# Simple and Fast BlockQuicksort using Lomuto's Partitioning Scheme

Martin Aumüller Nikolaj Hass

October 30, 2018



**Abstract**

This paper presents simple variants of the BlockQuicksort algorithm described by Edelkamp and Weiss (ESA 2016). The simplification is achieved by using Lomuto's partitioning scheme instead of Hoare's crossing pointer technique to partition the input. To achieve a robust sorting algorithm that works well on many different input types, the paper introduces a novel two-pivot variant of Lomuto's partitioning scheme. A surprisingly simple twist to the generic two-pivot quicksort approach makes the algorithm robust. The paper provides an analysis of the theoretical properties of the proposed algorithms and compares them to their competitors. The analysis shows that Lomuto-based approaches incur a higher average sorting cost than the Hoare-based approach of BlockQuicksort. Moreover, the analysis is particularly useful to reason about pivot choices that suit the two-pivot approach. An extensive experimental study shows that, despite their worse theoretical behavior, the simpler variants perform as well as the original version of BlockQuicksort.

## 1 Introduction

Sorting a sequence of elements in an efficient way is one of Computer Science's fundamental problems. It is one of the most important core routines in many algorithms, and does not seem to lose relevance as we get more and more data to work on every day.

While it has been more than 55 years since the quicksort algorithm was described by Hoare in [11], variants of it are still relevant today. This especially shows when looking at sorting algorithms implemented in the standard libraries of modern programming languages, such as C++, which uses a worst-case aware quicksort variant called introsort [15], Oracle's Java, which uses a two-pivot quicksort algorithm [23], and Microsoft's C#, which uses a traditional median-of-three quicksort approach.

In the pursuit of improving the performance of quicksort-based implementations, a long line of papers [16, 23, 13, 3, 21] have looked at the benefits of introducing more than one pivot in the algorithm. Here, [13, 3, 21] showed that the memory access behavior improves with a small number of pivots — an effect that cannot be achieved by other more traditional means such as choosing a pivot from a sample of elements [18]. However, better cache behavior leaves the problem of branch mispredictions unsolved, which is one of the largest bottleneck's in today's processor architectures.

As shown by Brodal and Moruz in [6], mispredicting conditional branches, i.e., mispredicting the outcome of the comparison of two elements of the input, are a necessity in the regime of comparison-efficient sorting algorithms. This means other tools are needed to reduce the performance penalty that these mispredictions incur, which can be as large as 15 cycles on a modern Intel i7 CPU [10], and even larger on other architectures [12]. As we will describe in Section 2, under certain circumstances modern processors can avoid branches using so-called conditional move-operations, among others, which were used in a number of implementations of quicksort variants [17, 8, 7, 4]. More detailed information about these methods is deferred to the related work part of the introduction.

In the branch-free regime of [17, 7, 4], a lot of complexity has been introduced to the algorithm. In fact, Axtmann et al. [4] describe code complexity as the main drawback of their approach, stat-

ing "*formal verification of the correctness of the implementation might help to increase trust in the remaining cases.*" [4, p. 12]. The BlockQuicksort approach of Edelkamp and Weiss [7] has a very elevated code complexity as well, using about 300 lines of code for the partitioning procedure.

The main contribution of this paper are simple, fast, and robust variants of branch-free quicksort-based algorithms. Thus, introducing more complexity is not necessarily required to improve performance. In more detail, the paper presents simpler variants of the BlockQuicksort algorithm described in [7]. The difference to BlockQuicksort lies in utilizing a partitioning scheme that Bentley in [5] attributed to Nico Lomuto, rather than using the traditional "crossing pointer" partitioning scheme by Hoare as in [7]. The description of the algorithms is provided in Section 3. In there, we describe a one-pivot and a two-pivot approach. Introducing an additional pivot allows us to make the algorithm robust and to avoid some of the common performance issues of the Lomuto partitioning scheme, such as the presence of many equal elements [7].

Theoretical properties of the proposed algorithms are discussed in Section 4. The result of this analysis allows us, among others, to reason about good pivot choices for the two-pivot variant. Comparing the Lomuto-based algorithms to the Hoare-based approach used in standard BlockQuicksort, we notice a slightly worse comparison count and a slightly worse memory behavior with regard to the "scanned elements/memory accesses" cost measure discussed in [13, 3, 21]. This makes the results of the experiments in Section 5 quite surprising, which show that the proposed variants can compete with BlockQuicksort. In the same section, we speculate about the reasons for this based on actual measurements from the CPU.

### 1.1 Related Work

**Tuning Quicksort Algorithms** Quicksort implementations (see Section 2 for a description of the quicksort algorithm) usually use insertionsort to sort subproblems that are below a certain size threshold [18]. Moreover, the pivot is usually chosen from a small sample of elements. In practice, it is usually chosen as the median in a sample of three elements, while theory suggests to choose it as the median of $\Theta(\sqrt{n})$ elements to minimize comparisons [14].

**Super-Scalar Samplesort** Sanders and Winkel [17] provided an engineered implementation of samplesort, a variant of quicksort in which many pivots are used to distribute the input to buckets. They used conditional moves to traverse the heap-like tree used to find the bucket of an element. The algorithm uses two phases, classification and rearranging elements, and needs extra space in the size of the input.

**Tuned Quicksort** Elmarsy et al. [8] described an engineered quicksort variant that uses a branch-free variant of Lomuto's partitioning scheme using conditional moves.

**BlockQuicksort** BlockQuicksort as introduced by Edelkamp and Weiss in [7] generalizes Hoare's partitioning scheme using blocks to handle misplaced elements. Filling these two blocks, one block for each of the two pointers used in Hoare's partitioning scheme, with references to misplaced elements is done branch-free using conditional moves. When these blocks are near-full, elements are moved into a correct position with regard to the pivot choice. When the two pointers are close at the end of the partitioning phase, the code has to handle many edges cases to produce a correct partition. To this end, BlockQuicksort uses a "clean-up phase" when partitioning is done.

**Multi-Pivot Quicksort** [9, 16, 23, 13, 1, 2, 3, 21] studied benefits of using a small number of pivots in quicksort. This line of research showed that this approach improves the memory access pattern drastically [3, 21], an improvement that cannot be achieved by other means such as choosing a good pivot by sampling input elements. In short, multi-pivot quicksort allows to make fewer accesses to array cells to sort the input, an effect that translates into better running times. The most popular multi-pivot algorithm is the two-pivot YBB algorithm [23] used in Oracle's Java since version 7.

**IPS$^4$o** [4] provides an in-place samplesort variant of [17] by combining it with the idea of using blocks from BlockQuicksort. Instead of two blocks, they use a large number of blocks, one for each

**Algorithm 1** One-Pivot-Quicksort

**procedure** OnePivotQuicksort($A[1..n]$)

```
1: if n > 1 then
2:     choosePivot(A[1..n]) ▷ Pivot resides in A[n]
3:     i ← partition(A[1..n])
4:     OnePivotQuicksort(A[1..i − 1])
5:     OnePivotQuicksort(A[i + 1..n])
```

bucket. These blocks are filled in a branch-free way using the classification strategy from [17]. All elements are moved in blocks and a final rearrangement step is needed to swap blocks to correct positions. Again, a lot of care is needed in the cleanup phase. IPS$^4$o lends itself to parallelization, as demonstrated by the experiments in [4].

**1.2 Our Contributions** This paper contains the following contributions:

- We present variants of BlockQuicksort using Lomuto's partitioning scheme (Algorithm 4, Algorithm 5) and evaluate them experimentally. Comparing the implementations, the proposed variants save lines of code by a factor 5 to 8 times compared to [7] and are easier to understand.
- We show how using a simple twist in two-pivot quicksort makes the algorithm more robust with respect to different input types.
- We analyze theoretical properties of our variants, which among others allows us to reason about good pivot choices.

