allocates memory on a preferred set of NUMA nodes. For example, a user could create an array with the bind allocation policy such that the i'th stripe of the array is pinned to NUMA node i.

Benchmarks of sequential algorithms usually allocate and initialize the input array with a single thread with the default allocation policy (local allocation). The memory pages of the array are thus all allocated on a single NUMA node (*local arrays*). Local arrays are slow for many parallel algorithms because the NUMA node holding the array becomes a bottleneck. It is therefore recommended to use a different layout for parallel (sorting) algorithms. For example, the authors of RADULS2 recommend to use an array where the i'th stripe of the array is first touched by thread i (*striped array*). Another example is RegionSort for which the authors recommend to invoke the application with the interleaved allocation policy. We call arrays of those applications *interleaved arrays*. Orestis and Ross allocate for their benchmarks [61] on machines with m NUMA nodes m subarrays where subarray i is pinned to NUMA node i.

We execute the benchmark of each algorithm with the following four input array types.

- For the *local array*, we allocate the array with the function `malloc` and a single thread initializes the array.
- For the *striped array*, we allocate the array with `malloc` and thread i initializes the i'th stripe of the input array.
- For the *interleaved array*, we activate the process-wide interleaved allocation policy using the Linux tool `numactl`.
- The *NUMA array* [3] uses a refined NUMA-aware array proposed by Lorenz Hübschle-Schneider[6]. The NUMA array pins the stripe i of the array to NUMA node i. This approach is similar to the array used by Orestis and Ross except that the NUMA array is a continuous array.

---

[6]https://gist.github.com/lorenzhs

	A1x16				A1x64				I2x16				I4x20			
	LA	IA	SA	NA	LA	IA	SA	NA	LA	IA	SA	NA	LA	IA	SA	NA
ASPaS	1.01	1.00	1.00	**1.00**	**1.00**	1.01	1.01	1.01	1.22	1.05	**1.01**	1.02	4.59	1.11	**1.00**	1.39
MCSTLmwm	**1.00**	1.01	1.01	1.01	**1.00**	1.02	1.01	1.02	1.13	**1.03**	1.06	1.03	2.28	**1.02**	1.16	1.18
MCSTLbq	1.02	1.03	1.02	**1.01**	1.04	1.05	**1.02**	1.04	1.49	**1.00**	1.08	1.02	3.67	**1.01**	1.28	1.30
IPS^{4}o	1.01	**1.00**	1.01	1.03	1.01	1.01	1.01	**1.00**	1.27	**1.00**	1.12	1.01	3.43	**1.00**	1.32	1.08
PBBS	1.01	**1.00**	1.00	1.00	1.01	1.00	**1.00**	1.01	1.09	**1.00**	1.03	1.01	1.47	**1.00**	1.17	1.12
PS^{4}o	1.01	**1.00**	1.01	1.02	**1.00**	1.00	1.01	1.01	1.13	**1.00**	1.11	1.04	2.28	**1.01**	1.19	1.23
TBB	1.02	1.02	**1.01**	1.02	1.02	**1.01**	1.01	1.02	1.10	1.02	1.09	**1.01**	1.12	**1.03**	1.12	1.05
IPS^{2}Ra	1.01	1.01	**1.01**	1.02	1.01	**1.00**	1.01	1.02	1.45	1.02	1.14	**1.00**	4.88	**1.01**	1.45	1.04
PBBR	**1.00**	1.01	1.01	1.00	1.03	**1.01**	1.01	1.02	1.11	**1.01**	1.03	1.04	2.33	**1.01**	1.27	1.44
RADULS2	**1.00**	1.01	1.01	1.01	1.08	1.09	1.09	**1.00**	1.23	**1.01**	1.03	1.09	4.80	**1.01**	1.53	2.86
RegionSort	1.00	1.01	1.01	**1.00**	**1.00**	1.01	1.01	1.01	1.28	**1.00**	1.07	1.05	4.18	**1.04**	1.22	1.36

Table 2. Average slowdowns of the local array (LA), the interleaved array (IA), the striped array (SA), and the NUMA array (NA) for different parallel sorting algorithms on different machines. We only consider uint64 data types with at least $2^{21}t$ bytes and input distribution Uniform.

Table 2 shows the average slowdown of each array type for each algorithm on our machines. As expected, the machines with a single CPU, A1x16 and A1x64, do not benefit from NUMA allocations. On the NUMA machines, the local array performs significantly worse than the other arrays. Depending on the algorithm, the average slowdown of the local array is a factor of up to 1.49 larger than the average slowdown of the respectively best array on I2x16. On I4x20, the local array performs even worse: Depending on the algorithm, the average slowdown of the local array is a factor of 1.12 to 4.88 larger.

The interleaved array (significantly) outperforms the other arrays for most algorithms or shows similar slowdown factors (± 0.02) on the NUMA machines, i.e. I2x16 and I4x20. Only ASPaS is on these machines with the striped array noticeable faster than with the interleaved array. However, ASPaS shows large running times in general. On I2x16, the NUMA machine with 32 cores, the average slowdown ratios of the striped array and the NUMA array to the interleaved array are relatively small (up to 1.12). On I4x20, which is equipped with 80 cores, the average slowdown ratio of the striped array (NUMA array) to the interleaved array increases in the worst case to 1.44 (to 2.83).

Our algorithm IPS^{4}o has almost the same average slowdowns when we execute the algorithm with the interleaved or the NUMA array. Other algorithms, e.g., our closest competitors RADULS2 and RegionSort, are much slower on I4x20 when executed with the NUMA array. The reason is that a thread of our algorithm predominantly works on one stripe of the input array allocated on a single NUMA node.

In conclusion, the local array should not be used on NUMA machines. The interleaved array is the best array on these machines with just a few minor exceptions. The NUMA array and the striped array perform better than the local array on NUMA machines and in most cases worse than the interleaved array. Unless stated otherwise, we report results obtained with the interleaved input array.

7.4 Evaluation of the Parallel Task Scheduler

The version of IPS^{4}o proposed in the conference article of this publication (IPS^{4}oNT) [6] uses a very simple task scheduling. I.e., tasks with more than n/t elements are all executed with t threads (so-called parallel tasks) and sequential tasks are assigned to threads greedily in descending order according to their size. The task scheduler of IPS^{4}o, described in Section 4.2, has three advantages. First, the number of threads processing a parallel task decreases as the size of the task decreases. This

means that we can process small parallel subtasks more efficiently. Second, voluntary work sharing is used to balance the load of sequential tasks between threads. Finally, thread i predominantly accesses elements from $A[in/t .. (i + 2)n/t - 1]$ in sequential tasks and in classification phases (see Lemmas 4.5 and 4.6). Thus, the access pattern of IPS^{4}o significantly reduces memory access to the nonlocal NUMA nodes when the striped array or the NUMA array is used.

Table 3 compares IPS^{4}o with IPS^{4}oNT. On machines with multiple NUMA nodes, i.e., I2x16 and I4x20, both algorithms are much slower when the local array is used. This is not surprising as the input is read in this case from a single NUMA node. On machine I4x20, I2x16, and A1x16, IPS^{4}o shows a slightly smaller average slowdown than IPS^{4}oNT for the same array type. The improvements come from the voluntary work sharing. Both algorithms do not execute parallel subtasks as $t \ll k$.

In the remainder of this section, we discuss the results obtained on machine I4x20. These results are perhaps the most interesting: Compared to the other machines, on I4x20 tasks with more than n/t elements occur regularly on the second recursion level of the algorithms as the number of threads is only slightly smaller than k. Thus, both algorithms actually perform parallel subtasks. In contrast to IPS^{4}oNT, IPS^{4}o uses thread groups whose size is proportional to the size of parallel tasks. Thus, we expect IPS^{4}o to be faster than IPS^{4}oNT for any array type.

We want to point out that the advantage of IPS^{4}o is caused by the handling of its parallel subtasks, not by the voluntary work sharing: When no parallel subtasks are executed, the running times do not differ much. However, our experiments show that IPS^{4}o performs much better than IPS^{4}oNT in cases where parallel tasks occur on the second recursion level. We now discuss the running time improvements separately for each array type.

With the interleaved array, IPS^{4}o reports the fastest running times. For this array, the average slowdown ratio of IPS^{4}oNT to IPS^{4}o is 1.13. For the interleaved array, we expect that parallel subtasks oftentimes cover multiple memory pages. Thus, both algorithms can utilize the bandwidth of multiple NUMA nodes when executing parallel subtasks. We assume that this is the reason that IPS^{4}oNT is not much slower than IPS^{4}o with interleaved arrays. For the NUMA array the average slowdown ratio increases to 2.54 – IPS^{4}oNT becomes much slower. The reason for this slowdown is that the subarray associated with a parallel subtask will often reside on a single NUMA node. IPS^{4}oNT executes such tasks with all t threads which then leads to a severe memory bottleneck. Additionally, subtasks of this task can be assigned to any of these threads. IPS^{4}o on the other hand executes the task with a thread group of appropriate size. Threads of this thread group also process resulting subtasks (unless they are rescheduled to other threads).

Let us now compare the striped array with the NUMA array. While IPS^{4}oNT exhibits about the same (bad) performance with both arrays, IPS^{4}o becomes 22 % slower when executed with the striped array (but still almost twice as fast as IPS^{4}oNT). A reason for the slowdown of IPS^{4}o might be that the striped array does not pin memory pages. Thus, during the block permutation, many memory pages are moved to other NUMA nodes. This is counterproductive since they are later accessed by threads on yet another NUMA node.

If a local array is used, the NUMA node holding it becomes a severe bottleneck – both IPS^{4}oNT and IPS^{4}o become several times slower. IPS^{4}o suffers less from this bottleneck (slowdown factor 1.09 rather than 1.51 for IPS^{4}oNT), possibly because a thread i of IPS^{4}o accesses a similar array stripe in a child task T' as in a parent task T. Thus, during the execution of T, some memory pages used by T' might be migrated to the NUMA node of i (recall that local arrays are not pinned).

In conclusion, IPS^{4}o is (much) faster than IPS^{4}oNT for any array type tested here. IPS^{4}o shows the best performance for the interleaved array and the NUMA array, with the interleaved array performing slightly better. Both arrays allocate memory pages distributed among the NUMA nodes,

	local array		interleaved array		striped array		NUMA array	
	ips^{4}oNT	ips4o	ips^{4}oNT	ips4o	ips^{4}oNT	ips4o	ips^{4}oNT	ips4o
A1x16	1.07	1.00 (3.62)	1.06	1.00 (3.59)	1.06	1.00 (3.61)	1.05	1.00 (3.67)
A1x64	1.04	1.00 (4.47)	1.04	1.00 (4.46)	1.04	1.00 (4.47)	1.05	1.00 (4.44)
I2x16	1.03	1.01 (5.65)	1.02	1.01 (4.47)	1.04	1.04 (5.01)	1.02	1.01 (4.52)
I4x20	1.51	1.09 (17.46)	1.13	1.00 (5.22)	1.84	1.00 (6.90)	2.54	1.00 (5.66)

Table 3. This table shows the average slowdown of IPS^{4}o and IPS^{4}oNT to the best of both algorithms for different array types and machines. The numbers in parentheses show the average running times of IPS^{4}o divided by $n/t \log_2 n$ in nanoseconds. We only consider uint64 data types with at least $2^{21}t$ bytes and input distribution Uniform.

and, compared to the striped array, pin the memory pages to NUMA nodes. For these arrays, the average slowdown ratio of IPS^{4}oNT to IPS^{4}o is between 1.13 and 2.54.

7.5   Parallel Algorithms

In this section, we compare parallel algorithms for different machines, input distributions, input sizes, and data types. We begin with a comparison of the average slowdowns of IPS^{4}o, IPS^{2}Ra, and their competitors for ten input distributions executed with six different data types (see Section 7.5.1). This gives a first general view of the performance of our algorithms as the presented results are aggregated across all machines. Afterwards, we compare the algorithms for input distribution Uniform with data type uint64 on different machines: We consider scaling with input sizes in Section 7.5.2 and scaling with the number of utilized cores in Section 7.5.3. Then, we discuss in Section 7.5.4 the running times for an interesting set of input distributions and data types, again by scaling the input size. In Section 7.5.5, we discuss the performance profiles of our algorithms and their most promising competitors. Finally, we separately compare IPS^{4}o to IMSDradix, which is only implemented in a very explorative manner and thus only works in some special cases (see Section 7.5.6).

7.5.1   Comparison of Average Slowdowns. Table 4 shows average slowdowns of parallel algorithms for different data types and input distributions aggregated over all machines and input sizes with at least $2^{21}t$ bytes. For the average slowdowns separated by machine, we refer to Appendix C, Tables 12 to 15. In this section, an *instance* describes the inputs of a specific data type and input distribution. We say that "algorithm A is faster than algorithm B (by a factor of C) for some instances" if the average slowdown of B is larger than the average slowdown of A (by a factor of C) for these instances.

Overall, the results show that IPS^{4}o is much faster than its competitors in most cases except some "easy" instances and some instances with uint32 data types. Except for some uint32 instances, IPS^{4}o is even significantly faster than its fastest radix sort competitor RegionSort. This indicates that parallel sorting algorithms are memory bound for most inputs, except for data types that only have a few bytes. In most cases, IPS^{4}o also outperforms our radix sorter IPS^{2}Ra. IPS^{2}Ra is faster for some instances with uint32 data types and, as expected, IPS^{2}Ra is faster for Uniform instances. IPS^{2}Ra has a better ranking than our fastest in-place radix sort competitor RegionSort. Thus, our approach of sorting data with parallel block permutations seems to perform better than the graph-based approach of RegionSort.

Comparison to IPS^{4}o. IPS^{4}o is the fastest algorithm for 30 out of 42 instances. IPS^{4}o is outperformed for 8 instances having "easy" input distributions, i.e, Sorted, ReverseSorted and Zero. For now

Type	Distribution	IPS^4o	PBBS	PS^4o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS^2Ra
double	Sorted	1.42	10.96	2.02	15.47	13.36	**1.06**				42.23	
double	ReverseSorted	**1.06**	1.34	1.98	1.76	11.00	3.01				5.34	
double	Zero	1.54	12.83	1.80	14.55	166.67	**1.06**				41.78	
double	Exponential	**1.00**	1.82	1.97	2.60	3.20	10.77				4.97	
double	Zipf	**1.00**	1.96	2.12	2.79	3.55	11.56				5.33	
double	RootDup	**1.00**	1.54	2.22	2.52	3.88	5.54				6.28	
double	TwoDup	**1.00**	1.93	1.88	2.45	2.99	5.52				4.44	
double	EightDup	**1.00**	1.82	2.01	2.48	3.19	10.37				5.02	
double	AlmostSorted	**1.00**	1.73	2.40	5.12	2.18	3.54				6.37	
double	Uniform	**1.00**	2.00	1.85	2.53	2.99	9.16				4.39	
Total		**1.00**	1.82	2.06	2.83	3.10	7.46				5.21	
Rank		1	2	3	4	5	7				6	
uint64	Sorted	1.45	10.56	1.80	15.65	13.50	**1.09**	6.72	56.24	33.08		8.83
uint64	ReverseSorted	**1.17**	1.42	2.23	2.01	12.27	3.40	1.34	8.07	4.65		1.76
uint64	Zero	1.69	13.58	1.87	15.02	171.86	**1.13**	1.36	51.61	32.50		1.16
uint64	Exponential	**1.04**	1.74	2.10	2.62	3.41	10.38	1.79	1.58	2.58		1.20
uint64	Zipf	**1.00**	1.82	2.16	2.69	3.60	10.48	1.61	16.80	6.04		1.68
uint64	RootDup	**1.00**	1.47	2.24	2.52	3.84	5.78	1.59	9.89	7.00		1.54
uint64	TwoDup	**1.07**	1.91	2.04	2.54	3.20	5.83	1.30	10.00	3.89		1.34
uint64	EightDup	**1.02**	1.69	2.06	2.42	3.25	9.54	1.37	12.45	5.00		1.44
uint64	AlmostSorted	**1.11**	1.88	2.73	5.75	2.54	4.15	1.36	9.84	5.87		1.55
uint64	Uniform	1.13	2.10	2.14	2.80	3.32	9.57	1.59	1.41	1.49		**1.03**
Total		**1.05**	1.79	2.20	2.91	3.28	7.54	1.51	6.17	4.07		1.38
Rank		1	4	5	6	7	10	3	9	8		2
uint32	Sorted	**1.77**	10.03	2.77	11.64	14.68	1.91	5.28	7.86			4.98
uint32	ReverseSorted	1.51	1.84	2.46	2.03	11.96	5.17	1.22	1.44			**1.17**
uint32	Zero	1.59	15.94	1.95	19.35	286.17	**1.18**	1.50	73.11			1.20
uint32	Exponential	1.31	2.85	2.34	3.68	4.55	17.62	1.57	2.02			**1.02**
uint32	Zipf	**1.05**	2.54	2.06	3.22	4.05	15.68	1.33	6.39			1.41
uint32	RootDup	**1.09**	1.78	2.26	2.62	3.92	6.16	1.37	7.50			1.42
uint32	TwoDup	1.40	3.18	2.32	3.59	4.35	9.10	1.24	1.83			**1.02**
uint32	EightDup	1.23	2.84	2.26	3.41	4.24	16.24	1.33	1.84			**1.08**
uint32	AlmostSorted	1.38	2.08	2.63	5.66	3.22	4.54	1.32	1.62			**1.08**
uint32	Uniform	1.41	3.26	2.28	3.68	4.45	14.52	1.36	1.61			**1.03**
Total		1.26	2.59	2.30	3.60	4.09	10.75	1.36	2.49			**1.14**
Rank		2	6	4	7	8	9	3	5			1
Pair	Sorted	1.39	9.38	1.82	15.05	15.50	**1.03**	5.75	20.15	52.30		8.02
Pair	ReverseSorted	**1.09**	1.47	2.06	2.22	10.46	3.15	1.35	3.21	8.24		1.77
Pair	Zero	1.66	14.10	1.77	15.21	118.30	**1.08**	1.21	11.71	54.52		1.16
Pair	Exponential	1.12	1.77	2.22	2.76	3.09	6.92	1.92	**1.07**	9.52		1.39
Pair	Zipf	**1.00**	1.62	2.04	2.53	2.79	6.30	1.62	7.35	9.87		1.77
Pair	RootDup	**1.01**	1.58	2.08	2.81	3.84	4.88	1.58	4.35	11.76		1.52
Pair	TwoDup	**1.02**	1.67	2.02	2.44	2.96	4.10	1.43	4.88	7.54		1.48
Pair	EightDup	**1.02**	1.59	2.05	2.41	2.83	6.01	1.40	6.98	8.81		1.57
Pair	AlmostSorted	**1.05**	1.95	2.69	5.67	3.24	3.88	1.37	4.27	10.94		1.65
Pair	Uniform	1.08	1.81	2.12	2.62	2.93	6.15	1.67	1.20	5.36		**1.04**
Total		**1.04**	1.71	2.16	2.90	3.08	5.35	1.56	3.46	8.87		1.47
Rank		1	4	5	6	7	9	3	8	10		2
Quartet	Uniform	**1.01**	1.29	2.08	2.40	2.93	4.42					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.05**	1.14	2.14	2.35	3.18	3.55					
Rank		1	2	3	4	5	6					

Table 4. Average slowdowns of parallel algorithms for different data types and input distributions. The slowdowns average over the machines and input sizes with at least $2^{21}t$ bytes.

on, we consider only these instances: TBB detects Sorted and Zero inputs as sorted and returns immediately. RegionSort detects that the elements of Zero inputs only have zero bits, and thus, also return immediately for Zero inputs. It is therefore not surprising that TBB and RegionSort sort easy inputs very fast. TBB (RegionSort) is for 4 (for 3) Zero instances better than IPS^{4}o. TBB is also better for 3 Sorted instances, i.e., with double, uint64, and Pair data types. Also, RegionSort is faster than IPS^{4}o for the ReverseSorted uint32 instance. Our algorithm also detects these instances but with a slightly larger overhead.

