In-place Parallel Super Scalar Samplesort (IPS^{4}o)

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders

Karlsruhe Institute of Technology, Karlsruhe, Germany
`{michael.axtmann,sascha.witt,sanders}@kit.edu`

---- **Abstract** --

We present a sorting algorithm that works in-place, executes in parallel, is cache-optimal, avoids branch-mispredictions, and performs work $O(n \log n)$ for arbitrary inputs with high probability. We ran extensive experiments and show that our algorithm scales linearly in the number of cores on various multi-socket machines with 32 cores. On large inputs, we outperform our closest in-place competitor by a factor of 2.25 to 2.53 and our closest non-in-place competitor by a factor of 1.26 to 1.89. Even sequentially executed, we outperform our closest sequential competitor, BlockQuicksort, by a factor of 1.19 to 1.23 for large inputs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases shared memory, parallel sorting, in-place algorithm, comparison-based sorting, branch prediction

1 Introduction

Sorting an array $A[1..n]$ of n 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 (for example in Kruskal's algorithm for minimum spanning trees). 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., [5, 29, 30, 31]), instruction parallelism (e.g., [28]), branch prediction (e.g., [10, 18, 19, 28]), caches (e.g., [5, 7, 11, 28]), or virtual memory (e.g., [17, 25]). 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. 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 – it needs $O(n \log n)$ expected work, it can be parallelized [30, 31], it can be implemented to avoid branch mispredictions [10] and it is reasonably cache-efficient. Perhaps most importantly, quicksort works (almost) in-place[1] which is of crucial importance for very large inputs. This feature rules out many contenders. Further algorithms are eliminated by the requirement to work for arbitrary data types and arbitrary input types. This makes algorithms based on integer sorting like radix sort (e.g., [21]) or using specialized hardware (e.g., GPUs or SIMD instructions) much less attractive, since these algorithms cannot be used in a reusable library like the C++ standard library where they have to work for arbitrary data types. Another portability issue is that the algorithm should use no code

---

[1] 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 sublinear in the input size. We adopt this convention but use the term *almost in-place* when we want to make clear what we mean. Quicksort needs logarithmic additional space.

specific for the processor architecture or the operating system. For example, non-temporal writes and allocation of more virtual memory than physical memory as previously used for radix sort [27] could also be used for speeding up comparison-based algorithms. One aspect of making an algorithm in-place is that such "tricks" are not needed. Hence, this paper focusses on comparison-based portable algorithms and also considers how the algorithms can be made robust for arbitrary inputs, e.g., with a large number of repeated keys.

The main contribution of this paper is to propose a new algorithm – **In-place Parallel Super Scalar Samplesort** (IPS^{4}o)^2 – that combines enough advantages to become an attractive replacement of quicksort. Our starting point is *super scalar samplesort* (s^3-sort) [28] which already provides a very good sequential non-in-place algorithm that is cache-efficient, allows considerable instruction parallelism and avoids branch mispredictions. s^3-sort is a variant of samplesort, 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. We parallelize this algorithm using t threads and make it more robust by taking advantage of inputs with many identical keys. Our main innovation is to make the algorithm in-place. The first phase of IPS^{4}o distributes the elements to k *buffer* blocks. When a buffer 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 step handles empty blocks and half-filled buffer blocks. The distribution 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 are small enough, they can be solved independently in parallel.

After discussing related work in Section 2 and introducing basic tools in Section 3, we describe our new algorithm IPS^{4}o in Section 4. Section 5 makes an experimental evaluation. An overall discussion and possible future work is given in Section 6. The appendix gives further experimental data and proofs.

2   Related Work

Variants of Hoare's quicksort [15,24] 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 the pivot are in the right part. The subproblems are solved recursively. A variant of quicksort (with a fallback to heapsort to avoid worst case scenarios) is currently used in the C++ standard library of GCC [24]. Some variants of quicksort use two or three pivots [22,32] and achieve improvements of around 20% in running time over the single-pivot case. Dual-pivot quicksort [32] is the default sorting routine in Oracle Java 7 and 8. The basic principle of quicksort remains, but elements are partitioned into three or four subproblems instead of two. Increasing the number of subproblems (from now on called *buckets*) even further leads to samplesort [5,6]. Unlike single- and dual-pivot quicksort, samplesort is usually not in-place, but it is well-suited for parallelization and more cache-efficient.

Super scalar samplesort [28] (s^3-sort) improves on 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 [19]. By replacing conditional branches with conditionally executed machine instructions, branch

---

^{2} The Latin word "ipso" means "by itself" referring to the in-place feature of IPS^{4}o.

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^3-sort is up to two times faster than quicksort (`std::sort`), at the cost of $O(n)$ additional space. BlockQuicksort [10] applies similar ideas to single-pivot quicksort, resulting in a very fast in-place sorting algorithm. In fact, we use BlockQuicksort to sort small subproblems in the base case of our recursion, as experiments showed it to be faster than other alternatives.

Super scalar samplesort has also been adapted for efficient parallel string sorting [4]. Our implementation is influenced by that work with respect to parallelization and handling equal keys. Moreover, we were also influenced by an implementation of s^3-sort written by Lorenz Hübschle-Schneider. A prototypical implementation of sequential non-blocked in-place s^3-sort in a student project by our student Florian Weber motivated us to develop IPS^{4}o.

The best practical comparison-based multi-core sorting algorithms we have found are based on multi-way mergesort [30] and samplesort [29], respectively. The former algorithm is used in the parallel mode of the C++ standard library of GCC. Parallel in-place algorithms are based on quicksort so far. Intel's Thread Building Blocks library [26] contains a variant that uses only sequential partitioning. The MCSTL library [30] contains two implementations of the more scalable parallel quicksort by Tsigas and Zhang [31].

There is a considerable amount of work by the theory community on (strictly) in-place sorting (e.g., [11, 12]). However there are few – mostly negative – results on transferring these results into practice. Katajainen and Teuhola [20] report that in-place mergesort is slower than heapsort, which is quite slow for big inputs due to its cache-inefficiency. Chen [8] reports that in-place merging takes about six times longer than non-in-place merging. There is previous work on (almost) in-place multi-way merging or data distribution. However, few of these papers seem to address parallelism. There are also other problems. For example, the multi-way merger in [14] needs to allocate very large blocks to become efficient. By contrast, the block size of IPS^{4}o does not depend on the input size. In-place data distribution, e.g., for radix sort [9], is often done element by element. Using this for samplesort would require doing the expensive element classification twice and would also make parallelization difficult.

3 Preliminaries

(Super Scalar) Samplesort Samplesort [13] can be viewed as a generalization of quicksort which uses multiple pivots to split the input into k *buckets* of about equal size. A robust way for determining the pivots is to sort $\alpha k - 1$ randomly sampled input elements. The pivots $s_1, \ldots s_{k-1}$ are then picked equidistantly from the sorted sample. Element e goes to bucket b_i if $s_{i-1} \le e < s_i$ (with $s_0 = -\infty$ and $s_k = \infty$). The main contribution of s^3-sort [28] is to eliminate branch mispredictions for element classification. Assuming k is a power of two, the pivots are stored in an array a representing a complete binary search tree: $a_1 = s_{k/2}$, $a_2 = s_{k/4}$, $a_3 = s_{3k/4}, \ldots$ More generally, the left successor of a_i is a_{2i} and its right successor is a_{2i+1}. Thus, navigating this tree is possible by performing a conditional instruction for incrementing an array index. We adopt (and refine) this approach to element classification but change the organization of buckets in order to make the algorithm in-place.