We believe the algorithms to be so simple to be described in a textbook on Algorithm Engineering.

## 2 Preliminaries

**Outline of a One-Pivot Quicksort Algorithm** We assume that the input to be sorted resides in an array $A[1..n]$. Classical quicksort (Algorithm 1) sorts the array $A$ as follows. If $n \leq 1$, do nothing. Otherwise, choose an element $p$ from $A$ as pivot. Next, *partition* the input such that $p$ resides in $A[i]$, $A[1..i-1]$ contains elements smaller than $p$, and $A[i+1..n]$ contains elements at least as large as $p$. Then, sort $A[1..i-1]$ and $A[i+1..n]$ recursively.

**Algorithm 2** Lomuto Partitioning Scheme

**procedure** LomutoPartition($A[1..n]$)

```
1: p ← A[n]
2: i ← 1
3: for j ← 1; j < n; j ← j + 1 do
4:     if A[j] < p then
5:         Swap A[i] and A[j]
6:         i ← i + 1
7: Swap A[i] and A[n]
8: return i
```

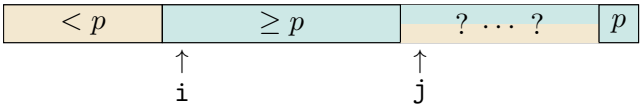


Figure 1: Lomuto invariant: $A[1..\mathtt{i}-1]$ consists of elements smaller than $p$, $A[\mathtt{i}..\mathtt{j}-1]$ consists of elements at least as large as $p$; $A[\mathtt{j}..n-1]$ has not been looked at, which is depicted by filling this part of the array with both colors.

**Lomuto's Partitioning Scheme** Traditionally, one uses the crossing pointer technique described by Hoare [11] to partition the input. A simpler partitioning method communicated to Bentley by Lomuto[1] is given as Algorithm 2.

Lomuto's partitioning scheme uses two additional variables i and j and maintains the invariant displayed in Figure 1. The variable j is incremented from 1 to $n-1$ using a **for** loop. When $A[\mathtt{j}]$ is inspected, it is compared to the pivot p. If it is smaller than the pivot, $A[\mathtt{i}]$ and $A[\mathtt{j}]$ are swapped, and i is incremented.

As pointed out in [5, 20], Lomuto's partitioning scheme is "not as fast as Hoare's version". Theoretically speaking, this is because on average, where the average is taken over all $n!$ permutations of the set $\{1, \ldots, n\}$, it makes three times more swaps than Hoare's partitioning scheme [19] and "scans" 50% more elements [20].

**Outline of a Two-Pivot Quicksort Algorithm** Two-pivot quicksort (Algorithm 3) sorts the array $A[1..n]$ as follows. If $n \leq 1$, do nothing.

[1] To quote [5, Page 110]: *A reader of a preliminary draft [of [5]] complained that the standard two-index method is in fact simpler than Lomuto's, and sketched some code to make his point: I stopped looking after I found two bugs.*

**Algorithm 3** Two-Pivot-Quicksort

**procedure** TwoPivotQuicksort($A[1..n]$)

1: **if** $n > 1$ **then**
2: choosePivot($A[1..n]$) ▷ Pivots $A[1] \leq A[n]$
3: $(\mathtt{i}, \mathtt{j}) \leftarrow$ partition($A[1..n]$)
4: TwoPivotQuicksort($A[1..\mathtt{i}-1]$)
5: TwoPivotQuicksort($A[\mathtt{i}+1..\mathtt{j}-1]$) ▷ Sec. 3.3
6: TwoPivotQuicksort($A[\mathtt{j}+1..n]$)

Otherwise, choose two elements $p, q$ with $p \leq q$ from $A$ as pivots. Next, partition the input such that $p$ resides in position $i$ and $q$ resides in position $j$, $A[1..i-1]$ contains elements smaller than $p$, $A[i+1..j-1]$ contains elements $x$ with $p \leq x \leq q$ and $A[j+1..n]$ contains elements larger than $q$. Then, sort $A[1..i-1]$, $A[i+1..j-1]$, and $A[j+1..n]$ recursively.

Many different partitioning methods for two-pivot quicksort exist, see, for example [23, 2].

**Avoiding Branch Mispredictions** Today, most CPUs use instruction-level parallelism. As described in [10, Section 3.13], the biggest problems in exploiting parallelism come from mispredicted branches or cache misses. Branch mispredictions may occur when the code contains conditional jumps, such as *if* statements. When reaching a branch, the CPU decides which branch it follows based on a branch predictor and loads the instructions following the branch into its pipeline. When the direction of the branch is mispredicted, the pipeline has to be flushed and the other direction has to be executed.

In comparison-based sorting algorithms, most branches occur when input elements are compared to each other. Following [7], on modern hardware these branches can be avoided as follows in C++:

- By using conditional moves (`CMOVcc`), which have the form `i = (x < y) ? j : i;`
- By casting a boolean to an integer (`SETcc`), which has the form `int i = (x < y);`

## 3 Lomuto BlockQuicksort

**3.1 One Pivot** Algorithm 4 (BlockLomuto1) describes the full partitioning method that can be plugged into the general quicksort procedure described as Algorithm 1. See Figure 2 to see the invariant that is kept by the partitioning method.

**Algorithm 4** One-Pivot Block Partitioning

**procedure** BlockLomuto1($A[1..n]$)

**Require:** $n > 1$, Pivot in $A[n]$

1: $\mathtt{p} \leftarrow A[n]$;
2: **integer** block$[0, \ldots, B-1]$, $\mathtt{i}, \mathtt{j} \leftarrow 1$, $\mathtt{num} \leftarrow 0$
3: **while** $\mathtt{j} < n$ **do**
4: $\mathtt{t} \leftarrow \min(B, n - \mathtt{j})$;
5: **for** $\mathtt{c} \leftarrow 0; \mathtt{c} < \mathtt{t}; \mathtt{c} \leftarrow \mathtt{c} + 1$ **do**
6: block$[\mathtt{num}] \leftarrow \mathtt{c}$;
7: $\mathtt{num} \leftarrow \mathtt{num} + (\mathtt{p} > A[\mathtt{j} + \mathtt{c}])$;
8: **for** $\mathtt{c} \leftarrow 0; \mathtt{c} < \mathtt{num}; \mathtt{c} \leftarrow \mathtt{c} + 1$ **do**
9: Swap $A[\mathtt{i}]$ and $A[\mathtt{j} + \text{block}[\mathtt{c}]]$
10: $\mathtt{i} \leftarrow \mathtt{i} + 1$
11: $\mathtt{num} \leftarrow 0$;
12: $\mathtt{j} \leftarrow \mathtt{j} + \mathtt{t}$;
13: Swap $A[\mathtt{i}]$ and $A[n]$;
14: **return** i;

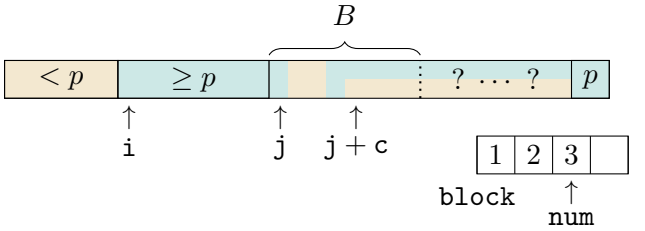


Figure 2: BlockQuicksort with Lomuto's partitioning scheme (Algorithm 4). Picture depicts the situation where Algorithm 4 is currently on Line 7 with $\mathtt{c} = 4$. So far, the block contains the indexes 1 and 2, representing that $A[\mathtt{j}+1]$ and $A[\mathtt{j}+2]$ are smaller than $p$. In general, given that `c` has value $c$ and `num` is $num$, block$[0..num-1]$ contains the indexes (relative to `j`) of all misplaced elements in $A[\mathtt{j}..\mathtt{j}+c]$. Unlabeled elements have only half the width of the array cell depicting $p$ in the picture.

