DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Branch-Free Parallel In-Place Quicksort

Simon Lang

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Branch-Free Parallel In-Place Quicksort

Sprungfreier Paralleler In Situ Quicksort

Author:	Simon Lang
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Viktor Leis
Submission Date:	15.03.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.03.2018 Simon Lang

Acknowledgments

I want to thank my loving partner Mareike Haller for all her support during the time I wrote this thesis. She was the one enabling all of this by taking off all my day-to-day work from me. Furthermore, I want to thank Christoph Hausmann and Dr. Wolfgang Maison for their commitment and their very valuable feedback. Finally, I want to thank my advisor Dr. Viktor Leis for answering all of my questions and providing the founding idea for this thesis to me.

Thank you all for your great support!

Abstract

In this master's thesis we provide a novel approach to adapt Quicksort to the requirements of modern computer architecture. One problem of classical Quicksort implementations is that their algorithmic best-case is also their worst-case in terms of branch prediction. To tackle this problem we present BF-Quicksort which avoids those hardly predictable branches and therefore aims to increase Quicksort's overall performance. Furthermore, we apply parallelization techniques to our proposed algorithm to enable concurrent execution on modern multi-core processors. We do all this while still maintaining Quicksort's in-place property. At the end of this thesis we validate the proposed properties of our algorithm in theory and in practice. Our experiments for all of our introduced algorithms yield increased performance in almost all of our benchmarks. This is why we conclude that branch-free Quicksort implementations are worth to be considered as an alternative to classical Quicksort approaches.

In dieser Masterarbeit stellen wir einen neuen Ansatz vor, der Quicksort an die Anforderungen moderner Rechnerarchitekturen anpasst. Ein Problem, mit dem klassische Quicksort-Implementierungen zu kämpfen haben ist, dass ihr algorithmisches Best-Case-Szenario das Worst-Case-Szenario für eine gute Sprungvorhersage ist. Um dieses Problem zu lösen präsentieren wir BF-Quicksort, einen Algorithmus der schwer vorhersagbaren Sprungbefehle vermeidet und damit seine Leistung steigert. In einem nächsten Schritt parallelisieren wir unseren Algorithmus, um die Vorteile von modernen Mehrkern-Systemen nutzen zu können. Bei jeder dieser Anpassungen achten wir darauf, dass Quicksort seine in-situ Eigenschaft weiterhin behält. Am Ende dieser Arbeit validieren wir unseren Algorithmus sowohl theoretisch, als auch praktisch. Unsere Experimente mit allen präsentierten sequentiellen Algorithmen und unserem parallelen Partitionierer erzielen im Allgemeinen gute Resultate in unserer Testumgebung. Daher sind wir der Überzeugung, dass sprungfreie Quicksort-Implementierungen eine gute Alternative zu herkömmlichen Varianten sein können.

"There's still one good thesis left in Quicksort."
-D. E. Knuth in [26]

Contents

1 Introduction

Since the development of Quicksort by Tony Hoare [15] it became one of the most important sorting algorithms and an essential part of computer science as we know it today. Over the years Quicksort has received many improvements (e.g. [12] [27] [25] [8] [11]) and is an integral part of well known systems like the C standard library's `qsort` [16] and Java's standard sort function for arrays: Dual Pivot Quicksort [9]. Nowadays it is more important than ever to process huge data sets as fast as possible, because of recent developments in how and in which amounts we generate them (e.g. IoT, sensors in factories and mobile devices, etc.). Therefore, research on improving the performance of Quicksort is still ongoing and an open field for innovations [34] [3] [10] [6] [28]. Although open for innovations, the basic principles in Quicksort remain about the same: First we choose one element from the input as pivot and start moving elements from the input into two partitions: One containing all elements smaller than the pivot and one containing all larger elements. In a recursive call we then apply these partitioning steps to all partitions generated this way, until the input is sorted. One recent study on improving the performance of Quicksort by Kaligosi and Sanders [20] uncovered that suboptimal choices of pivots (*skewed pivots*) lead to better performance in standard Quicksort implementations. At first glance this finding may seem odd, because the algorithm's performance per se gets worse for pivots further apart from the input set's median. From an algorithmic perspective a skewed pivot increases the Quicksort's number of recursions, but the reason why we can observe an increase in the algorithms overall performance regardless of the algorithmic impediment lies in modern computer architecture: Microprocessors try to figure the outcome of a conditional jump statement before its actual evaluation to enable instruction prefetching. Usually those *branch predictors* have a high success rate of over 90% [32] for deciding whether a given jump instruction will be taken or not. Sadly those circuits work particularly bad in case of Quicksort: To get the optimal amount of comparisons and iterations for Quicksort, we always aim for the exact median as a pivot in all of its iterations. Consequently, in each partitioning phase 50% of all elements go to the left of the pivot and the other half to its right. This algorithmically good behavior is technically bad, because it means that a branch predictor has a chance of about 50% to provide the correct guess for where to place the currently examined element and which instructions to prefetch in order to make this happen. Such a perfect execution of Quicksort would lead to a maximum of 0.5 mispredictions per element. Kaligosi et. al.'s measurements show that this worst-case scenario in terms of mispredicted jumps also applies to real world algorithms. Each such *branch miss* introduced by a wrong prediction usually causes a flush in the CPU's instruction pipeline, which in turn quickly becomes very time consuming for large input counts. Kaligosi et. al. utilize these facts in their proposed Quicksort implementation, by skewing the pivot on purpose by $\frac{n}{10}$ elements to increase the branch predictor's accuracy and therefore the algorithm's overall performance.

In this thesis we propose an alternative approach to this problem, by entirely avoiding branches in the partitioning phase of Quicksort. We do this to keep the algorithm's number of recursions as low as possible while still taking modern CPU architecture into consideration. We name the proposed algorithm BF-Quicksort, which is short for "Branch-Free Buffer Filling Quicksort". The main contributions of this thesis are:

- Some of the fastest freely available sequential and parallel implementations of branch-free in-place partitioning and Quicksort
- Analyses and tuning of the variable factors of the proposed algorithms
- Comparisons of all introduced algorithms to prevailing competitors

To achieve all of this, we first analyze the paper on parallel Quicksort, which provides the founding idea for our approach and one of the papers from which our sequential implementation hugely benefited. In the following section, we cover the preliminaries in which we explain branch prediction, in-place sorting, and Quicksort in more detail. The fourth section contains our proposed implementation of BF-Quicksort and our main findings. After that, we analyze our algorithm in the next two sections in both theory and practice. We use our findings of those sections to further tune our algorithm and to compare our performance to recent partitioning and sorting algorithms. At the end of our thesis we conclude and summarize our findings and give a brief outlook on further research in the field of branch-fee parallel sorting.

2 Related Work

2.1 Tsigas & Zhang

The work of Tsigas and Zhang [30] largely provides the theoretical foundation of this thesis. In their work, they provide a fast and scalable implementation of parallel Quicksort. They propose a fine-grained 3+1 phase approach to parallelize the algorithm: In a first step the algorithm partitions concurrently over the input and performs a sequential clean-up afterwards. In the next step Tsigas et. al. assign threads to both generated partitions relative to their size. This procedure is then repeated until the number of processors assigned to a partition is equal to one. As soon as this is the case, the single processor calls sequential Quicksort on its partition and performs dynamic load balancing once it finishes its workload. We describe this algorithm in more detail in Section 4.2. This very successful algorithm is also included in GNUs *GCC parallel mode* [29], which is a collection of parallel implementations for popular standard algorithms, like partitioning and sorting. For parallelization of BF-Quicksort we use their proposed 3+1 phase algorithm and implement it via the thread pool pattern. Moreover, the first two phases are not only interesting in terms of parallelism: They also inspired us to use swap buffers in single-threaded partition, which in turn enables our branch-free Quicksort.

2.2 Edelkamp & Weiß

The independent paper on *sequential BlockQuicksort* of Edelkamp and Weiß [10] provides some similar ideas on how to approach an efficient branch-free Quicksort implementation as we propose. Their work especially facilitates BF-Quicksort in is sequential base case. However, our original idea for BF-Quicksort came up independently from this paper and therefore the first versions of our algorithm where not influenced by Edelkamp et. al.'s work. After discovering the paper, we gratefully enhanced our algorithm by their findings: We make use of their provided code framework for pivot selection and their cleaned-up implementation of the GCC standard library's Insertionsort.

Similar to BF-Quicksort, *BlockQuicksort* also makes use of swap buffers: In a first phase it iterates over the input data in static blocks, filling swap buffers with mispositioned elements, until it reaches the end of the respective block. After this scanning phase their algorithm switches to a swapping phase, in which it empties its (partially) filled buffers, by swapping their contained elements to the correct partitions. Instead of using fixed size blocks, our algorithm iterates freely over the input and switches to the swapping phase only if our swap buffers are completely filled. Furthermore, our buffers are dynamically sized: We calculate a L1-Cache optimal size for the buffers at the beginning of our algorithm. In direct comparison, our algorithm performs for our benchmarks at least similar or better than BlockQuicksort. Furthermore, we apply an additional pattern detector for inputs containing only identical elements and take branch-free Quicksort to the next level by introducing parallelism as stated above.

3 Preliminaries

To introduce BF-Quicksort it is important to understand all keywords in the title of our thesis. The section on branch prediction covers technical details and indicates the main problem under which comparison-based Quicksort implementations suffer. All of our explanations in the first section are based on the chapters 2.3.4, 2.4.3, and 7.2 of the book by Brinkschulte and Ungerer on microcontrollers and microprocessors [32] if not stated otherwise. In the next section we cover the definition of in-place for the purpose of our thesis. This property of algorithms is covered by most Quicksort implementations and is one fundamental part of Quicksort's overall success. At the end of this chapter we go deeper into the general idea of Quicksort and combine the findings of both previous sections.

3.1 Branch Prediction

Modern superscalar processors cover a tremendous amount of optimization in order to increase their overall performance. To improve instruction throughput on a single machine, instructions are not executed at once: Each instruction is split into several steps, which each can be executed in a single clock cycle. To keep track of the execution order for those partial instructions, modern CPU architecture introduces a queue to serialize incoming instruction steps sequentially. This *instruction pipeline* is part of the Microprocessor and provides *stages*, which are specifically responsible to hold a particular instruction step. For example a 5 stage pipeline may support the following steps: *Instruction Fetch*, *Instruction Decode/Register Fetch*, *Execute/Address Calculation*, *Memory Access*, and *Write Back*. Ideally, all stages of the pipeline are used by a different instruction on each stage. This leads to parallel execution of instructions on hardware level and therefore a speedup of up to $\frac{n \cdot k}{k+n-1}$, where n is the number of parallel executed instructions and k is the number of pipeline stages. If the amount of incoming instructions is equal to infinity and no other side-effects occur, the gained speedup is equal to k. Therefore, each processor tries to keep its pipeline as full as possible all the time, by fetching a new instruction at each clock cycle if possible.

This process of *instruction prefetching* increases the overall performance of the CPU, but induces *pipeline hazards*, which may reduce the speedup. Generally speaking a hazard in computer architecture refers to a situation in which an unwanted event called *pipeline conflict* could happen. If encountered, such conflicts interrupt the clocked instruction flow through the pipeline. There are three types of hazards: *Data Hazards*, *Structural Hazards*, and *Control Flow Hazards*. For this thesis the latter type of hazards is of particular interest, since it is impeding classical Quicksort implementations by causing delays in loading correct instructions into the pipeline.