4 In-Place Parallel Super Scalar Samplesort (IPS^{4}o)

IPS^{4}o is based on the ideas of s^3-sort. It is a recursive algorithm, where each step divides the input into k buckets, such that each element of bucket b_i is smaller than all elements of b_{i+1}. As long as problems with at least $\beta \frac{n}{t}$ elements exist, we partition those problems one after
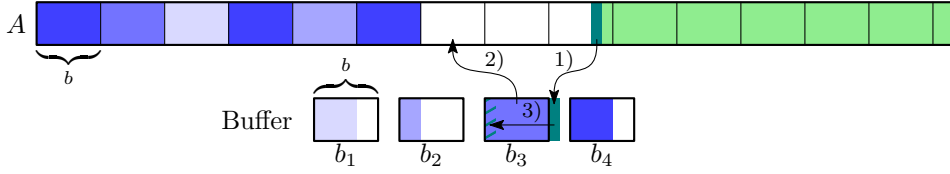
Figure 1 Local classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. Here, the next element (in dark green) has been determined to belong to bucket b_3. As that buffer block is already full, we first flush it into the array A, then write the new element into the now empty buffer.
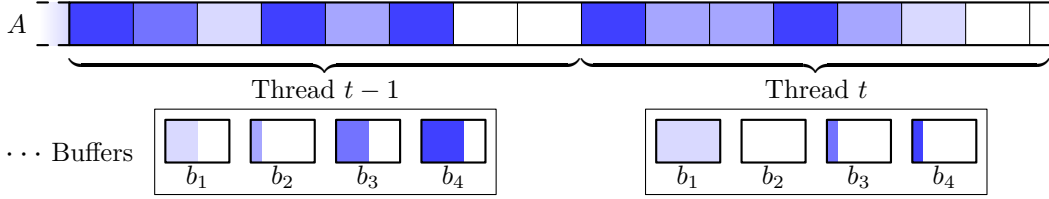
Figure 2 Input array and block buffers of the last two threads after local classification.

another with t threads in parallel. Here, β is a tuning parameter. Then we assign remaining problems in a balanced way to threads, which sort them sequentially.

The partitioning consists of four phases. **Sampling** determines the bucket boundaries. **Local classification** groups the input into blocks such that all elements in each block belong to the same bucket. **Block permutation** brings the blocks into the globally correct order. These blocks are then permuted to bring them into the correct order. Finally, we perform some **cleanup** around the bucket boundaries. The following sections will explain each of these phases in more detail.

Sampling.   The sampling phase is similar to the sampling in s^3-sort. The main difference is that we swap the sample to the front of the input array to keep the in-place property even if the oversampling factor α depends on n.

4.1   Local 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 previous phase, each element in the stripe is classified into one of the k buckets, then moved into the corresponding local buffer block. If this buffer is already full, it is first flushed 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 flushed – otherwise no full buffer could exist.

In this way, each thread creates blocks of b elements belonging to the same bucket. Figure 1 shows a typical situation during this phase. To achieve the in-place property, we do not track which bucket each flushed block belongs to. However, we do keep count of how many elements are classified into each bucket, since we need this information in the following
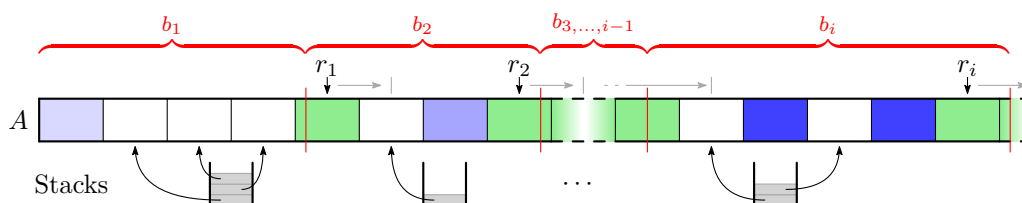
Figure 3 Invariant during block permutation. In each bucket b_i, blocks to the left of r_i are empty (white) or already correct (blue), while all blocks to the right of r_i are unprocessed (green).

phases. This information can be obtained almost for free as a side effect of maintaining the buffer blocks. Figure 2 depicts the input array after local classification. Each stripe contains a number of flushed blocks, followed by a number of empty blocks. The remaining elements are still contained in the buffer blocks.
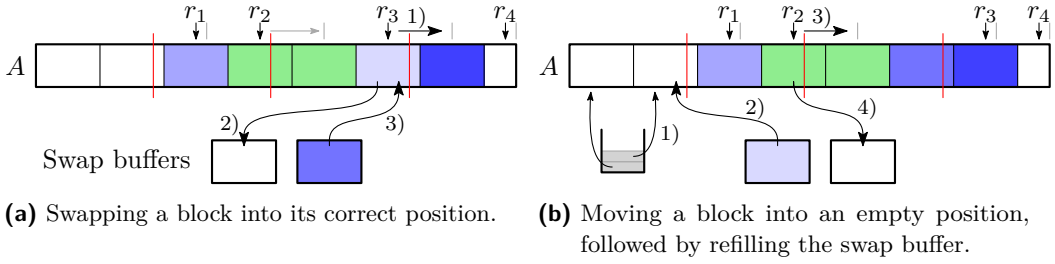
4.2 Block Permutation

In this phase, the blocks in the input array will be rearranged such that they appear in the correct order. From the previous phase we know, for each stripe, how many elements belong to each bucket. We perform a prefix sum operation to compute the exact boundaries of the buckets in the input array. In general, these will not coincide with the block boundaries. For the purposes of this phase, we will ignore this: We mark the end of each bucket b_i with an end pointer e_i, rounded up to the next block. If that would result in a position outside the input array, we instead round down to the previous block (to avoid overflows). For each b_i, a read pointer r_i is 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 r_i and e_i. To keep track of empty blocks in each bucket b_i, we store references to them on a stack S_i.

We allocate a single empty *overflow block* and add it to the stack of the last bucket which contains more than b elements – buckets with less than b elements have all their elements in the buffer, so they have no block in the input array and therefore cannot cause overflow in this phase. Doing this allows us to avoid explicit overflow checks: Instead of writing to the final block, which may extend beyond the bounds of the input array, the algorithm will automatically write to the overflow block.

During the block permutation, we maintain the following invariant, visualized in Figure 3:
- In each bucket b_i, each block to the left of r_i is correctly placed, i.e., contains only elements belonging to b_i, or is an empty block and is referenced by stack S_i.
- In each b_i, each block to the right of r_i (including the block r_i points to) is unprocessed. At the start of the block permutation, we establish this invariant by moving all empty blocks to the front of the input array, in parallel (see Appendix A for details). The read pointers r_i are then set to the first non-empty block in each bucket, or e_i if there are none.

We are now ready to start the block permutation. Each thread maintains two local swap buffers. 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 increments the read pointer r_p (atomically), reads the block it pointed to into one of its swap buffers, and adds a reference to the now empty block to stack S_p. If b_p contains no more unprocessed blocks (i.e., $r_p = e_p$), 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 this phase ends. The starting points for the threads are distributed

(a) Swapping a block into its correct position.

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

Figure 4 Block permutation examples.

across that cycle to reduce contention.

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 Figure 4:

- $r_{dest} < e_{dest}$ In this case, r_{dest} points to an unprocessed block in bucket b_{dest}. The thread increases r_{dest}, reads the unprocessed block into its empty swap buffer, and writes the other one into its place.

- $r_{dest} = e_{dest}$ In this case, no unprocessed block remains in bucket b_{dest}. The thread takes a reference to an empty block from stack S_{dest}, writes its block to that position, 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. We omitted this from the above description for the sake of clarity. In some cases, this reduces the number of block moves significantly.