Algorithm 4 is the straight-forward generalization of the standard Lomuto partitioning scheme discussed in the previous section. In addition to the pivot $p$ and the two indexes `i` and `j`, the algorithm uses an array `block` that can store $B$ indexes, and a variable `num`. Except when there are less than $B$ elements left to consider (see Line 4), the algorithm considers $B$ elements of the input at a time in the **while** loop that starts on Line 3. First, in Lines 5–7, it fills the `block` array with the indexes

**Algorithm 5** Two-Pivot Block Partitioning

**procedure** BlockLomuto2($A[1..n]$)

**Require:** $n > 1$, Pivots in $A[1] \leq A[n]$

```
1:  p ← A[1]; q ← A[n];
2:  integer block[0, ..., B − 1]
3:  i, j, k ← 2, num_<p, num_≤q ← 0
4:  while k < n do
5:      t ← min(B, n − k);
6:      for c ← 0; c < t; c ← c + 1 do
7:          block[num_≤q] ← c;
8:          num_≤q ← num_≤q + (q ≥ A[k + c]);
9:      for c ← 0; c < num_≤q; c ← c + 1 do
10:         Swap A[j + c] and A[k + block[c]]
11:     k ← k + t;
12:     for c ← 0; c < num_≤q; c ← c + 1 do
13:         block[num_<p] ← c;
14:         num_<p ← num_<p + (p > A[j + c]);
15:     for c ← 0; c < num_<p; c ← c + 1 do
16:         Swap A[i] and A[j + block[c]]
17:         i ← i + 1
18:     j ← j + num_≤q;
19:     num_<p, num_≤q ← 0;
20: Swap A[i − 1] and A[1];
21: Swap A[j] and A[n];
22: return (i − 1, j);
```

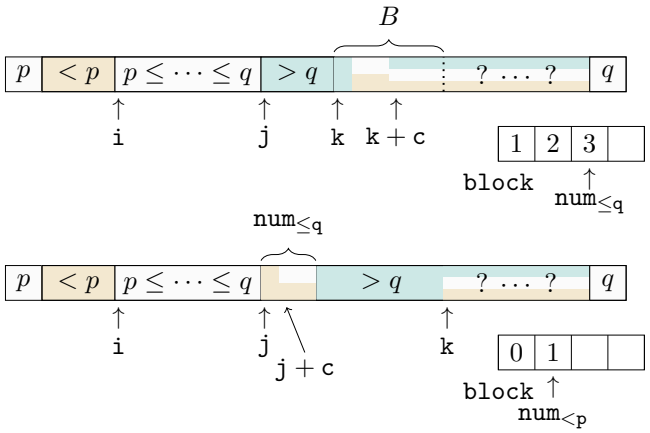


Figure 3: Lomuto block partitioning with two pivots. Top: Algorithm 5 compares elements with $q$ and is on Line 8 with $\mathtt{c} = 3$, cf. Figure 2. Bottom: Algorithm compares the $\mathtt{num}_{\leq \mathtt{q}}$ elements at most as large as $q$ to $p$. Algorithm is on Line 14 with $\mathtt{c} = 1$.

of the elements that are smaller than the pivot, i.e., the elements that are misplaced. This is done in a branch-free way by the cast to an integer. Next, in Lines 8-10, all misplaced elements are moved to a final position in the array, `num` is reset to 0, and `j` is advanced by one block size. After the loop ends, the pivot is put into place (Line 13) and its position is returned.

We stress that Algorithm 4 is the full algorithm as used in the experiments. It is easy to describe and, as we will see, performs as well as the BlockQuicksort variant using Hoare's partitioning scheme described in [7]. The reader is invited to compare BlockLomuto1 to their [7, Algorithm 3], which omits the complicated rearrangement phase.

**3.2 Two Pivots** Algorithm 5 (BlockLomuto2) describes a two-pivot version of Algorithm 4, see Figure 3. Compared to the latter, it uses one more index to store the beginning of a segment and one more `num` variable to store the number of misplaced elements in a block. The difference to BlockLomuto1 lies in the Lines 12–18 of Algorithm 5. Directly after moving misplaced elements (with respect to $q$) into a consecutive segment of the array, the algorithm checks the $\mathtt{num}_{\leq \mathtt{q}}$ elements in this segment immediately to $p$ using the same block to store misplaced elements. This can be done since all misplaced elements w.r.t. $q$ have been moved. It then moves misplaced elements smaller than $p$ in the very same fashion. After the rearrangement phase ends, the two pivots are swapped into place and their position is returned.

**3.3 Handling Equal Elements** Algorithm 5 ensures that all elements equal to $p$ or $q$ are stored between these pivots in the resulting partition. Thus, if $p$ equals $q$, the call on Line 5 of Algorithm 3 can be avoided. So, from now on, Line 5 is guarded by the statement `if (p != q)`.

## 4 Theoretical Properties of the Algorithms

In this section we analyze the theoretical properties of the proposed algorithms with respect to the number of element comparisons they make and their memory access pattern. The analysis is provided in a general way that allows us to choose the pivot(s) from a sample of the array elements.

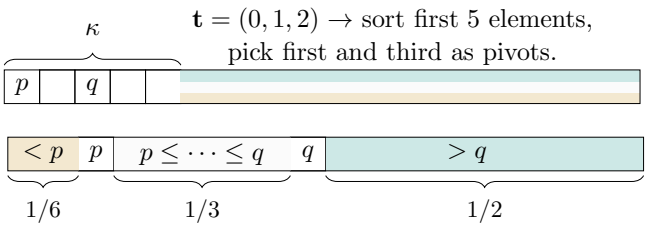


Figure 4: Top: Sampling step with $\mathbf{t} = (0, 1, 2)$ and two pivots. Bottom: Assuming a random permutation, expected sizes of different groups under the pivot choice. The 5 samples split the input into 6 different parts of equal size (in expectation). Thus, a fraction of 1/6 of the remaining elements are smaller than $p$, 1/3 are in between $p$ and $q$, and 1/2 are larger than $q$.

**4.1 Setup of the Analysis** The input is a random permutation of the set $\{1, \ldots, n\}$ which resides in an array $A[1..n]$. Fix an integer $k \in \{1, 2\}$ which denotes the number of pivots. Fix a vector $\mathbf{t} = (t_0, \ldots, t_k) \in \mathbb{N}^{k+1}$. Let $\kappa := \kappa(\mathbf{t}) = k + \sum_{0 \le i \le k} t_i$ be the number of samples. We assume that $\kappa$ is a constant independent of $n$. The general outline of a $k$-pivot quicksort algorithm is then as follows: If $n \le \kappa$, sort $A$ directly. Otherwise, sort the first $\kappa$ elements and then set $p_i = A[i + \sum_{j<i} t_j]$, for $1 \le i \le k$. Next, partition the input $A[\kappa + 1..n]$ with respect to the pivots $p_1, \ldots, p_k$. Subsequently, by a constant number of swaps, move the elements residing in $A[1..\kappa]$ to correct final locations. Finally, sort the $k+1$ subproblems recursively. (Algorithm 1 and Algorithm 3 have to be slightly adapted for this to hold.) See Figure 4 for an example.