In this paragraph, we do not consider "easy" instances. IPS^{4}o is significantly faster than our competitors for 23 out of 30 instances (> 1.15). For 3 instances, IPS^{4}o performs similar (± 0.06) to **RegionSort** (AlmostSorted distributed uint32 instance and Uniform distributed uint32 instance) and **PBBR** (Exponential distributed Pair instance). RegionSort is the only competitor that is noticeably faster than IPS^{4}o, at least for one instance, i.e., TwoDup distributed uint32 inputs (factor 1.13). Overall, IPS^{4}o is faster than its respectively fastest competitor by a factor of 1.2, 1.4, 1.6, and 1.8 for 22, 13, 8, and 5 noneasy instances, respectively. If we only consider comparison-based competitors, IPS^{4}o is faster by a factor of 1.2, 1.4, 1.6, and 1.8 for 29, 28, 22, and 10 noneasy instances, respectively. The values become even better when we only consider in-place comparison-based competitors. In this case, the IPS^{4}o is faster by a factor of 2.15 for all noneasy instances.

IPS^{4}o is much faster than **PS^{4}o**. The only difference between these algorithms is that IPS^{4}o implements the partitioning routine in-place whereas PS^{4}o is non-in-place. We note that the algorithms share most of their code, even the decision tree is the same. The reason why PS^{4}o is slower than IPS^{4}o is that IPS^{4}o is more cache efficient than PS^{4}o: For example, PS^{4}o has about 46 % more L3-cache misses than IPS^{4}o for Uniform distributed uint64 inputs with 2^{27} elements whereas the number of instructions and the number of branch (misses) of PS^{4}o are similar to the ones of IPS^{4}o. The sequential results presented in Section 7.2 support this conjecture as the gap between the sequential versions is smaller than the gap between the parallel versions.

Comparison to IPS^{2}Ra. Our in-place radix sorter IPS^{2}Ra performs slightly better than our fastest competitor, **RegionSort**. IPS^{2}Ra is faster than RegionSort for 11 out of 21 noneasy instances. In particular, IPS^{2}Ra is faster by a factor of 1.2, 1.4, and 1.6 for 9, 4, and 1 noneasy instances and the factor is never smaller than 0.8.

IPS^{2}Ra outperforms **IPS^{4}o** for instances with uint32 data types and some Uniform distributed instances. For uint32 instances, IPS^{2}Ra is faster than IPS^{4}o by a factor of 1.37. Interestingly, IPS^{2}Ra is not much faster than IPS^{4}o for Uniform instances with more than 32-bit elements. This indicates that the evaluation of the branchless decision tree is not a limiting factor for these data types in IPS^{4}o. For the remaining instances (data types with more than 32-bit elements and instances which noneasy distributions other than Uniform) IPS^{4}o is significantly faster than IPS^{2}Ra.

From now on, we do not present results for ASPaS, TBB, PS^{4}o, and MCSTLmwm. In regard to non-in-place comparison-based competitors, the algorithms ASPaS, PS^{4}o, and MCSTLmwm perform worse than PBBS. For non-in-place comparison-based competitors, the parallel quicksort algorithm TBB is for noneasy instances slower than the quicksort implementation MCSTLbq.

7.5.2   Running Times for Uniform Input. In this section, we compare IPS^{4}o and IPS^{2}Ra to their closest parallel competitors for Uniform distributed uint64 inputs. Figure 13 depicts the running times separately for each machine. The results of the algorithms obtained for double inputs are similar to the running times obtained for uint64 inputs. We decided to present results for uint64 inputs as our closest parallel competitors for data types with "primitive" keys, i.e., RegionSort, PBBR, and RADULS2, do not support double inputs.

Fig. 13. Running times of parallel algorithms sorting uint64 values with input distribution Uniform executed on different machines.

We outperform all comparison-based algorithms significantly for medium and large input sizes, e.g., by a factor of 1.49 to 2.45 for the largest inputs depending on the machine. For in-place competitors, the factor is even 2.55 to 3.71. In general, all competitors are very inefficient for small input sizes except the non-in-place competitors PBBS and PBBR. However, the performance of PBBS and PBBR significantly decreases for larger inputs. Exploiting integers in IPS^{2}Ra slightly improves the performance for medium and large input sizes compared to IPS^{4}o. For small input sizes, exploiting integers makes IPS^{2}Ra more efficient than IPS^{4}o. Our radix sorters RADULS2 and RegionSort are only competitive for large input sizes. Still, they are very inefficient even for these input sizes on I4x20, our largest machine. In particular, they are 2.72 respectively 3.08 times slower than IPS^{2}Ra for the largest input size on this machine.

We sort twice as much data as our non-in-place competitors (PBBS, RADULS2, and PBBR) which run out of memory for 2^{32} elements on A1x16. Also, the results in Table 4, Section 7.5.1, show that

inputs with uint64 Uniform inputs are "best case" inputs for RADULS2. Other input distributions and data types are sorted by RADULS2 much less efficient.

Fig. 14. Speedup of parallel algorithms with different number of threads relative to our sequential implementation $I1S^4o$ on different machines, sorting 2^{30} elements of uint64 values with input distribution Uniform.

7.5.3 Speedup Comparison and Strong Scaling. The goal of the speedup benchmark is to examine the performance of the parallel algorithms with increasing availability of cores. Benchmarks with $2i$ threads are executed on the first i cores, starting at the first NUMA node until it is completely used. Then we continue using the cores of the next NUMA node, and so on. Here, we mean by cores "physical cores" that run two hardware threads on our machines and we use NUMA nodes as a synonym for CPUs.[7] In result, the benchmark always takes advantage of the "full capacity" of a core with hyper-threading. Preliminary experiments have shown that all algorithms are slowed down when we use only one thread per core.

Figure 14 depicts the speedup of parallel algorithms executed on different numbers of cores relative to our sequential implementation $I1S^4o$ on our machines for Uniform inputs. We first compare our algorithms to the non-in-place radix sorter **RADULS2**. This competitor is fast for Uniform inputs but it is slow for inputs with skewed key distributions and inputs with duplicated

---

[7]Many Linux tools interpret a CPU with two hardware threads per core as two distinct NUMA nodes – one contains the first hardware thread of each core and the other contains the second hardware threads of each core.

keys (see Table 4 in Section 7.5.1). On the machines with one CPU, A1x16 and A1x64, RADULS2 is faster when we use only a fraction of the available cores. When we further increase the available cores on these machines, the speedup of RADULS2 stagnates and our algorithms, IPS^{4}o and IPS^{2}Ra, catch up until they have about the same speedup. RADULS2 also outperforms all algorithms on our machine with four CPUs, I4x20, when the algorithms use only one CPU. On the same machine, the performance of RADULS2 stagnates when we expand the algorithm to more than one CPU. When RADULS2 uses all CPUs, it is even a factor of 2.54 slower than our algorithm IPS^{4}o. We have seen the same performance characteristics when we executed IPS^{4}oNT on this machine. IPS^{4}o solved this problem of IPS^{4}oNT with a more sophisticated memory and task management. Thus, we conclude that the same problems also result in performance problems for RADULS2. Our algorithms IPS^{4}o and IPS^{2}Ra use the memory on this machine more efficiently and do not get memory-bound – the speedup of our algorithms increases on I4x20 linearly.

The in-place radix sorter **RegionSort** seems to have similar problems as RADULS2 on I4x20. Even worse, the speedup of RegionSort stagnates on three out of four machines when the available cores increase. When all cores are used, the speedup of RegionSort is a factor of 1.11 to 2.70 smaller than the speedup of IPS^{2}Ra. On three out of four machines, our radix sorter **IPS^{2}Ra** has a larger speedup than our samplesort algorithm **IPS^{4}o** when we use only a few cores. For more cores, their speedups converge on two machines, even though IPS^{2}Ra performs significantly fewer instructions.

On the machines with one CPU, A1x16 and A1x64, IPS^{4}o has a speedup of 8.37 respectively 40.92. This is a factor of 1.46 respectively 1.85 more than the fastest **comparison-based competitor**. On the machine with four CPUs, I4x20, and on the machine with two CPUs, I2x16, the speedup of IPS^{4}o is 20.91 respectively 17.49. This is even a factor of 2.27 respectively 2.17 more than the fastest comparison-based competitor.

In conclusion, our in-place algorithms outperform their comparison-based competitors significantly on all machines independently of the number of assigned cores. For example, IPS^{4}o yields a speedup of 40.92 on the machine A1x64 whereas PBBS only obtains a speedup of 22.17. As expected, the fastest competitors for the (Uniform) input used in this experiment are radix sorters. The fastest radix sort competitor, non-in-place RADULS2, starts with a very large speedup when only a few cores are in use. For more cores, RADULS2 remains faster than our algorithms on one machine (I2x16). On two machines (A1x64 and A1x16), the speedup of RADULS2 converges to the speedups of our algorithms. And, on our largest machine with four CPUs (I4x20), the memory management of RADULS2 seems to be not practical at all. On this machine, RADULS2 is even a factor of 2.54 slower than IPS^{4}o. The in-place radix sort competitor RegionSort is in all cases significantly slower than our algorithms. The speedup of IPS^{2}Ra is larger than the one of IPS^{4}o when they use only a few cores of the machine. However, the speedup levels out when the number of cores increases in most cases.

7.5.4 Input Distributions and Data Types. In this section, we compare our algorithms to our competitors for different input distributions and data types by scaling the input size. We show results of Uniform inputs for the data types double, Pair, and 100B. For a discussion of Uniform distributed uint64 data types, we refer to Section 7.5.2. For the remaining input distributions, we use the data type uint64 as a convenient example: In contrast to double, uint64 is supported by all algorithms in Fig. 15. Additionally, we assume that uint64 is more interesting than uint32 in practice. We decided to present in Fig. 15 results obtained on machine A1x64 as our competitors have the smallest absolute running time on this machine. For more details, we refer to Figs. 19 to 22 in Appendix C which report the results separately for each machine.

For many inputs, our IPS^{4}o is faster than IPS^{2}Ra. For most inputs, IPS^{4}o (and to some extend IPS^{2}Ra) is much faster than RegionSort, our closest competitor. For example, IPS^{4}o is up to a factor

Fig. 15. Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x64. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

of 1.61 faster for the largest inputs ($n = 2^{37}/D$) and up to a factor of 1.78 for inputs of medium size ($n = 2^{29}/D$). The results show that radix sorters are often slow for inputs with many duplicates or skewed key distributions (i.e., Zipf, Exponential, EightDup, RootDup). Yet, our algorithm seems to be the least affected by this. Our algorithms outperform their comparison-based competitors significantly for all input distributions and data types with $n \geq 2^{28}/D$. For example, IPS^{4}o outperforms PBBS by a factor of 1.25 to 2.20 for the largest inputs. Only for small inputs, where the algorithms are inefficient anyway, our algorithms are consistently outperformed by one algorithm (non-in-place PBBS). The remainder of this section compares our algorithms and their competitors in detail.

The non-in-place comparison-based **PBBS** is slower than IPS^{4}o for small inputs ($n \leq 2^{27}/D$). We note that all algorithms are inefficient for these small inputs. However, for inputs where the algorithms become efficient and for large inputs, IPS^{4}o significantly outperforms PBBS. For example, PBBS is a factor of 1.25 to 2.20 slower than IPS^{4}o for the largest input size. The difference between PBBS and IPS^{4}o is the smallest for 100B inputs. This input has very large elements which are moved only twice by PBBS due to its $\sqrt{n}$-way partitioning strategy. We see this as an important turning point. While the previous state-of-the-art comparison-based algorithm worked non-in-place, it is now robustly outperformed by our in-place algorithms for inputs that are sorted efficiently by parallel comparison-based sorting algorithms.

The in-place comparison-based **MCSTLbq** is significantly slower than our algorithms for all inputs. For example, MCSTLbq is a factor of 2.46 to 3.87 slower than IPS^{4}o for the largest input size. We see this improvement as a major contribution of our paper.

The non-in-place radix sorter **PBBR** is tremendously slow for all inputs with skewed inputs and inputs with identical keys. In particular, its running times exceed the limits of Fig. 15 for AlmostSorted, RootDup, TwoDup, and Zipf inputs. Exceptions are Uniform inputs with Pair data type: For these inputs, PBBR is faster than our algorithms for small input sizes and performs similar for medium and large inputs. However, this advantage disappears for other uniformly distributed inputs (see Table 4 in Section 7.5.1).

The non-in-place radix sorter **RADULS2** is a factor of 2.20 to 2.72 slower than IPS^{4}o for the largest input size. For smaller inputs, its performance is even worse for almost all inputs.

Even though IPS^{2}Ra outperforms the in-place radix sorter **RegionSort** for almost all inputs, IPS^{4}o is even faster. Thus, we concentrate our analysis on comparing RegionSort to IPS^{4}o rather than IPS^{2}Ra. For input data types supported by RegionSort, i.e., integer keys, it is our closest competitor. Overall, we see that the efficiency of RegionSort slightly degenerates for inputs larger than $n > 2^{32}$. The performance of IPS^{4}o remains the same for these large input sizes. RegionSort performs the best for AlmostSorted and TwoDup distributed inputs. For these inputs, RegionSort is competitive to IPS^{4}o in most cases. However, RegionSort performs much worse than IPS^{4}o for the remaining inputs, e.g., random inputs (Uniform), skewed inputs (Exponential and Zipf), and inputs with many duplicates (e.g., RootDup). For these distributions, RegionSort is slower than IPS^{4}o by a factor of 1.17 to 1.61 for the largest input size and becomes even less efficient for smaller inputs, e.g., RegionSort is slower than IPS^{4}o by factors of 1.29 to 1.68 for $n = 2^{27}$.

IPS^{4}o is competitive or faster than **IPS^{2}Ra** for all inputs. IPS^{4}o and IPS^{2}Ra perform similarly for inputs of medium input size which are Uniform, TwoDup, RootDup, and AlmostSorted distributed. Still, for these inputs, the performance of IPS^{2}Ra (significantly) decreases for large inputs ($n > 2^{32}$) in most cases. For inputs with very skewed key distributions, i.e., Exponential and Zipf, IPS^{4}o is significantly faster than IPS^{2}Ra.

Fig. 16. Pairwise performance profiles of IPS^{4}o to PBBS, MCSTLbq, RegionSort, and IPS^{2}Ra. The performance plots with PBBS and MCSTLbq use all data types. The performance plots with RegionSort and IPS^{2}Ra use inputs with unsigned integer keys (uint32, uint64, and Pair data types). The results were obtained on all machines for all input distributions with at least $2^{21}t$ bytes except Sorted, ReverseSorted, and Zero.

7.5.5 Comparison of Performance Profiles. In this section, we compare the pairwise performance profiles of IPS^{4}o with the (non)in-place comparison-based MCSTLbq (PBBS), and the radix sorter RegionSort as well as the pairwise performance profiles of IPS^{2}Ra and RegionSort. The profiles are shown in Fig. 16. We do not compare our algorithms to the radix sorters PBBR and RADULS2 as these are non-in-place and as their profiles are much worse than the profiles of the in-place radix sorter RegionSort. Overall, the performance of IPS^{4}o is much better than the performance of any other sorting algorithm. When we only consider radix sorters, the performance profile of IPS^{2}Ra is better than the one of RegionSort.

IPS^{4}o performs significantly better than **PBBS**. For example, PBBS sorts only 2.4 % of the inputs at least at fast as IPS^{4}o. Also, there is virtually no input for which PBBS is at least a factor of 1.50 faster than IPS^{4}o. In contrast, IPS^{4}o sorts 66 % of the inputs at least a factor of 1.50 faster than PBBS.

The performance profile of **MCSTLbq** is even worse than the one of PBBS. IPS^{4}o is faster than MCSTLbq for virtually any inputs. IPS^{4}o is even three times faster than MCSTLbq for almost 50 % of the inputs.

The performance of IPS^{4}o is also significantly better the performance of **RegionSort**. For example, IPS^{4}o sorts 74 % of the inputs faster than RegionSort. Also, RegionSort sorts only 9 % of the inputs at least a factor of 1.25 faster than IPS^{4}o. In contrast, IPS^{4}o sorts 44 % of the inputs at least a factor of 1.25 faster than RegionSort.

Among all pairwise performance profiles, the profiles of **IPS^{4}o and IPS^{2}Ra** are the closest. Still, IPS^{4}o performs better than IPS^{2}Ra. For example, IPS^{4}o sorts 62 % of the inputs faster than IPS^{2}Ra. Also, IPS^{2}Ra outperforms IPS^{4}o for 16 % of the inputs by a factor of 1.25 or more. On the other hand, IPS^{4}o outperforms IPS^{2}Ra for 31 % of the inputs by a factor of 1.25 or more.

7.5.6 Comparison to IMSDradix. We compare our algorithm IPS^{4}o to the in-place radix sorter IMSDradix [61] separately as the available implementation works only in rather special circumstances – 64 threads, $n > 2^{26}$, integer key-value pairs with values stored in a separate array. Also, that implementation is not in-place and requires a very specific input array: On a machine with m NUMA nodes, the input array must consist of m subarrays with a capacity of $1.2n/m$ each.