We know that all blocks belonging to a bucket fit there; this follows from the fact that we round up bucket boundaries and that all elements not constituting a whole block are still in the buffers from the previous phase. However, there can still be rare cases where we hold a block belonging to b_i, but $r_i = e_i$ and S_i is empty. This happens if another thread is currently reading the last unprocessed block from b_i: It has increased r_i, but not yet added the block to S_i. In this case, we simply wait until it has done so, then continue normally.

Care has to be taken to ensure that the read pointers r_i and stacks S_i can be read and modified concurrently. We need the stacks to track the empty blocks in each partition because they may not be contiguous; other, correctly placed blocks may be interspersed (as can be seen in Figure 3). This happens, for example, when a thread reads a block from its primary bucket, then another thread (or the same) swaps in a block coming from another bucket, and then the first thread again reads a block from that bucket.

4.3 Cleanup

After the block permutation, some elements may still be in incorrect positions. This is due to the fact that we only moved blocks, which may span bucket boundaries. We call the partial block at the beginning of a bucket its *head* and the partial block at its end its *tail*.

We assign consecutive buckets evenly to threads; if $t > k$, some threads will not receive any buckets, but those that do only need to process a single bucket each. Each thread reads the head of the first bucket of the next thread 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 consist of the elements in the head of b_{i+1} (or the swap buffer, for the last bucket), the partially filled buffers from the local classification phase (of all threads), and, for the corresponding bucket, the overflow
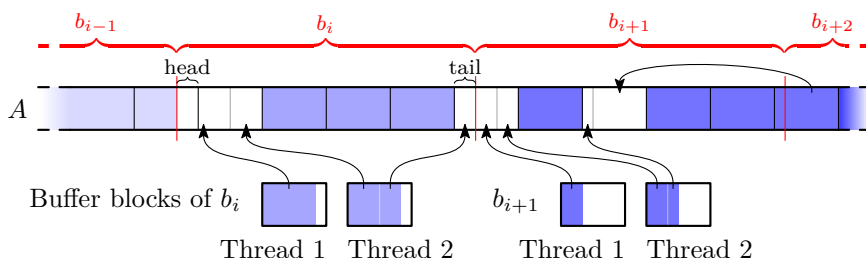
Figure 5 An example of the steps performed during cleanup.

buffer. Empty slots consist of the head of b_i and any empty blocks referenced by stack S_i. Although the concept is relatively straightforward, the implementation is somewhat involved, due to the many parts that have to be brought together. Figure 5 shows an example of the steps performed during this phase. Afterwards, all elements are back in the input array and correctly partitioned, ready for recursion.

4.4 The Case of Many Identical Keys

Having inputs with many identical keys can be a problem for samplesort, since this might move a large fractions of the keys through many levels of recursion. We turn such inputs into *easy* instances by introducing separate buckets for elements identical to pivots (keys occurring more then $\frac{n}{k}$ times are likely to become pivots). Finding out whether an element has to go into an equality bucket (and which one) can be implemented using a single additional comparison [4] and, once more, without a conditional branch. Equality buckets can be skipped during recursion and thus are not a load balancing problem.

4.5 Analysis

Algorithm IPS^{4}o inherits from s^3-sort that it has virtually no branch mispredictions (this includes the comparisons for placing elements into equality buckets discussed in subsection 4.4). More interesting is the parallel complexity. Here, the main issue is the number of accesses to main memory. We analyze this aspect in the parallel external memory (PEM) model [2], where each of the t threads has a private cache of size M and access to main memory happens in blocks of size B. In Appendix B we prove:

▶ **Theorem 1.** *Assuming $b = \Theta(tB)$ (buffer block size), $M = \Omega(ktB)$, $n_0 = \mathcal{O}(M)$ (base case size), $\alpha \in \Omega(\log t) \cap \mathcal{O}(t)$ (oversampling factor), and $n = \Omega\big(\max(k,t)t^2B\big)$, IPS^{4}o has an I/O-complexity of $O(\frac{n}{tB} \log_k \frac{n}{n_0})$ block transfers with high probability.*

Basically, Theorem 1 tells us that IPS^{4}o is asymptotically I/O efficient if certain rather steep assumptions on cache size and input size hold. Lifting those could be an interesting theoretical question and we would have to see how absence of branch mispredictions and the in-place property can be combined with previous techniques [2, 5]. However, it is likely that the constant factors involved are much larger than for our simple implementation. Thus, the constant factors will be the main issue in bringing theory and practice further together. To throw some light on this aspect, let us compare the constant factors in I/O-volume (i.e., data flow between cache and main memory) for the sequential algorithms IS^{4}o (IPS^{4}o with $t = 1$) and s^3-sort. To simplify the discussion, we assume a single level of recursion, $k = 256$ and 8-byte elements. In Appendix B we show that IS^{4}o needs about $48n$ bytes of I/O volume,

whereas s^3-sort needs (more than) $86n$ – almost twice that of IS^4o. This is surprising since on first glance, the partitioning algorithm of IS^4o writes the data twice, whereas s^3-sort does this only once. However, this is more than offset by "hidden" overheads of s^3-sort like memory management, allocation misses, and associativity misses.

Finally, we consider the memory overhead of IPS^4o. In Appendix B we show:

▶ **Theorem 2.** *IPS^{4}o requires additional space* $\mathcal{O}(kbt + \log_k n/n_0)$.

In practice, the term $\mathcal{O}(kbt)$ (mostly for the distribution buffers) will dominate. However, for a strictly in-place algorithm in the sense of algorithm theory, we need to get rid of the $\mathcal{O}(\log n)$ term which depends on the input size. We discuss this separately in subsection 4.6.

4.6   From Almost In-Place to Strictly In-Place

We now explain how the space consumption of IPS^4o can be made independent of n in a rather simple way. We can restrict ourselves to the sequential case, since only $\mathcal{O}(\log_k p)$ levels of parallel recursion are needed to arrive at subproblems that are solved sequentially. We require the partitioning operation to mark the beginning of each bucket by storing the largest element of a bucket in its first entry. By searching the next larger element, we can then find the end of the bucket. Note that this is possible in time logarithmic in the bucket size using exponential/binary search. We assume that the corresponding function *searchNextLargest* returns $n + 1$ if no larger elements exists – this happens for the last bucket. The following pseudocode uses this approach to emulate recursion in constant space for sequential IS^4o.

```
i := 1                                              −− first element of current bucket
j := n + 1                                          −− first element of next bucket
while i < n do
    if j − i < n_0 then smallSort(a, i, j − 1);   i := j        −− base case
    else partition(a, i, j − 1)                     −− partition first unsorted bucket
    j := searchNextLargest(A[i], A, i + 1, n)       −− find beginning of next bucket
```

4.7   Implementation Details

The strategy for handling identical keys described in subsection 4.4 is enabled conditionally: After the splitters have been selected from the initial sample, we check for and remove duplicates. Equality buckets are only used if there were duplicate splitters.

For buckets under a certain base case size n_0, we stop the recursion and fall back on BlockQuicksort. Additionally, we use an adaptive number of buckets on the last level of the recursion, such that the expected size of the final buckets remains reasonable. For example, instead of performing a 256-way partitioning to get 256 buckets of size 16, we might perform a 64-way partitioning to get 64 buckets of size 64. The latter is faster because there is less overhead, and 64 elements can still be sorted quickly. Furthermore, 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 another pass over the data.

IPS^4o has several parameters that can be used for tuning and adaptation. We performed our experiments using (up to) $k = 256$ buckets, an oversampling factor of $\alpha = 0.2 \log n$, an overpartitioning factor of $\beta = 1$, a block size of $b = 256$ elements, and a base case size of $n_0 = 2048$ elements. In the sequential case, we avoid the use of concurrent stacks and atomic read pointers which are needed in the parallel algorithm. All algorithms are written in C++

and compiled with version 5.2.0 of the GNU compiler collection, using the optimization flags "`-mcx16 -march=native -O3`". For parallelization, we employ OpenMP.