**4.2 Cost Measures** We measure cost in two ways: we count the number of *comparisons* with the pivot(s) and we count the number of array cells *accessed* by the respective algorithm when sorting an input containing $n$ elements.

With regard to comparisons, we define the following random variables: Let $\mathrm{PCMP}_n$ denote the number of comparisons with elements against pivot $p$ in Algorithm 4, i.e., the number of times Line 7 is reached. Let $\mathrm{CMP}_n$ be the number of comparisons over the whole recursion. Let $\mathrm{PCMP}'_n$ denote the number of comparisons with elements against the pivots $p$ and $q$ in Algorithm 5 (Line 7 and Line 12) and define $\mathrm{CMP}'_n$ accordingly.

With regard to array accesses, we define the cost as the number of times a cell of the array has been accessed. (This is identical to the notion of *scanned elements* used in [21].) More precisely, for Algorithm 4, the random variable $\mathrm{PMA}_n$ counts the number of times we reach Line 7 (reading $A[\mathtt{j} + \mathtt{c}]$) and Line 9 (reading $A[\mathtt{i}]$). For Algorithm 5, $\mathrm{PMA}'_n$ counts the sum of the number of times Line 7, Line 12, and Line 14 are reached, i.e., individual accesses of `i`, `j`, and `k` in the array. We let by $\mathrm{MA}_n$ and $\mathrm{MA}'_n$ denote the cost over the whole recursion, respectively.

**From Partitioning Cost to Sorting Cost** Computing the average sorting cost from a given average partitioning cost is straight-forward if the latter is bounded by $a \cdot n + O(1)$, see [2].

For one-pivot quicksort with pivot $p$, we let by $a_0 := p - 1, a_1 := n - p$ denote the number of elements smaller/larger than the pivot. For two-pivot quicksort with pivots $p$ and $q$, we set $a_0 := p - 1, a_1 := q - p, a_2 := n - q$ to denote group sizes in the partition. For a given sequence $\mathbf{t} = (t_0, \ldots, t_k) \in \mathbb{N}^{k+1}$ we define $H(\mathbf{t})$ by

$$H(\mathbf{t}) = \sum_{i=0}^{k} \frac{t_i + 1}{\kappa + 1} \left( H_{\kappa+1} - H_{t_i+1} \right), \tag{4.1}$$

where $H_\ell$ denotes the $\ell$-th harmonic number. Let $\mathrm{P}_n$ denote the random variable which counts the cost of a single partitioning step, and let $\mathrm{C}_n$ denote the cost over the whole sorting procedure. Following [3], the average sorting cost $\mathrm{E}(C_n)$ follows the recurrence

$$\mathrm{E}(\mathrm{C}_n) = \mathrm{E}(\mathrm{P}_n) + \sum_{a_0 + \cdots + a_k = n-k} (\mathrm{E}(\mathrm{C}_{a_0}) + \cdots + \mathrm{E}(\mathrm{C}_{a_k})) \cdot \Pr(\langle a_0, \ldots, a_k \rangle), \tag{4.2}$$

where $\langle a_0, \ldots, a_k \rangle$ is the event that the group sizes are exactly $a_0, \ldots, a_k$. The probability of this event for a given vector $\mathbf{t}$ is $\frac{\binom{a_0}{t_0} \cdots \binom{a_k}{t_k}}{\binom{n}{\kappa}}$. Now, a result by Hennequin [9, Proposition III.9] says that for fixed $k$ and $\mathbf{t}$ and average partitioning cost $\mathrm{E}(P_n) = a \cdot n + O(1)$ recurrence (4.2) has the solution

$$\mathrm{E}(C_n) = \frac{a}{H(\mathbf{t})} n \ln n + O(n). \tag{4.3}$$

The sampling technique used here does not preserve randomness in subproblems, since a few elements have already been sorted during the pivot sampling step. For the analysis, we ignore that the unused samples have been seen and get only an estimate on the sorting cost. See [16] for a detailed analysis of this situation.

### 4.3 Analysis

**BlockLomuto1 (Algorithm 4)** Every element in the input that has not been sampled is compared with the pivot exactly once (Line 7), so we get $\mathrm{PCMP}_n = n + O(1)$. Conditioned on the pivot being $p$, there are exactly $p-1$ elements that are smaller than $p$ in the input. Each of these elements is accessed exactly once in Line 9, thus we get $\mathrm{PMA}_n = n+p+O(1)$. If the pivot is chosen according to a vector $\mathbf{t} = (t_0, t_1)$, we expect that there are $\frac{t_0+1}{t_0+t_1+2} \cdot n + O(1)$ many elements smaller than $p$. Thus, we get $\mathrm{E}(\mathrm{PMA}_n) = \left(1 + \frac{t_0+1}{t_0+t_1+2}\right) \cdot n + O(1)$ memory accesses on average.

**BlockLomuto2 (Algorithm 5)** Each unsampled element from the input is compared exactly once to $q$ (Line 7). Each element that is smaller than $q$ is then compared to $p$ (Line 12). If the pivots $p$ and $q$ are chosen according to $\mathbf{t} = (t_0, t_1, t_2) \in \mathbb{N}^3$, we expect that a fraction of $\frac{t_0+t_1+2}{t_0+t_1+t_2+3}$ elements are smaller than $q$. Thus, we obtain $\mathrm{E}(\mathrm{PCMP}'_n) = \left(1 + \frac{t_0+t_1+2}{t_0+t_1+t_2+3}\right) \cdot n + O(1)$. Regarding memory accesses, fix the pivots $p, q$. Every array cell is accessed by Line 7 of the algorithm. All array cells $A[j]$ with $j < q$ are accessed a second time in Line 12. Finally, all array cells $A[j]$ with $j < p$ are accessed a third time in Line 14 of the algorithm. These considerations yield $\mathrm{E}(\mathrm{PMA}'_n) = \left(1 + \frac{2t_0+t_1+3}{t_0+t_1+t_2+3}\right) \cdot n + O(1)$.

**Comparison with One-Pivot Hoare Partitioning** BlockQuicksort [7] uses Hoare's partitioning scheme. It is immediate that in Hoare's partitioning scheme every element is compared once to the pivot, and every array cell is accessed exactly once, as well.

### 4.4 Comparison to IPS$^4$o

IPS$^4$o with $2^\ell$ pivots makes $\ell$ comparisons per element. Thus, it makes $\ell \cdot n + O(1)$ comparisons in the partitioning step. A