Control Flow Hazards occur if there are conditional jumps in the fetched instruction block. Those jumps lead to skips of several instructions when executed. Since instructions are prefetched before their actual evaluation, a *prefetcher* has to guess the outcome of conditional jumps. Wrong predicted and taken jump instructions generally lead to non-prefetched instructions and therefore invalidate the already loaded instruction set. As a consequence, those jumps cause a *pipeline stall*, which means filling the pipeline with NOPs (placeholder operations that cause the CPU to IDLE) until the target instructions are loaded. This induces high latency of up to three clock cycles for a 5 stage pipeline. Since modern CPU architectures implement far deeper pipelines, the caused delay is even more severe, resulting in approximately 10-30 clock cycles loss per mispredicted jump [33].

On average every seventh instruction in a modern program is a conditional jump that causes a pipeline stall if taken. Some constructs introducing conditional jumps into programs are `if`-, `for`-, and `while`-statements. In worst-case we can observe a stall of several clock cycles every seven instructions without a sophisticated strategy to handle instruction prefetching for those statements. The circuit that tries to solve this problem is called *branch predictor*. A branch predictor implements techniques to figure out whether a conditional jump (the *branch*) will be taken or not before its actual evaluation. This prediction process has to be done as early as possible in order to prefetch the correct instruction set into the instruction pipeline. Correct branch prediction results in a *branch hit* and wrong prediction in a *branch miss*. For example, the branch prediction for `for`- and `while`-statements can be determined *statically* and is usually evaluated to “branch taken”, since loops are very likely of being executed several times in a row. But the evaluation of `if`-statements taking is way more complicated and taking the correct guess requires *dynamic* strategies, since there are several factors influencing the chances of taking the jump. One such factor is the history of the branch: *"Was it taken before?"* and *"How often was it taken?"*. There exist several concrete approaches for predicting branches that take these questions into consideration, like One-level and Two-level branch predictors. Further details to the realization of those circuits are out of scope for this thesis and are explained in more detail in Ungerer’s book on microcontrollers [32]. If implemented in one of the referred ways, the resulting branch predictors are usually yielding very good results with an accuracy of over 90%.

Sadly, those numbers do not hold for all algorithms in equal shares. Quicksort in particular is an exception to this. In Section 3.3 we take a closer look on why this is the case. In such situations it often makes sense to completely avoid branches by providing a fixed order of instructions, which can be safely preloaded into the instruction pipeline. According to the Intel Optimization Reference Manual [17] this can be done by transforming conditional branches into one of the following constructs:

1. Casting the condition's bool result to `int`:

   ```
   int res = predicate(var1, var2);
   ```

   By applying this technique we obtain the result of the predicate in form of a 0 (false) or a 1 (true). We can now use these numbers e.g. to index an array or increment counters.

2. Using conditional moves (CMOVcc and FCMOVcc on Intel's x86 architecture [17]):

   ```
   var = predicate(var1, var2) ? trueValue : falseValue;
   ```

   This technique allows us to store a specific value in a variable, depending on the outcome of the predicate. The conditional move instructions have to be supported by the CPU though. Fortunately, most common CPUs implement this instruction set (e.g. Intel x86 and ARM ARMv8 [2]).

In fact, Elmasry and Katajainen showed that it is theoretically possible to transform every program to a semantically equivalent program with just a constant number of branch misses [11]. But introducing such branch-free code comes at a cost: The resulting code usually needs more instructions to implement the same logic. This means that enhancing a certain code snippet to be branch-free is a thin line between gain of performance by avoiding wrongly predicted branches and losing performance by adding additional instructions. In our thesis we make particular use of Technique 1. Code 3.1 contains an example for an unoptimized method containing branches, which can be converted to the second branch-free method by applying this technique. The semantics of those code segments and the actual use of the procedure are described more closely in Section 4.1.

3.2 In-place

An algorithm is considered to work in-place if the extra memory required for it is constant in its input space. This means that, aside from the input there is only $\mathcal{O}(1)$ additional memory needed. Usually this strict upper bound is relaxed to $\mathcal{O}(\log n)$, because very few algorithms fulfill the hard criterion: Even a counter for an input

Code 3.1: BF Check and Advance to Left With and Without Branches

```
void checkAndAdvanceToLeft(Set input, int rightIter, // With branches
        int[] toLeft, int toLeftCnt, Predicate pred) {
    if (pred(input[rightIter])): // If the predicate is fulfilled
        toLeft[toLeftCnt] = rightIter; // Store the element
        toLeftCnt++; // And increase the number of stored elements
    rightIter--; // Go to the next element
}

void checkAndAdvanceToLeft(Set input, int rightIter, // Without branches
        int[] toLeft, int toLeftCnt, Predicate pred) {
    bool res = pred(input[rightIter]); // Store predicate result in res
    toLeft[toLeftCnt] = rightIter; // Store element and possibly override
    toLeftCnt += res; // If res==0, we will override the stored
                  // element in the next iteration
    rightIter--; // Go to the next element
}
```

array requires $\mathcal{O}(\log n)$ extra space in bits [5]. For our thesis we also use the relaxed condition.

3.3 Quicksort and Partitioning

The original idea of Quicksort (see Code 3.2 and Code 3.3) by Tony Hoare [15] is the following: In a first step we choose some element at random from the input and use it as our pivot element. In the following *partition* procedure we swap all elements smaller or equal than the pivot to a position left of it and all greater elements to its right[1] (see Code 3.2). We recursively apply this algorithm on both partitions and the resulting subpartitions until there is just one or no element left per partition (see Code 3.3). At this point the input is sorted and the algorithm terminates.

Quicksort is considered as one of the most efficient sorting algorithms although its worst-case complexity is in $\mathcal{O}(n^2)$. This worst-case scenario is very unlikely, because it only occurs for very bad pivot choices, e.g. if the pivot equals the minimum or maximum element of the input set in each iteration, like it would be the case for inputs for which all elements are the same. Already for a reasonable high chance

[1]Assuming sorting from small to large elements

Code 3.2: Hoare Partition

```
 1 int partition(Set input, Predicate pred) {
 2     int leftIter = input.beginIndex();
 3     int rightIter = input.endIndex();
 4 
 5     while (leftIter<rightIter):
 6         while (!pred(input[leftIter])): leftIter++;
 7         while (pred(input[rightIter])): rightIter--;
 8         if (leftIter<rightIter):
 9             swap(input[leftIter], input[rightIter]);
10             leftIter++; rightIter--;
11 
12     return leftIter;
13 }
```

that the pivot is selected as an element of the middle 50% percentile Quicksort's best- and average-case complexities are in $\mathcal{O}(n \log n)$ [27]. In practice this and the fact that Quicksort requires very few instructions in its inner loop, are the reason for the overall very good performance of Quicksort. As already discussed in Section 3.2, Quicksort works in-place, which makes it even more attractive for sorting data in scenarios where hardware resources are limited, either by architectural constraints, e.g. found in the embedded devices sector or by the amount of data to be processed e.g. found in the sector of Big Data analysis.

As Code 3.2 and Code 3.3 suggest, the most time consuming part of Quicksort is the `partition` call. In this part we execute $\mathcal{O}(n \log n)$ comparisons and $\mathcal{O}(\frac{1}{3} lnn)$ swaps on average [27]. Compared to the recursive `sort`-call, which consists of few operations executed $\mathcal{O}(\log n)$ times on average, the partitioning phase takes most time of the algorithm. In practice the *glibc* Quicksort implementation `qsort` already spends over 50% of its time partitioning data when called for 10^3 elements. This rate increases even further, the more elements the algorithm is supposed to sort. Figure 3.1 shows some samples of this increase in more detail, whereby we want to note that `qsort` implements a median of three pivot selection strategy in this scenario. From this follows that optimizing the partition phase is crucial for a good performing Quicksort. For this purpose, it is important to understand that the loop which checks whether elements belong to the left or right, is the hotspot of the overall algorithm and the main source of branch misses. In our example lines 6 and 7 of Code 3.2 are of particular interest.

Code 3.3: Quicksort Recursion

```
void sort(Set input) {
    T pivot = choosePivot(input);
    // Large elements will get swapped to the right of the pivot
    int middle = partition(input, (T x){pivot<x});

    Set leftPartition = input.getSubset(begin, middle);
    Set rightPartition = input.getSubset(middle+1, end);
    if (leftPartition.size() > 1): sort(leftPartition);
    if (rightPartition.size() > 1): sort(rightPartition);
}
```

As already discussed in Section 3.1, the CPU tries to predict the outcome of those conditional jumps before their actual evaluation. In the case of Quicksort, this is the point at which modern branch predictors struggle guessing the correct outcome of conditional jumps (see Figure 3.2b). Predicting the correct branch behavior is particualy hard for Quicksort, because as we are aiming for a pivot as close as possible to the median of the input set, we also increase the number of branch misses the closer we get to this median, up to a misprediction rate of 50% [20]. We observe this behavior, because of the definition of the median: A median is the element that separates all lower elements from all higher elements, for which the number of lower elements is equal to the number of higher elements with a maximum deviation of one element for even input counts. This results in a 50/50-chance to predict whether a yet unevaluated comparison to the exact median pivot element will yield true or false, resulting in almost 0.5 branch misses per element, depending on the used branch prediction strategy [20]. During our profiling experiments, we also found that these expected values also practically hold for GCC's `partition` for which we observed approximately 0.49 branch misses per element. To reduce the number of branch misses and therefore the number of costly pipeline flushes, Kaligosi and Sanders introduced α-skewed pivot elements. Such skewed pivots are on purpose offset from the median by a factor of $\alpha \leq \frac{1}{2}$, whereby Kaligosi et. al. found that a skew factor of $\frac{1}{10}$ performs particularly well, reducing the overall branch misses for the exact pivot and $e^{26} \approx 19.57 \cdot 10^{10}$ elements by over 30%, resulting in a speedup of approximately 11,8% (also see Figure 3.2). This finding is particularly interesting in theory, because it explains why not so accurate, but fast pivot selection strategies like median of three often perform better in Quicksort than strategies that yield higher quality pivots at the cost of being more time consuming.