Distribution	I2x16	
	IPS^{4}o	IMSDradix
Sorted	**1.00**	77.00
ReverseSorted	**1.00**	7.23
Zero	**1.00**	
Exponential	**1.00**	2.31
Zipf	**1.00**	35.64
RootDup	**1.00**	7.10
TwoDup	**1.00**	41.03
EightDup	**1.00**	44.58
AlmostSorted	**1.00**	10.47
Uniform	**1.00**	1.68
Total	**1.00**	10.95
Rank	1	2

Table 5. Average slowdowns of IPS^{4}o and IMSDradix for Pair data types, different input distributions, and machine I2x16 with inputs containing at least $2^{21}t$ bytes. IMSDradix breaks for Zero input.

The experiments in [60] pin subarray i to NUMA node i. We nevertheless see the comparison as important since IMSDradix uses a similar basic approach to block permutation as IPS^{4}o.

Table 5 shows the average slowdowns of IPS^{4}o and IMSDradix for different input distributions executed on I2x16. We did not run IMSDradix on A1x64, I4x20, and A1x16 as these machines do not have 64 hardware theads. The results show that IPS^{4}o is much faster than IMSDradix for all input distributions. For example, the average slowdown ratio of IMSDradix to IPS^{4}o is 7.10 for RootDup input on I2x16. Note that IMSDradix breaks for Zero input and its average slowdown are between 35.64 and 44.58 for some input distributions with duplicated keys (TwoDup, EightDup, and Zipf) and with a skewed key distribution (Zipf). For Sorted input, IMSDradix is also much slower because IPS^{4}o detects sorted inputs.

7.6 Phases of Our Algorithms

Figure 17 shows the running times of the sequential samplesort algorithm I1S^{4}o and the sequential radix sorter I1S^{2}Ra as well as their parallel counterparts IPS^{4}o and IPS^{2}Ra. The running times are split into the four phases of the partitioning step (sampling, classification, permutation, and cleanup), the time spent in the base case algorithm, and overhead for the remaining components of the algorithm such as initialization and scheduling. In the following discussion of the sequential and parallel execution times, we report numbers for the largest input size unless stated otherwise.

Sequential Algorithms. The running time curves of I1S^{2}Ra are less smooth than those for I1S^{4}o because this code currently lacks the same careful adaptation of the distribution degree k

The time for **sampling** in the partitioning steps of I1S^{4}o is relatively small, i.e., 7.88 % of the total running time. For I1S^{2}Ra, no sampling is performed.

The **classification phase** of I1S^{4}o takes up about half of the total running time. The **permutation phase** is a factor of about eight faster than its classification phase. As both phases transfer about the same data volume and as the classification phase performs a factor of $\Theta(\log k)$ more local work ($\log k = 8$), we conclude that the classification phase is bounded by its local work. It is interesting to note that this was very different in 2004. In the 2004 paper [64], data distribution dominated element classification. Since then, peak memory bandwidth of high-performance processors has increased much faster than internal speed of a single core. The higher investment in

Fig. 17. Accumulated running time (normalized by $t8n \log n$) of the phases of our sequential samplesort and radix sort algorithms (top left and top right) and their parallel counterparts (bottom left and bottom right) obtained on machine A1x64 for uint64 values with input distribution Uniform.

memory bandwidth was driven by the need to accommodate the memory traffic of multiple cores. Indeed, we will see below that for parallel execution, memory access once more becomes crucial. Classification and permutation phases of $I1S^2Ra$ behave similarly as for $I1S^4o$. Since the local work for classification is much lower for radix sort, the running time ratio between these two phases is smaller yet, with 2.43 still quite high.

The **cleanup** takes less than five percent of the total running time of $I1S^4o$ and less than two percent of $I1S^2Ra$. The sequential algorithms spend a significant part of the running time in the **base case**. The base case takes 36.71 % of the total running time. For $I1S^2Ra$ the base case even dominates the overall running time (70.29 %) because it performs less work in the partitioning steps and because it uses larger base cases. The **overhead** caused by the data structure construction and task scheduling is negligible in the sequential case.

Parallel Algorithms. The partitioning steps of the parallel algorithms are significantly slower than the ones of the sequential algorithms. In particular, the work needed for the permutation phase increases by a factor of 11.59 for IPS^4o and 19.88 for IPS^2Ra. Since the permutation phase does little else than copying rather large blocks, the difference is mainly caused by memory bottlenecks.

Since memory access costs now dominate the running time, the performance advantage of radix sort over samplesort decreases when executed in parallel instead of sequentially. For other input

distributions as well as other data types, the parallel radix sort is even slower than parallel sample sort (see Section 7.5.1). In other words, the price paid for *less local work* in classification (for radix sort) is more data transfers due to *less accurate classification*. In the parallel setting, this tradeoff is harmful except for uniformly distributed keys and small element sizes.

When the input size below $n = 2^{26}$, the time for the classification phase and the additional overhead dominates the total running time. We can avoid this overhead. In our implementation, a "sorter object" and an appropriate constructor allows us to separate the data structure creation from the sorting. When we exclude the time to create and initialize the sorter object, sorting with IPS⁴o becomes significantly faster, i.e., a factor of 3.74 for $n = 2^{24}$.

8 CONCLUSION AND FUTURE WORK

In-place Super Scalar Samplesort (IPS⁴o) and In-place Super Scalar Radix Sort (IPS²Ra) are among the fastest sorting algorithms both sequentially and on multi-core machines. The algorithms can also be used for data distribution and local sorting in distributed memory parallel algorithms (e.g., [5]).

Both algorithms are the fastest known algorithms for a wide range of machines, data types, input sizes, and data distributions. Exceptions are small inputs (which fit into cache), a limitation to a fraction of the available cores (which profit from nonportable SIMD instructions), and almost sorted inputs (which profit from sequential adaptive sorting algorithms). Even in those exceptions, our algorithms, which were not designed for these purposes, are surprisingly close to more specialized implementations. One reason is that for large inputs, memory access costs overwhelmingly dominate the total cost of a parallel sorting algorithm so that saving elsewhere has little effect.

Our comparison based algorithm parallel algorithm IPS⁴o even mostly outperforms the integer sorting algorithms, despite having a logarithmic factor overhead with respect to executed instructions. Memory access efficiency of our algorithms is also the reason for the initially surprising observation that our in-place algorithms outperform algorithms that are allowed to use additional space.

Both algorithms significantly outperform the fastest parallel comparison-based competitor, PBBS, on almost all inputs. They are also significantly better than the fastest sequential comparison-based competitor, BlockPDQ, except for sorted and almost-sorted inputs.

The fastest radix sort competitors are SkaSort (sequential) and RegionSort (parallel). Our radix sorter is significantly faster than SkaSort and competitive to RegionSort. Also, our parallel samplesort algorithm is significantly faster than RegionSort for all inputs. Exceptions are some 32-bit inputs. Our parallel samplesort algorithm even sorts uniform distributed inputs significantly faster than RegionSort if the keys contain more than 32-bits.

Radix sorters which take advantage of non-portable hardware features, e.g., IppRadix (vector instructions) and RADULS2 (non-temporal writes), are very fast for small (Uniform distributed) data types. IppRadix for example sorts 32-bit unsigned integers very fast and RADULS2 is very fast for 64-bit unsigned integers. However, the interesting methods developed for these algorithms have little impact on larger data types and "hard" input distributions and thus, we perform better overall.

We compare the algorithms for input arrays with various NUMA memory layouts. With our new locality aware task scheduler, IPS⁴o is robustly fast for all NUMA memory layouts.

Future Work

Several improvements of our algorithms can be considered which address the remaining cases where our algorithms are outperformed. For small inputs, not in-place variants of our algorithms with preallocated data structures, smaller values of the distribution factor k and smaller block sizes

could be faster. For small inputs, the base case sorter becomes also more relevant. Here we could profit from several results on fast sorting for very small inputs [10, 13, 16]. Also, we would like to speed up the branchless decision tree with vector instructions. Preliminary results have shown improvements of up to a factor of 1.25 for I1S^{4}o with a decision tree using AVX-512 instructions. However, a general challenge remains how data-parallel instructions can be harnessed for sorting data with large keys and associated information and how to balance portability and efficiency.

With respect to the volume of accessed memory, which isa main distinguishing feature of our algorithms, further improvements are conceivable. One option is to reconsider the approach from most radix sort implementations and of the original super scalar samplesort [64] to first determine exact bucket sizes. This is particularly attractive for radix sorters since computing bucket indices is very fast. Then one could integrate the classification phase and the permutation phase of IPS^{4}o. To make this efficient, one should still work with blocks of elements moved between local buffer blocks and the input/output array. For samplesort, one would approximate bucket sizes using the sample and a cleanup would be required. Another difficulty may be a robust parallel implementation that avoids contention for all input distributions.

A more radical approach to reducing memory access volume would be to implement the permutation phase in sublinear time by using the hardware support for virtual memory. For large inputs, one could make data blocks correspond to virtual memory pages. One could then move around blocks by just changing their virtual addresses. It is unclear to us though whether this is efficiently (or even portably) supported by current operating systems. Also, the output might have an unexpected mapping to NUMA nodes which might affect the performance of subsequently processing the sorted array.

Our radix sorter IPS^{2}Ra is currently a prototype meant for demonstrating the usefulness of our scheduling and data movement strategies independently of a comparison based sorter. It could be made more robust by adapting the function for extracting bucket indices to various input distributions (which can be approximated analyzing a sample of the input). This could in particular entail various compromises between the full-fledged search tree of IPS^{4}o and the plain byte extraction of IPS^{2}Ra. For example, one could accelerate the search tree traversal of super scalar samplesort by precomputing a lookup table of starting nodes that are addressed by the most significant bits of the key. One could also consider the approach from the LearnedSort algorithm [47] which addresses a large number of buckets using few linear functions. Perhaps, approximate distribution-learning approaches can be replaced by fast and accurate computational-geometry algorithms. Existing geometry algorithms [19, 40] might have to be adapted to use a cost function that optimizes the information gain from using a small number of piece-wise linear functions.

Adaptive sorting algorithms are an intriguing area of research in algorithms [24]. However, implementations such as Timsort currently cannot compete with the best nonadaptive algorithms except for some extreme cases. Hence, it would be interesting to engineer adaptive sorting algorithms to take the performance improvements of fast nonadaptive algorithms (such as ours) into account.

The measurements reported in this paper were performed using somewhat non-portable implementations that use a 128-bit compare-and-swap instruction specific to x86 architectures (see also Section 6). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.

Coming back to the original motivation for an alternative to quicksort variants in standard libraries, we see IPS^{4}o as an interesting candidate. The main remaining issue is code complexity. When code size matters (e.g., as indicated by a compiler flag like -Os), one could use IPS^{4}o with fixed k and a larger base case size. Formal verification of the correctness of the implementation

might help to increase trust in the remaining cases.

REFERENCES

[1] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *20th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 197–206. https://doi.org/10.1145/1378533.1378573

[2] Martin Aumüller and Nikolaj Hass. 2019. Simple and Fast BlockQuicksort using Lomuto's Partitioning Scheme. In *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*, Stephen G. Kobourov and Henning Meyerhenke (Eds.). SIAM, 15–26. https://doi.org/10.1137/1.9781611975499.2

[3] Michael Axtmann. 2020. NUMA Array. https://github.com/ips4o/NumaArray. Accessed: 2020-09-01.

[4] Michael Axtmann. 2020. (Parallel) Super Scalar Sample Sort. https://github.com/ips4o/ps4o. Accessed: 2020-09-01.

[5] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. 2015. Practical Massively Parallel Sorting. In *27th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 13–23. https://doi.org/10.1145/2755573.2755595

[6] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th European Symposium on Algorithms (ESA)*, Vol. 87. LIPIcs, 9:1–9:14. https://doi.org/10.4230/LIPIcs.ESA.2017.9

[7] Huang Bing-Chao and Donald E Knuth. 1986. A one-way, stackless quicksort algorithm. *BIT Numerical Mathematics* 26, 1 (1986), 127–130.

[8] Timo Bingmann. 2018. *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools*. Ph.D. Dissertation. Karlsruher Institut für Technologie (KIT). https://doi.org/10.5445/IR/1000085031

[9] Timo Bingmann, Andreas Eberle, and Peter Sanders. 2017. Engineering Parallel String Sorting. *Algorithmica* 77, 1 (2017), 235–286. https://doi.org/10.1007/s00453-015-0071-1

[10] Timo Bingmann, Jasper Marianczuk, and Peter Sanders. Aug. 2020. Engineering Faster Sorters for Small Sets of Items. Computing Research Repository (CoRR). arXiv:2002.05599

[11] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *22nd Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 189–199. https://doi.org/10.1145/1810479.1810519

[12] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1996. A Comparison of Sorting Algorithms for the Connection Machine CM-2, In 3rd Symposium on Parallel Algorithms and Architectures (SPAA). *Commun. ACM* 39, 12es, 273–297. https://doi.org/10.1145/113379.113380

[13] Berenger Bramas. 2017. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *IJACSA* 8, 10 (2017), 337–344. https://doi.org/10.14569/ijacsa.2017.081044

[14] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. 2007. Engineering a cache-oblivious sorting algorithm. *ACM J. Exp. Algorithmics* 12 (2007), 2.2:1–2.2:23. https://doi.org/10.1145/1227161.1227164

[15] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. 2015. PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort. *Proc. VLDB Endow.* 8, 12 (2015), 1518–1529. https://doi.org/10.14778/2824032.2824050

[16] Michael Codish, Luís Cruz-Filipe, Markus Nebel, and Peter Schneider-Kamp. 2017. Optimizing sorting algorithms by using sorting networks. *Formal Aspects Comput.* 29, 3 (2017), 559–579. https://doi.org/10.1007/s00165-016-0401-3

[17] Intel Corporation. 2020. Intel® Integrated Performance Primitives. https://software.intel.com/en-us/ipp-dev-reference. Version 2020 Initial Release.

[18] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Program.* 91, 2 (2002), 201–213. https://doi.org/10.1007/s101070100263

[19] David H Douglas and Thomas K Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *artogr. Int. J. Geogr. Inf. Geovisualization* 10, 2 (1973), 112–122.

[20] Branislav Durian. 1986. Quicksort Without a Stack. In *Mathematical Foundations of Computer Science (Lecture Notes in Computer Science, Vol. 233)*. Springer, 283–289. https://doi.org/10.1007/BFb0016252

[21] Stefan Edelkamp and Armin Weiß. 2016. BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. In *24th European Symposium on Algorithms (ESA)*, Vol. 57. LIPIcs, 38:1–38:16. https://doi.org/10.4230/LIPIcs.ESA.2016.38

[22] Stefan Edelkamp and Armin Weiß. 2019. Worst-Case Efficient Sorting with QuickMergesort. In *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 1–14. https://doi.org/10.1137/1.9781611975499.1

[23] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. 2012. Branch Mispredictions Don't Affect Mergesort. In *11th Symposium on Experimental Algorithms (SEA)*, Vol. 7276. Springer, 160–171. https://doi.org/10.1007/978-3-642-30850-5_15

[24] Vladimir Estivill-Castro and Derick Wood. 1992. A Survey of Adaptive Sorting Algorithms. *ACM Comput. Surv.* 24, 4 (1992), 441–476. https://doi.org/10.1145/146370.146381

[25] Philip J. Fleming and John J. Wallace. 1986. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Commun. ACM* 29, 3 (1986), 218–221. https://doi.org/10.1145/5666.5673

[26] Gianni Franceschini. 2004. Proximity Mergesort: optimal in-place sorting in the cache-oblivious model. In *15th Symposium on Discrete Algorithms (SODA)*. SIAM, 291–299.

[27] Gianni Franceschini and Viliam Geffert. 2005. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. *J. ACM* 52, 4 (2005), 515–537. https://doi.org/10.1145/1082036.1082037

[28] Rhys S. Francis and L. J. H. Pannan. 1992. A parallel partition for enhanced parallel QuickSort. *Parallel Comput.* 18, 5 (1992), 543–550. https://doi.org/10.1016/0167-8191(92)90089-P

[29] W. Donald Frazer and A. C. McKellar. 1970. Samplesort: A Sampling Approach to Minimal Storage Tree Sorting. *J. ACM* 17, 3 (1970), 496–507. https://doi.org/10.1145/321592.321600

[30] Edward H. Friend. 1956. Sorting on Electronic Computer Systems. *J. ACM* 3, 3 (1956), 134–168. https://doi.org/10.1145/320831.320833

[31] Viliam Geffert and Jozef Gajdos. 2010. Multiway in-place merging. *Theor. Comput. Sci.* 411, 16-18 (2010), 1793–1808. https://doi.org/10.1016/j.tcs.2010.01.034

[32] Fuji Goro and Morwenn. 2014. TimSort. https://github.com/timsort/cpp-TimSort.git. Accessed: 2020-09-01.

[33] REFRESH Bioinformatics Group. 2017. RADULS2. https://github.com/refresh-bio/RADULS. Accessed: 2020-09-01.

[34] Xiaojun Guan and Michael A. Langston. 1991. Time-Space Optimal Parallel Merging and Sorting. *IEEE Trans. Computers* 40, 5 (1991), 596–602. https://doi.org/10.1109/12.88483

[35] Philip Heidelberger, Alan Norton, and John T. Robinson. 1990. Parallel Quicksort Using Fetch-and-Add. *IEEE Trans. Computers* 39, 1 (1990), 133–138. https://doi.org/10.1109/12.46289

[36] C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (1962), 10–15. https://doi.org/10.1093/comjnl/5.1.10

[37] Kaixi Hou, Hao Wang, and Wu-chun Feng. 2015. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *29th International Conference on Supercomputing (ICS)*. ACM, 383–392. https://doi.org/10.1145/2751205.2751247

[38] Bing-Chao Huang and Michael A. Langston. 1992. Fast Stable Merging and Sorting in Constant Extra Space. *Comput. J.* 35, 6 (1992), 643–650. https://doi.org/10.1093/comjnl/35.6.643

[39] Lorenz Hübschle-Schneider. 2016. Super Scalar Sample Sort. https://github.com/lorenzhs/ssssort. Accessed: 2020-09-01.

[40] Hiroshi Imai and Masao Iri. 1986. An optimal algorithm for approximating a piecewise linear function. *J. Inf. Process.* 9, 3 (1986), 159–162.

[41] Joseph JáJá. 2000. A perspective on Quicksort. *Comput. Sci. Eng.* 2, 1 (2000), 43–49. https://doi.org/10.1109/5992.814657

[42] Tomasz Jurkiewicz and Kurt Mehlhorn. 2014. On a Model of Virtual Address Translation. *Journal of Experimental Algorithmics (JEA)* 19, 1 (2014), 1.9:1–1.9:28. https://doi.org/10.1145/2656337

[43] Kanela Kaligosi and Peter Sanders. 2006. How Branch Mispredictions Affect Quicksort. In *14th European Symposium on Algorithms (ESA)*, Vol. 4168. Springer, 780–791. https://doi.org/10.1007/11841036_69

[44] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. 1996. Practical In-Place Mergesort. *Nord. J. Comput.* 3, 1 (1996), 27–40.