| Algo | AS | Cost | Best (cost, **t**) |
|---|---|---|---|
| $H_1$ | | cmp | $2.00n \ln n$, $(0,0)$ |
| $H_1$ | 0 | ma | $2.00n \ln n$, $(0,0)$ |
| $H_1$ | | cmp + ma | $4.00n \ln n$, $(0,0)$ |
| $H_1$ | | cmp | $1.71n \ln n$, $(1,1)$ |
| $H_1$ | 2 | ma | $1.71n \ln n$, $(1,1)$ |
| $H_1$ | | cmp + ma | $3.43n \ln n$, $(1,1)$ |
| $H_1$ | | cmp | $1.62n \ln n$, $(2,2)$ |
| $H_1$ | 4 | ma | $1.62n \ln n$, $(2,2)$ |
| $H_1$ | | cmp + ma | $3.24n \ln n$, $(2,2)$ |
| $H_1$ | | cmp | $1.53n \ln n$, $(5,5)$ |
| $H_1$ | 10 | ma | $1.53n \ln n$, $(5,5)$ |
| $H_1$ | | cmp + ma | $3.06n \ln n$, $(5,5)$ |
| $L_1$ | | cmp | $2.00n \ln n$, $(0,0)$ |
| $L_1$ | 0 | ma | $3.00n \ln n$, $(0,0)$ |
| $L_1$ | | cmp + ma | $5.00n \ln n$, $(0,0)$ |
| $L_1$ | | cmp | $1.71n \ln n$, $(1,1)$ |
| $L_1$ | 2 | ma | $2.57n \ln n$, $(1,1)$ |
| $L_1$ | | cmp + ma | $4.29n \ln n$, $(1,1)$ |
| $L_1$ | | cmp | $1.62n \ln n$, $(2,2)$ |
| $L_1$ | 4 | ma | $2.38n \ln n$, $(1,3)$ |
| $L_1$ | | cmp + ma | $4.05n \ln n$, $(2,2)$ |
| $L_1$ | | cmp | $1.53n \ln n$, $(5,5)$ |
| $L_1$ | 10 | ma | $2.22n \ln n$, $(4,6)$ |
| $L_1$ | | cmp + ma | $3.78n \ln n$, $(4,6)$ |
| $L_2$ | | cmp | $2.00n \ln n$, $(0,0,0)$ |
| $L_2$ | 0 | ma | $2.40n \ln n$, $(0,0,0)$ |
| $L_2$ | | cmp + ma | $4.40n \ln n$, $(0,0,0)$ |
| $L_2$ | | cmp | $1.73n \ln n$, $(0,1,2)$ |
| $L_2$ | 3 | ma | $1.92n \ln n$, $(0,1,2)$ |
| $L_2$ | | cmp + ma | $3.65n \ln n$, $(0,1,2)$ |
| $L_2$ | | cmp | $1.62n \ln n$, $(1,1,3)$ |
| $L_2$ | 5 | ma | $1.88n \ln n$, $(0,2,3)$ |
| $L_2$ | | cmp + ma | $3.51n \ln n$, $(1,1,3)$ |
| $L_2$ | | cmp | $1.55n \ln n$, $(2,3,6)$ |
| $L_2$ | 11 | ma | $1.77n \ln n$, $(1,3,7)$ |
| $L_2$ | | cmp + ma | $3.32n \ln n$, $(2,3,6)$ |

Table 1: Best asymptotic expected sorting cost of Hoare's one-pivot ($H_1$), Lomuto's one-pivot ($L_1$), and Lomuto's two-pivot ($L_2$) algorithm with regard to a given sample size AS (in addition to the pivot(s)). For each cost measure (cmp–comparisons, ma–memory accesses), we explicitly state the sample vector $\mathbf{t}$ for which this cost is achieved.

rough estimate for the number of array accesses is obtained by counting three array accesses per array position; the first during classification, a second when a block is full and is moved back into the array, the third when a block is moved to a final position during the final rearrangement phase.

**4.5 Putting Everything Together** Using (4.3) with the cost formulas derived above, we can reason about the expected sorting cost of Algorithms 4/5, and compare them to the two other approaches. Table 1 gives an overview over the minimum cost achievable with certain sample sizes; we present a short summary.

First, Lomuto's partitioning scheme is inferior to Hoare's, in particular with regard to the number of memory accesses. However, choosing pivots from a sample greatly improves the number of accesses, both for the one- and the two-pivot variant. While BlockLomuto2 makes slightly more comparisons, the number of memory accesses improves by around $0.5n \ln n$ for the sample sizes considered. The pivot choices that achieve the minimum cost are the median of the sample (Hoare's scheme), the median or the element one larger than it (BlockLomuto1), and a skewed pivot choice for BlockLomuto2. E.g., $(1, 1, 1)$ accesses around 9% more array cells than $(0, 1, 2)$. As a rule of thumb, the calculations motivate to take the larger pivot $q$ as the median in a sample, and to choose the smaller pivot $p$ as the median of all the sampled elements smaller than $q$. For a sample size of 5, BlockLomuto2 makes a factor of 1.18 more memory accesses than Hoare.

Using 128 pivots from a sample of 257 elements, taking every second element as a pivot, IPS$^4$o outperforms its competitors by a large margin. (The same behavior was observed for other samplesort-based approaches in [1].) On average, it makes around $1.51n \ln n + O(n)$ comparisons and $0.65n \ln n + O(n)$ memory accesses. In comparison, BlockLomuto2 with a sample of 257 elements makes $1.45n \ln n + O(n)$ comparisons, but incurs $1.69n \ln n + O(n)$ memory accesses, on average.

## 5 Experimental Evaluation

This section presents the evaluation of the experiments we conducted with regard to the performance and other properties of the proposed algorithms. The implementation was written in C++, performance counters were obtained using PAPI[2]. Our code, the Jupyter notebook that makes all computations transparent and which includes additional plots not found in the paper, as well as raw results are available at `https://bitbucket.org/alenex19_paper48/submission/`.

**Input Distributions** We ran benchmarks with the following input distributions: Permutation, Sawtooth, RandomDup, Sorted, Reversed, Equal, and EightDup. Permutation chooses the input as a random permutation of $\{1, \ldots, n\}$; Sawtooth sets $A[i] = i \mod \sqrt{n}$, RandomDup uses $A[i] = \texttt{uniform}(n) \mod \sqrt{n}$; Sorted sets $A[i] = i$, Reversed sets $A[i] = n - i - 1$, and Equal uses $A[i] = 1$. Finally, EightDup sets $A[i] = i^8 + n/2 \mod n$, following [7], resulting in inputs that duplicate the median many times when $n$ is a power of two. All inputs consisted of 64-bit integers. For each input size, we ran each algorithm on the same 600 inputs drawn from a certain input distribution. All figures in the following contain the average over these trials. We call a running time improvement *significant* if it was observed in at least 95% of the trials.

**Machine Details** We ran the experiments on two different machines: `Xeon` and `i7`. `Xeon` is set up with two 14-core Intel Xeon E5-2690 v4 CPUs clocked at 2.6 GHz, 35MB L3 Cache and 512GB of RAM. `Xeon` was running Ubuntu 16.10 with Linux kernel 4.4 and the code was compiled with `gcc` version 5.4.0. The compiler flags were `-O3 -march=native -funroll-loops`. All runs used a single core and a single thread. There are marginal differences to results obtained on `i7`, thus we moved the discussion for this architecture to Appendix A.

**Competitors.** To compare the running time to other implementations, we used the implementation of BlockQuicksort [7] (BlockQS) available at `https://github.com/weissan/BlockQuicksort`

[2] `http://icl.utk.edu/papi/`

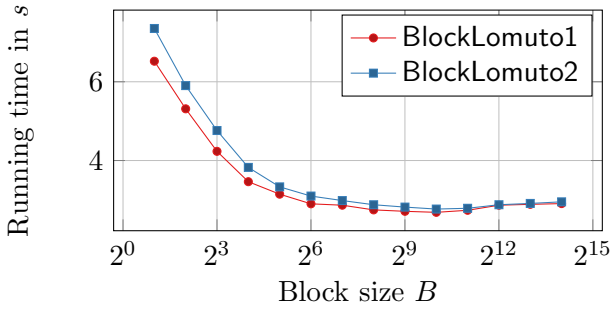


Figure 5: Influence of block size $B$ to the running time for $n = 2^{26}$, `Xeon` on Permutation.

and the implementation of IPS$^4$o [4] retrieved from `https://github.com/SaschaWitt/ips4o`. Furthermore, we used std::sort from `gcc` as the baseline implementation to compare all algorithms to. The implementations of BlockLomuto1 and BlockLomuto2 save lines of code by a factor of 5 and 8, resp., compared to BlockQS under a similarly dense coding style.