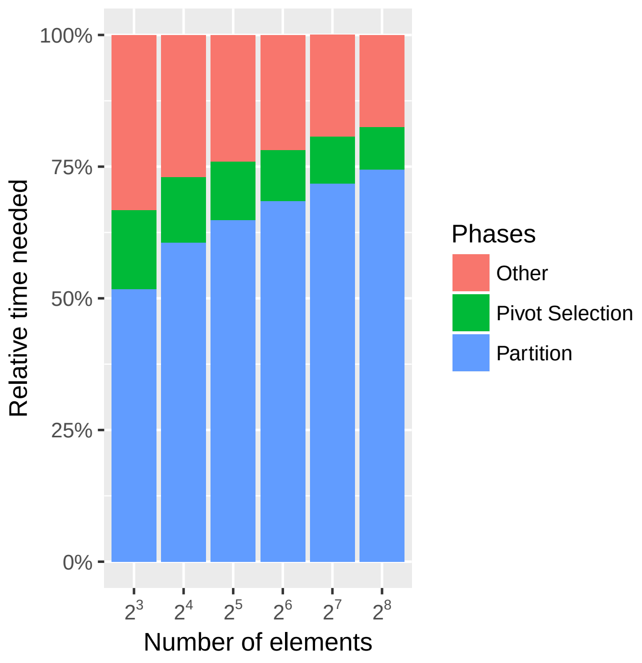
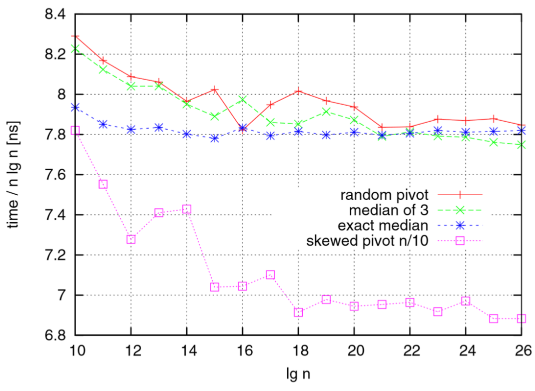
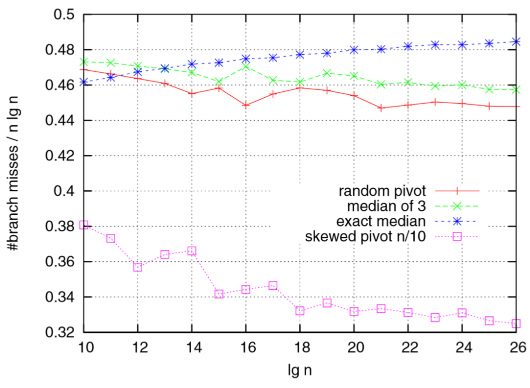


Figure 3.1: Relative time distribution of `qsort`'s Quicksort phases

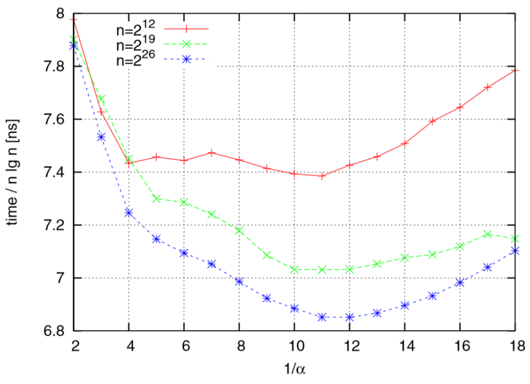
All of the presented findings demonstrate how important a careful pivot selection is. In an ideal Quicksort run we would always select the median of the input set as our pivot for each of its iterations. Sadly, this pivot selection strategy is impracticable, because it is very costly ($\in \Theta(n)$) to determine the exact pivot. Because of this Hoare suggested in his original paper on Quicksort to use a random sample of the input as pivot [15]. This strategy soon turned out to be suboptimal, because of its bad expected behavior [27]. This is why Sedgewick introduced a better approximation for the real median to Quicksort: Selecting the pivot as median of a random sample of three elements. This median of three approach reduces the number of expected comparisons for Quicksort from $1.386 n log n$ to $1.188 n log n$ [27], but comes at the expense of increasing the expected number of swaps by 3% [8]. After that, more sophisticated pivot selection strategies came up, such as the *Ninther* [31], proposed to be used in Quicksort as pivot selection strategy by Bentley et. al. [8]. In this approach we select the pivot as the median of three medians of three, which provides an even better approximation of the real median. Since such better approximations are more time consuming to calculate, Bentley et. al. proposed to use better and more expensive pivot selection strategies only if the number of elements in the current iteration exceeds a certain threshold. In their paper they used median of three for less than 40 elements and the *Ninther* for



(a) Execution time of Quicksort using random, median-of-three, exact median and $\frac{1}{10}$-skewed pivots



(b) Number of branch misses in Quicksort using random, median-of-three, exact median and $\frac{1}{10}$-skewed pivots



(c) Comparison of the impact of different pivot skew factors

Figure 3.2: α-skewed pivots and their influence on the performance of Quicksort found by Kaligosi et. al. [20]

larger inputs. As already discussed, these improvements in selecting a better pivot yielded only minor performance gains, because of the effects described by Kaligosi and Sanders. For our branch-free Quicksort implementation we expect that better pivot samples will lead to better results.

4 BF-Quicksort

In this chapter we present our solutions and main findings for the hardware induced problems of Quicksort when selecting pivots close to the median of the input set. In the previous chapter we identified hardly predictable branches as the main problem, which we solve by entirely avoiding branches in the critical section of Quicksort. Furthermore, we optimize Quicksort by implementing some of the optimizations proposed by Sedgewick [25], Bentley and McIlroy [8], and Edelkamp and Weiss [10]. After that, we take BF-Quicksort to the next level by implementing parallel execution for the algorithm. We split our explanations into two parts: The first part is about our sequential algorithm and the second part about parallel BF-Quicksort. All optimizations we introduce in the sequential part also apply to our parallel implementation.

4.1 Sequential BF-Quicksort

4.1.1 The Algorithm

The sequential implementation of BF-Quicksort consists of the same phases descibed in Section 3.3: At the beginning we select a suitable pivot element, next we start our branch-free partitioning procedure, and finally we call the algorithm recursively on both partitions. The main difference to classical Quicksort and one of our main contributions is our implementation of the partitioning procedure, which consists of three sub-phases: *Filling the Swap Buffers*, *Neutralizing the Swap Buffers*, and *Rearranging the Remainder*. Code 4.1 is an implementation in pseudo-code with the respective phases annotated in comments and Figure 4.1 shows an exemplified execution of our partitioning algorithm *BF-Partition*. Before we actually call `partition` on the input, we implement one of Sedgewick's [25] proposed refinements: We swap the chosen pivot to the last available input index and call `partition` from *begin* to *end-1* where *begin* is the first and *end* is the last element of the input.

Filling the Swap Buffers: In the first phase we introduce two iterators `leftIter` starting at index 0 and `rightIter` starting at end-1, which we use to index our input set. BF-Quicksort avoids hard to predict branches, by separating this first phase from the *Neutralizing the Swap Buffers* phase. To realize this, we introduce two swap buffers: `toLeft` and `toRight`, whereby each of these buffers stores up to `blockSize` elements.

The purpose of the buffers is to hold the indexes of all elements on either side that shall be swapped to the respective other side. We start this phase by filling the `toLeft` buffer as we start comparing the element pointed to by `rightIter` to our pivot. We add this element to `toLeft` if it is smaller than the pivot and proceed to the left. We implement this behavior by the already presented code snippet Code 3.1. We repeat this procedure for `rightIter` as long as `toLeft` has remaining capacities and as long as we do cross `leftIter`. After that, we fill the other buffer in the same way until both buffers are full or until we have compared each element of the input to the pivot. The *Filling the Swap Buffers* phase is the most critical phase in our sequential and parallel implementations, because it contains the most frequently executed code segments of the whole Quicksort algorithm. A lightheaded implementation at this point would result in very high overhead and many branch misses. This is why we completely avoid branches by applying Technique 1 described in Section 3.1: We remove all hard to predict conditional branches from this section, by changing the first method in Code 3.1 to the latter. We also apply the same technique for our `checkAndAdvanceToRight` call.

Neutralizing the Swap Buffers: Similar to the previous phase, this phase is part of the main partitioning loop. At this point we empty both buffers by swapping all elements they contain. We do this for the leftmost and rightmost elements of the input first, since those elements are most likely in the wrong partition. At this stage of our algorithm, the indexes of those outermost elements are located at positions 0 of both buffers. We repeat this main loop of filling the buffers and swapping their contents until there are no more elements left to compare. It is worth noticing that this phase empties both buffers completely in all iterations. An exception to this is the last iteration, in which at most one buffer with at most `bufferSize` unswapped elements can remain. If both buffers are empty, the algorithm terminates. Otherwise it enters phase three to rearrange the remaining elements.

Our approach differs from the proposed method described by Edelkamp et. al. in these first two phases. In contrast to our algorithm they fix the amount of elements they process before entering the swap phase. This leads to the fact that their buffers are not full most of the time they start swapping them and can contain carry elements for the next filling iteration. In worst-case for a perfect pivot their buffers are only half full in each neutralization phase. This is why our proposed approach only enters the *Neutralizing the Swap Buffers* phase, as soon as both buffers are completely filled. Therefore, we induce less overhead in managing the buffers and less switching between the first two phases. We also found that our approach practically yields better results for large input counts as described in Subsection 6.1.5.

Rearranging the Remainder: The invariant for this optional phase is that exactly one of our swap buffers is empty. In this phase we rearrange the remaining elements of this *unneutralized* buffer to their final destinations. In order to realize this we have to know the separating index of the two future partitions. For this purpose we define this position as the index of the leftmost element in the right partition. In order to find this position, we introduce a variable `lCnt`, which counts the number of elements that will be in the left partition. In fact, we added this variable already in the first phases of the algorithm to count all elements skipped by the left iterator and all elements belonging to the left partition. We can do this very cheaply, by using the integer results of the comparisons between the pivot and the right iterator and by doing some simple arithmetics for the other iterator. Since we have already compared each element to the pivot at this stage, we can guarantee that `lCnt` is the actual index for the partition separating element. From there on we use this variable as the starting point for our last phase, in which we start iterating to the side the *unneutralized* buffer indicates: Left if `toLeft` is not empty and vice versa. As soon as we encounter an element that should not belong to the respective partition, we swap it with one of the misplaced elements in our buffer. To enable correct positioning, we swap central elements found in this phase with the outermost elements contained by the buffer first. As soon as the buffer is empty or the index of the element in the buffer is smaller/larger (respectively to the *unneutralized* buffer: `toLeft/toRight`) than the index indicated by `lCnt`, the algorithm terminates returning `lCnt`. This phase induces at most a constant amount additional swaps $\in \mathcal{O}(blockSize)$ and at most a constant amount of additional compare instructions $\in \mathcal{O}(2 \cdot blockSize)$. We need this additional phase though, because the index at which the two iterators pass each other is not necessarily the actual separating index, as it would be the case for the naive Hoare partitioning algorithm. This can lead to elements that get into the buffers although they are already correctly positioned. An easy example for this is a very bad selected pivot, for which the right partition would only contain two elements, with one being selected as the pivot and one at the position end-1. In this example the iterators would cross each other at the index (end-1)-(`blockSize`-1), whereas the index separating the partitions would be end-2 (the initially removed pivot is at index end). At this point the `toLeft` buffer contains one element, which is already at its correct location and the `toRight` buffer is empty. From here we can see that our last phase will immediately terminate, because the index stored at `toLeft[0]` is larger than `lCnt`.

At the end of the partitioning phase we swap the pivot element to the partition separating position located at `input[lCnt]`. This results in a fully partitioned set, in which the pivot is already at its sorted position. From this point we start the algorithm over for both partitions and recursively sort the input this way, as we already explained for the classical Quicksort algorithm in Section 3.3.