[45] Pok-Son Kim and Arne Kutzner. 2008. Ratio Based Stable In-Place Merging. In *Theory and Applications of Models of Computation (TAMC)*, Vol. 4978. Springer, 246–257. https://doi.org/10.1007/978-3-540-79228-4_22

[46] Marek Kokot, Sebastian Deorowicz, and Maciej Dlugosz. 2017. Even Faster Sorting of (Not Only) Integers. In *5th International Conference on Man-Machine Interactions (ICMMI)*, Vol. 659. Springer, 481–491. https://doi.org/10.1007/978-3-319-67792-7_47

[47] Ani Kristo. 2020. LearnedSort. https://github.com/learnedsystems/LearnedSort. Accessed: 2020-09-01.

[48] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *International Conference on Management of Data (SIGMOD)*. ACM, 1001–1016. https://doi.org/10.1145/3318464.3389752

[49] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. 2014. Multi-Pivot Quicksort: Theory and Experiments. In *16th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 47–60. https://doi.org/10.1137/1.9781611973198.6

[50] Shoshana Marcus. 2015. Review of: A Guide to Experimental Algorithmics by Catherine C. McGeoch. *SIGACT News* 46, 1 (2015), 20–22. https://doi.org/10.1145/2744447.2744453

[51] Charles U. Martel and Dan Gusfield. 1989. A Fast Parallel Quicksort Algorithm. *Inf. Process. Lett.* 30, 2 (1989), 97–102. https://doi.org/10.1016/0020-0190(89)90116-6

[52] Mike McFadden. 2014. WikiSort. https://github.com/BonzaiThePenguin/WikiSort. Accessed: 2020-09-01.

[53] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. 1993. Engineering Radix Sort. *Comput. Syst.* 6, 1 (1993), 5–27.

[54] Kurt Mehlhorn and Peter Sanders. 2003. Scanning Multiple Sequences Via Cache Memory. *Algorithmica* 35, 1 (2003), 75–93. https://doi.org/10.1007/s00453-002-0993-2

[55] David R. Musser. 1997. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* 27, 8 (1997), 983–993.

[56] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. RegionSort. https://github.com/omarobeya/parallel-inplace-radixsort. Accessed: 2020-09-01.

[57] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, ACM, 213–224.

[58] Orson Peters. 2015. Pattern-defeating quicksort. https://github.com/orlp/pdqsort. Accessed: 2020-09-01.

[59] Tim Peters. 2002. Timsort. http://svn.python.org/projects/python/trunk/Objects/listsort.txt. Accessed: 2020-03-31.

[60] Orestis Polychroniou. 2014. In-place MSB. http://www.cs.columbia.edu/~orestis/publications.html. Accessed: 2020-09-01.

[61] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *International Conference on Management of Data (SIGMOD)*. ACM, 755–766. https://doi.org/10.1145/2588555.2610522

[62] Naila Rahman. 2002. *Algorithms for Hardware Caches and TLB*. Vol. 2625. Springer, 171–192. https://doi.org/10.1007/3-540-36574-5_8

[63] James Reinders. 2007. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly.

[64] Peter Sanders and Sebastian Winkel. 2004. Super Scalar Sample Sort. In *12th European Symposium on Algorithms (ESA)*, Vol. 3221. Springer, 784–796. https://doi.org/10.1007/978-3-540-30140-0_69

[65] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *24th Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 68–70. https://doi.org/10.1145/2312005.2312018

[66] Johannes Singler, Peter Sanders, and Felix Putze. 2007. MCSTL: The Multi-core Standard Template Library. In *Euro-Par*, Vol. 4641. Springer, 682–694. https://doi.org/10.1007/978-3-540-74466-5_72

[67] Malte Skarupke. 2016. I Wrote a Faster Sorting Algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/. Accessed: 2020-03-31.

[68] Malte Skarupke. 2016. Ska Sort. https://github.com/skarupke/ska_sort. Accessed: 2020-09-01.

[69] Virginia Tech SyNeRGy Lab. 2018. ASPaS. https://github.com/vtsynergy/aspas_sort. Accessed: 2020-09-01.

[70] Philippas Tsigas and Yi Zhang. 2003. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE Computer Society, 372. https://doi.org/10.1109/EMPDP.2003.1183613

[71] Jan Wassenberg and Peter Sanders. 2011. Engineering a Multi-core Radix Sort. In *Euro-Par*, Vol. 6853. Springer, 160–169. https://doi.org/10.1007/978-3-642-23397-5_16

[72] Lutz M. Wegner. 1987. A Generalized, One-Way, Stackless Quicksort. *BIT Comput. Sci. Sect.* 27, 1 (1987), 44–48. https://doi.org/10.1007/BF01937353

[73] Armin Weiss. 2016. BlockQuicksort. https://github.com/weissan/BlockQuicksort. Accessed: 2020-09-01.

[74] Armin Weiss. 2016. Yaroslavskiy's Dual-Pivot Quicksort. https://github.com/weissan/BlockQuicksort/blob/master/Yaroslavskiy.h++. Accessed: 2020-09-01.

[75] Jokob Wenzel. 2019. Intel Threading Building Blocks with CMake build system. https://github.com/wjakob/tbb. TBB 2019 Update 6.

[76] Vladimir Yaroslavskiy. 2010. Question on sorting. http://mail.openjdk.java.net/pipermail/core-libs-dev/2010-July/004649.html. Accessed: 2020-09-01.

A DETAILS OF THE ANALYSIS

A.1 Limit Number of Recursions

In this section, we prove the following theorem:

THEOREM A.1. *Let $M \geq 1$ be a constant. Then, after $O\left(\log_k \frac{n}{M}\right)$ recursion levels, all non-equality buckets of IPS^{4}o have size M with a probability of at least $1 - n/M$ for an oversampling ratio of $\alpha = \Theta(c \log k)$.*

We first show the Lemmas A.2 and A.3 which are used to prove Theorem A.1. Let e be an arbitrary but fixed element of a task with n elements in IPS^{4}o. A "successful recursion step" of e is a recursion step that assigns the element to a bucket of size $3n/k$.

LEMMA A.2. *The probability of a successful recursion step of an arbitrary but fixed element is at least $1 - 2k^{-c/12}$ for an oversampling ratio of $\alpha = c \log k$.*

PROOF. We bound the probability that a task of n elements assigns an arbitrary but fixed element e_j to a bucket containing at most $3n/k$ (a successful recursion step). Let $[e_1 .. e_n]$ be the input of the task in sorted order, let $R_r = [e_r .. e_{r+1.5n/k-1}]$ be the set containing e_k and the $1.5n/k$}'th larger elements, and let $[s_1 .. s_{\alpha k}]$ be the selected samples. The boolean indicator X_{ik} that sample s_i is an element of R_k is defined as

$$X_{ij} = \begin{cases} 1, & s_i \in R_k \\ 0, & \text{else.} \end{cases}$$

The probability $\Pr[X_{ik} = 1] = 1.5 \frac{n}{k} \cdot \frac{1}{n} = \frac{1.5}{k}$ is independent of the sample s_i as the samples are selected with replacement. Thus, the expected value of the number of samples selected from R_k is $X_k = \sum_{i=1}^{\alpha k} X_{ik}$ is $E[X_k] = 1.5/k \cdot \alpha k = 1.5\alpha$. We use the Chernoff bound to limit the probability of less than α samples in R_k to $\Pr[X_k < \alpha] = \Pr[X_k < (1 - 1/3)E[X_k]] < e^{-1/2(1/3)^2 E[X_k]} = e^{-1/12\alpha}$. When R_j as well as $R_{j-1.5n/k}$ both provide at least α samples, R_j as well as $R_{j-1.5n/k}$ provide a splitter and e_j is in a bucket containing at most $3n/k$ elements. The probability is $\Pr[X_j \geq S \wedge X_{j-1.5n/k} \geq S] = 1 - \Pr[X_j < S \vee X_{j-1.5n/k} < S] \geq 1 - \Pr[X_j < S] - \Pr[X_{j-1.5n/k} < S] > 1 - 2e^{-1/12\alpha} = 1 - 2k^{-1/12c}$. □

LEMMA A.3. *Let c be a constant, let $\alpha = c \log k$ be the oversampling ratio of IPS^{4}o ($c \geq 36 - 2.38/\log(0.34 \cdot k)$), and let IPS^{4}o execute $2 \log_{k/3} \frac{n}{M}$ recursion levels. Then, an arbitrary but fixed input element of IPS^{4}o passes at least $\log_{k/3} \log \frac{n}{M}$ successful recursion levels with a probability of at least $1 - (n/M)^{-2}$.*

PROOF. We execute IPS^{4}o $2 \log_{k/3} \frac{n}{M}$ recursion levels and bound the probability that an arbitrary but fixed input element passes at least $\log_{k/3} \log \frac{n}{M}$ successful recursion levels. This experiment is a Bernoulli trial as we have exactly two possible outcomes, "successful recursion step" and "non-successful recursion step", and the probability of success is the same on each level. Let denote the random variable X as the number of non-successful recursion steps after $2 \log_{k/3} \frac{n}{M}$ recursion levels, p the probability of a non-successful recursion step, and let $c \geq 36 - 2.38/\log(0.34 \cdot k)$. The

probability I

$$
\begin{aligned}
I = \mathbb{P}[X > 2\log\frac{n}{M} - \log\frac{n}{M}] &\leq \mathbb{P}[X > \log\frac{n}{M}] \\
&\leq \sum_{j>\log\frac{n}{M}} \binom{2\log\frac{n}{M}}{j} p^j (1-p)^{2\log\frac{n}{M}-j} \leq \sum_{j>\log\frac{n}{M}} \left(\frac{2e\log\frac{n}{M}}{j}\right)^j p^j \\
&\leq \sum_{j>\log\frac{n}{M}} \left(\frac{2e\log\frac{n}{M}}{\log\frac{n}{M}}\right)^j p^j \leq \sum_{j>\log\frac{n}{M}} (2e)^j \left(2k^{-1/12c}\right)^j \\
&\leq \sum_{j>\log\frac{n}{M}} \left(4ek^{-1/12c}\right)^j = \frac{\left(4ek^{-1/12c}\right)^{\log\frac{n}{M}+1}}{1-4ek^{-1/12c}} \\
&\leq \frac{\left(\frac{n}{M}\right)^{-1/12c+\log(4e)}}{1-4ek^{-1/12c}} \leq \left(\frac{n}{M}\right)^{-2}
\end{aligned}
\tag{2}
$$

defines an upper bound of the probability that a randomly selected input element passes $2\log_{k/3} n/M$ recursion levels without passing $\log_{k/3}\frac{n}{M}$ successful recursion levels. For the sake of simplicity, all logarithms of the equation above are to the base of $k/3$. The third "$\leq$" uses $\binom{n}{k} \leq (en/k)^k$, the fifth "$\leq$" uses Lemma A.2 and the "=" uses the geometric series. □

PROOF OF THEOREM A.1. We first assume that $M \geq k^2 n_0$ holds. In this case, we select $kc\log k$ samples. Let $l = \log_{k/3}\frac{n}{M}$ and let e be an arbitrary but fixed input element of IPS^{4}o after $2l$ recursion levels. Lemma A.3 tells us that e has passed at least l successful recursion steps with a probability of at least $1 - (n/M)^{-2}$ when IPS^{4}o has performed $2\log_{k/3}\frac{n}{M}$ recursion levels. Element e is, in this case, in a bucket containing more than $n(3/k)^l = M$ elements as each successful recursion step shrinks the bucket by a factor of at least $3/k$. Let $E = [e_1 .. e_n]$ be the input elements of IPS^{4}o in sorted order and let $Q = \{e_{iM}|1 \leq i < n/M \land i \in \mathbb{N}\}$ every n/M'th element. We now examine buckets containing elements in Q after $2l$ recursion levels. The probability that any element Q is in a bucket containing more than M elements is less than $n/M \cdot (n/M)^{-2} = (n/M)^{-1}$ – this follows from the former insight and the Boole's inequality. In other words, the probability that all elements in Q are in buckets containing less than M elements is larger than $1 - M/n$. As this holds for all elements in Q, every n/M'th element in the sorted output, the probability that all elements after $2l$ recursion level are in buckets containing less than M elements is larger than $1 - M/n$. □

A.2 Comparing the I/O Volume of I1S^{4}o and S^{4}o.

We compare the first level of I1S^{4}o and S^{4}o for inputs with 8-byte input elements. We assume a oracle with 1-byte entries for S^{4}o. Furthermore, we assume that the input does not fit into the private cache.

Both algorithms read and write the data once for the base case – $16n$ bytes of I/O volume. Each level of I1S^{4}o reads and writes all data once in the classification phase and once in the permutation phase – $32n$ bytes per level. Each level of S^{4}o reads the elements twice and writes them once only in its distribution phase – $24n$ bytes per level.

Additionally, S^{4}o writes an oracle sequence that indicates the bucket for each element in the classification phase and reads the oracle sequence in the distribution phase – $2n$ bytes per level. The algorithm also has to allocate the temporary arrays. For security reasons, that memory is zeroed by the operating system – $9n$ bytes.[8] If the number of levels is odd, S^{4}o has to copy the sorted result

---

[8]In current versions of the Linux kernel this is done by a single thread and thus results in a huge scalability bottleneck.

Subroutine	Types	Reps	Sum in n Bytes
S⁴o			
Copy back	r + w + wa	once	16 + 8
Base Case	r + w	once	16
Init Temp Array + Oracle	w	once	9
Classification: Oracle	w + wa	per level	1 + 1
Classification: Array	r	per level	8
Redistribution: Oracle	r	per level	1
Redistribution: Array	r + w + wa	per level	16+8
I1S⁴o			
Base Case	r + w	once	16
Classification	r + w	per level	16
Redistribution	r + w	per level	16

Table 6. I/O volume of read (r) and write (w) operations broken down into subroutines of I1S⁴o and S⁴o. Additionally, potential write allocate operations (wa) are listed.

back to the input array – $16n$ bytes. For now, I1S⁴o (S⁴o) has an I/O volume of $32n$ ($26n$) byte per level and $16n$ ($41n$) bytes once.

When S⁴o writes to the temporary arrays or during copying back, cache misses happen when an element is written to a cache block that is currently not in memory. Depending on the cache replacement algorithm, a *write allocate* may be performed – the block is read from the memory to the cache even though none of the data in that block will ever be read. Detecting that the entire cache line will be overwritten is difficult as S⁴o writes to the target buckets element by element. This amounts to an I/O volume of up to $9n$ bytes per level and $8n$ bytes once. I1S⁴o does not perform write allocates. The classification phase essentially sweeps a window of size $\Theta(bk)$ through the memory by reading elements from the right border of the window and writing elements to the left border. The permutation phase reads a block from the memory and replaces the "empty" memory bock with a cached block afterwards. Finally, we get for I1S⁴o (S⁴o) a total I/O volume of $32n$ ($35n$) byte per level and $16n$ ($49n$) bytes once – S⁴o with one level has a factor of 1.75 more I/O volume than I1S⁴o. Table 6 shows the I/O volume of the subroutines in detail.

Furthermore, S⁴o may suffer more conflict misses than I1S⁴o due to the mapping of data to cache lines. In the distribution phase, S⁴o reads the input from left to right but writes elementwise to positions in the buckets which are not coordinated. For the same reasons, S⁴o may suffer more TLB misses. I1S⁴o, on the other hand, essentially writes elements to cached buffer blocks (classification) and swaps blocks of size b within the input array (block permutation). For an average case analysis on scanning multiple sequences, we refer to [54].

Much of this overhead can be reduced using measures that are non-portable (or hard to make portable). In particular, non-temporal writes eliminate the write allocates and also help to eliminate the conflict misses. One could also use a base case sorter that does the copying back as a side-effect when the number of recursion levels is odd. When sorting multiple times within an application, one can keep the temporary arrays without having to reallocate them. However, this may require a different interface to the sorter. Overall, depending on many implementation details, S⁴o may require slightly or significantly more I/O volume.

B FROM IN-PLACE TO STRICTLY IN-PLACE

We now explain how the space consumption of IPS^{4}o can be made independent of n in a rather simple way by adapting the strictly in-place approach of quicksort. We do not consider the space requirement for storing parallel tasks as those tasks are processed immediately. However, we require the (sequential and parallel) partitioning steps to mark the beginning of each subtask by storing the largest element of a subtask in its first position. When a thread has finished its last parallel task, the elements covered by its sequential tasks can be described with two indices c_l and c_r:

LEMMA B.1. *When thread i starts its sequential phase, the sequential tasks assigned to thread i cover a consecutive subarray of the input array, i.e., there is no gap between the tasks in the input array.*

For reasons of better readability, we appended the proof of Lemma B.1 to the end of this chapter. The proof of this lemma also implicitly describes the technique to calculate the values of c_l and c_r.

In the sequential phase, the thread has to sort the elements $A[c_l, c_r - 1]$, which are partitioned into its sequential tasks. The boundaries of the sequential tasks are implicitly represented by the largest element at the beginning of each task. Starting a search at the leftmost task, the first element larger than the first element of a task defines the first element of the next task. Note that the time required for the search is only logarithmic to the task size when using an exponential/binary search. We assume that the corresponding function *searchNextLargest* returns $n + 1$ if no larger elements exist – this happens for the last task. The function *onlyEqualElements* checks whether a task only contains identical elements. We have to skip these "equal" tasks to avoid an infinite loop. The following pseudocode uses this approach to emulate recursion in constant space on the sequential tasks.

$n := e - b$	−− total size of sequential tasks
$i := b$	−− first element of current task
$j := searchNextLargest(A[i], A, i + 1, n)$	−− first element of next task
while $i < n$ **do**	
else if $onlyEqualElements(A, i, j - 1)$; **then** $i := j$	−− skip equal tasks
else if $j - i < n_0$ **then** $smallSort(a, i, j - 1)$; $i := j$	−− base case
else $partition(a, i, j - 1)$	−− partition first unsorted task
$j := searchNextLargest(A[i], A, i + 1, n)$	−− find beginning of next task

The technique which we described – making the space consumption of IPS^{4}o independent of n – used the requirement that the sequential tasks of a thread cover the consecutive subarray $A[c_l, c_r - 1]$ for some c_l and c_r. In the following, we show that this requirement holds.

PROOF OF LEMMA B.1. Let thread i process a parallel task $T[l, r]$ with thread group $[\underline{t} .. \overline{t})$ in one of the following states:

- *Left*. Thread i is the leftmost thread of the thread group, i.e., $i = \underline{t}$.
- *Right*. Thread i is the rightmost thread of the thread group, i.e., $i = \overline{t} - 1$.
- *Middle*. Thread i is not the leftmost or rightmost of the thread group, i.e., $\underline{t} < i < \overline{t} - 1$.