**5.1 Block Size** Figure 5 shows how the block size influences the sorting time for the one- and two-pivot variant for $2 \leq B \leq 2^{14}$. Both variants benefit from large block sizes, decreasing the running time to sort the input of $n = 2^{26}$ items from around 7 seconds to around 3 seconds. The minimum is attained for both variants for a block that can hold up to 1024 items, but there is no big difference for block sizes around this value. Consequently, we set the block size to $B = 1024$ for both implementations.

**5.2 Pivot Strategies** We implemented different pivot selection strategies and compared them to each other. Figure 6 shows the result of this experiment. For BlockLomuto1, we compare direct choice, median of 3, and median of medians of 5, which groups 25 elements in 5 groups, chooses the median in each, and chooses the median of these medians as pivot, see [7]. For BlockLomuto2, we compare direct choice, first two of three elements, first and third of 5, second and fourth of 5, and the adaption of the median of medians strategy described above, in which we take the first and third element in the sample of 5 medians.

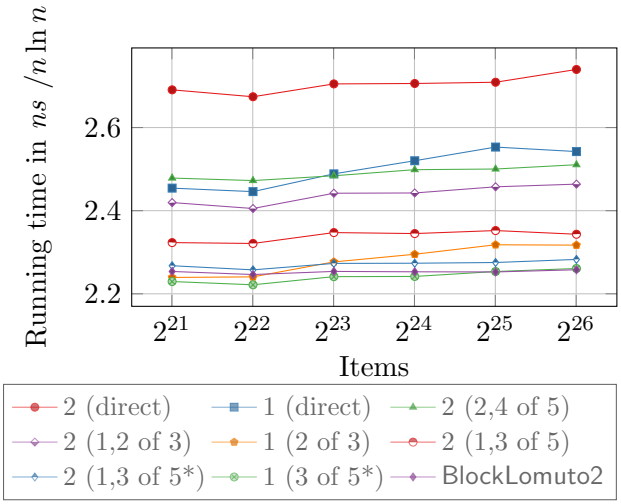


Figure 6: Running times on Permutation on `Xeon` for different pivot choices. Legend entries of the form `p (R of S)` are read as follows: For the `p`-pivot variant, we sort `S` elements and choose the elements `R` in the sorted sample, e.g., `(2, 4 of 5)` means that the second and the fourth element in a sorted sample of five elements are chosen as pivots; a `*` refers to the median of medians sampling.

The plot shows that the pivot selection strategy has big influence on the running time. In particular, choosing the pivot directly from the input without sampling performs much worse than even simple pivot selection strategies. For example, for BlockLomuto1, there is a difference of a factor 1.14 in running time when choosing the pivot directly from the input compared to choosing it as the median of 5 medians. BlockLomuto2 is a factor of 1.16 times slower without sampling than choosing the first two elements in a sample of three elements, i.e., looking at one additional element. It is by a factor of 1.22 slower when pivots are chosen among the medians of five elements each. In particular, choosing skewed pivots as predicted in Section 4 provides an improvement for the two-pivot variant. Choosing the second and fourth element is a factor of 1.07 slower than choosing the first and third element.

For the final implementation, we used a slightly more elaborate pivot selection strategy that switches between pivot strategies based on the input size. The best thresholds to switch strate-

gies have been obtained experimentally as well. While improvements in running time are consistent, they do not exceed a factor of 1.01 compared to the fastest variant that does not switch strategies, as examplified through the difference between BlockLomuto2 and 2 (1, 3 of 5*) in Figure 6.

**5.3 Further Tuning** Apart from choosing the block size and pivot selection strategy based on the experiments described above, the implementations used in the experiments are identical to Algorithm 4 and Algorithm 5. In contrast to the observations made in [7, Section 3.2], unrolling the main loop did not increase performance. We suspect that this is due to the simplicity of the code that allows the compiler to unroll the code. Furthermore, we do not perform the cyclic rotations described in [7] since they actually increased running time. We suspect that this is due to the CPU pipelining the load-/store instruction issued in Line 9 of Algorithm 4, whereas the cyclic shifts described in [7] introduce read/write dependencies.

**5.4 Running Times** We discuss the running times observed on `Xeon` plotted in Figure 7. In the following, running time differences are always stated for the data points associated with $2^{27}$ items.

On Permutation, IPS$^4$o has the fastest running time. On average, BlockLomuto1 and BlockLomuto2 are a factor of 1.15 times slower, BlockQS is 1.20 times slower; stdsort is a factor of 2.22 slower than IPS$^4$o. With regard to significant running time differences, these factors decrease in absolute value by about 0.02.

For inputs containing many duplicates (SawTooth, RandomDup, EightDup, Equal), BlockLomuto1 cannot compete with the other algorithms; a drawback of Lomuto's partitioning scheme that had already been identified in [7]. Thus, we omit it in the plots. In the respective order of these input types, it was 221, 280, 18571, 21010 times slower than IPS$^4$o. On the other hand, BlockLomuto2 is robust on all of these input types, being fastest on RandomDup and Equal, and being close in performance to BlockQS on the other two. Only on Sorted and Reversed it is around a factor 2 to 3 slower than the fastest variant. For both of these input types, stdsort performs very well. Furthermore, we note that average running times predict the difference between implementations very well: Even for random structured inputs (RandomDup), significant differences that occurred in at least 95% of the trials made differences only about a factor of 0.02 smaller than average running time differences.

Despite their simplicity, the BlockLomuto variants show very good performance. In particular, the two-pivot variant is robust on different input types, achieving robustness with the small twist introduced in Section 3. On the other hand, BlockQS achieves robustness with elaborate additional checks of the input after the main partitioning step finishes.

**5.5 Practical Observations** Comparing our running time results to the considerations in Section 4, it is quite surprising that BlockLomuto1 and BlockLomuto2 can compete in performance with BlockQS. In Table 2 we present selected measurements we got from running experiments. Experiments are run on Permutation to report on the average-case behavior of implementations.

From Section 4, we expect that IPS$^4$o and BlockQS use fewer instructions than the Lomuto-variants discussed in this paper. Furthermore, they are expected to have a better cache behavior.

From Table 2 we see that with regard to L1 cache misses, BlockQS incurs the fewest misses, followed by stdsort, BlockLomuto2, IPS$^4$o, and BlockLomuto1. So, IPS$^4$o uses too many blocks for all of them to reside in L1 cache. Looking at L2 cache misses, IPS$^4$o incurs fewer misses than all other algorithms, as predicted. BlockLomuto2 has a better cache behavior than BlockLomuto1, but behaves worse than BlockQS, which again follows nicely the memory accesses computed in Section 4.

BlockLomuto1 and BlockLomuto2 branch conditionally in about the same dimension as IPS$^4$o and BlockQS, and there is a big difference to the number of branches in stdsort. These branches are easy to predict, as shown by the low misprediction rates. Here, LomutoBlock variants make fewer mispredictions than BlockQS, and there is a big gap to the misprediction rate of IPS$^4$o; stdsort mispredicts