Code 4.1: BF Sequential Partition

```
int partition(Set input, Predicate pred) {
    // Initialization
    int blockSize = CACHE_SIZE/(8*sizeOf(Set.type))<512 ?
        CACHE_SIZE/(8*sizeOf(Set.type)) :
        512;
    int leftIter = input.beginIndex();
    int rightIter = input.endIndex();
    T[] toLeft = [], toRight = []; // Our swap buffers
    int toLeftCnt = 0, toRightCnt = 0; // Fillfactor
    int lCnt = 0; // Used in to determine the splitting point
    // Main Loop
    while (leftIter<=rightIter):
        toLeftCnt = 0, toRightCnt = 0;

        // Phase 1: Filling the Swap-Buffers
        while (toLeftCnt<blockSize&&leftIter<=rightIter):
            checkAndAdvanceToLeft(rightIter, pred, toLeft, toLeftCnt);
        lCnt += toLeftCnt;
        int leftStart = leftIter;
        while (toRightCnt<blockSize&&leftIter<=rightIter):
            checkAndAdvanceToRight(leftIter, pred, toRight, toRightCnt);
        lCnt += (leftIter-leftStart)-toRightCnt;

        // Phase 2: Swapping
        int minCnt = minimum(toLeftCnt, toRightCnt);
        if (minCnt>0):
            for (int off=0; off<minc; off++):
                swap(toLeft[off], toRight[off]);

    // Phase 3: Rearranging of the Remainder
    rearrangeRemainder(input, toRight, toLeft, lCnt, pred);
    return lCnt;
}
```

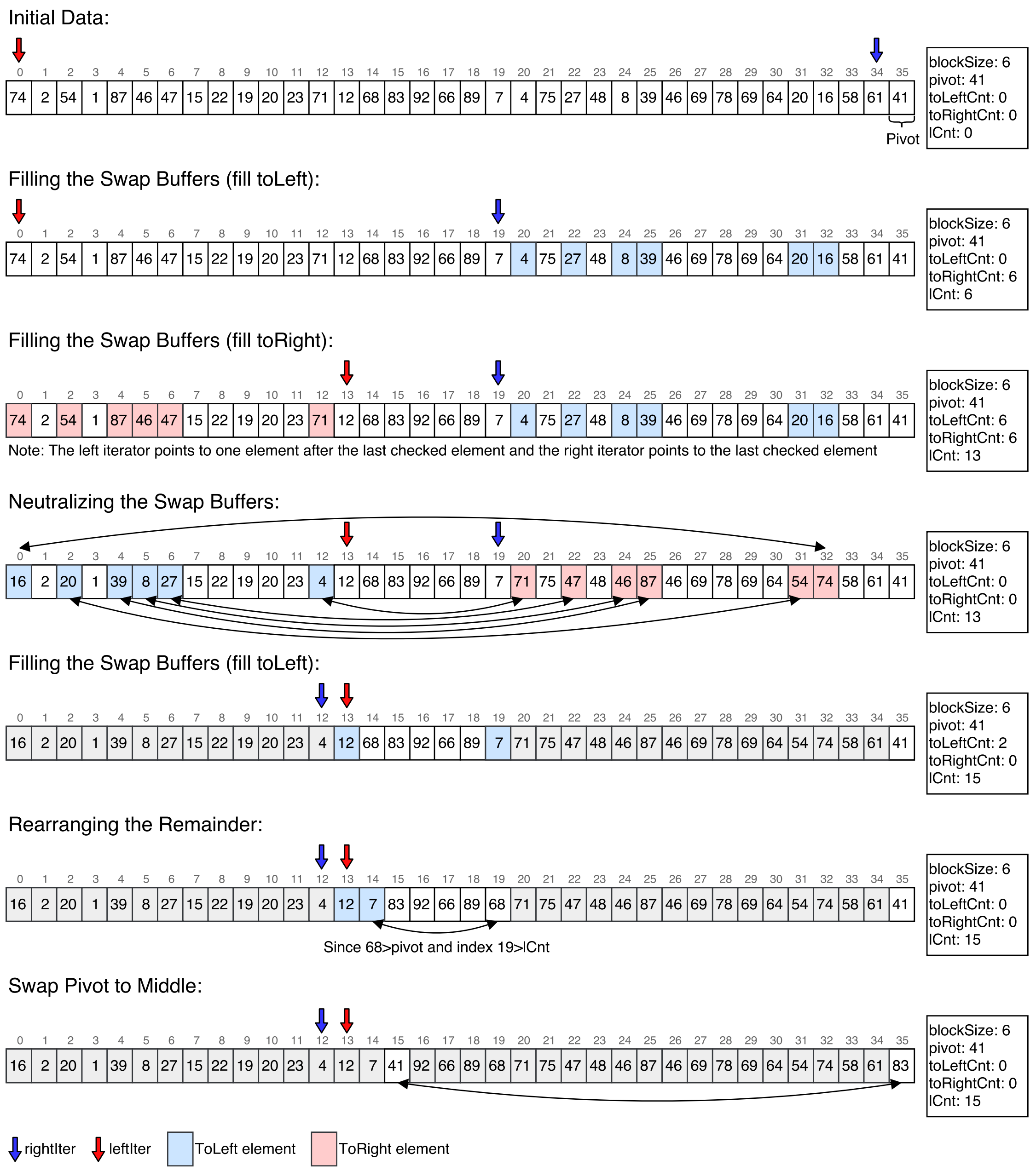


Figure 4.1: Step-by-step graphical representation of our partition procedure

4.1.2 Further Tuning of the Algorithm

As described by Sedgewick [25] it makes sense to switch to *Insertionsort* for small input counts. We also apply this technique to BF-Quicksort and switch the algorithm for less than 20 remaining elements. This improvement comes at the cost of one additional branch miss per element. We found that this improvement makes sense though, because it brought more performance gain than loss.

Because of the overhead of creating and managing buffers and avoiding branches, BF-Partition requires a certain input size to work efficiently. This is why we swap our algorithm with the branch-free Hoare partitioner proposed by Edelkamp et. al. [10], which performs better for small input counts < 1024 elements as we can see from the results of our the experiments in Subsection 6.1.2. We chose this particular algorithm, because it works still branch-free and enables us to also maintain this property for our Quicksort algorithm.

Furthermore, we also apply loop unrolling for the *Filling the Swap Buffers* phase, as suggested by Edelkamp et. al.. This simple improvement reduces the amount of compare instructions we need for the inner `while`-loops. We found an unroll size of 16 to yield the best results on our system.

In our next optimization we transform loops in which we swap two sets of elements with each other into cyclic permutations, as described by Abhyankar and Ingle [1]:

Code 4.2: Swap Loop

```
void swapWithSwapLoop(Set toRight, Set toLeft, int n):
    for (int i=0; i<n; i++):
        swap(input[toRight[i]], input[toLeft[i]]);

void swapWithCyclicPermutations(Set toRight, Set toLeft, int n):
    T temp = input[toRight[0]];
    input[toRight[0]] = input[toLeft[0]];
    for (int i=1; i<n; i++):
        input[toLeft[i-1]] = input[toRight[i]];
        input[toRight[i]] = input[toLeft[i]];
    input[toLeft[i-1]] = temp;
```

This transformation does not guarantee that an element located at `toLeft[0]` is swapped with an element in `toRight[0]`. But this is not problematic in our case, because this transformation still guarantees that small indexes will be swapped before high indexes, as long as they also would have been swapped in the untransformed loop.

BF-Quicksort overall performs well for random inputs. But there also exist several other input permutations, for which it is worth detecting the input pattern and choosing a sorting strategy accordingly [22]. As a proof of concept we included a pattern detector in our algorithm that detects whether all or some inputs of a given partition are the same. This is a very fundamental improvement for Quicksort, because we avoid one possible worst-case of the algorithm, which happens for inputs solely consisting of the same elements (we refer to those inputs as *all equal* inputs).

For our algorithm we use two different approaches to that problem: The first one was proposed by Edelkamp et. al., which we will refer as *duplicate check* (dc) and the second approach is developed by us, which we refer to as *all equal check* (aec). The latter approach is a more coarse grained attempt to solve the given problem. In our implementation we execute those pattern detectors on subset level in every Quicksort iteration, right after we finished partitioning the input. This allows us to also detect inputs with many equal elements even on deeper recursion levels, for which it is also more probable to encounter those partitions. It is worth noticing that we only use those pattern detectors in our Quicksort implementations, whereas we do not make use of them in the standalone partitioning algorithm although they may require to adapt the partitioning procedure specifically for Quicksort.

The duplicate check approach as proposed in the implementation of [10] works as follows: After finishing the partitioning phase we start scanning all elements to the right of `lCnt` for elements equal to the pivot. We do this as long as the amount of identified elements equal to the pivot (which is equal to one at initialization) is at least twice as big as the number of elements we overall compare to the pivot in this phase. This means that we compare two additional elements for equality to the pivot for each element equal to the pivot that we already found. As soon as we encounter such an element we swap it to index `lCnt`$+\#ofFoundPivots + 1$. This way we generate a third partition only containing elements equal to the pivot that we can skip in our next iteration of Quicksort.

We found that this approach may be very time consuming for many equal elements in the input or just few duplicate elements. It is very tempting to implement more sophisticated detection algorithms to better handle this kind of problem, but those approaches usually are more time consuming than coarse approaches. This is why we try to achieve a decent all equal detection mechanism at minimum cost. For this we assume that whenever the input contains many equal elements we also select the recurring element as our pivot with high probability. From here we only have to compare the sizes of the partitioned input's subsets to zero and the size of the input. If one of those comparisons yields true we know that it is very likely that we have a partition containing all equal elements. At this point we simply check whether this is the case, by iterating over the non-empty partition as long as we can find that our conjecture is true. In the case that the respective partition exclusively contains equal elements, we can safely stop further recursions to that partition and immediately return.

4.2 Parallel BF-Quicksort

Our parallel implementation of BF-Quicksort is based on the paper from Tsigas & Zhang [30]. Therefore, we introduce the same 3+1 parallel phase model for our algorithm: *Parallel Partition of the Data*, *Sequential Partition of the Data*, *Process Partition*, and *Sequential Sorting with Helping*. The first three phases are executed recursively, whereas the last phase is a dynamically load balanced call to sequential BF-Quicksort. Since this procedure introduces a different kind of blocks, we will further refer to blocks as the elements contained by the sequential algorithm's swap buffers and introduce the term chunks, which refers to the elements assigned to a thread during the *Parallel Partition of the Data* phase.

Parallel Partition of the Data: In the first phase of the algorithm we initialize p threads: One thread for each of the p logical processors. Next one of these threads picks a pivot, swaps it to the last index, and splits the input set into k equally sized chunks. If the input is not divisible by k, all remaining elements are grouped into a separate and smaller chunk. Next, all threads start working in parallel and fetch two unprocessed (*charged*) chunks from both ends of the input so that each thread owns one left chunk and one right chunk. After that all threads perform a modification of our sequential partitioning algorithm on these two chunks: As long as both iterators do not reach the end of one of the chunks, everything behaves analogously to our sequential scenario. As soon as one or both iterators reach the end of their respective chunk, the contents of both buffers are swapped until at least one buffer is empty. We call the chunk only containing correctly positioned elements *neutralized*. After that, we substitute each neutralized chunk by fetching a new chunk from the respective end of the input and restart this procedure by filling the swap buffers. The process of getting a new chunk is guarded by a mutex, whereby the critical section consists of at most two compare instructions, three assignments, one addition, and two subtractions. This process is repeated in all threads until there are at most p not neutralized chunks in the input and therefore no more chunks to fetch. In the last step of this phase we store the start and end indexes of the remaining $\mathcal{O}(p)$ charged chunks for each thread. At this point we have neutralized LN left and RN right chunks.

Sequential Partition of the Data: In this phase we process all remaining charged chunks. For this we swap them as close as possible to the future splitting element in order to get one coherent charged section in the center of the input. As opposed to Tsigas et. al., we leave one step here: The neutralization of the remaining chunks. Our measurements indicate better performance for leaving this step. At this point all elements from index 0 to (*chunkSize* $\cdot$ *LN*) and from index (*chunkSize* $\cdot$ *RN*) to *end-1* are partitioned correctly. We then process the remaining elements with our in Section 4.1 introduced sequential partitioning procedure.

Process Partition: At the beginning of this phase we swap back the previously removed pivot from the last index to the splitting point. From there on we have fully partitioned the input and apply phase one to three recursively on both partitions. For a better load balancing we do this by assigning a number of threads proportional to the size of the subproblem to each subroutine. If a call to the recursive algorithm would receive one or fewer threads, we switch to the last parallel phase.