5 Experimental Results

We present the results of our in-place parallel sorting algorithm IPS^{4}o. We compare the results of IPS^{4}o with its in-place competitors, parallel sort from the Intel® TBB library [26] (TBB), parallel unbalanced quicksort from the GCC STL library (MCSTLubq), and parallel balanced quicksort from the GCC STL library (MCSTLbq). We also give results on the parallel non-in-place sorting algorithms, parallel samplesort from the problem based benchmark suite [29] (PBBS) and parallel multiway mergesort from the GCC STL library [30]. We also ran sequential experiments and present the results of IS^{4}o, the sequential implementation of IPS^{4}o. We compare the results of IS^{4}o with its sequential competitors, a recent implementation [16] of non-in-place Super Scalar Samplesort [28] (s^3-sort) optimized for modern hardware, BlockQuicksort [10] (BlockQ), and Dual-Pivot Quicksort [32] (DualPivot).

We ran benchmarks with six input types: Uniformly distributed (*Uniform*), exponentially distributed (*Exponential*), and almost sorted (*AlmostSorted*), proposed by Shun et. al. [29]; and *RootDup*, *TwoDup*, and *EightDup* from Edelkamp et. al. [10]. The input type RootDup sets $A[i] = i \mod \lfloor\sqrt{n}\rfloor$, TwoDup sets $A[i] = i^2 + \frac{n}{2} \mod n$, and EightDup sets $A[i] = i^8 + \frac{n}{2} \mod n$. We ran benchmarks with 64-bit floating point elements. For $n < 2^{30}$, we perform each measurement 15 times and for $n \geq 2^{30}$, we perform each measurement twice. Unless stated otherwise, we report the average over all runs.

We ran our experiments on a machine with two sockets (*M2*) and a machine with four sockets (*M4*). Machine M2 has two Intel Xeon E5-2683 v4 16-core processors and machine M4 has four Intel Xeon E5-4640 8-core processors. Both machines are equipped with 512 GiB of memory. We ran sequential algorithms on one core and parallel algorithms on one socket (*M2-1Socket* and *M4-1Socket* respectively) and on all available sockets (*M2-2Socket* and *M4-4Socket* respectively). We use the `taskset` tool to set the CPU affinity accordingly. We tested all parallel algorithms on input type Uniform with and without hyper-threading on all machine instances. Each algorithm took advantage of hyper-threading or took about the same time to sort the input. Thus, we give results of all algorithms with hyper-threading.

Sequential Algorithms. Figure 7 shows the running time of sequential algorithms on input type Uniform executed on machine instance M2. We see that IS^{4}o is faster than its closest competitor, BlockQ, by a factor of 1.08 to 1.19 for inputs larger than the base case size ($n \geq 2^{13}$). For $n \leq 2^{12}$, IS^{4}o performs similarly to BlockQ, as those input sizes are sorted with the base case sorter BlockQ.

As expected, the running times for inputs with a moderate number of unique keys (TwoDup and Exponential) are similar to the running times for Uniform (compare Figure 9 (a-b) in Appendix C). But when we sort input types with few unique keys (RootDup), IS^{4}o gets even faster by a factor of about two. The running times of DualPivot also decrease for this input type and become comparable for large inputs ($n \geq 2^{28}$). However, IS^{4}o now outperforms its closest competitor, BlockQ, by a factor of up to 1.53, whereas IS^{4}o outperformed BlockQ on input type Uniform just by a factor of up to 1.19. (see Figure 9 (c) in Appendix C). Only input type AlmostSorted is hard for IS^{4}o; for example, DualPivot outperforms IS^{4}o by a factor of 1.99 for $n = 2^{30}$ (see also Figure 9 (d) in Appendix C).

In Appendix C, we also give running times on machine M4 (see Figure 10 for Uniform and see Figure 9 (e-h) for Exponential, TwoDup, RootDup, and AlmostSorted). We see

that IS^{4}o runs faster on machine M4, e.g., IS^{4}o now outperforms BlockQ on Exponential by a factor of 1.23 ($n = 2^{30}$), compared to a factor of 1.12 on M2. Generally, IS^{4}o performs similarly to the fastest competitor for large inputs (M2) or is faster than its competitors (M4) on most of the input types. Thus, we decided that our parallel algorithm IPS^{4}o uses IS^{4}o as its sequential sorting algorithm.

Parallel Algorithms.   Figure 6 (a-d) presents experiments of parallel algorithms on Uniform on different machine instances. We see that IPS^{4}o outperforms its competitors on M2-1Socket, M2-2Socket, and M4-1Socket by a factor of 1.75 to 2.31 for $n = 2^{32}$. For $n = 2^{25}$, IPS^{4}o still outperforms its competitors by a factor of 1.47 to 2.21. But in all cases, IPS^{4}o is faster than its closest in-place competitor, MCSTLbq, by a factor of at least 1.75. On M4-4Socket, IPS^{4}o is a factor of 1.26 faster than its closest non-in-place competitor PBBS for $n = 2^{32}$. However, IPS^{4}o outperforms its closest in-place competitor on that machine instance by a factor of at least 1.96 for any reasonable input size. When we sort smaller inputs ($n < 2^{24}$), the non-in-place algorithms become competitive but the in-place competitors become disproportionately slow.

In Figure 6 (e-g), we present running times of parallel algorithms on input types with duplicates (RootDup, TwoDup, and Exponential) on machine instance M2-2Socket. We do not give running times on EightDup as the running times are similar to the running times on Exponential. For a moderate number of unique keys (TwoDup and Exponential), IPS^{4}o still outperforms its in-place competitors by a factor of at least 2.25 for any reasonable input size and its non-in-place competitors by a factor of at least 1.54 for $n \geq 2^{25}$. We also see that the non-in-place algorithms are almost as fast as IPS^{4}o if we sort inputs which contain few unique keys (RootDup). However, IPS^{4}o still outperforms its non-in-place competitors by a factor of at least 4.28 on this input for $n \geq 2^{24}$. Figure 6 (h) shows experiments of parallel algorithms on AlmostSorted on machine instance M2-2Socket. On AlmostSorted, the fastest non-in-place algorithm, TBB, performs similar to IPS^{4}o for any reasonable input size. The non-in-place algorithms MCSTLbq and MCSTLubq become comparable for $n \geq 2^{27}$.