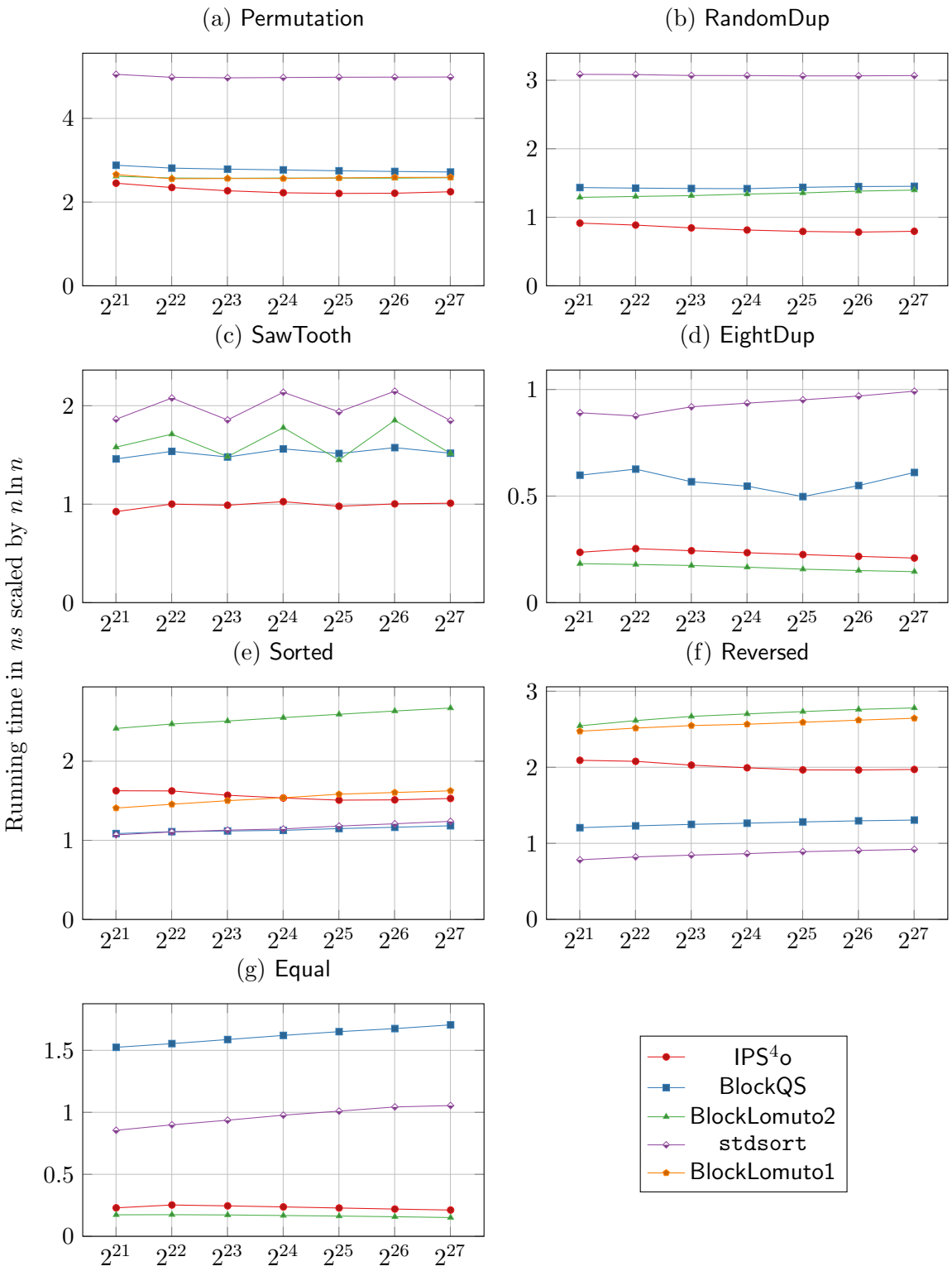


Figure 7: Running time plots on `Xeon`. The $x$-axis represents the number of items, the $y$-axis shows the running time in nanoseconds scaled by $n \ln n$.

| Algorithm | L1/L2 CM | CB (% MP) | INS | WOF (%, cycles) |
|---|---|---|---|---|
| IPS$^4$o | .147 / .085 | 1.080 ( 6.1%) | 12.870 | 2.921 (41.3%, 6.834) |
| BlockLomuto1 | .169 / .143 | 0.926 (11.7%) | 16.456 | 3.001 (38.7%, 7.758) |
| BlockLomuto2 | .144 / .121 | 0.884 (10.5%) | 17.467 | 2.579 (32.7%, 7.878) |
| BlockQS | .125 / .102 | 0.877 (14.7%) | 17.566 | 3.240 (38.7%, 8.354) |
| stdsort | .137 / .115 | 2.128 (29.0%) | 11.769 | 11.509 (74.0%, 15.550) |

Table 2: CPU counter measurements for $n = 2^{27}$ items on 600 trials on Permutation. We use the abbreviations "CM" for "Cache Misses", "INS" for "Instructions", "CB" for "conditional branch instructions", "MP" for "mispredicted", and "WOF" for "cycles without instruction finished". All values are normalized by $n \ln n$.

more branches, since comparisons to the pivot have a prediction rate of around 50%.

With regard to the instruction count, we see that IPS$^4$o makes by far the fewest instructions among "branch-free" variants. Somehow surprisingly, the Lomuto-variants incur fewer instructions than BlockQS. We suspect that this is because of their simpler structure that simplifies bookkeeping. Looking at cycles in which no instruction was finished (because of, e.g., memory stalls or branch mispredictions), Lomuto-based variants have slightly fewer of them compared to BlockQS; around 33–40% of the total amount of cycles are of this type for branch-free variants, while they make up 74% of the cycles for stdsort.

In conclusion, we see that the theoretical differences between IPS$^4$o/BlockQS and the Lomuto-based variants translate into practice, but their easier structure make them competitive to BlockQS. No variant can compete with IPS$^4$o, both in theory and in practice.

## 6 Conclusion

This paper introduced simple variants of the BlockQuicksort algorithm by [7] using block-based versions of Lomuto's partitioning scheme. The implementation was shown to be competitive in running time to the implementation of [7]. A novel twist to the general two-pivot quicksort approach made the proposed two-pivot variant particularly robust with regard to different input distributions. The paper presented theoretical properties of the algorithms and verified them through experiments.

Due to their simple structure, the proposed algorithms are particularly suited to test further optimizations. For example, it would be nice to inspect how much can be gained from implementing different block operations such as filling the block or rearranging elements through vectorized SIMD operations. In another line of research, having a simple implementation could allow to formally verify the correctness of the implementation. Additionally, a theoretical analysis of handling equal elements as described in Section 3.3 seems interesting [22].

We remark that a 3-pivot variant of Lomuto's scheme did not give any improvements with regard to observed running times; the same is true for a two-pivot variant of Hoare's scheme used in BlockQuicksort. In particular, handling all the edges cases correctly made this algorithm very complicated.

**Acknowledgments** We thank Armin Weiss for a useful hint in the two-pivot version. We also thank Timo Bingmann who provided some source code used in the testing framework. Furthermore, we thank the anonymous reviewers for their suggestions that helped us in improving the presentation of this paper.

## References

[1] Aumüller, M.: On the Analysis of Two Fundamental Randomized Algorithms - Multi-Pivot Quicksort and Efficient Hash Functions. Ph.D. thesis, Technische Universität Ilmenau, Germany (2015), http://www.db-thueringen.de/servlets/DocumentServlet?