In order to achieve the behavior described in the above three steps and to balance work loads among threads we introduce a thread pool. Furthermore, this data structure avoids the overhead connected with creating and destroying threads in the first recursion levels. We use this datastructure to hold all threads in a continuous ready state, as long as there is not enough parallel work to be done. As soon as another thread adds a task to the work stack of the thread pool one of the waiting threads wakes up and starts processing it. This work stack also stores tasks as long as all threads are working. If a threat would then run out of work, it fetches the next task from the stack and continues processing.

Sequential Sorting with Helping: In this phase we call an adapted sequential implementation of BF-Quicksort for the given subset. In this routine the respective thread works its way down the recursion tree by recursively calling it self for one of both partitions. For the other partition it adds the sorting task to the global work stack of our thread pool. As soon as all threads finish the global workload, our algorithm terminates.

5 Theoretical Analysis

Before experimentally evaluating our algorithm we provide some insights in BF-Quicksort's proposed attributes of being branch-free and in-place. For this we provide rough theoretical estimations on the upper bounds for possible branch misses and memory requirements.

5.1 Branch-Free

In this section we prove that our algorithm avoids branches and is branch-free to a degree of $\mathcal{O}(nlogn)$ on average, when choosing a pivot selection strategy of median of three or better. In the first theorem we provide the upper bound on branch-misses for our sequential algorithm in general. The proof follows a similar pattern as the one presented in [10]. After that, we extend our first theorem to the parallel case. In our proofs we refer to pseudo code snippets we provided in previous sections.

Theorem 1. *Let C be the average number of comparisons of Quicksort with a pivot sample size of one. Then, BF-Quicksort with blocksize B and an arbitrary but fixed pivot selection strategy induces at most*

$$\frac{2}{B} \cdot C + \mathcal{O}(n)$$

branch mispredictions on average.

Proof. Part 1: To proof our theorem we first show that our partitioning procedure induces at most $\frac{2}{B}$ branch misses *per comparison to the pivot*. In its main procedure (Code 4.1) BF-Partition contains four loops located in Lines 12, 16, 20, 26, and one method containing a loop in Line 31. Furthermore, we have an implicit `if`-statement in the `minimum`-call in Line 25. The loop in Line 12 causes only a constant amount of branch misses, because modern branch predictors assume that the loop condition holds all the time [32] and therefore only induces one branch miss as soon as we exit the loop. Lines 16, 20, and 27 follow the same principle: Each of those loops causes one branch miss every time we exit them. Additionally we have at most one branch miss per main loop iteration for the `minimum`-call. In each iteration of the main loop we compare at least $2B$ elements to the pivot. Because each such loop induces at maximum 4 branch misses, we can follow that there are at maximum $\frac{4}{2B} = \frac{2}{B}$ branch misses *per compared* *element to the pivot.* We can safely neglect the amount of branch misses induced by Line 31, since they are upper bounded by $\mathcal{O}(B) \in \mathcal{O}(1)$: The size of the remainder is at most B and with our remainder handling we never have to compare more than $2B \in \mathcal{O}(1)$ elements to empty the buffer: This worst-case scenario happens, whenever we enter the *Rearranging the Remainder* phase with a full swap buffer and all currently mispositioned elements are next to each other starting from the position where both iterators *leftIter* and *rightIter* met.

Part 2: We declare n as the number of elements used as input for Quicksort and m as the number of elements used in an arbitrary, but fixed iteration of Quicksort $\Rightarrow n = m$ for the first iteration. For selecting the pivot element we can expect to induce no branch misses, because of its branch-free implementation [10]. As we can see for Lines 8 and 9 in Code 3.3, we induce at most a constant amount of branch misses per Quicksort iteration. Also our additional check to when to switch to the Hoare partitioner introduces only one extra branch miss per iteration. As described in [10], Edelkamp and Weiss' branch-free Hoare partitioner also induces very few branch misses: $\frac{6}{H} \in \mathcal{O}(1)$, with H being the chosen block size for their particular partitioner, which is 128 in our case. In our implementation we additionally switch to Insertionsort for $m < 20$ elements. Each such call induces one branch miss per element. Since we can upper bound the number of calls to the Quicksort recursion to n, we can conclude that Quicksort introduces at most $\mathcal{O}(n)$ additional branch misses plus $\mathcal{O}(n)$ branch misses for Insertionsort. From this follows Theorem 1.

$\square$

Corollary 1. *From Theorem 1 and Sedgewick's observation that Quicksort with a pivot selection strategy of median of three or better has* $1.18 n log n + \mathcal{O}(n)$ *comparisons on average [27], we can conclude that BF-Quicksort induces less than*

$$\frac{3}{B} n log n + \mathcal{O}(n)$$

branch misses on average.

Theorem 2. *For the parallel case we can follow that we also have at most*

$$\frac{2}{B} \cdot C + \mathcal{O}(n)$$

branch misses. Let K be the size of our parallel chunks and p be the number of used processors, then more specifically the parallel phase induces

$$\lfloor \frac{n}{K} \rfloor - 2p + \mathcal{O}(n) \in \mathcal{O}(n)$$

additional branch misses.

Proof. Theorem 2 follows, because we are only inducing $\mathcal{O}(n)$ additional comparisons to our partitioning procedure due to the following changes: We introduce the *Parallel Partition of the Data* phase and the *Sequential Partition of the Data* phase. The latter phase obviously only induces the branch misses associated with the sequential phase plus at maximum kp misses for the loops that swap the remaining chunks to the center, and $2p$ misses for the conditional call to Insertionsort to sort the chunks relative to their distance to the remaining chunk. Where k is the number of chunks. Because $\mathcal{O}(2p + kp) \in \mathcal{O}(1)$, we can ignore those misses for our further analysis

The parallel phase of our partitioning algorithm consists of p sequential partitioning phases, in which we fetch $\lfloor \frac{n}{K} \rfloor - 2p$ times new chunks. This chunk fetching can induce at most this many branch misses, because every time a thread fails to acquire the lock to get a new chunk, another thread is acquiring a new chunk under the assumption that fetching a new chunk is faster than retying to acquire the lock (in fact this statement also holds if fetching a new chunk is only constantly slower than retrying to acquire the lock). From this follows that the total number of possible branch misses increases by $\lfloor \frac{n}{K} \rfloor - 2p \in \mathcal{O}(n)$.

In our parallel Quicksort recursion we only have one additional branch miss for the check when to switch to the sequential case and one branch miss per enqueue of one task into our thread pool. This also holds for our sequential phase with helping, which works similarly. All of these numbers can again be upper bounded to at maximum of n calls. From this we can conclude Theorem 2. $\square$

5.2 In-Place

Theorem 3. *BF-Quicksort works in-place only requiring*

$$\mathcal{O}((2p+1)K + 2Bp + logn)$$

additional space.

Proof. Our implementation of Quicksort differs from classical implementations in terms of memory in two points: We use swap buffers to swap elements in a separate phase and we use chunks to split work in the parallel partition phase. For all other operations we use containers of constant size. In the sequential part the size of our buffers can be considered constant, because it is calculated without taking the number of input elements into consideration (see Subsection 6.1.1 for our approach to this bound and an experimental showcase to alternative block sizes): $B = minimum\{\frac{L_1}{8T}, 512\} \in \mathcal{O}(1)$. Therefore, the additional space requirement for this stage is $\mathcal{O}(2B)$.

Also in our parallel case we need only $\mathcal{O}(2pK + K)$ additional space to hold the $2p$ chunks and swapping the remaining chunks to their final destination. Each such chunk has a size of $K = \frac{L_2}{T} \leq L_2 \in \mathcal{O}(1)$. Additionally each thread holds two buffers, which further increases the space requirements by $\mathcal{O}(2Bp)$

Apart from that points BF-Quicksort does not differ in terms of additional space requirements from classical Quicksort approaches. With the findings of [25] to the space requirements of Quicksort we follow Theorem 3. $\square$

6 Experimental Results

If not specified differently, we ran our experiments on an Intel Core i7-7900X Skylake CPU with 3.30 GHz (4.3 GHz with Turbo Boost), 10 physical cores with support for hyper-threading, and up to 20 logical cores. Each physical core has 32KB of L1 data cache and 1 MB of L2 cache. All cores share a 13.75 MB L3 cache and are connected to 126 GB RAM. The system runs a 64-bit Ubuntu Linux distribution (17.10). Our programs are written in C++ and compiled using GNU's gcc compiler (7.2.0) with flags: *-std=c++-14 -O3 -march=native*.

For performance measurements we read events from the Linux Kernel's perf interface using jevents. In addition to that, we utilize Intel's VTune Amplifier [18] to measure memory bandwidth utilization. We generated all random inputs with the 64 bit Mersenne Twister pseudo-random generator (std::mt19937_64), using a fixed but pseudo-randomly chosen set of seeds. If not stated otherwise we repeated each experiment at least 15 times, changing inputs for each repetition, because we found that some of the profiled algorithms are sensible to the concrete distribution of the input. We present the average of all of those runs.

For our experiments we use the following input types and notation:

- *uintX*: An X bit long unsigned integer. In our implementation we use the types: uint8_t, uint16_t, uint32_t, and uint64_t.
- *recordX*: A data structure of X Bytes, of which we use the first four Byte as an ID to compare records to each other. We choose X to be 16, 32, 64, and 128. In our implementation we defined the type as struct Record<X>, having an uint32_t `id` member and an uint8_t[$X - 4$] array of allocated, but uninitialized data as payload.

For our experiments it is important to notice that the input types uint8 and uint16 cause skew of the inputs, because of their limited range. We expect for those input types that our proposed all equal check brings a significant performance boost compared to non pattern detecting approaches. We annotate BF-Quicksort for experiments where we have activated this property with *aec* and with *dc* if implemented with duplicate check (see Subsection 4.1.2).

To provide a better overview we divided our tests into two categories: Sequential and Parallel. Since Quicksort heavily relies on an efficient partitioning procedure we execute all independent partitioning experiments before all Quicksort experiments, in order to determine the optimal configuration for BF-Quicksort.

6.1 Sequential

This section contains all experiments relating to our sequential Quicksort implementation. We executed all described experiments in the same order as we present them in this section. This is why all improvements found in earlier experiments are already part of our algorithm in subsequent experiments.

6.1.1 Block Sizes

In our first experiment we find a suitable choice for our swap buffers (`toLeft` and `toRight`), used in the partitioning phase of BF-Quicksort. For this we fix the input size to 2^{26} elements and vary the block sizes in a range of 2^5 to 2^{11}. We measure the partitioning phase for all presented input types. The term block size refers to the size of one of the buffers as introduced in Chapter 4. Generally speaking, we can make optimal use of the separate swap phase and therefore avoid unnecessary interrupts in processing the input, by increasing the buffer size. But as an opposing effect, we clutter the L1-Cache with our buffered data. As a consequence we have to find a balance of the size for our buffers. We do this opposed to Edelkamp and Weiss, which fix their buffer sizes with a global constant.