In Appendix C, we give running times on various input types on machine instances M2-1Socket and M4-1Socket (see Figure 13) as well as machine instance M4-4Socket (see Figure 12). We see that the running time ratios of our IPS^{4}o to its competitors on the single socket machines are similar to the ratios on machine instance M2-2Socket. Note that on the machine instance M4-4Socket, the running times of the non-in-place competitors catch up with the running times of IPS^{4}o, but IPS^{4}o is faster than any in-place competitor on any input instance by a factor of 2.13 to 3.70 for $n = 2^{32}$.
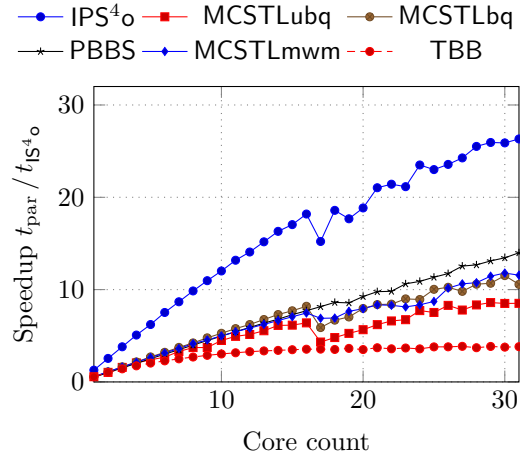
Figure 8 depicts the speedup of parallel algorithms with different numbers of cores relative to our sequential implementation IS^{4}o on machine M2, sorting 2^{30} elements of input type Uniform (see also Figure 11 in Appendix C for measurements on machine M4). We see that IPS^{4}o outperforms its competitors on any number of cores. We also see that IPS^{4}o on 32 cores outperforms its sequential implementation by a factor of 27.54, whereas its fastest competitor, the not-in-place PBBS algorithm, outperforms IS^{4}o just by a factor of 14.44. The in-place parallel quicksort implementations, MCSTLubq and MCSTLbq, scale similar to PBBS up to 16 cores but begins lagging behind PBBS for larger number of cores. Table 1 in Appendix C shows speedups of IPS^{4}o (IS^{4}o) relative to the fastest parallel (sequential) in-place and non-in-place algorithms on different input types executed on different machine instances for $n = 2^{32}$ ($n = 2^{30}$).
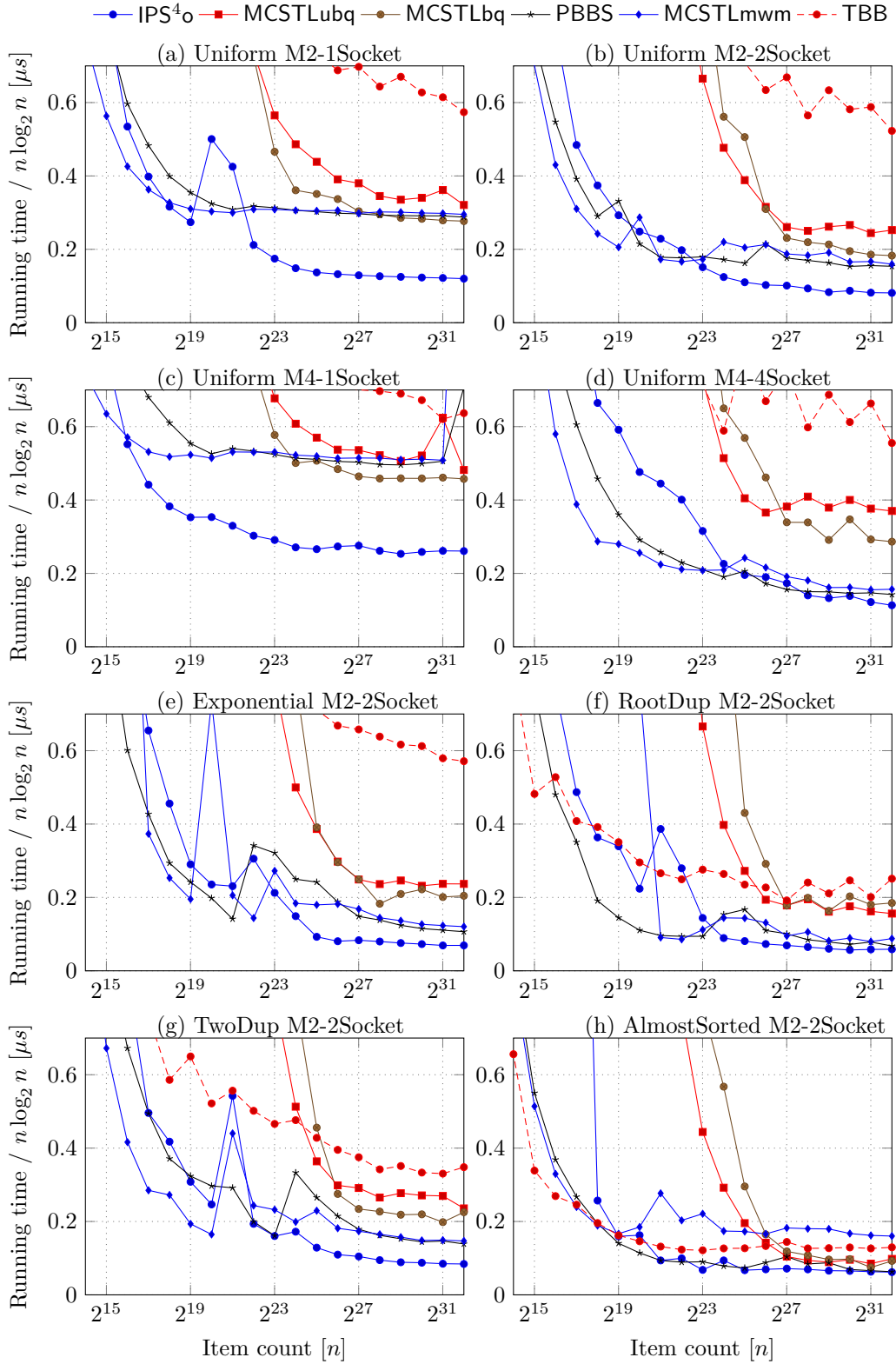
Figure 6 Running times of parallel algorithms on different input types executed on different machine instances.
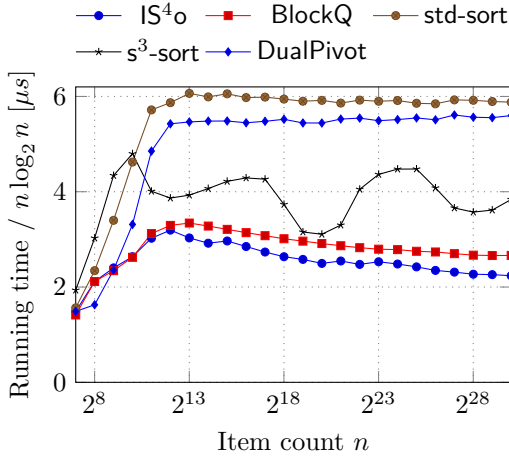
Figure 7 Running times of sequential algorithms on input type Uniform executed on machine instance M2.

Figure 8 Speedup of parallel algorithms with different number of cores relative to our sequential implementation IS^{4}o on M2, sorting 2^{30} elements of input type Uniform.

6 Conclusion and Future Work

In-place super scalar samplesort (IPS^{4}o) is the fastest comparison-based multi-core parallel sorting algorithm for large inputs. Compared to previous in-place algorithms, it is about twice as fast because it is both cache-efficient and avoids branch mispredictions. IPS^{4}o even outperforms non-in-place algorithms because it saves on overhead for memory allocation, associativity misses and write allocate misses. The algorithm can also be used for data distribution and local sorting in distributed memory parallel algorithms (e.g., [3]).

Several improvements of IPS^{4}o can be considered. Besides careful adaptation of parameters like k, b, α, and the choice of base case algorithms, one would like to avoid contention on the bucket pointer in the block permutation phase when t is large. At the price of easy portability further improvements are possible. Perhaps the most important measure would be to make IPS^{4}o aware of non-uniform memory access costs (NUMA) depending on the memory module holding a particular piece of data. This can be done by preferably assigning pieces of the input array to "close-by" cores both for local classification and when switching to sequential sorting. In situations with little NUMA effects, we could ensure that our data blocks correspond to pages of the virtual memory. Then, one can replace block permutation with relabelling the virtual memory addresses of the corresponding pages.

Coming back to the original motivation for an alternative to quicksort in standard libraries, we see IPS^{4}o as an interesting candidate to be used for large inputs. A remaining issue is the complexity of the code – IPS^{4}o is considerably more complicated than quicksort. When code size matters (e.g., as indicated by a compiler flag like -Os), quicksort should still be used. Formal verification of the correctness of the implementation might also help to increase acceptance. Note that formal verification even uncovered a long-standing bug in the quicksort implementation of the Java standard library once [1].