[2] Aumüller, M., Dietzfelbinger, M.: Optimal partitioning for dual-pivot quicksort. ACM Trans. Algorithms 12(2), 18:1–18:36 (2016), http://doi.acm.org/10.1145/2743020

[3] Aumüller, M., Dietzfelbinger, M., Klaue, P.: How good is multi-pivot quicksort? ACM Trans. Algorithms 13(1), 8:1–8:47 (2016), http://doi.acm.org/10.1145/2963102
[4] Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: In-place parallel super scalar samplesort (ipsssso). In: 25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria. pp. 9:1–9:14 (2017), https://doi.org/10.4230/LIPIcs.ESA.2017.9
[5] Bentley, J.L.: Programming pearls. Addison-Wesley (1986)
[6] Brodal, G.S., Moruz, G.: Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In: Proc. of the 9th International Workshop on Algorithms and Data Structures (WADS'05). pp. 385–395. Springer (2005), http://dx.doi.org/10.1007/11534273_34
[7] Edelkamp, S., Weiß, A.: Blockquicksort: Avoiding branch mispredictions in quicksort. In: 24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark. pp. 38:1–38:16 (2016), https://doi.org/10.4230/LIPIcs.ESA.2016.38
[8] Elmasry, A., Katajainen, J., Stenmark, M.: Branch mispredictions don't affect mergesort. In: Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings. pp. 160–171 (2012), http://dx.doi.org/10.1007/978-3-642-30850-5_15
[9] Hennequin, P.: Analyse en moyenne d'algorithmes: tri rapide et arbres de recherche. Ph.D. thesis, Ecole Politechnique, Palaiseau (1991)
[10] Hennessy, J.L., Patterson, D.A.: Computer Architecture - A Quantitative Approach, 5th Edition. Morgan Kaufmann (2012)
[11] Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–15 (1962)
[12] Kaligosi, K., Sanders, P.: How branch mispredictions affect quicksort. In: Proc. of the 14th Annual European Symposium on Algorithms (ESA'06). pp. 780–791. Springer (2006)
[13] Kushagra, S., López-Ortiz, A., Qiao, A., Munro, J.I.: Multi-pivot quicksort: Theory and experiments. In: Proc. of the 16th Meeting on Algorithms Engineering and Experiments (ALENEX'14). pp. 47–60. SIAM (2014)
[14] Martínez, C., Roura, S.: Optimal sampling strategies in quicksort and quickselect. SIAM J. Comput. 31(3), 683–705 (2001)
[15] Musser, D.R.: Introspective sorting and selection algorithms. Softw., Pract. Exper. 27(8), 983–993 (1997)
[16] Nebel, M.E., Wild, S., Martínez, C.: Analysis of pivot sampling in dual-pivot quicksort: A holistic analysis of yaroslavskiy's partitioning scheme. Algorithmica pp. 1–52 (2015)
[17] Sanders, P., Winkel, S.: Super scalar sample sort. In: Proc. of the 12th Annual European Symposium on Algorithms (ESA'04). pp. 784–796. Springer (2004)
[18] Sedgewick, R.: Implementing quicksort programs. Commun. ACM 21(10), 847–857 (1978)
[19] Wild, S.: https://cs.stackexchange.com/questions/11458/ accessed on August 8, 2018.
[20] Wild, S.: Dual-pivot quicksort and beyond: Analysis of multiway partitioning and its practical potential. Ph.D. thesis, Technische Universität Kaiserslautern (2016)
[21] Wild, S.: Dual-pivot and beyond: The potential of multiway partitioning in quicksort. it - Information Technology 60(3), 173–177 (2018), https://doi.org/10.1515/itit-2018-0012
[22] Wild, S.: Quicksort is optimal for many equal keys. In: Proceedings of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2018, New Orleans, LA, USA, January 8-9, 2018. pp. 8–22 (2018), https://doi.org/10.1137/1.9781611975062.2
[23] Wild, S., Nebel, M.E.: Average case analysis of java 7's dual pivot quicksort. In: Proc. of the 20th European Symposium on Algorithms (ESA'12). pp. 825–836 (2012)

## A Running Time Plots for i7

The machine i7 was equipped with a 4-core Intel Core i7-4790 clocked at 3.6 GHz, with 8 MB L3 Cache and 32GB RAM. i7 was running CentOS 7 with Linux kernel 3.10 and code was compiled using gcc version 5.1.1. Figure 8 presents an overview over the running time measurements that we obtained. While i7 turns out to be faster than Xeon, differences in running time are comparable to Section 5.

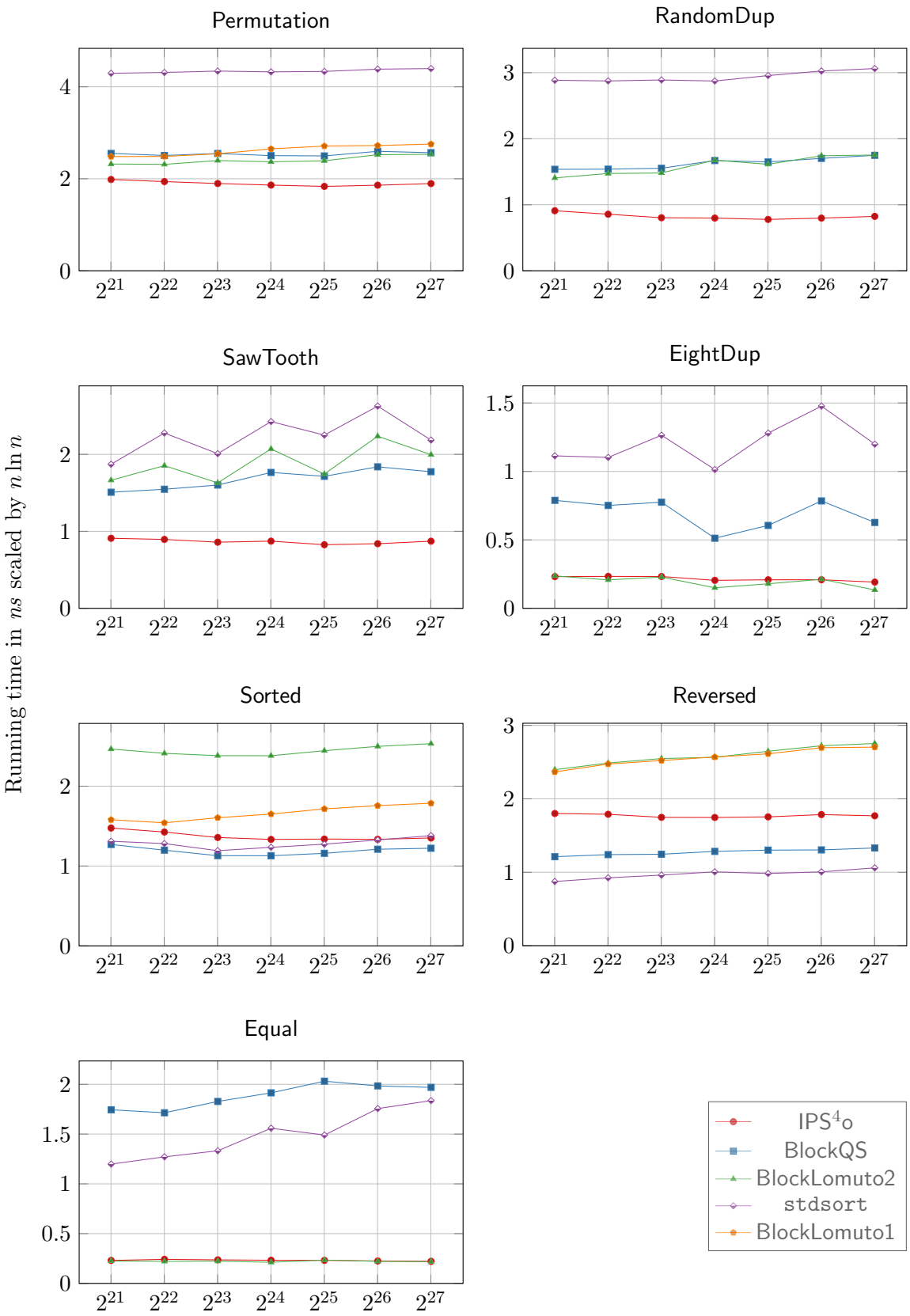


Figure 8: Running time plots on `i7`. The $x$-axis represents the number of items, the $y$-axis shows the running time in nanoseconds scaled by $n \ln n$.