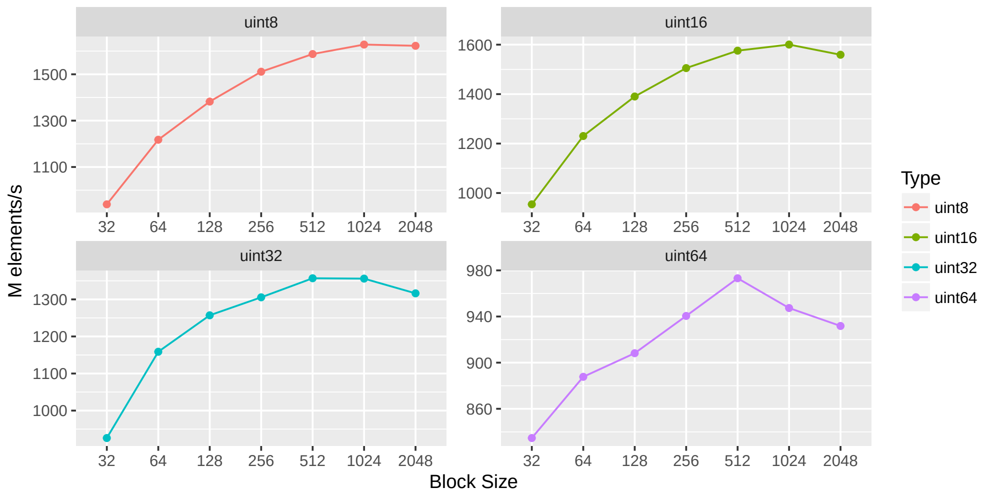


Figure 6.1: Comparison of various block sizes for BF-Quicksort's partitioning subroutine on integer inputs

From our experiment with integers (Figure 6.1) we can see that those small-sized inputs greatly benefit from large buffers as expected. The first point for which we can get the maximal performance for all those types lies around 512 elements per buffer. After that, the performance keeps its level for uint8 and uint16. For uint32, uint64, and bigger inputs we can observe a performance drop e.g. at 1024 elements for uint64. We can pinpoint the threshold for those drops to the size at which our buffers fulfill the following criterion:

$$\begin{aligned} 2B \cdot T \cdot E \geq L_1 \Leftrightarrow B &\geq \frac{L_1}{2T \cdot E}; \\ \text{where } B &= \text{Buffer Size;} \\ T &= \text{Size of the analyzed type in Bytes;} \\ E &= \text{Expected amount of elements skipped, until} \\ & \text{the next element gets into one buffer} + 1; \\ L_1 &= \text{Size of the L1-Cache in Bytes} \end{aligned}$$

The left side of this inequation represents the amount of elements we process before we enter the swapping phase. With its first part $2B \cdot T$ is representing the total amount of Bytes we need for our two buffers. Since we are not only processing elements going into our buffers, we have to also consider all other elements we iterate over. We do this by the factor E. This factor is important, because as soon as we process more than L1-Cache size elements, we may push our buffered elements out of the L1-Cache, which in turn causes L1-Cache-Misses as soon as we enter the swapping phase. For this experiment we can set $E = 2$, because we choose the exact median of the input set as pivot. The reason why we do not see this behavior for the two smaller uint types is that it gets overshadowed by the positive effects of bigger buffers. This happens, because loading small-sized elements is less costly than loading bigger elements: More missed elements fit into one load to be fetched from L2-Cache or slower memory. Accordingly this effect gets even worse for bigger sized inputs. Therefore, we repeated the experiment for 32 to 128 Byte records.

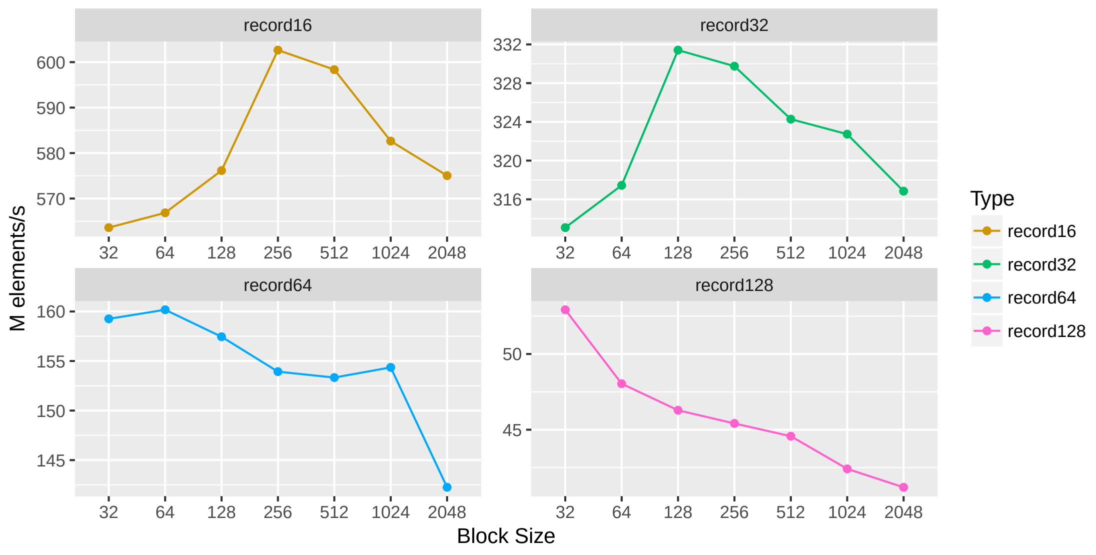


Figure 6.2: Comparison of various block sizes for BF-Quicksort's partitioning subroutine on record inputs

Our results show that the proposed threshold also applies to record input types. Moreover we observe an additional drop for those inputs as soon as buffer sizes using our proposed formula do come close L2-Cache size. Namely those drops occur after block size 1024 for record64 and after block size 512 for record128. Even though the drop from 1024 to 2048 for record64 seems to be steeper than the drop from 512 to 1024 for record 128, the relative drop is the same. For our future experiments we choose the buffer size according to our findings to be factor 2 below the proposed threshold, but at least 512 elements: $B = minimum\{\frac{L_1}{4T \cdot E}, 512\}$ with $E = 2$. For our purposes it is no problem to be optimistic about the factor E, because we always aim for an as good as possible pivot in BF-Quicksort. For example a median of three selection of the pivot already yields a probability of 68.75% [27] to get a pivot in the middle half percentile of the input, for which $E \leq 3$ already holds.

6.1.2 Partition Comparison

In this experiment we compare our proposed partitioning algorithm to other existing algorithms. For this we selected the following partitioners:

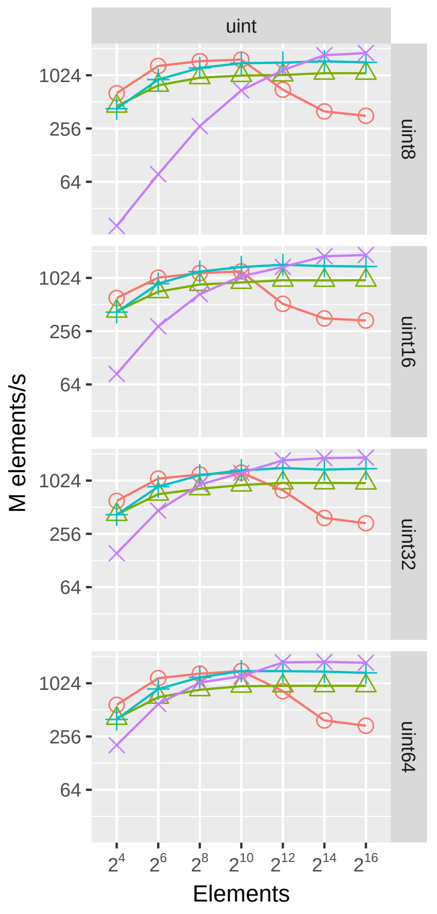
- *GCC's standard implementation of partition* (stdPar): There exists several implementations of partition, but this particular implementation is one of the most famous ones. It neither implements branch avoiding techniques nor makes use of swap buffers.

- *Branch-Free Lomuto Partition* (lomPar): This branch-free implementation of Lomuto partition was proposed by Katajainen in their paper on branch-free sorting [21]. In contrast to Edelkamp et. al.'s and our algorithm, branch-free Lomuto partition does not make any use of swap buffers and "simply" implements the proposed branch avoiding techniques on top of Lomuto's famous partitioning algorithm, named after and attributed to Nico Lomuto and published by Bentley in [7].
- *Blocked Hoare partition* (hoaPar): As already discussed in Section 2.2 *BlockQuicksort* and therefore its partitioner also make use of buffers to delay swapping in order to become branch-free. We consider this algorithm as our closest sequential in-place competitor.

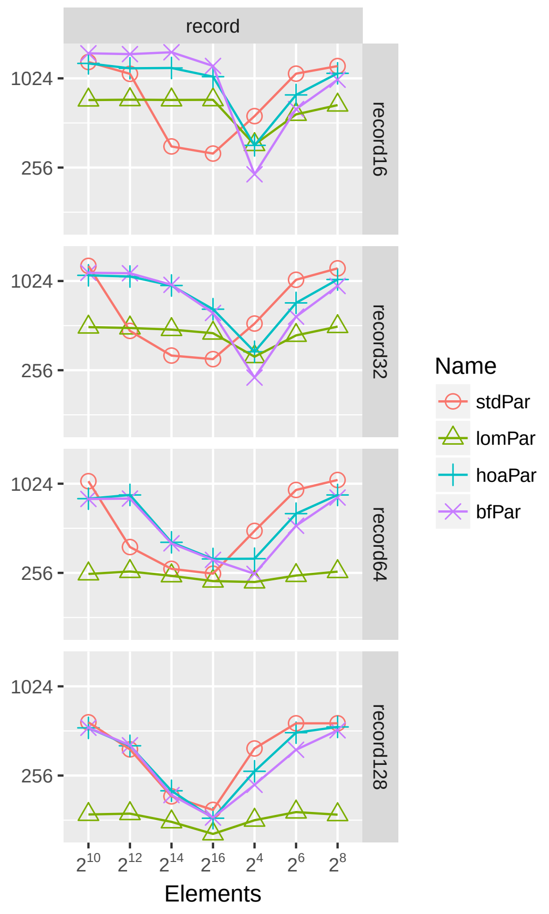
We split this experiment into two parts: The first part is about small to medium input counts between 2^4 and 2^{16} and the second part is about large input counts between 2^{18} and 2^{28}.

Small inputs

For our first experiment with small input sizes we repeatedly execute the partitioning algorithms on a randomly selected input set. We do this until the algorithms partitioned exactly 128MB data. After that, we summed up the times and repeated the experiment for different input sets. We present our results in million elements per second, for which we divided the accumulated time by the total number of elements processed in all runs, which is equal to $\frac{128MB}{T}$ where T denotes the size of the input type in Bytes.