────── **References** ──────

1 Wolfgang Ahrendt, Bernhard Beckert, et al., editors. *Deductive Software Verification – The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.

2 Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *20th Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 197–206. ACM, 2008.

3 Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *27th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, 2015. `doi:10.1145/2755573.2755595`.

4 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *European Symposium on Algorithms*, pages 169–180. Springer, 2013.

5 Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199. ACM, 2010.

6 Guy E Blelloch, Charles E Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM, 1991.

7 G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.

8 Jing-Chao Chen. A simple algorithm for in-place merging. *Information Processing Letters*, 98(1):34 – 40, 2006. URL: `http://www.sciencedirect.com/science/article/pii/S002001900500339X`, `doi:http://dx.doi.org/10.1016/j.ipl.2005.11.018`.

9 Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. Paradis: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment*, 8(12):1518–1529, 2015.

10 Stefan Edelkamp and Armin Weiss. Blockquicksort: Avoiding branch mispredictions in quicksort. In *24th European Symposium on Algorithms (ESA)*, volume 57 of *LIPIcs*, 2016.

11 Gianni Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 291–299, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=982792.982833`.

12 Gianni Franceschini and Viliam Geffert. An in-place sorting with o(nlog n) comparisons and o(n) moves. *J. ACM*, 52(4):515–537, July 2005. URL: `http://doi.acm.org/10.1145/1082036.1082037`, `doi:10.1145/1082036.1082037`.

13 W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970. URL: `http://doi.acm.org/10.1145/321592.321600`, `doi:10.1145/321592.321600`.

14 Viliam Geffert and Jozef Gajdoš. Multiway in-place merging. In Mirosław Kutyłowski, Witold Charatonik, and Maciej Gębala, editors, *17th Symposium on Fundamentals of Computation Theory (FCT)*, volume 5699 of *LNCS*, pages 133–144. Springer, 2009. URL: `http://dx.doi.org/10.1007/978-3-642-03409-1_13`, `doi:10.1007/978-3-642-03409-1_13`.

15 Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

16 Lorenz Hübschle-Schneider. Super scalar sample sort. `https://github.com/lorenzhs/ssssort`, retrieved September 15, 2016.

17 Tomasz Jurkiewicz and Kurt Mehlhorn. On a model of virtual address translation. *Journal of Experimental Algorithmics (JEA)*, 19, 2015.

18 K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *14th European Symposium on Algorithms (ESA)*, volume 4168 of *LNCS*, pages 780–791, 2006.

19   Kanela Kaligosi and Peter Sanders. How branch mispredictions affect quicksort. In *European Symposium on Algorithms*, pages 780–791. Springer, 2006.

20   Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nord. J. Comput.*, 3(1):27–40, 1996.

21   Marek Kokot, Sebastian Deorowicz, and Maciej Dlugosz. Even faster sorting of (not only) integers. *arXiv preprint arXiv:1703.00687*, 2017.

22   Shrinu Kushagra, Alejandro López-Ortiz, J. Ian Munro, and Aurick Qiao. Multi-pivot quicksort: Theory and experiments. In *Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 47–60, Philadelphia, PA, USA, 2014. AMS. URL: `http://dl.acm.org/citation.cfm?id=2790174.2790180`.

23   K. Mehlhorn and P. Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.

24   David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

25   N. Rahman. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter Algorithms for Hardware Caches and TLB, pages 171–192. Springer, 2003.

26   James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

27   Peter Sanders and Jan Wassenberg. Engineering a multi-core radix sort. In *17th Euro-Par Conference*, volume 6853 of *LNCS*, pages 160–169. Springer, 2011.

28   Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *12th European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 784–796. Springer, 2004.

29   Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012.

30   J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-74466-5_72`.

31   P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *PDP*, pages 372–381. IEEE Computer Society, 2003. URL: `http://doi.ieeecomputersociety.org/10.1109/EMPDP.2003.1183613`.

32   Vladimir Yaroslavskiy. Dual-pivot quicksort. *Research Disclosure*, 2009.

A   Details of the Algorithm

Empty block movement   To rearrange the empty blocks at the start of the block permutation phase, we first compute how many exist by subtracting the sum of the number of blocks each thread flushed from the total number of blocks. Then, each thread computes how many of the empty blocks in its stripe it has to move and how many are already correctly placed (because they will still be empty after all empty blocks have been moved). We also compute two prefix sums, one over the number of blocks flushed by each thread and one over the number of empty blocks each thread has to move. Using this information, each thread computes the exact positions at which it should place the empty blocks from its stripe, then performs the moves.

B   Details of the Analysis

Proof of Theorem 1. It can be shown using Chernoff bounds that oversampling ratio $\alpha = \Omega(\log kt)$ is sufficient to produce (non-equality) buckets of size $\mathcal{O}(N/k)$ with high probability for subproblems of size N. Hence, $\mathcal{O}(\log_k n/n_0)$ levels of recursion suffice with high probability. On the other hand, for $\alpha = \mathcal{O}(t)$, even sequentially processing the sample does not become a bottleneck.

During a block distribution phase, each thread reads $\mathcal{O}(n/tb)$ logical data blocks, writes them to the buffers in its private cache and moves them back to main memory eventually.

The same asymptotic cost occurs for moving blocks during block permutation. Each thread performs $\mathcal{O}(n/tb)$ successful acquisitions of the next block in a bucket. Charging $\mathcal{O}(t)$ I/Os for this accounts for possible contention with other threads. Overall, we get cost $\mathcal{O}(n/b) = \mathcal{O}(n/tB)$. Similarly, there are k unsuccessful acquisitions before termination is determined, for which we charge an overall cost of $\mathcal{O}(kt)$ I/Os. Since we assume $n = \Omega(kt^2B)$, we have $k = \mathcal{O}(n/t^2B)$ and hence $\mathcal{O}(kt) = \mathcal{O}(n/tB)$.

In the cleanup phase, we consider a case distinction with respect to k and t. If $k \leq t$, then each thread processes at most one bucket and it has to move elements from $t+2$ distribution buffers and bucket boundaries. This amounts to a cost of $\mathcal{O}(tb/B) = \mathcal{O}(t^2)$ I/Os. Since $n = \Omega(t^3B)$, we get $t^2 = \mathcal{O}(n/tB)$. If $k > t$, then each thread processes $\mathcal{O}(k/t)$ buckets with a total cost of $\mathcal{O}(k/t \cdot t^2) = \mathcal{O}(kt)$. Since $n = \Omega(kt^2B)$, we have $kt = \mathcal{O}(n/tB)$.   ◀

Comparing the I/O volume of IS^{4}o and s^3-sort.   Both algorithms read and write the data once for the base case – $16n$ bytes (of I/O volume). IS^{4}o reads and writes all data both during data distribution and block permutation phase – $32n$ bytes or $48n$ bytes overall. s^3-sort reads the element twice but writes them only once in its distribution algorithm – $24n$ bytes. This sounds like a slight advantage. However, now we come to overheads unique to s^3-sort. First, the algorithm reads and writes a sequence of oracle that indicate the bucket for each element – $2n$ bytes. s^3-sort has to copy the sorted result data back to the input array – $16n$ bytes. It also has to allocate the temporary arrays. For security reasons, that memory is zeroed by the operating system – $9n$ bytes.[3] When writing to the temporary arrays or during copying back, there are so called allocate misses that happen when an element is written to a cache block that is currently not in memory – that block is read to the cache because the CPU does not know that none of the data in that block will ever be read. This amounts to an I/O volume

---

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

of up to $17n$ bytes. Furthermore, s^3-sort may suffer more associativity misses than IS^{4}o – the relative positions of the buckets in the temporary array are not coordinated while IS^{4}o essentially sweeps a window of size $\approx bk$ through the memory during the distribution phase. For an average case analysis refer to [23]. Even ignoring the latter overhead we get a total I/O volume of $86n$ byte – more than twice as much as IS^{4}o. Many of these overheads can be reduced using measures that are non-portable (or hard to make portable). In particular, non-temporal writes eliminate the allocation misses and also help to eliminate the oracles and the associativity misses. Also one could use a base case sorter that does the copying back as as 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 IS^{4}o may require slightly more I/O volume than s^3-sort or significantly less.