We claim that the sequential tasks assigned to thread i fulfill the following propositions when (and directly before) thread i processes $T[l, r]$:

- $T[l, r]$ *was processed in state Left (Right)*. Thread i does not have sequential tasks or its sequential tasks cover a consecutive subarray of the input array, i.e., there is no gap between the tasks. In the latter case, the rightmost task ends at position $l - 1$ with $l - 1 \in (in/t, (i + 1)n/t - 1)$ (leftmost task begins at position r with $r \in (in/t, (i + 2)n/t - 1)$).
- $T[l, r]$ *was processed in state Middle*. Thread i does not have sequential tasks.

Assume for now that these propositions hold – we will prove them later. We use the proposition to show that Lemma B.1 holds when a thread i starts processing its sequential tasks: Let $T[l, r)$ be the last parallel task of thread i, executed with the thread group $[\underline{t} .. \bar{t})$. No matter in which state $T[l, r)$ has been executed, the sequential tasks of thread i cover a consecutive subarray of the input array at the beginning of its sequential phase.

- $T[l, r)$ *was processed in state Right.* As $i = \bar{t} - 1$, we add all subtasks of $T[l, r)$ which start in $A[in/t - 1, r - 1]$ to thread i. No gap can exist between these subtasks as they cannot be interrupted by a parallel task. Also, before we assign the new subtasks to thread i, the leftmost sequential task of thread i begins at position r (see proposition). Then, the rightmost sequential subtask of $T[l, r)$ which start in $A[in/t - 1, r - 1]$ ends at $A[r - 1]$. Thus, after the subtasks were added, there is no gap between the sequential tasks of thread i.

- $T[l, r)$ *was processed in state Left.* As $i = \underline{t}$, we add all subtasks of $T[l, r)$ which start in $A[l, (i + 1)n/t - 1]$ to thread i. No gap can exist between these subtasks as they cannot be interrupted by a parallel task. Also, before thread i adds the new subtasks, the rightmost sequential task ends at position $l - 1$ (see proposition). Then, the leftmost subtask of $T[l, r)$ which starts in $A[l, (i + 1)n/t - 1]$ begins at $A[l]$. Thus, after the subtasks were added, there is no gap between the sequential tasks of thread i.

- $T[l, r)$ *processed in state middle.* We add all sequential subtasks to thread i which start in $A[in/t, (i + 1)n/t - 1]$. No gap can exist between these subtasks as they cannot be interrupted by a parallel task. Also, the subtasks are added in sorted order from left to right.

We will now prove the propositions by induction.

Base case. When a thread i processes its first parallel task $T[0, n)$, thread i does not have any sequential tasks.

Inductive step. We assume that the induction hypothesis holds when thread i was executing the parallel task $T[l, r)$ with the thread group $[\underline{t} .. \bar{t})$. We also assume thread i and others execute the next parallel task $T[l_s, r_s)$. We note that $T[l_s, r_s)$ is a subtask of $T[l, r)$. We have to prove that the induction hypothesis still holds after we have added thread i's sequential subtasks of $T[l, r)$ to the thread, i.e., when thread i executes the subtask $T[l_s, r_s)$.

- *Thread i has executed $T[l, r)$ in the state Middle and thread i is executing $T[l_s, r_s)$ in the state Middle.* From the induction hypothesis, we know that thread i did not have sequential tasks when $T[l, r)$ was executed. We have to show that thread i did not get sequential subtask of $T[l, r)$, after $T[l, r)$ has been executed. A sequential subtask $T[a, b)$ would have been added to thread i, if $i = \min(at/n, t - 1)$. We show that no $T[a, b)$ with this property exists. As thread i is not the rightmost thread of $T[l, r)$, we have $i < \bar{t} - 1$. This means that a sequential subtask $T[a, b)$ is only assigned to thread i if $i = \lfloor at/n \rfloor$ holds, i.e., $a \in [in/t, (i + 1)n/t)$ is required. However, there is no sequential subtask of $T[l, r)$ which begins in the i'th stripe of the input array: As thread i is not the leftmost thread of $T[l_s, r_s)$, the parallel subtask $T[l_s, r_s)$ contains the subarray $A[in/t, (i + 1)n/t - 1]$ completely (see Lemma 4.5). Thus, a second (sequential) subtask $T[a, b)$ with $in/t \le a < (i + 1)n/t$ cannot exist.

- *Thread i has executed $T[l, r)$ in the state Middle and thread i is executing $T[l_s, r_s)$ in the state Right.* From the induction hypothesis, we know that thread i did not have sequential tasks when $T[l, r)$ was executed. As thread i was not the rightmost thread of $T[l, r)$, we have $i < \bar{t} - 1$. This means that a sequential subtask $T[a, b)$ of $T[l, r)$ is only assigned to thread i if $i = \lfloor at/n \rfloor$ holds, i.e., $a \in [in/t, (i + 1)n/t)$ is required. However, as thread i is not the leftmost thread of $T[l_s, r_s)$, $T[l_s, r_s)$ completely contains $A[in/t, (i + 1)n/t - 1]$. Thus, there is no sequential subtask $T[a, b)$ with $a \in [in/t(i + 1)n/t)$ – we do not add sequential tasks of $T[l, r)$ to thread i.

- *Thread i has executed $T[l, r)$ in the state Middle and thread i is executing $T[l_s, r_s)$ in the state Left.* From the induction hypothesis, we know that thread i did not have sequential tasks when $T[l, r)$ was executed. Also, as thread i is not the rightmost thread of $T[l, r)$, we have $i < \bar{t} - 1$. This means that a sequential subtask $T[a, b)$ is only assigned to thread i if $i = \lfloor at/n \rfloor$ holds, i.e., $a \in [in/t, (i+1)n/t)$ is required. Thus, if there is no sequential subtask $T[a, b)$ of $T[l, r)$ with $a \in [in/t, (i+1)n/t)$, the thread i does not get sequential subtasks and the induction step is completed in the case here. Otherwise, if sequential subtasks $T[a, b)$ exist with $a \in in/t, (i+1)n/t)$, they are added to thread i and we have to show that the propositions hold afterwards: All subtasks $T[a, b)$ which begin in $A[in/t, (i+1)n/t]$ are sequential subtasks, except one parallel subtask, $T[l_s, r_s)$. Thus, there is no gap between these sequential subtasks. As thread i is the leftmost thread of $T[l_s, r_s)$, we know that $l_s \in (in/t, (i+1)n/t)$ and that $r_s \geq (i+2)n/t$. Thus, the rightmost sequential subtask ends at $A[l_s - 1]$ with $l_s - 1 \in (in/t, (i+1)n/t - 1)$.

- *Thread i has executed $T[l, r)$ in the state Left (Right) and thread i executes $T[l_s, r_s)$ in the state Left (Right).* From the induction hypothesis, we know that $l - 1$ (that r) is narrowed by $l - 1 \in (in/t, (i+1)n/t)$ (by $r \in [in/t, (i+2)n/t)$) before tasks of $T[l, r)$ are added to thread i. As thread i is the leftmost (rightmost) thread of $T[l_s, r_s)$, we can narrow the begin l_s (end r_s) of $T[l_s, r_s)$ also by $l_s - 1 \in (in/t, (i+1)n/t)$ (by $r_s \in [in/t, (i+2)n/t)$). Thus, $T[l, r)$ creates subtasks whereof one subtask starts at $A[l]$ (at $A[r_s]$), one subtask ends at $A[l_s - 1]$ (at $A[r-1]$), and subtasks cover the remaining elements in between without gaps. These subtasks are sequential subtasks as $\lfloor (l_s - 1)t/n \rfloor - \lfloor lt/n \rfloor \leq \lfloor ((i+1)n/t - 1)t/n \rfloor - \lfloor (in/t)t/n \rfloor = 0$ (as $\lfloor (r-1)t/n \rfloor - \lfloor r_s t/n \rfloor \leq \lfloor ((i+2)n/t - 1)t/n \rfloor - \lfloor (in/t)t/n \rfloor = 1$). And, these sequential subtasks are all added to thread i, as they start in the subarray $A[in/t, (i+1)n/t - 1]$ (in the subarray $A[in/t, (i+2)n/t - 1]$, the +2 is used as thread i is the rightmost thread of $T[l_s, r_s)$). Note that, in the penultimate sentence, we used the inequality $l \leq in/t$ (the inequality $r \leq (i+2)n/t$ from the induction hypothesis. When these subtasks were added to thread i, the sequential tasks of thread i still cover a consecutive sequence of elements: On the one hand, the leftmost (rightmost) sequential subtask starts at $A[l]$ (ends at $A[r-1]$,) and the new sequential subtasks have no gaps in between. On the other hand, we know from the induction hypothesis that the rightmost (leftmost) sequential task of thread i had ended at position $l - 1$ (had started at position r) and that the old sequential tasks of thread i had not had gaps in between.

□

C MORE MEASUREMENTS

Fig. 18. Running times of sequential algorithms of uint32 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, WikiSort, and LearnedSort cannot be seen as their running times exceed the plot.

Type	Distribution	I1S⁴o	BlockPDQ	BlockQ	1S⁴o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	LearnedSort	IPS²Ra
double	Sorted	1.11	1.77	20.31	1.04	11.54	17.14	**1.03**	51.78	2.90	16.90	49.53	46.50	
double	ReverseSorted	**1.04**	1.87	15.20	1.08	5.57	6.40	1.07	25.97	6.08	9.18	24.82	22.38	
double	Zero	1.14	1.93	17.48	1.11	1.29	13.56	**1.02**	2.93	3.72	13.09	18.38	13.69	
double	Exponential	**1.05**	1.10	1.24	1.28	2.31	2.56	4.15	3.79	3.98	1.27	1.12	2.33	
double	Zipf	1.19	1.32	1.44	1.45	2.86	3.07	4.77	4.23	4.84	1.26	**1.14**	3.14	
double	RootDup	**1.10**	1.39	1.73	1.62	1.51	2.50	1.50	5.43	2.81	1.70	2.43	3.33	
double	TwoDup	1.21	1.33	1.40	1.42	2.50	2.73	2.93	3.27	3.12	**1.11**	1.14	3.15	
double	EightDup	**1.02**	1.08	1.36	1.27	2.46	2.78	4.28	4.29	4.07	1.21	1.35	2.38	
double	AlmostSorted	2.22	**1.05**	1.87	2.79	1.57	1.62	1.25	5.85	2.26	2.05	4.22	4.39	
double	Uniform	1.13	1.24	1.27	1.32	2.47	2.57	3.62	2.94	3.56	1.14	**1.07**	2.13	
Total		1.23	**1.21**	1.46	1.53	2.19	2.51	2.89	4.15	3.43	1.36	1.55	3.04	
Rank		2	1	4	5	7	8	9	12	11	3	6	10	
uint64	Sorted	1.10	1.77	17.70	1.06	8.93	16.60	**1.06**	42.12	2.82	17.88	61.80	79.74	11.03
uint64	ReverseSorted	**1.02**	1.73	14.04	1.08	4.92	6.25	1.03	22.35	6.16	10.28	35.04	40.14	6.71
uint64	Zero	1.10	1.50	15.49	1.04	1.08	12.05	**1.04**	2.35	3.34	12.65	16.71	15.46	1.29
uint64	Exponential	1.09	1.22	1.35	1.40	2.65	2.84	4.69	3.74	4.62	1.23	1.37	2.50	**1.05**
uint64	Zipf	1.45	1.71	1.92	1.93	3.54	3.82	6.08	5.02	6.23	1.60	1.50	3.07	**1.04**
uint64	RootDup	**1.06**	1.44	1.77	1.70	1.43	2.55	1.64	5.16	3.24	1.70	2.28	3.33	1.08
uint64	TwoDup	1.55	1.84	1.89	2.00	3.34	3.57	4.02	4.07	4.29	1.37	2.32	3.07	**1.00**
uint64	EightDup	1.20	1.32	1.56	1.58	2.77	3.15	5.07	4.76	5.03	1.49	2.68	2.71	**1.02**
uint64	AlmostSorted	2.13	**1.06**	1.85	3.03	1.52	1.71	1.35	5.36	2.41	2.37	6.14	8.39	1.23
uint64	Uniform	1.28	1.47	1.51	1.63	2.84	2.97	4.24	3.18	4.32	1.17	1.57	4.86	**1.05**
Total		1.36	1.42	1.68	1.84	2.45	2.86	3.42	4.40	4.14	1.52	2.24	4.99	**1.06**
Rank		2	3	5	6	8	9	10	12	11	4	7	13	1
uint32	Sorted	2.84	4.29	49.59	2.87	24.36	60.07	**1.94**	121.52	6.43	35.09	48.04	263.84	27.84
uint32	ReverseSorted	1.55	2.27	20.24	1.46	6.38	11.16	**1.01**	31.74	5.86	9.79	29.71	61.66	8.44
uint32	Zero	2.56	3.97	48.98	2.53	2.26	33.81	**1.94**	6.54	9.05	20.41	12.16	37.89	3.12
uint32	Exponential	1.54	1.85	2.07	1.89	4.37	4.57	7.00	5.93	6.71	1.47	**1.03**	4.73	1.18
uint32	Zipf	1.89	2.31	2.65	2.40	5.27	5.67	8.57	7.40	8.91	1.33	1.20	5.23	**1.18**
uint32	RootDup	1.19	1.55	1.97	1.85	1.63	2.76	1.44	5.98	3.15	1.23	1.52	3.86	**1.11**
uint32	TwoDup	1.93	2.46	2.50	2.46	5.05	5.07	5.07	5.20	5.47	1.22	1.46	5.22	**1.10**
uint32	EightDup	1.34	1.64	1.99	1.77	4.17	4.56	6.50	5.74	6.43	1.22	1.83	3.19	**1.01**
uint32	AlmostSorted	2.65	1.25	2.21	3.50	1.83	2.74	**1.14**	6.79	2.45	2.08	4.92	10.36	1.33
uint32	Uniform	1.75	2.05	2.06	2.04	4.10	4.23	5.89	4.55	5.91	1.41	**1.00**	5.72	1.32
Total		1.70	1.83	2.19	2.21	3.46	4.09	4.09	5.88	5.15	1.40	1.58	6.56	**1.17**
Rank		4	5	6	7	8	10	9	12	11	2	3	13	1
Pair	Sorted	1.12	1.57	13.51	1.04	7.57	12.35	**1.02**	28.08	2.31	13.08			8.61
Pair	ReverseSorted	1.11	1.41	9.31	**1.01**	3.78	4.63	1.05	14.28	7.20	6.49			5.00
Pair	Zero	1.16	1.65	10.91	1.05	1.08	10.21	**1.03**	1.97	2.74	9.02			1.22
Pair	Exponential	1.15	2.05	1.29	1.38	2.11	2.45	4.26	3.19	4.18	1.25			**1.05**
Pair	Zipf	1.45	2.75	1.67	1.82	2.69	2.84	4.82	3.73	5.27	1.48			**1.02**
Pair	RootDup	1.20	1.46	1.68	1.71	1.44	2.30	1.86	4.39	3.78	1.60			**1.03**
Pair	TwoDup	1.74	3.04	1.83	2.01	2.90	3.10	3.74	3.56	4.41	1.47			**1.00**
Pair	EightDup	1.30	2.39	1.53	1.65	2.28	2.65	4.51	3.97	4.93	1.46			**1.01**
Pair	AlmostSorted	2.73	**1.02**	2.29	3.40	1.86	2.06	2.29	5.47	3.91	2.58			1.48
Pair	Uniform	1.41	2.54	1.47	1.71	2.46	2.48	3.82	2.88	4.22	1.24			**1.00**
Total		1.50	2.06	1.66	1.88	2.20	2.53	3.43	3.81	4.36	1.54			**1.07**
Rank		2	6	4	5	7	8	9	10	11	3			1
Quartet	Uniform	1.06	1.91	1.26	1.39	1.92	1.78	3.08	2.01	3.22	**1.04**			
Rank		2	6	3	4	7	5	9	8	10	1			
100B	Uniform	1.21	1.16	1.13	1.51	1.52	1.21	2.02	1.55	2.65	**1.09**			
Rank		4	3	2	6	7	5	9	8	10	1			