(a) Comparison in million elements per second for uint inputs



(b) Comparison in million elements per second for record inputs

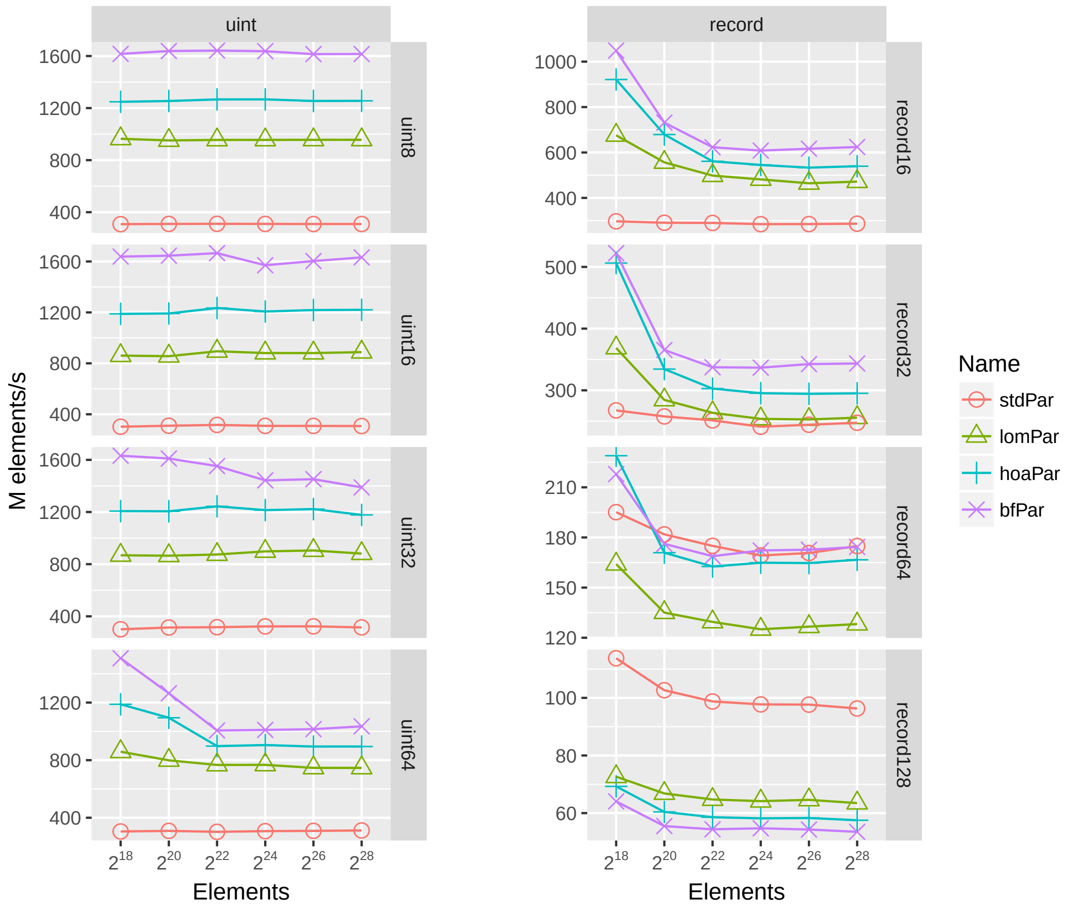
Figure 6.3: Comparison of different sequential partitioning algorithms for small input counts

From the results depicted in Figure 6.3, we can see that for small input counts all branch-free algorithms cause more overhead, than gain. Even the non-buffer using Lomuto approach does not perform well in this context. This is because of the already mentioned overhead in avoiding "simple" `if`-statements, causing more instructions to be executed (see Section 3.1). For such small input counts we can also observe the negative influence of dynamically calculating the buffer sizes with our improved algorithm. This effect is outweighed by its gain in the context of larger input types and increasing numbers of inputs.

Our both plots for different input types show that GCC's standard partitioner outperforms all selected branch-free approaches for fewer than 2^{10} elements and that hoarPar is the next better performing algorithm in this context. Both of these findings are very important for BF-Quicksort, because it heavily relies on good performing partitioners for small input counts in order to be efficient (see Section 3.3). This is why we select a fallback partitioner for few input elements. To maintain our proposed branch-free property, we choose the Hoare Partitioner of Edelkamp et. al. for this purpose. An alternative approach would be to choose stdPar, but in fact our experiments with the latter partitioner as fallback did not yield significant improvements. From here on we use this fallback in our *Quicksort* procedure, whereas our standalone partitioner remains unaffected from these findings.

Large inputs

In this part of our experiment we execute the proposed algorithms only once for each input and calculate the throughput for each partitioner measured this way.



(a) Partition comparison for uint inputs (b) Partition comparison for record inputs

Figure 6.4: Partition comparison for large input counts

From the plots in Figure 6.4 we see that we outperform GCC's standard partitioning procedure at least by a factor of 3.32 for integer inputs and still perform better for up to record64. We even beat our closest competitor hoaPar at least by a factor of 1.78 for uint8 and uint16 inputs. BF-Partition also performs better for all other input types less than record128. Interestingly, the performance of our algorithm does not converge to the performance of stdPar, instead it rapidly decreases for input types larger than record64. This is why we further analyzed the respective numbers:

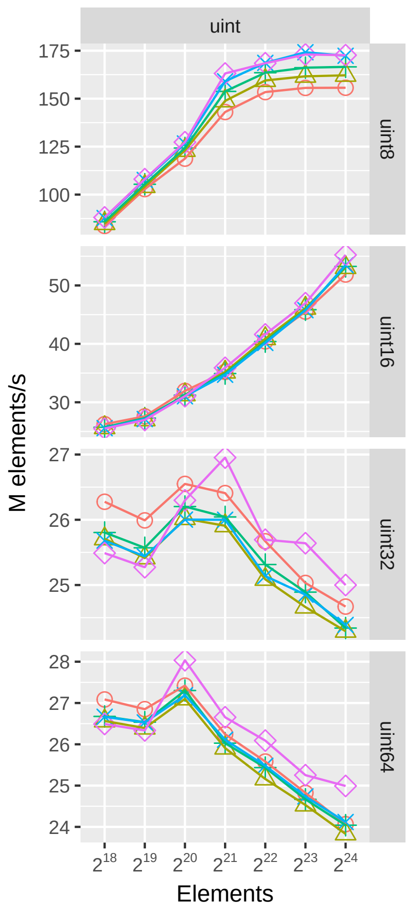
name	type	time[ms]	cycles	LLCmiss	L1miss	instruc	brmiss
bfPar	record16	108.85	7.06	0.25	0.26	9.12	0.00
stdPar	record16	235.76	15.24	0.25	0.25	7.25	0.50
bfPar	record32	195.97	13.15	0.50	0.51	10.43	0.01
stdPar	record32	275.15	18.05	0.50	0.50	8.50	0.50
bfPar	record64	388.67	25.36	1.00	1.01	13.07	0.01
stdPar	record64	393.24	25.40	1.00	1.00	10.99	0.49
bfPar	record128	1236.34	82.90	1.97	1.56	18.40	0.03
stdPar	record128	687.34	44.82	2.00	1.83	16.00	0.49

Table 6.1: Results of the sequential partitioning experiment for 2^{26} record inputs. All numbers except for time[ms] are divided by the amount of elements.

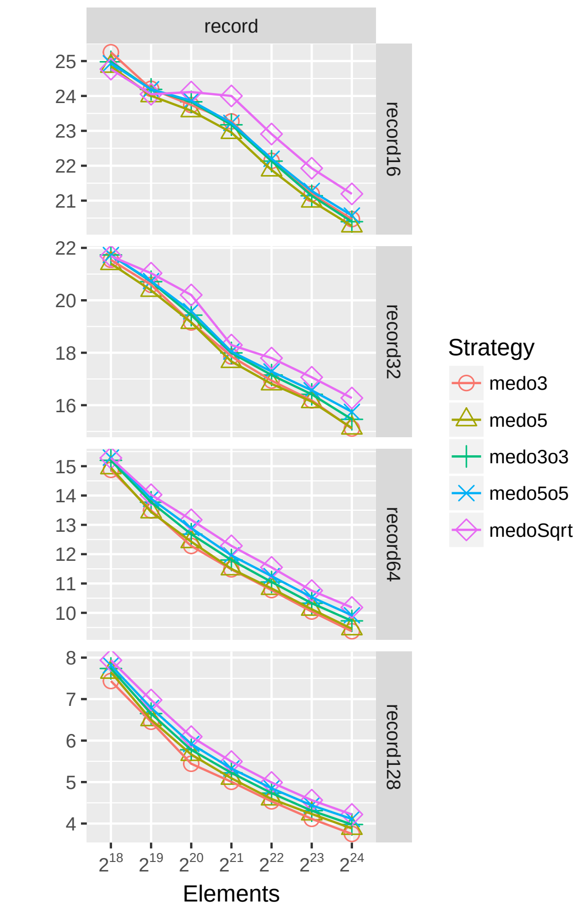
Unfortunately, those counters did not provide more general insight to the reason why we can observe this effect, because neither the instruction counter nor the amount of used cycles in conjunction with cache misses or branch misses explains this huge drop. Because of this we tried compiliing our code with another compiler to see whether the problem persists. For this purpose we selected *clang++* as an alternative using the proposed flags stated at the beginning of this chapter. The numbers we described in Table 6.1 did only change relatively for the code generated by this compiler. Because of time constraints we stopped at this point and leave this phenomenon up to further research. A possible approach to get down to the problem is to use Intel's Architecture Code Analyzer (IACA) [14] to statically examine the performance bottlenecks of hoaPar and our algorithm.

6.1.3 Pivot Selection

The appropriate pivot selection strategy is an integral part of Quicksort, because it influences its recursion depth. As already described in Section 3.3, we want to be as close to the exact median of the input as possible. Unfortunately, the more sophisticated pivot selection strategies are and the closer they come to the actual pivot, the more time it takes to compute them. This is why we compare different approaches in our algorithm to determine the optimal compromise between calculating the pivot and algorithmic gain in terms of running time. We expect a better running time for better choices of pivots as described more closely in Section 3.3. To test this, we compare random pivot selection to different approaches of median of *a* pivot selection to each other. For our purposes we define the median of *a* as the median between *a* deterministically at random selected elements and the median of b medians of a as the median of b elements selected by b median of a strategies. In our experiment we compare random pivot selection (rnd), median of 3 (medo3), median of 5 (medo5), median of 3 medians of 3 (medo3o3) (also called Ninther [31]), median of 5 medians of 5 (medo5o5), and median of $\sqrt{n}$ (medoSqrt) to each other. As already described more sophisticated strategies are costly and this is why they transitively fall back to less costly approaches for small inputs: medoSqrt falls back to medo5o5, medo5o5 falls back to medo3o3, and medo3o3 falls back to medo3. Furthermore, medo5 also falls back to medo3 for sufficiently small inputs. We gratefully use Edelkamp et. al.'s findings and implementation of some of those strategies. For our purposes we further enhance them to a better fit for BF-Quicksort. We also compared alternative approaches like median of $log_2(n)$, median of $log_e(n)$, and median of $log_{10}(n)$ which fall back to median of three for small input counts. We left their lines in this plot, to keep it readable and because these strategies did not perform better than medo5o5 or medoSqrt.



(a) Different pivot selection strategies for uint inputs



(b) Different pivot selection strategie for record inputs

Figure 6.5: Comparison of different pivot selection strategies for BF-Quicksort

As expected, our findings show that a more sophisticated selection of pivots facilitates branch-free Quicksort, as opposed to the initially mentioned results of Kaligosi and Sanders [20] on non branch-free Quicksort. All proposed pivot selection strategies are pretty close to each other, with random pivot selection as an exception. For all input distributions we can observe median of $\sqrt{n}$ and median of 5 medians of 5 to yield the best results. This is why we select median of $\sqrt{n}$ as the pivot selection strategy for BF-Quicksort. For all of our future experiments we use the term BF-Quicksort without further annotations for our algorithm that implements this median of $\sqrt{n}$ approach and uses our proposed all equal pattern detector.

6.1.4 Branch Misses

In this section we provide the experiment for Theorem 1 in Section 5.1. Furthermore, we compare our algorithm to other well known sorting algorithms to get the greater picture. For this we introduce GCC's standard sorting algorithm (stdqs), which is a variation of Quicksort, called Introsort. This type of algorithm falls back to Heapsort for great recursion depths and is often considered as the reference in-place algorithm to beat performancewise [4]. We also introduce our already mentioned closest in-place branch-free competitor BlockQuicksort (blqs) implemented by Edelkamp et. al., for which we already have discussed the main differences to sequential BF-Quicksort in Section 2.2.