Proof of Theorem 2. The main space overhead is for k buffer blocks of size b for each of t threads. This bound also covers smaller amounts of memory for the search tree ($\mathcal{O}(k)$), swap buffers and overflow buffers ($\mathcal{O}(bt)$), read pointers ($\mathcal{O}(kB$ if we avoid false sharing), end pointers, and bucket boundary pointers. The stacks of empty blocks overall have at most $kt + 1$ entries. Since they are implemented as linked lists, we get $\mathcal{O}(kt)$ space for them. All of these data structures can be used for all levels of recursion. The term $\mathcal{O}(\log_k n/n_0)$ stems from the space for the recursion stack itself.   ◄

C   More Measurements

Machine	Algo	Competitor	Uniform	Expo	AlmostS	RootDup	TwoDup
M2	IS^{4}o	both	1.19	1.12	0.50	0.99	1.17
M4	IS^{4}o	both	1.23	1.23	0.56	1.22	1.25
M2-1Socket	IPS^{4}o	in-place	2.31	2.04	0.93	2.33	2.16
		non-in-place	2.40	2.01	1.11	1.66	2.11
M4-1Socket	IPS^{4}o	in-place	1.75	1.61	0.95	1.81	1.78
		non-in-place	2.71	2.94	2.10	3.31	2.48
M2-2Socket	IPS^{4}o	in-place	2.25	2.95	1.49	2.66	2.68
		non-in-place	1.89	1.54	1.02	1.15	1.65
M4-4Socket	IPS^{4}o	in-place	2.53	3.16	2.13	3.03	2.70
		non-in-place	1.26	1.23	1.08	1.09	1.12

Table 1 The first two rows show the speedups of IS^{4}o relative to the fastest sequential in-place and non-in-place competitor on different input types executed on machine M2 and M4 for $n = 2^{30}$. The last rows show the speedups of IPS^{4}o relative to the fastest parallel in-place and non-in-place competitor on different input types executed on different machine instances for $n = 2^{32}$.
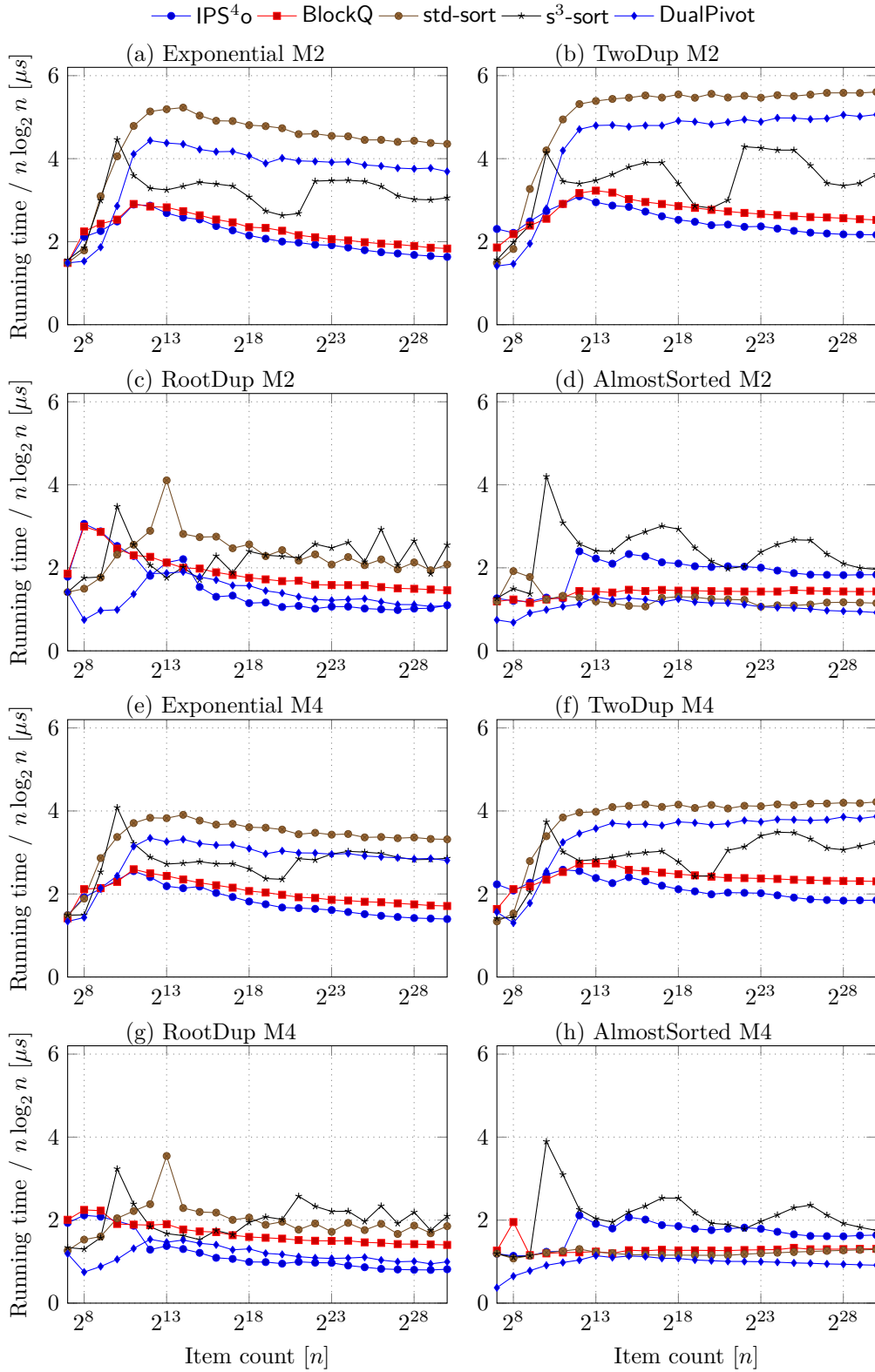
**Figure 9** Running times of sequential algorithms on different input types executed on machine instances M2 and M4.
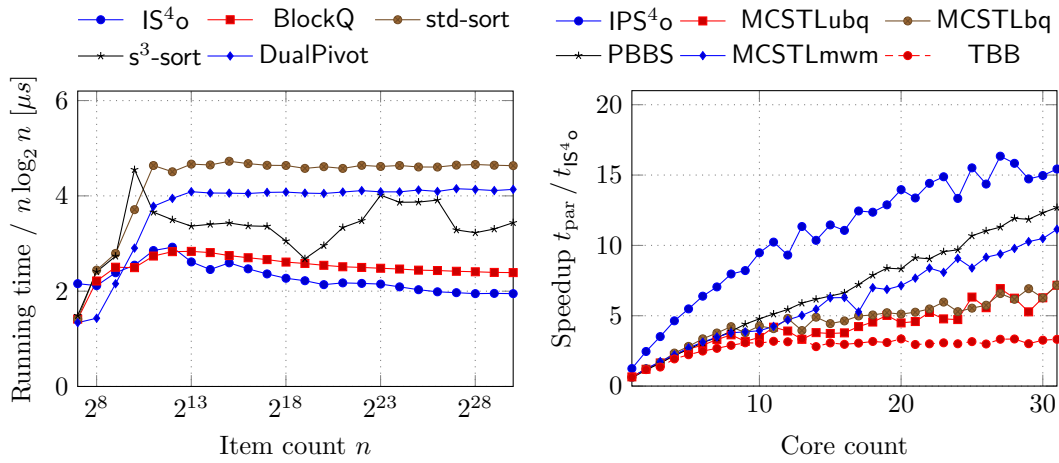
Figure 10 Running times of sequential algorithms on input type Uniform executed on machine instance M4.