Table 7. Average slowdowns of sequential algorithms for different data types and input distributions on I4x20. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	$I1S^4o$	BlockPDQ	BlockQ	$1S^4o$	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	LearnedSort	IPS^2Ra
double	Sorted	**1.01**	1.76	35.11	1.09	15.38	21.09	1.10	76.15	2.58	28.91	72.53	65.62	
double	ReverseSorted	1.02	1.87	15.19	**1.01**	5.04	5.53	1.10	26.62	6.54	10.39	25.48	22.04	
double	Zero	**1.03**	1.83	31.36	1.09	1.25	17.36	1.10	2.50	3.36	21.07	32.62	18.57	
double	Exponential	**1.02**	1.29	1.59	1.23	2.45	2.69	4.36	4.42	4.48	1.46	1.38	1.86	
double	Zipf	**1.05**	1.46	1.73	1.26	2.70	2.86	4.64	4.43	4.86	1.34	1.27	2.00	
double	RootDup	**1.13**	1.99	2.51	1.75	1.61	2.43	1.34	6.93	3.54	2.44	2.98	3.22	
double	TwoDup	1.11	1.38	1.46	1.27	2.40	2.46	2.82	3.45	3.11	1.11	**1.08**	2.27	
double	EightDup	**1.00**	1.38	1.73	1.25	2.60	3.09	4.62	5.25	4.82	1.60	1.70	3.02	
double	AlmostSorted	2.20	1.48	2.47	2.90	1.58	1.58	**1.01**	6.79	2.58	2.51	3.53	4.61	
double	Uniform	1.06	1.25	1.34	1.22	2.36	2.39	3.58	3.00	3.61	1.18	**1.05**	2.03	
Total		**1.18**	1.45	1.79	1.48	2.20	2.45	2.78	4.69	3.77	1.58	1.66	2.90	
Rank		1	2	6	3	7	8	9	12	11	4	5	10	
uint64	Sorted	1.27	1.84	35.98	**1.00**	15.29	23.94	1.40	70.86	3.38	39.54	100.69	121.56	16.53
uint64	ReverseSorted	**1.01**	1.79	13.80	1.01	4.27	5.30	**1.01**	20.68	6.63	11.26	28.76	34.85	6.19
uint64	Zero	1.24	1.79	35.08	**1.00**	1.19	15.68	1.41	2.31	4.18	24.10	36.80	17.26	1.35
uint64	Exponential	1.06	1.42	1.72	1.31	2.32	2.72	4.82	4.20	4.88	1.29	1.79	2.13	**1.04**
uint64	Zipf	1.78	2.52	3.14	2.24	4.21	4.77	7.84	6.78	8.28	2.37	2.38	3.43	**1.00**
uint64	RootDup	1.62	2.81	3.93	2.90	2.23	3.47	2.32	9.39	5.50	3.38	3.92	4.93	**1.00**
uint64	TwoDup	2.05	2.81	3.01	2.44	4.45	4.86	5.68	6.00	6.44	2.14	3.06	4.39	**1.00**
uint64	EightDup	1.42	1.72	2.48	1.70	3.15	3.97	6.55	6.13	6.40	2.25	4.15	2.71	**1.02**
uint64	AlmostSorted	2.19	1.26	2.45	3.20	1.56	1.65	**1.13**	5.94	2.94	2.89	6.59	8.09	1.18
uint64	Uniform	1.44	1.95	2.07	1.74	3.10	3.43	5.37	4.19	5.44	1.38	2.09	5.14	**1.02**
Total		1.61	1.98	2.60	2.13	2.84	3.37	4.11	5.88	5.47	2.13	3.13	5.44	**1.03**
Rank		2	3	6	4	7	9	10	13	12	5	8	11	1
uint32	Sorted	2.15	3.19	67.26	**2.10**	31.01	47.01	2.18	149.90	5.53	62.99	38.38	262.40	32.72
uint32	ReverseSorted	1.24	2.08	18.46	1.32	6.12	7.30	**1.07**	29.95	6.22	11.91	16.88	49.89	7.38
uint32	Zero	2.32	3.12	81.68	2.37	**2.02**	33.24	2.34	5.00	8.11	28.89	16.21	37.73	2.66
uint32	Exponential	1.49	1.99	2.59	1.91	3.63	4.11	7.14	6.41	7.00	1.55	**1.05**	3.52	1.05
uint32	Zipf	1.93	3.06	3.89	2.60	5.45	5.94	9.91	8.60	9.80	2.04	1.28	4.50	**1.06**
uint32	RootDup	1.74	3.34	4.51	3.14	2.60	4.03	2.14	11.16	5.37	2.89	2.20	5.97	**1.00**
uint32	TwoDup	2.27	3.18	3.51	2.88	5.32	5.86	6.77	7.21	7.00	1.69	1.24	6.96	**1.02**
uint32	EightDup	1.55	2.17	2.84	1.93	3.92	4.67	7.66	7.48	7.41	1.82	2.13	4.46	**1.02**
uint32	AlmostSorted	2.82	1.69	2.91	4.26	1.96	1.92	**1.00**	7.71	2.97	2.72	4.37	10.75	1.39
uint32	Uniform	1.75	2.33	2.66	2.31	4.27	4.50	6.87	5.22	6.57	1.67	**1.02**	5.87	1.16
Total		1.89	2.46	3.21	2.62	3.67	4.21	4.74	7.50	6.25	2.00	1.66	7.24	**1.09**
Rank		3	5	7	6	8	9	10	13	11	4	2	12	1
Pair	Sorted	1.03	1.77	23.06	**1.01**	12.03	17.28	1.04	44.20	2.29	24.60			13.90
Pair	ReverseSorted	1.05	1.18	8.67	**1.04**	3.83	4.29	1.04	13.90	7.30	7.52			5.51
Pair	Zero	1.02	1.66	18.17	**1.02**	1.09	14.29	1.03	2.20	2.84	15.14			1.27
Pair	Exponential	1.13	2.04	1.28	1.17	1.92	2.18	3.83	3.30	4.46	1.14			**1.08**
Pair	Zipf	1.46	2.80	1.74	1.56	2.64	2.91	5.04	3.91	5.82	1.54			**1.02**
Pair	RootDup	1.49	1.78	2.44	1.96	1.77	2.60	2.04	5.89	4.98	2.24			**1.00**
Pair	TwoDup	1.65	2.98	1.81	1.67	2.85	2.96	3.82	3.65	4.76	1.52			**1.00**
Pair	EightDup	1.32	2.28	1.63	1.40	2.29	2.61	4.91	4.22	5.01	1.74			**1.00**
Pair	AlmostSorted	3.63	**1.00**	3.48	4.50	2.54	2.64	2.41	7.49	5.13	3.90			2.09
Pair	Uniform	1.42	2.63	1.65	1.53	2.60	2.65	4.23	3.21	4.77	1.22			**1.05**
Total		1.60	2.10	1.91	1.78	2.34	2.64	3.58	4.32	4.97	1.75			**1.14**
Rank		2	6	5	4	7	8	9	10	11	3			1
Quartet	Uniform	1.15	1.89	1.46	1.34	1.91	1.98	3.37	2.38	3.89	**1.01**			
Rank		2	5	4	3	6	7	9	8	10	1			
100B	Uniform	1.52	1.35	1.45	1.54	2.17	1.45	2.42	2.06	3.75	**1.01**			
Rank		5	2	4	6	8	3	9	7	10	1			

Table 8. Average slowdowns of sequential algorithms for different data types and input distributions on A1x16. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	I1S⁴o	BlockPDQ	BlockQ	1S⁴o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	LearnedSort	IPS²Ra
double	Sorted	1.07	1.83	31.60	**1.01**	14.48	28.79	1.02	92.63	3.93	27.46	94.22	75.36	
double	ReverseSorted	**1.03**	1.70	19.10	1.06	5.72	7.75	1.14	36.85	7.45	12.07	37.44	27.41	
double	Zero	1.09	1.79	26.88	**1.01**	1.15	19.12	1.05	3.02	4.87	21.95	28.47	19.26	
double	Exponential	**1.00**	1.11	1.23	1.29	2.34	2.69	4.44	4.42	4.19	1.19	1.65	2.21	
double	Zipf	**1.08**	1.23	1.38	1.48	2.79	3.11	5.01	4.70	4.79	1.08	1.39	2.40	
double	RootDup	**1.04**	1.23	1.52	1.51	1.23	2.12	1.34	6.01	2.57	1.59	2.39	2.93	
double	TwoDup	1.20	1.35	1.38	1.52	2.60	2.85	3.19	3.79	3.18	**1.02**	1.45	2.69	
double	EightDup	**1.00**	1.06	1.31	1.33	2.43	2.90	4.64	5.02	4.35	1.16	1.61	2.49	
double	AlmostSorted	2.29	**1.01**	2.07	2.76	1.60	1.87	1.30	7.43	2.22	2.26	5.61	4.73	
double	Uniform	1.05	1.20	1.20	1.34	2.43	2.54	3.82	3.26	3.63	**1.01**	1.78	2.11	
Total		1.18	**1.16**	1.42	1.55	2.13	2.55	3.00	4.79	3.45	1.28	2.00	2.98	
Rank		2	1	4	5	7	8	10	12	11	3	6	9	
uint64	Sorted	1.11	1.83	28.42	**1.02**	13.30	26.46	1.05	84.18	3.33	34.08	116.92	130.41	16.69
uint64	ReverseSorted	1.06	1.73	17.76	**1.02**	5.28	7.00	1.04	32.52	7.72	13.81	43.72	47.28	7.59
uint64	Zero	1.10	1.68	25.90	1.03	1.11	17.94	**1.01**	2.73	4.42	22.45	28.05	17.99	1.47
uint64	Exponential	**1.02**	1.15	1.24	1.38	2.23	2.59	4.42	4.09	4.14	1.11	2.06	2.06	1.10
uint64	Zipf	1.28	1.56	1.71	1.85	3.20	3.61	5.76	5.23	5.72	1.27	1.75	2.05	**1.01**
uint64	RootDup	**1.06**	1.22	1.52	1.63	1.16	1.95	1.39	5.35	2.73	1.43	2.26	2.94	1.35
uint64	TwoDup	1.49	1.67	1.65	1.90	2.99	3.29	3.76	4.19	3.85	1.21	2.26	2.65	**1.00**
uint64	EightDup	1.16	1.20	1.47	1.58	2.51	3.00	4.95	5.04	4.74	1.36	2.54	2.27	**1.02**
uint64	AlmostSorted	2.37	**1.02**	1.91	3.08	1.60	1.76	1.41	6.81	2.39	2.69	7.30	8.47	1.21
uint64	Uniform	1.25	1.41	1.38	1.61	2.61	2.80	4.09	3.43	4.01	**1.00**	2.80	3.71	1.12
Total		1.32	1.30	1.54	1.80	2.21	2.64	3.25	4.77	3.79	1.37	2.66	4.32	**1.11**
Rank		3	2	5	6	7	8	10	13	11	4	9	12	1
uint32	Sorted	2.82	4.45	83.97	2.32	35.39	95.00	**2.00**	234.86	8.93	61.73	92.69	426.73	42.36
uint32	ReverseSorted	1.47	2.38	26.97	1.51	7.54	12.80	**1.00**	49.47	7.45	14.09	51.86	86.09	10.10
uint32	Zero	2.51	4.09	80.92	**1.99**	2.49	54.23	2.11	7.59	12.12	34.89	17.80	58.18	4.03
uint32	Exponential	1.29	1.53	1.67	1.62	3.20	3.56	5.92	5.77	5.61	1.13	1.11	2.80	**1.05**
uint32	Zipf	1.67	2.10	2.40	2.18	4.76	5.14	8.11	7.72	7.94	**1.09**	1.34	3.13	1.22
uint32	RootDup	1.35	1.43	1.85	1.70	1.41	2.42	1.26	6.66	2.76	**1.09**	1.67	3.61	1.61
uint32	TwoDup	1.86	2.16	2.22	2.15	4.12	4.40	4.86	5.67	4.89	**1.04**	1.78	3.97	1.18
uint32	EightDup	1.28	1.46	1.77	1.59	3.29	3.76	6.00	6.28	5.70	**1.04**	1.91	2.41	1.08
uint32	AlmostSorted	2.77	1.22	2.43	3.56	1.89	2.76	**1.09**	8.40	2.23	2.30	7.36	10.79	1.30
uint32	Uniform	1.35	1.62	1.56	1.63	3.20	3.25	4.79	4.01	4.53	**1.01**	1.12	3.99	1.20
Total		1.59	1.61	1.96	1.98	2.91	3.51	3.68	6.22	4.45	**1.19**	1.84	5.51	1.22
Rank		3	4	6	7	8	9	10	13	11	1	5	12	2
Pair	Sorted	1.08	1.72	19.73	1.02	9.96	19.02	**1.01**	51.62	2.65	20.73			11.81
Pair	ReverseSorted	1.07	1.18	10.15	1.08	3.48	4.63	**1.07**	17.97	7.55	7.46			5.07
Pair	Zero	1.12	1.64	17.10	**1.01**	1.05	16.22	1.12	2.13	3.05	13.49			1.15
Pair	Exponential	**1.04**	1.94	1.18	1.52	1.82	2.07	3.87	3.35	4.25	1.10			1.07
Pair	Zipf	1.39	2.68	1.53	2.06	2.53	2.72	4.83	4.09	5.25	1.25			**1.00**
Pair	RootDup	**1.05**	1.15	1.39	1.66	1.09	1.76	1.65	4.31	3.24	1.23			1.12
Pair	TwoDup	1.48	2.67	1.56	2.00	2.47	2.66	3.35	3.55	3.95	1.21			**1.02**
Pair	EightDup	1.24	2.20	1.40	1.78	2.08	2.47	4.52	4.35	4.85	1.37			**1.00**
Pair	AlmostSorted	3.40	**1.00**	2.62	4.12	2.17	2.50	2.80	7.94	4.55	3.17			1.66
Pair	Uniform	1.29	2.39	1.35	1.76	2.24	2.34	3.75	2.98	4.00	**1.04**			1.08
Total		1.43	1.88	1.53	2.01	2.00	2.34	3.37	4.16	4.22	1.37			**1.12**
Rank		3	5	4	7	6	8	9	10	11	2			1
Quartet	Uniform	1.22	2.00	1.31	1.82	2.00	2.02	3.37	2.39	3.71	**1.02**			
Rank		2	5	3	4	6	7	9	8	10	1			
100B	Uniform	1.51	1.30	1.29	1.88	1.84	1.38	2.39	1.98	3.40	**1.04**			
Rank		5	3	2	7	6	4	9	8	10	1			

Table 9. Average slowdowns of sequential algorithms for different data types and input distributions on I2x16. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	I1S⁴o	BlockPDQ	BlockQ	1S⁴o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	LearnedSort	IPS²Ra
double	Sorted	**1.03**	1.64	29.22	1.11	16.69	22.54	1.24	70.63	2.51	27.84	66.47	67.60	
double	ReverseSorted	1.01	1.56	11.86	**1.00**	4.75	4.98	1.04	21.65	4.68	8.75	19.44	18.32	
double	Zero	**1.01**	1.64	21.64	1.17	1.11	16.84	1.19	2.62	3.03	18.42	32.61	16.30	
double	Exponential	**1.03**	1.10	1.20	1.24	2.62	2.92	4.78	4.32	4.79	1.36	1.32	2.17	
double	Zipf	**1.02**	1.11	1.27	1.26	2.95	3.12	4.92	4.28	5.06	1.08	1.21	2.16	
double	RootDup	1.14	1.64	1.96	1.86	1.79	2.69	**1.14**	7.04	3.71	2.32	3.24	3.98	
double	TwoDup	1.18	1.26	1.31	1.37	2.80	2.95	3.25	3.48	3.54	1.09	**1.09**	2.31	
double	EightDup	**1.02**	1.06	1.29	1.25	2.72	3.13	4.86	4.93	4.82	1.38	1.52	2.51	
double	AlmostSorted	3.03	1.22	3.02	4.47	2.43	2.33	**1.00**	9.01	3.56	3.52	4.81	6.97	
double	Uniform	1.09	1.15	1.15	1.25	2.64	2.68	3.96	3.07	3.95	1.10	**1.04**	2.13	
Total		1.25	**1.21**	1.51	1.61	2.54	2.82	2.89	4.83	4.16	1.53	1.71	3.49	
Rank		2	1	3	5	7	8	9	12	11	4	6	10	
uint64	Sorted	1.33	1.92	29.77	**1.00**	16.75	24.57	1.03	59.38	2.81	37.63	79.94	113.03	16.30
uint64	ReverseSorted	1.01	1.54	10.29	**1.01**	4.11	4.67	1.01	15.95	4.66	9.84	20.49	27.22	5.05
uint64	Zero	1.37	2.06	24.96	1.10	1.18	16.98	**1.01**	2.57	3.81	22.53	39.42	16.94	1.39
uint64	Exponential	**1.04**	1.18	1.33	1.34	2.51	2.89	5.09	3.73	5.07	1.26	1.62	2.06	1.04
uint64	Zipf	1.73	2.06	2.39	2.33	4.80	5.32	9.03	6.36	9.19	2.15	2.51	2.73	**1.00**
uint64	RootDup	1.59	2.32	2.90	2.77	2.42	3.61	1.80	7.98	5.65	3.15	4.26	4.56	**1.00**
uint64	TwoDup	2.04	2.47	2.55	2.62	4.94	5.36	6.36	5.47	6.95	1.99	3.41	4.35	**1.00**
uint64	EightDup	1.37	1.52	1.91	1.77	3.43	4.06	6.89	5.57	6.84	2.13	3.39	2.48	**1.00**
uint64	AlmostSorted	2.93	1.19	2.94	4.64	2.39	2.41	**1.03**	7.08	3.83	4.03	7.29	11.08	1.66
uint64	Uniform	1.43	1.73	1.73	1.82	3.59	3.78	5.86	3.75	5.84	1.34	2.02	4.99	**1.00**
Total		**1.65**	1.71	2.17	2.30	3.30	3.78	4.17	5.51	6.00	2.12	3.13	5.84	
Rank		2	3	5	6	8	9	10	11	13	4	7	12	
uint32	Sorted	2.48	4.51	67.12	2.80	37.74	54.21	**1.93**	139.72	6.14	64.10	28.28	343.53	35.51
uint32	ReverseSorted	1.42	1.88	13.03	1.55	5.38	6.34	**1.06**	19.68	4.27	8.92	7.12	40.41	5.83
uint32	Zero	2.19	3.95	60.33	2.38	2.21	41.48	**1.96**	6.01	7.85	27.01	17.69	42.95	2.97
uint32	Exponential	1.60	1.84	2.13	1.97	4.12	4.76	7.67	5.90	7.78	1.51	**1.00**	3.45	1.09
uint32	Zipf	2.04	2.51	3.01	2.67	6.12	6.86	10.77	7.84	11.07	1.66	1.25	4.11	**1.06**
uint32	RootDup	1.67	2.48	3.21	2.83	2.56	4.07	1.50	9.10	5.14	2.41	2.08	5.62	**1.00**
uint32	TwoDup	2.65	3.13	3.32	3.13	6.46	7.09	7.72	6.89	8.31	1.63	**1.09**	6.71	1.10
uint32	EightDup	1.53	1.81	2.31	1.93	4.31	5.07	7.80	6.48	7.89	1.55	1.30	2.97	**1.00**
uint32	AlmostSorted	5.23	2.10	4.95	8.25	3.93	3.96	**1.00**	12.63	5.42	5.34	5.24	23.34	2.70
uint32	Uniform	2.21	2.53	2.60	2.57	5.38	5.77	8.32	5.38	8.39	1.75	**1.02**	7.46	1.29
Total		2.21	2.31	2.97	2.94	4.51	5.24	4.84	7.49	7.49	2.03	**1.53**	10.24	
Rank		4	5	7	6	8	10	9	11	12	3	2	13	
Pair	Sorted	1.03	1.65	20.71	1.04	12.67	18.52	**1.03**	35.04	2.41	23.54			12.43
Pair	ReverseSorted	1.08	1.18	6.89	1.07	3.77	3.90	**1.05**	10.76	5.49	6.82			4.58
Pair	Zero	**1.02**	1.65	13.38	1.04	1.06	12.81	1.03	2.03	2.78	12.80			1.24
Pair	Exponential	1.06	2.00	1.09	1.20	1.96	2.18	4.16	2.72	4.40	1.13			**1.04**
Pair	Zipf	1.53	3.18	1.63	1.74	3.06	3.31	5.96	3.77	6.50	1.58			**1.00**
Pair	RootDup	1.62	1.90	2.06	2.17	1.86	2.85	1.96	5.37	5.22	2.22			**1.00**
Pair	TwoDup	1.67	3.23	1.69	1.86	3.10	3.34	4.26	3.32	4.99	1.55			**1.00**
Pair	EightDup	1.24	2.37	1.34	1.44	2.31	2.70	4.96	3.64	5.16	1.72			**1.00**
Pair	AlmostSorted	3.51	**1.00**	3.26	4.62	2.70	2.91	1.96	6.48	5.18	4.17			2.03
Pair	Uniform	1.41	2.82	1.40	1.58	2.71	2.81	4.62	2.75	4.90	1.24			**1.00**
Total		**1.60**	2.21	1.68	1.90	2.48	2.85	3.69	3.83	5.16	1.77			
Rank		2	6	3	5	7	8	9	10	11	4			
Quartet	Uniform	1.13	1.82	1.24	1.28	1.96	1.84	3.04	1.95	3.67	**1.01**			
Rank		2	5	3	4	8	6	9	7	10	1			
100B	Uniform	1.53	1.38	1.40	1.68	2.10	1.42	2.25	1.79	3.50	**1.01**			
Rank		5	2	3	6	8	4	9	7	10	1			

Table 10. Average slowdowns of sequential algorithms for different data types and input distributions on A1x64. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	$1S^4o$	S^4oS
double	Sorted	**1.00**	35.77
double	ReverseSorted	**1.00**	16.15
double	Zero	**1.00**	17.03
double	Exponential	**1.00**	1.41
double	Zipf	**1.00**	1.34
double	RootDup	**1.02**	1.39
double	TwoDup	**1.00**	1.23
double	EightDup	**1.00**	1.52
double	AlmostSorted	**1.02**	1.13
double	Uniform	**1.00**	1.23
Total		**1.01**	1.32
Rank		1	2
uint64	Sorted	**1.00**	37.39
uint64	ReverseSorted	**1.00**	15.50
uint64	Zero	**1.00**	16.41
uint64	Exponential	**1.00**	1.41
uint64	Zipf	**1.00**	1.31
uint64	RootDup	**1.02**	1.33
uint64	TwoDup	**1.00**	1.24
uint64	EightDup	**1.00**	1.48
uint64	AlmostSorted	**1.04**	1.11
uint64	Uniform	**1.00**	1.24
Total		**1.01**	1.30
Rank		1	2
uint32	Sorted	**1.00**	39.40
uint32	ReverseSorted	**1.01**	15.16
uint32	Zero	**1.00**	20.16
uint32	Exponential	**1.00**	1.49
uint32	Zipf	**1.00**	1.38
uint32	RootDup	**1.01**	1.45
uint32	TwoDup	**1.00**	1.25
uint32	EightDup	**1.00**	1.56
uint32	AlmostSorted	**1.04**	1.14
uint32	Uniform	**1.00**	1.25
Total		**1.01**	1.35
Rank		1	2
Pair	Sorted	**1.00**	24.42
Pair	ReverseSorted	**1.00**	10.47
Pair	Zero	**1.00**	12.16
Pair	Exponential	**1.00**	1.32
Pair	Zipf	**1.01**	1.20
Pair	RootDup	**1.02**	1.24
Pair	TwoDup	**1.00**	1.19
Pair	EightDup	**1.00**	1.37
Pair	AlmostSorted	**1.04**	1.07
Pair	Uniform	**1.01**	1.16
Total		**1.01**	1.22
Rank		1	2
Quartet	Uniform	**1.01**	1.12
Rank		1	2
100B	Uniform	1.09	**1.04**
Rank		2	1

Table 11. Average slowdowns of $1S^4o$ and S^4oS for different data types and input distributions. The slowdowns average over the machines and input sizes with at least 2^{18} bytes.

Fig. 19. Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x64. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

Fig. 20. Running times of parallel algorithms on different input distributions and data types of size D executed on machine I2x16. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

Fig. 21. Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x16. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.
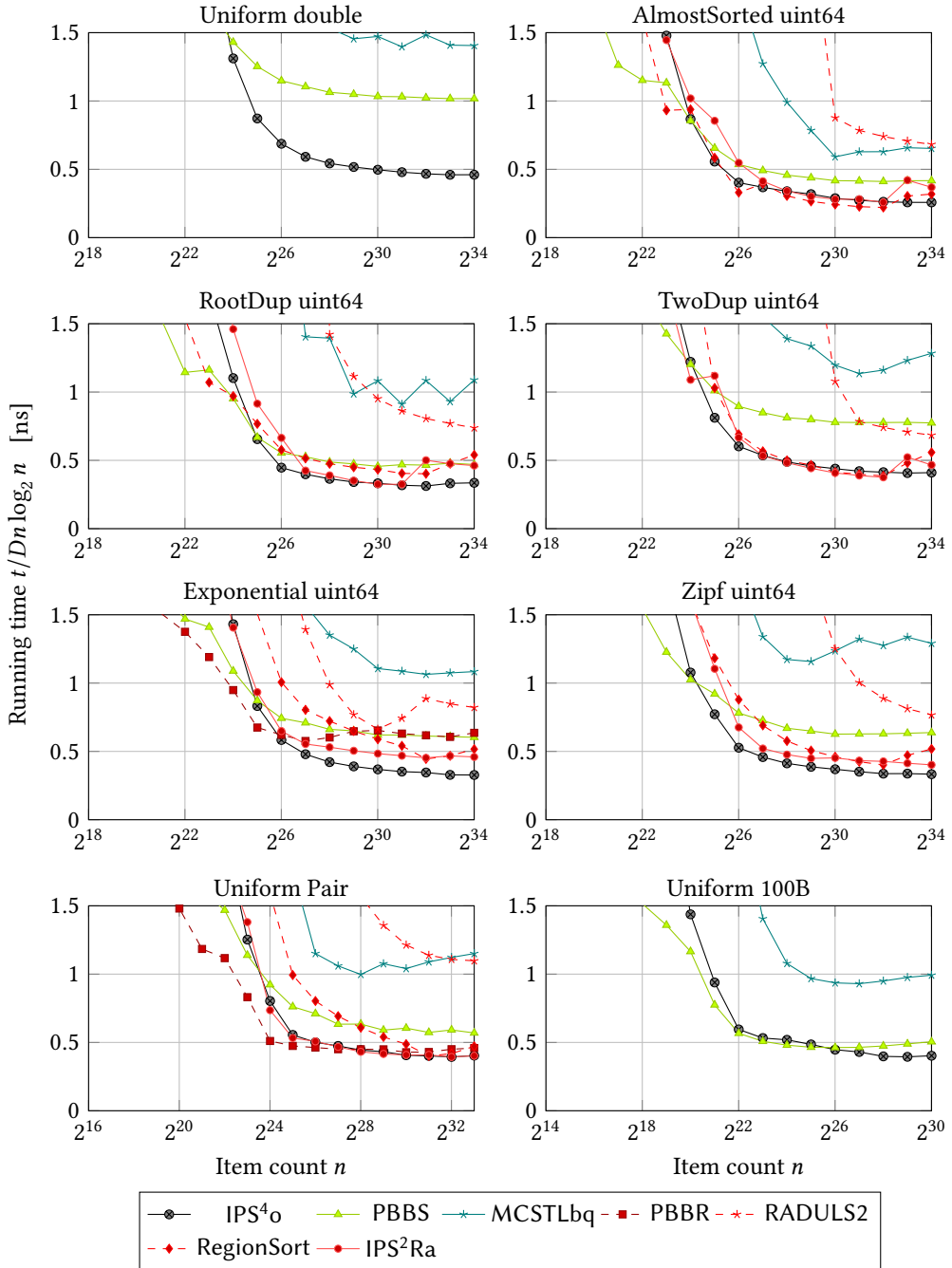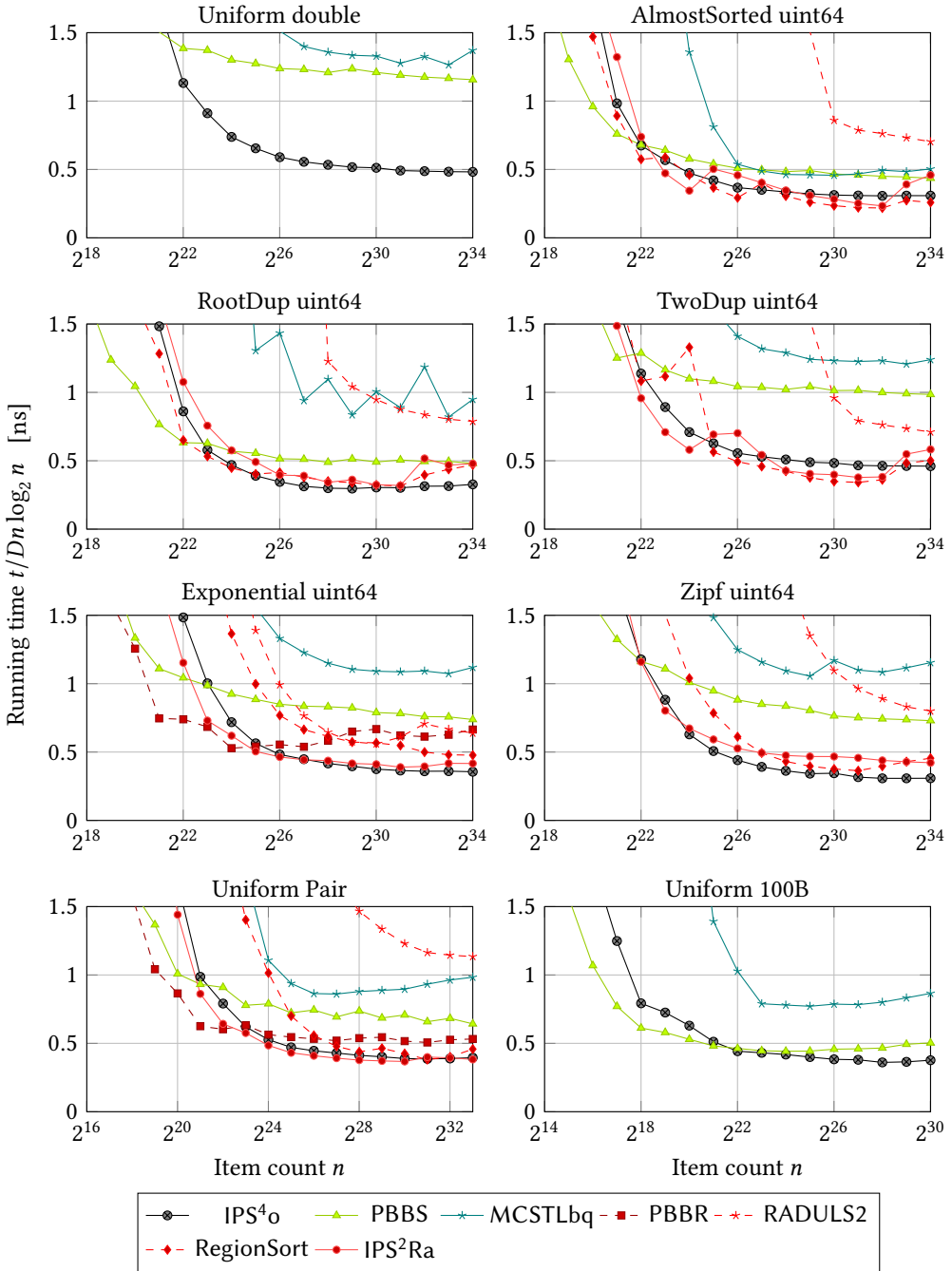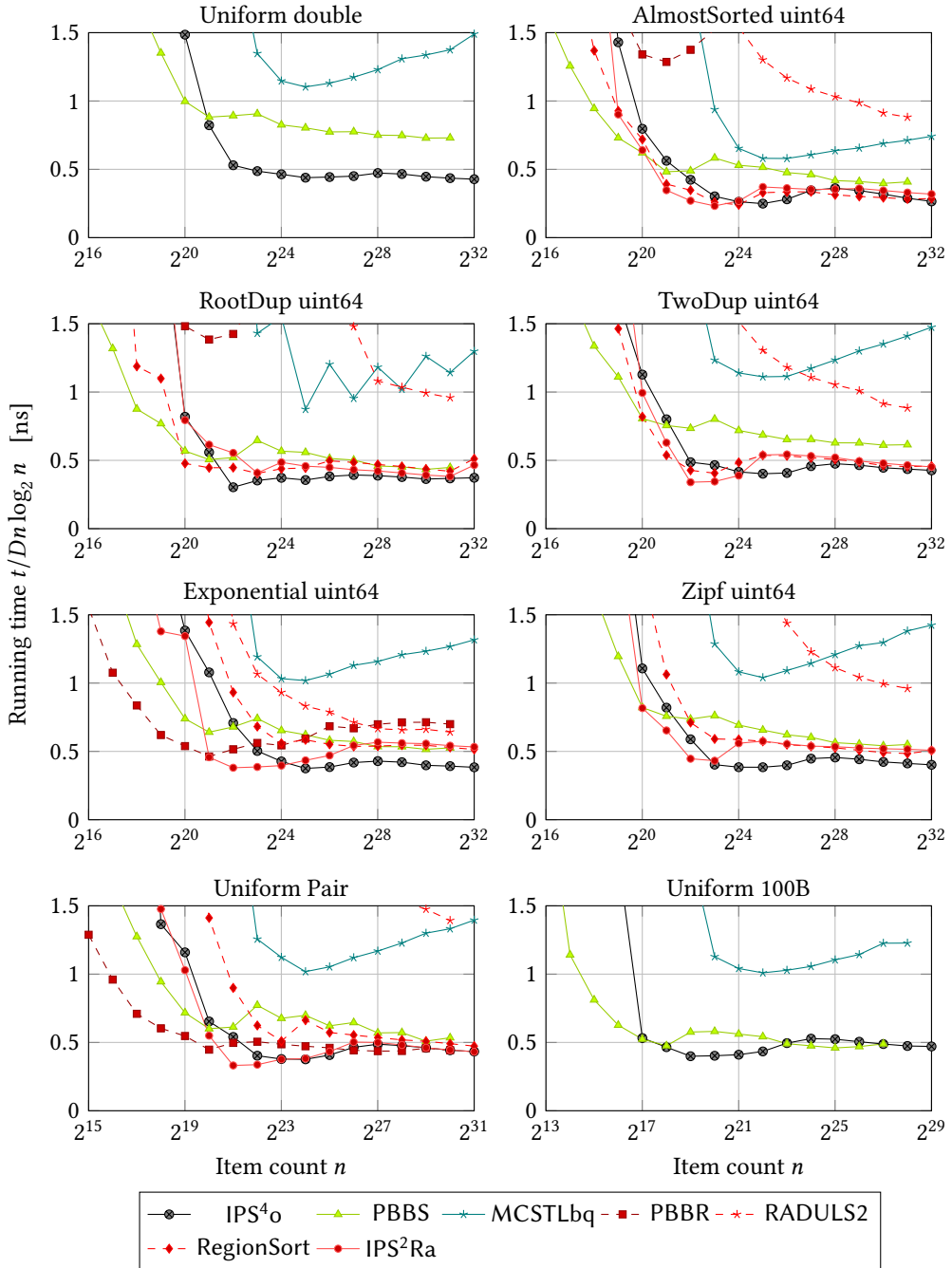
Fig. 22. Running times of parallel algorithms on different input distributions and data types of size D executed on machine I4x20. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

Type	Distribution	IPS⁴o	PBBS	PS⁴o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS²Ra
double	Sorted	1.04	14.24	1.36	17.74	22.71	**1.01**				65.29	
double	ReverseSorted	**1.09**	1.21	1.73	1.40	15.82	3.40				6.39	
double	Zero	1.04	12.29	1.30	19.88	319.78	**1.00**				64.20	
double	Exponential	**1.00**	1.82	1.87	2.45	3.37	15.64				5.57	
double	Zipf	**1.00**	1.89	1.98	2.51	3.25	16.17				5.99	
double	RootDup	**1.00**	1.45	2.02	2.20	3.74	6.33				6.78	
double	TwoDup	**1.00**	1.90	1.73	2.23	2.92	7.01				4.85	
double	EightDup	**1.00**	1.84	1.94	2.29	3.34	15.16				5.63	
double	AlmostSorted	**1.00**	1.50	2.12	4.12	2.57	3.43				7.15	
double	Uniform	**1.00**	1.96	1.70	2.33	3.00	12.65				4.77	
Total		**1.00**	1.75	1.90	2.53	3.15	9.57				5.76	
Rank		1	2	3	4	5	7				6	
uint64	Sorted	1.17	12.63	1.31	17.38	23.67	**1.00**	7.19	78.17	44.41		10.76
uint64	ReverseSorted	1.17	1.24	1.96	1.59	18.27	3.96	**1.07**	7.83	4.24		1.37
uint64	Zero	1.09	12.45	1.33	19.68	317.45	**1.00**	1.02	69.97	39.00		1.32
uint64	Exponential	**1.02**	1.61	1.97	2.39	3.32	14.06	1.63	1.51	2.61		1.29
uint64	Zipf	**1.00**	1.66	2.03	2.41	3.44	14.10	1.39	19.54	5.47		1.23
uint64	RootDup	**1.00**	1.35	2.06	2.20	3.62	7.54	1.33	9.04	5.70		1.23
uint64	TwoDup	**1.03**	1.74	1.87	2.25	3.11	7.34	1.11	10.11	3.43		1.08
uint64	EightDup	**1.00**	1.61	2.00	2.24	3.46	13.40	1.28	13.38	4.38		1.23
uint64	AlmostSorted	1.11	1.58	2.46	4.72	3.22	4.39	**1.05**	9.69	5.47		1.30
uint64	Uniform	1.11	1.96	2.01	2.59	3.15	13.15	1.40	1.26	1.27		**1.03**
Total		**1.04**	1.63	2.05	2.59	3.33	9.77	1.30	6.25	3.64		1.20
Rank		1	4	5	6	7	10	3	9	8		2
uint32	Sorted	**1.23**	9.48	1.74	10.22	17.28	2.09	4.87	7.39			4.96
uint32	ReverseSorted	1.67	1.84	2.56	1.87	16.85	8.19	**1.06**	1.39			1.16
uint32	Zero	1.09	13.43	1.35	22.64	474.99	**1.00**	1.01	89.40			1.38
uint32	Exponential	1.27	2.60	2.12	3.43	4.24	24.27	1.37	1.78			**1.00**
uint32	Zipf	1.06	2.32	1.94	3.07	3.90	22.24	1.16	6.03			**1.02**
uint32	RootDup	1.11	1.61	2.13	2.43	3.89	7.71	1.18	6.98			**1.08**
uint32	TwoDup	1.46	3.09	2.27	3.53	4.61	12.22	1.07	1.59			**1.00**
uint32	EightDup	1.24	2.66	2.13	3.21	3.99	23.06	1.16	1.54			**1.04**
uint32	AlmostSorted	1.51	1.99	2.60	5.20	3.69	5.49	1.12	1.52			**1.01**
uint32	Uniform	1.46	3.18	2.24	3.77	4.73	21.36	1.21	1.50			**1.01**
Total		1.29	2.43	2.20	3.44	4.13	14.54	1.18	2.30			**1.02**
Rank		3	6	4	7	8	9	2	5			1
Pair	Sorted	1.05	12.66	1.32	16.25	23.98	**1.00**	6.54	29.00	75.74		9.68
Pair	ReverseSorted	**1.11**	1.28	1.85	1.57	16.77	3.21	1.12	2.82	7.53		1.39
Pair	Zero	1.06	14.76	1.29	19.14	283.98	**1.00**	1.04	15.63	74.08		1.35
Pair	Exponential	1.21	1.59	2.27	2.33	3.41	8.89	1.93	**1.01**	9.32		1.62
Pair	Zipf	**1.00**	1.35	1.90	1.93	2.91	7.31	1.38	7.94	8.41		1.26
Pair	RootDup	**1.02**	1.24	1.87	2.00	3.49	5.12	1.26	3.38	9.61		1.28
Pair	TwoDup	**1.02**	1.37	1.88	1.86	2.84	4.37	1.24	4.69	6.20		1.17
Pair	EightDup	**1.02**	1.36	1.96	1.89	3.10	7.29	1.31	8.00	7.57		1.29
Pair	AlmostSorted	**1.07**	1.74	2.59	4.25	3.64	3.94	1.09	4.01	10.36		1.31
Pair	Uniform	1.08	1.50	1.98	2.04	2.94	7.26	1.47	**1.05**	4.39		1.07
Total		**1.06**	1.44	2.05	2.23	3.17	6.07	1.36	3.30	7.70		1.28
Rank		1	4	5	6	7	9	3	8	10		2
Quartet	Uniform	**1.03**	1.14	1.99	1.83	2.71	4.68					
Rank		1	2	4	3	5	6					
100B	Uniform	**1.05**	1.11	2.04	1.70	2.53	3.51					
Rank		1	2	4	3	5	6					

Table 12. Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine A1x64. The slowdowns average input sizes with at least $2^{21}t$ bytes.