Figure 11 Speedup of parallel algorithms with different number of cores relative to our sequential implementation $\mathsf{IS}^4\mathsf{o}$ on M4 sorting 2^{30} elements of input type Uniform.
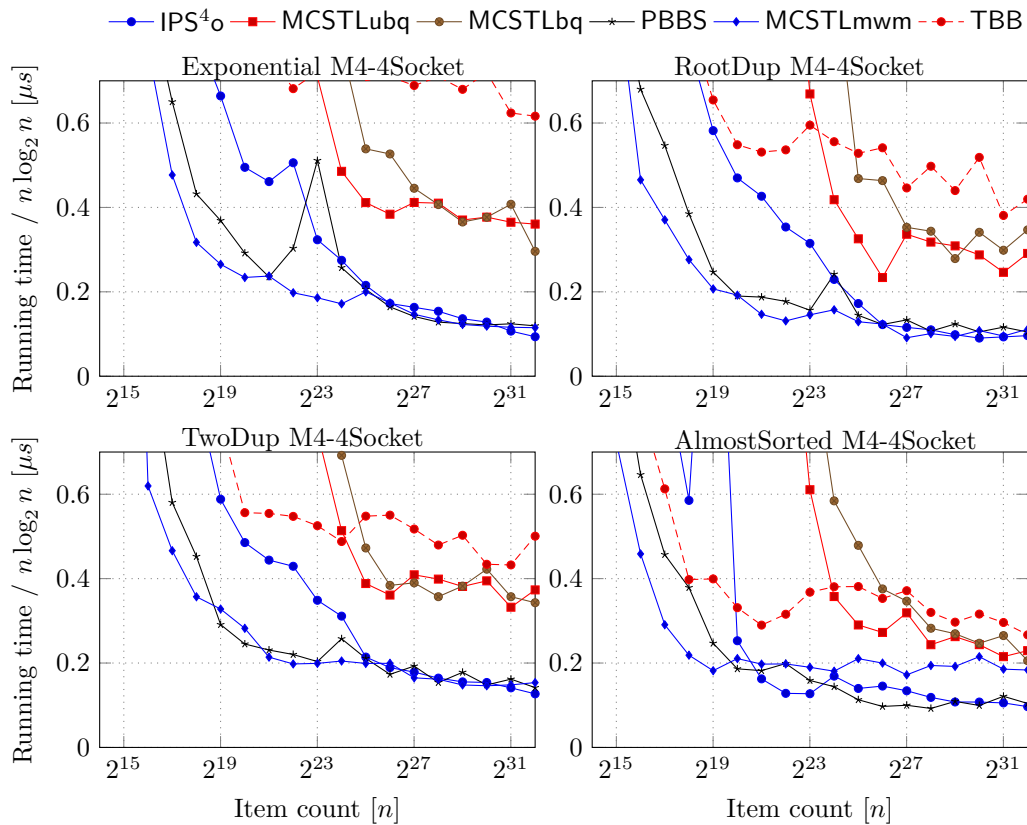
Figure 12 Running times of parallel algorithms on different input types executed on machine instance M4-4Socket.
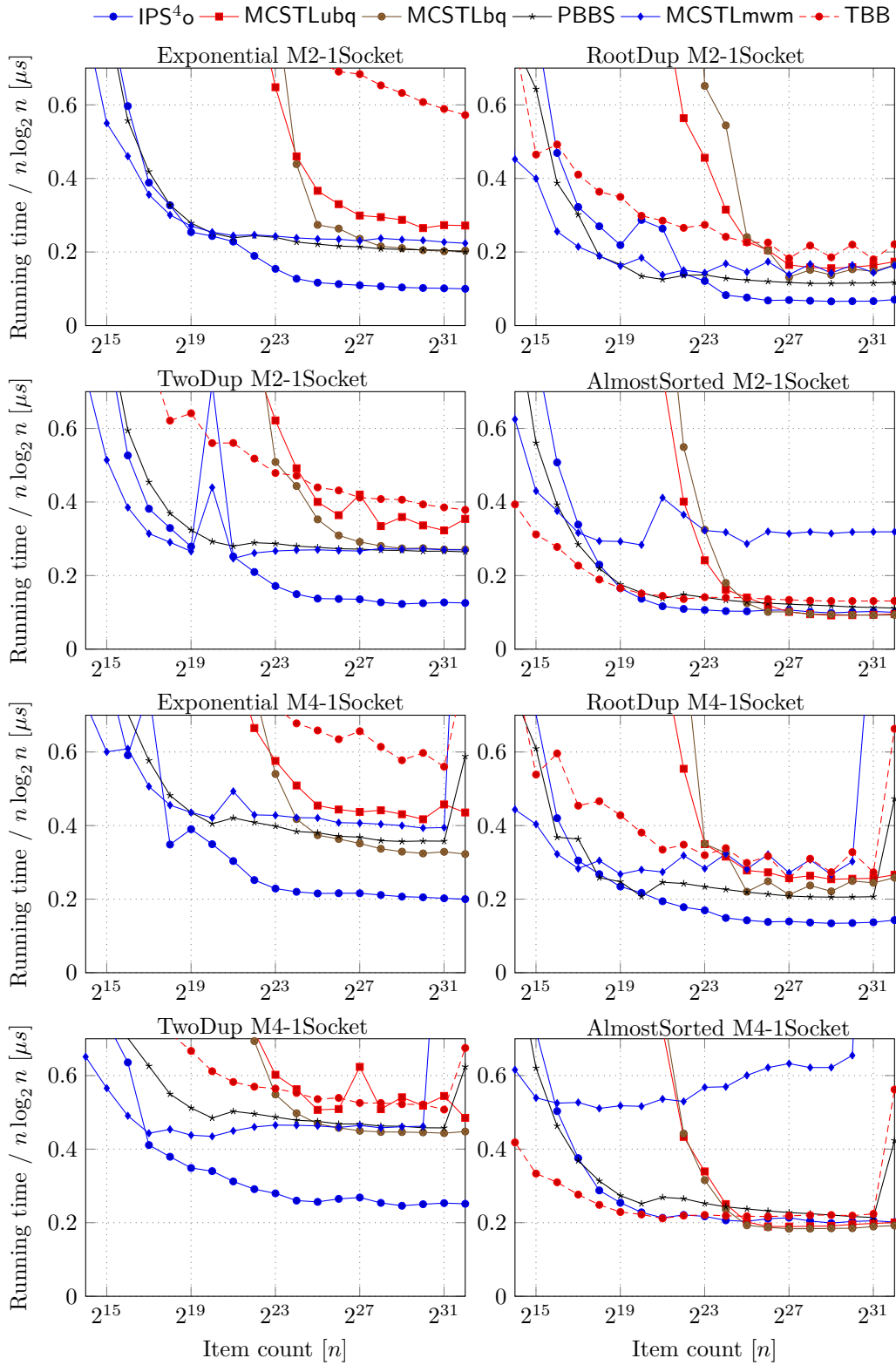
Figure 13 Running times of parallel algorithms on different input types executed on machine instances M2-1Socket and M4-1Socket.