Type	Distribution	IPS^{4}o	PBBS	PS^{4}o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS^{2}Ra
double	Sorted	2.47	18.27	2.81	22.46	19.43	**1.02**				60.33	
double	ReverseSorted	**1.05**	1.20	1.78	1.40	8.79	2.24				4.29	
double	Zero	2.17	17.91	2.30	26.99	292.91	**1.03**				60.63	
double	Exponential	**1.00**	2.06	1.92	2.76	2.79	11.65				4.17	
double	Zipf	**1.00**	2.36	2.08	3.16	3.33	13.48				4.65	
double	RootDup	**1.00**	1.63	2.29	2.62	3.53	5.71				5.71	
double	TwoDup	**1.00**	2.15	1.79	2.60	2.64	5.37				3.52	
double	EightDup	**1.00**	2.10	1.98	2.67	2.93	11.29				4.29	
double	AlmostSorted	**1.00**	1.47	2.07	4.21	1.57	2.72				5.05	
double	Uniform	**1.00**	2.23	1.78	2.70	2.65	9.35				3.43	
Total		**1.00**	1.97	1.98	2.92	2.71	7.54				4.34	
Rank		1	2	3	5	4	7				6	
uint64	Sorted	2.32	16.98	2.91	21.77	18.67	**1.01**	8.19	93.89	50.76		12.20
uint64	ReverseSorted	1.34	1.43	2.28	1.75	11.17	2.83	**1.00**	8.30	4.24		1.47
uint64	Zero	1.62	18.42	2.30	26.88	291.98	1.07	1.09	85.76	52.94		1.09
uint64	Exponential	**1.04**	1.98	2.06	2.79	3.01	11.02	1.58	1.45	1.96		1.08
uint64	Zipf	**1.00**	2.17	2.09	2.99	3.24	12.17	1.32	18.30	5.64		1.30
uint64	RootDup	**1.00**	1.55	2.27	2.53	3.50	5.64	1.17	9.66	6.40		1.26
uint64	TwoDup	1.19	2.36	2.10	2.91	3.12	6.18	**1.09**	11.02	3.39		1.15
uint64	EightDup	**1.05**	2.02	2.11	2.66	2.98	10.68	1.14	14.02	4.45		1.15
uint64	AlmostSorted	1.23	1.73	2.62	5.24	1.99	3.42	**1.04**	9.95	5.18		1.26
uint64	Uniform	1.21	2.50	2.15	3.09	3.11	10.31	1.36	1.54	1.15		**1.06**
Total		**1.10**	2.02	2.19	3.08	2.95	7.80	1.23	6.43	3.49		1.18
Rank		1	4	5	7	6	10	3	9	8		2
uint32	Sorted	3.33	14.36	3.55	14.59	18.15	**1.96**	6.28	8.67			6.47
uint32	ReverseSorted	1.94	2.12	2.80	2.07	12.90	5.32	**1.02**	1.28			1.14
uint32	Zero	1.97	19.35	1.99	32.52	473.11	**1.06**	1.08	105.42			1.09
uint32	Exponential	1.46	3.38	2.43	4.20	4.33	19.28	1.32	1.93			**1.00**
uint32	Zipf	1.10	2.99	2.03	3.73	3.90	17.67	**1.05**	5.83			1.10
uint32	RootDup	1.21	1.94	2.41	2.96	3.48	6.35	**1.01**	7.00			1.36
uint32	TwoDup	1.64	3.79	2.48	4.08	4.38	9.68	**1.04**	2.12			1.06
uint32	EightDup	1.38	3.48	2.43	3.95	4.29	18.54	1.10	1.84			**1.09**
uint32	AlmostSorted	1.72	2.25	2.76	6.11	2.87	4.77	1.17	1.34			**1.01**
uint32	Uniform	1.53	3.63	2.23	3.95	4.09	14.73	1.09	1.73			**1.08**
Total		1.42	2.98	2.39	4.06	3.87	11.54	1.11	2.43			**1.09**
Rank		3	6	4	8	7	9	2	5			1
Pair	Sorted	2.17	14.22	2.93	19.64	17.52	**1.03**	7.26	34.12	95.13		11.46
Pair	ReverseSorted	1.14	1.31	2.05	1.74	9.42	2.76	**1.03**	3.11	8.53		1.53
Pair	Zero	1.95	20.27	2.40	25.29	197.76	**1.03**	1.06	19.66	97.93		1.06
Pair	Exponential	**1.06**	1.49	2.02	2.36	2.45	6.71	1.52	1.07	8.42		1.20
Pair	Zipf	**1.00**	1.58	1.93	2.38	2.55	6.87	1.35	7.99	9.45		1.35
Pair	RootDup	**1.01**	1.34	2.05	2.31	3.06	4.89	1.16	4.30	10.97		1.13
Pair	TwoDup	**1.05**	1.65	1.99	2.30	2.48	4.08	1.15	4.92	6.91		1.17
Pair	EightDup	**1.02**	1.50	2.04	2.22	2.48	6.44	1.16	7.57	8.34		1.21
Pair	AlmostSorted	1.06	1.67	2.60	4.49	2.12	3.25	**1.03**	3.97	11.00		1.33
Pair	Uniform	1.06	1.73	2.01	2.41	2.39	6.19	1.36	1.32	4.74		**1.01**
Total		**1.04**	1.56	2.08	2.56	2.49	5.31	1.24	3.55	8.26		1.20
Rank		1	4	5	7	6	9	3	8	10		2
Quartet	Uniform	**1.00**	1.26	2.01	2.20	2.28	4.50					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.01**	1.16	1.94	1.93	2.16	3.33					
Rank		1	2	4	3	5	6					

Table 13. Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine I2x16. The slowdowns average input sizes with at least $2^{21}t$ bytes.

Type	Distribution	IPS^4o	PBBS	PS^4o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS^2Ra
double	Sorted	1.08	11.04	1.26	14.97	15.74	**1.00**				47.54	
double	ReverseSorted	**1.01**	1.15	1.45	1.53	3.63	1.48				5.11	
double	Zero	1.10	7.43	1.23	18.54	48.16	**1.00**				47.74	
double	Exponential	**1.00**	1.52	1.41	2.22	2.88	4.35				5.06	
double	Zipf	**1.00**	1.55	1.46	2.23	2.92	4.21				4.97	
double	RootDup	**1.00**	1.40	1.39	2.24	3.19	2.82				5.62	
double	TwoDup	**1.00**	1.62	1.43	2.11	2.81	2.68				4.64	
double	EightDup	**1.00**	1.52	1.46	2.29	2.85	4.42				5.20	
double	AlmostSorted	**1.00**	1.52	1.96	4.39	2.24	1.86				6.73	
double	Uniform	**1.00**	1.71	1.43	2.20	2.78	3.96				4.63	
Total		**1.00**	1.55	1.50	2.44	2.80	3.33				5.22	
Rank		1	3	2	4	5	6				7	
uint64	Sorted	1.05	10.76	1.21	14.83	15.67	**1.00**	6.04	42.27	27.60		7.78
uint64	ReverseSorted	1.07	1.23	1.58	1.64	3.86	1.58	**1.04**	5.03	3.12		1.27
uint64	Zero	1.06	7.41	1.17	18.31	47.38	**1.00**	1.01	34.98	30.32		1.06
uint64	Exponential	**1.03**	1.46	1.50	2.25	2.91	3.99	1.40	1.63	1.91		1.24
uint64	Zipf	**1.00**	1.47	1.49	2.22	2.94	3.82	1.29	6.92	3.55		1.27
uint64	RootDup	**1.00**	1.36	1.39	2.23	3.16	2.80	1.22	5.67	4.18		1.15
uint64	TwoDup	**1.04**	1.57	1.53	2.21	2.95	2.71	1.14	5.64	2.80		1.11
uint64	EightDup	**1.03**	1.45	1.50	2.29	2.90	4.06	1.22	7.00	3.27		1.22
uint64	AlmostSorted	1.08	1.66	2.16	4.78	2.42	2.05	**1.07**	6.63	4.23		1.18
uint64	Uniform	1.05	1.67	1.54	2.31	2.97	3.84	1.25	1.44	1.19		**1.02**
Total		**1.03**	1.52	1.57	2.51	2.88	3.23	1.22	4.08	2.79		1.17
Rank		1	4	5	6	8	9	3	10	7		2
uint32	Sorted	1.14	14.38	1.33	16.51	14.74	**1.13**	5.53	16.98			6.20
uint32	ReverseSorted	1.25	1.49	1.68	1.67	4.32	1.96	**1.02**	1.75			1.12
uint32	Zero	1.12	8.33	1.27	19.15	56.60	**1.00**	1.02	48.85			1.05
uint32	Exponential	1.10	2.20	1.60	2.70	3.18	6.38	1.27	1.64			**1.07**
uint32	Zipf	**1.02**	2.07	1.52	2.58	3.14	5.97	1.21	4.53			1.20
uint32	RootDup	**1.01**	1.68	1.53	2.31	3.21	3.51	1.19	5.33			1.13
uint32	TwoDup	1.12	2.36	1.57	2.61	3.14	3.75	1.10	1.75			**1.00**
uint32	EightDup	**1.04**	2.08	1.53	2.56	3.06	6.01	1.17	2.00			1.09
uint32	AlmostSorted	1.19	1.99	2.17	5.53	2.56	2.28	1.10	2.32			**1.04**
uint32	Uniform	1.17	2.56	1.57	2.79	3.17	5.69	1.13	1.43			**1.00**
Total		1.09	2.12	1.63	2.89	3.06	4.53	1.17	2.29			**1.07**
Rank		2	5	4	7	8	9	3	6			1
Pair	Sorted	1.06	12.94	1.20	14.57	15.62	**1.00**	6.13	17.80	55.45		7.71
Pair	ReverseSorted	**1.06**	1.51	1.54	1.68	3.87	1.51	1.10	2.11	6.43		1.28
Pair	Zero	1.07	9.92	1.15	18.42	44.53	**1.00**	1.01	8.84	59.46		1.09
Pair	Exponential	**1.04**	1.48	1.48	2.19	2.83	2.91	1.44	1.04	5.90		1.29
Pair	Zipf	**1.00**	1.45	1.45	2.09	2.78	2.77	1.32	3.22	6.72		1.31
Pair	RootDup	**1.00**	1.57	1.31	2.29	3.27	2.63	1.21	2.73	7.61		1.20
Pair	TwoDup	**1.01**	1.42	1.48	2.03	2.79	2.29	1.25	2.68	5.85		1.11
Pair	EightDup	**1.02**	1.48	1.49	2.19	2.81	3.01	1.17	3.28	6.60		1.26
Pair	AlmostSorted	**1.05**	1.98	2.06	4.18	2.44	1.86	1.09	2.75	8.50		1.15
Pair	Uniform	**1.04**	1.50	1.55	2.15	2.89	2.80	1.31	1.12	4.40		1.05
Total		**1.02**	1.54	1.53	2.37	2.82	2.58	1.25	2.20	6.39		1.19
Rank		1	5	4	7	9	8	3	6	10		2
Quartet	Uniform	**1.01**	1.19	1.45	1.96	2.55	2.17					
Rank		1	2	3	4	6	5					
100B	Uniform	**1.03**	1.12	1.48	2.00	2.43	2.09					
Rank		1	2	3	4	6	5					

Table 14. Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine A1x16. The slowdowns average input sizes with at least $2^{21}t$ bytes.

Type	Distribution	IPS⁴o	PBBS	PS⁴o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS²Ra
double	Sorted	1.38	4.36	3.57	8.71	3.91	**1.24**				11.66	
double	ReverseSorted	**1.08**	1.99	3.54	3.50	33.14	8.28				6.39	
double	Zero	2.23	15.45	2.95	3.79	161.14	**1.26**				11.06	
double	Exponential	**1.01**	1.89	3.03	3.00	4.00	17.67				5.42	
double	Zipf	**1.00**	2.05	3.38	3.34	5.28	20.29				6.38	
double	RootDup	**1.00**	1.68	3.85	3.18	5.64	9.79				8.00	
double	TwoDup	**1.00**	2.05	2.86	2.96	3.80	9.77				5.21	
double	EightDup	**1.00**	1.83	2.93	2.67	3.82	15.83				5.17	
double	AlmostSorted	**1.01**	2.83	4.03	9.66	2.62	10.32				7.06	
double	Uniform	**1.00**	2.08	2.75	2.97	3.76	15.88				5.24	
Total		**1.00**	2.03	3.23	3.56	4.02	13.66				5.99	
Rank		1	2	3	4	5	7				6	
uint64	Sorted	1.47	4.75	2.22	9.82	4.12	**1.45**	5.51	26.12	16.95		5.50
uint64	ReverseSorted	**1.10**	1.91	3.56	3.77	31.91	8.39	3.24	12.98	8.60		4.15
uint64	Zero	4.94	18.91	3.54	4.48	190.32	1.58	3.37	27.58	15.69		**1.21**
uint64	Exponential	**1.08**	1.96	3.20	3.14	4.87	19.91	3.05	1.80	4.86		1.23
uint64	Zipf	**1.00**	2.00	3.51	3.24	5.38	19.31	3.08	30.36	12.53		4.42
uint64	RootDup	**1.00**	1.65	3.87	3.27	5.71	9.94	3.84	19.67	16.42		3.43
uint64	TwoDup	**1.00**	1.97	2.89	2.83	3.73	9.85	2.22	15.49	7.37		2.55
uint64	EightDup	**1.01**	1.69	2.87	2.49	3.84	14.73	2.14	17.58	10.19		2.74
uint64	AlmostSorted	**1.00**	2.83	4.07	9.64	2.77	10.81	3.29	14.71	10.28		3.34
uint64	Uniform	1.15	2.30	3.20	3.31	4.30	16.91	2.87	1.40	3.02		**1.00**
Total		**1.03**	2.03	3.35	3.58	4.26	13.92	2.87	8.75	8.12		2.38
Rank		1	2	5	6	7	10	4	9	8		3
uint32	Sorted	**2.01**	4.80	7.19	7.18	9.39	3.03	4.44	3.46			2.84
uint32	ReverseSorted	**1.21**	1.93	2.91	2.68	23.08	8.82	2.15	1.42			1.27
uint32	Zero	2.75	29.11	4.61	8.71	533.66	1.97	5.36	52.76			**1.33**
uint32	Exponential	1.45	3.34	3.61	4.60	7.78	34.24	2.88	3.03			**1.02**
uint32	Zipf	**1.00**	2.81	3.06	3.52	5.84	26.87	2.31	10.95			3.26
uint32	RootDup	**1.00**	1.89	3.29	2.78	5.74	8.66	2.69	12.64			2.65
uint32	TwoDup	1.40	3.56	3.25	4.29	5.78	16.27	2.09	1.86			**1.03**
uint32	EightDup	1.27	3.25	3.27	4.07	6.40	28.26	2.25	2.07			**1.11**
uint32	AlmostSorted	**1.14**	2.06	3.05	5.78	4.15	7.46	2.25	1.52			1.28
uint32	Uniform	1.53	3.73	3.45	4.34	6.69	26.29	2.50	1.80			**1.00**
Total		**1.24**	2.86	3.28	4.11	5.96	18.42	2.41	3.04			1.44
Rank		1	4	6	7	8	9	3	5			2
Pair	Sorted	1.52	2.93	2.33	10.32	8.15	**1.07**	3.48	8.00	15.73		4.38
Pair	ReverseSorted	**1.07**	1.91	3.14	5.76	21.25	8.15	2.92	5.88	11.18		4.00
Pair	Zero	3.63	12.24	2.80	5.22	70.67	1.32	2.08	5.97	17.38		**1.15**
Pair	Exponential	1.20	2.90	3.64	5.12	4.04	14.27	3.53	**1.18**	18.28		1.53
Pair	Zipf	**1.00**	2.27	3.31	4.44	2.97	11.95	3.06	13.93	18.25		4.98
Pair	RootDup	**1.01**	2.58	3.81	6.39	6.71	9.15	4.01	9.30	24.77		3.42
Pair	TwoDup	**1.00**	2.46	3.05	4.22	4.07	7.30	2.54	9.19	13.48		3.48
Pair	EightDup	**1.02**	2.23	2.97	3.84	3.02	9.60	2.28	11.55	14.89		3.43
Pair	AlmostSorted	**1.00**	2.66	3.79	14.09	6.58	10.68	3.25	7.75	14.84		4.09
Pair	Uniform	1.13	2.81	3.37	4.69	3.77	12.17	3.20	1.32	9.45		**1.04**
Total		**1.05**	2.55	3.41	5.52	4.24	10.51	3.08	5.57	15.68		2.79
Rank		1	2	5	7	6	9	4	8	10		3
Quartet	Uniform	**1.01**	1.64	3.28	4.45	5.09	8.95					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.14**	1.17	3.61	4.73	8.11	7.00					
Rank		1	2	3	4	6	5					

Table 15. Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine I4x20. The slowdowns average input sizes with at least $2^{21}t$ bytes.