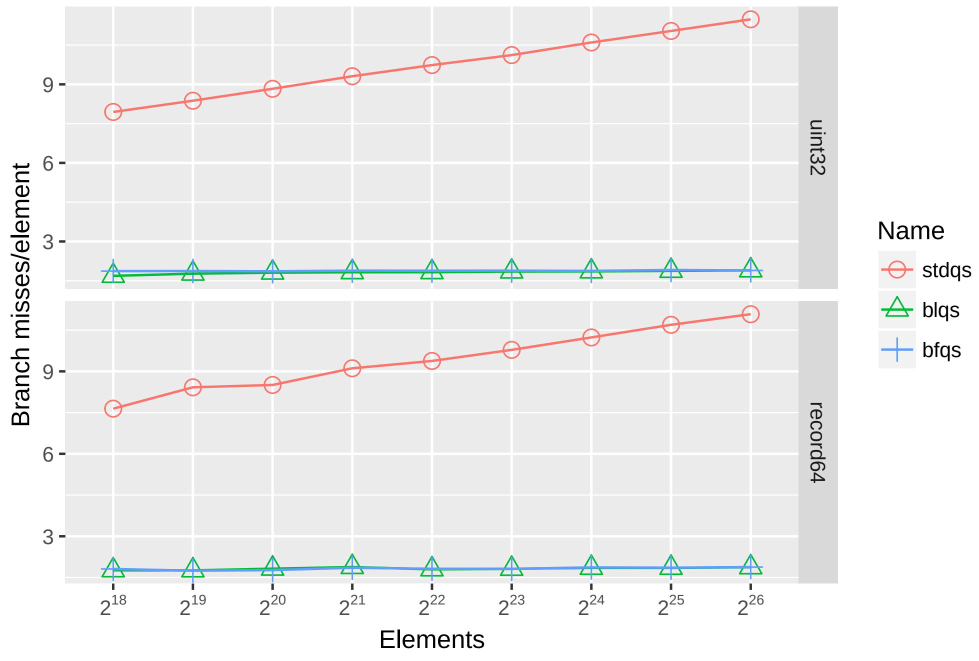


Figure 6.6: Comparison of different sorting algorithms and their branch misses

Since our experiments yield comparable results for all input types we present only an excerpt for uint32 and record64. From our results we can see that both branch-free algorithms only induce about 1.9 branch misses per element for all measured input counts, while stdqs's branch misses per element are linearly increasing with the input count. As already discussed, almost half of the branch misses in BF-Quicksort are induced by calls to Insertionsort. Although seemingly linear, the amount of branch misses for our algorithm is upper bounded logarithmically using a very small *nlogn* term as described in more detail in Section 5.1.

6.1.5 Quicksort Comparison

In our last experiments on sequential algorithms we set BF-Quicksort's overall performance in context to the in Subsection 6.1.4 already mentioned algorithms. Additionally we add the following algorithms to our comparison:

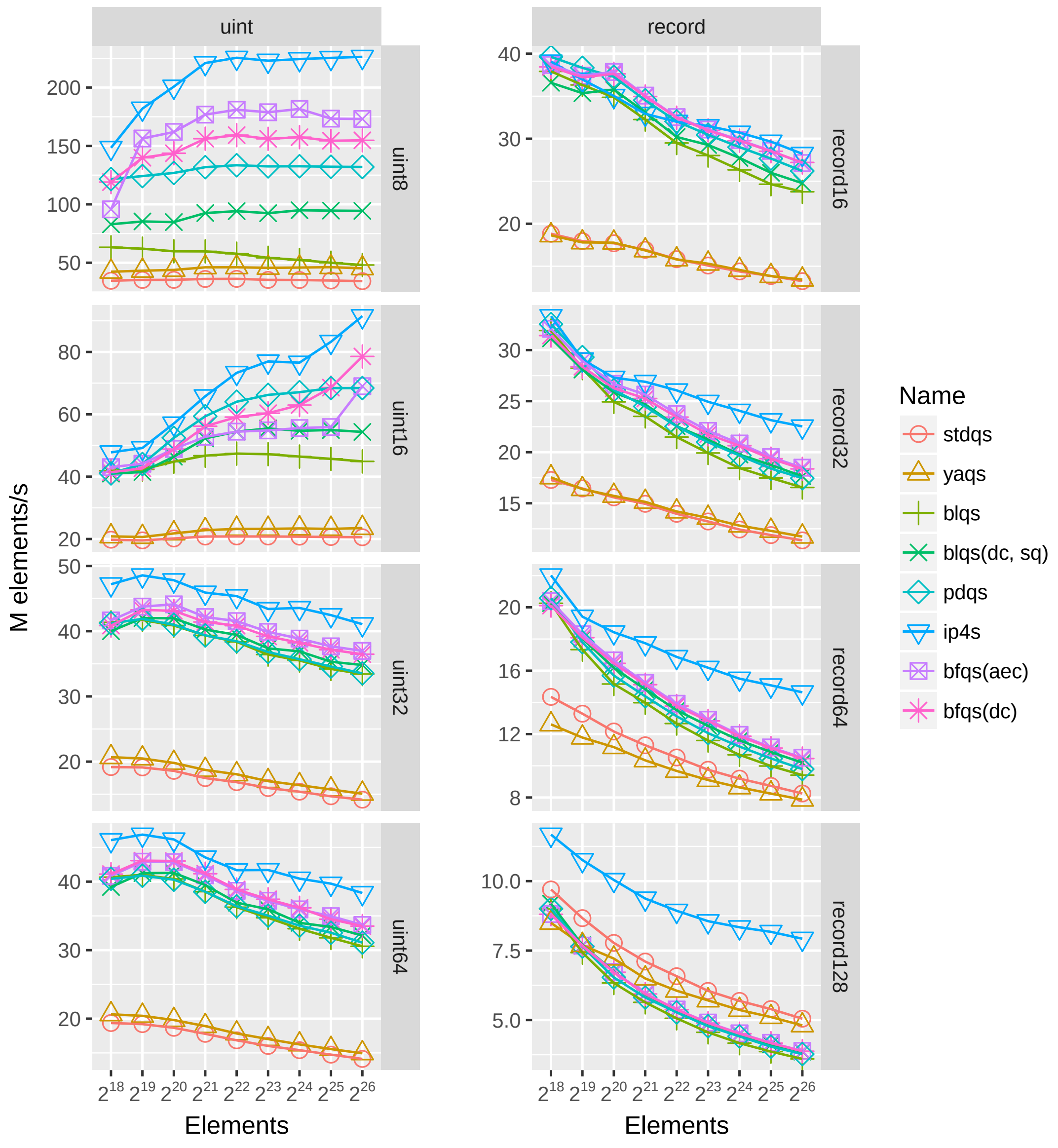
- *Blocked Quicksort with duplicate check and median of $\sqrt{n}$* (blqs(dc,sq)): For this comparison we further enhance the code provided by Edelkamp et. al., by switching their configuration to a more favorable than their standard configuration. We add a variant of their algorithm using their proposed duplicate check and a median of square root pivot selection strategy which is quite similar to the one we use in BF-Quicksort. Edelkamp et. al. also used this configuration in their paper on BlockQuicksort [10].

- *Yaroslavsky's Dual Pivot Quicksort* (yaqs): This algorithm proposed by Yaroslavsky is Java's standard sorting function for arrays since Java 7 [23]. It uses two pivots to determine three partitions in each iteration. This algorithm does not work in-place. For our measurements we use a C++ version of the algorithm converted by Edelkamp et. al.. This algorithm does not avoid branches.

- *Pattern Defeating Quicksort* (pdqs): Although to our knowledge there is no paper on this sorting algorithm implemented by Orson Peters [22] we found it during our research to be quite relevant: This algorithm is a modified version of the algorithm proposed by Edelkamp et. al. and detects patters in the input to improve its execution. Therefore, this algorithm is also branch-free and utilizes swap buffers for its partitioning procedure.

- *In-Place Super Scalar Samplesort* (ip4s): This algorithm proposed by Axtmann et. al. [4] is an evolution of the Super Scalar Samplesort proposed by Sanders and Winkel [24]. As the name suggests, this algorithm works in-place in contrast to its predecessor. In addition to that it does also avoid branches. These are the reasons for why we add this algorithm to our comparison although it is no classical Quicksort algorithm. For evaluating our results compared to theirs it is important to know that BlockQuicksort outperformed the predecessor of ip4s [10], whereas ip4s outperformed BlockQuicksort [4]. We found that this algorithm is one of the best freely available sequential in-place sorting algorithms to date.

- *BF-Quicksort with duplicate check*: As discussed in Subsection 4.1.2, we can use two different approaches for checking duplicates in BF-Quicksort. Our cheap all equal check and the more sophisticated duplicate check approach. For all of our future experiments we annotate our algorithm with *dc*, whenever it got executed with duplicate check instead of our all equal check and with *aec* if executed with enabled all equal check.



(a) Comparison for uint inputs (b) Comparison for record inputs

Figure 6.7: Comparison of different sequential sorting algorithms

Overall we can see that BF-Quicksort performs quite well for almost all inputs. Specifically compared to the highly tuned standard sort we achieve a speedup of up to 5.5 for one Byte inputs by utilizing our all equal check. Also for all other integer inputs our algorithm is approximately twice as fast as stdqs, which is a great success. Even more, we outperform our tuned closest in-place competitor blqs(dc, sq) for almost all input configurations by a few percent. This is why we can consider BF-Quicksort to be superior to BlockQuicksort for sufficiently large inputs and random input permutations of the given input types. We also see pdqs to perform better for medium uint16 input counts. We suspect that this is the case, because our all equal check does not take off for lower input counts and our duplicate check is not as efficient as the one implemented in pdqs. As soon as there are many large partitions with all equal elements, our algorithm performs starts performing way better. This is also the reason why we see pdqs performing better for uint16 inputs until the input count reaches 2^{26} elements.

The only algorithm that constantly outperforms BF-Quicksort is ip4s. We suspect that one reason for this is that avoiding branches especially makes sense in the context of Samplesort, which creates several partitions for the input. From this we conclude that it makes sense to further investigate in branch-free multi-pivot Quicksort implementations in future works.

6.2 Parallel

In this section we present all our tuning and comparison experiments for parallel BF-Quicksort. As described in Section 6.1 we also present our experiments in this section in the same order as we executed them. All improvements we proposed in an earlier experiment will be incorporated from there on for all subsequent experiments.

6.2.1 Chunk Size

Many factors influence the performance of our parallel partitioning algorithm. One significant factor is the appropriate selection of BF-Quicksort's chunk size. To find a suitable chunk size there are some points to consider:

- A big chunk size is good for BF-Quicksort's *Parallel Partition of the Data* phase, because threads do not have to fetch new chunks that often.
- A small chunk size is good for BF-Quicksort's *Sequential Partition of the Data* phase, because it determines the size of the sequential subproblem.
- Each core has its own L1 and L2 cache. To be efficient the maximum chosen chunk size should be smaller than the L2 cache, because the sequential buffers

already take up the whole L1 cache.

- Because modern computer architectures support hyper-threading, which means that several logical threads share a physical core, we can even further decrease that threshold to be about factor 2 below the size of the L2 cache.

Taken all these considerations into account, we see that there might exists a "sweet spot" for the chunk size, which is neither too big nor too small. Furthermore, we see that the chunk size has to be dependent on the amount of threads, executing our partitioning algorithm. Because of all of those reasons we came up with the following experiment to determine this spot:

We chose a size of $K_0 = \frac{L_2}{2Tp}$ as our guess for the appropriate chunk size. This formula fulfills all of the above mentioned criteria and is the center of the following plots, which we generated for the chunk sizes $\frac{1}{4}K_0$, $\frac{1}{2}K_0$, $\frac{3}{4}K_0$, K_0, $2K_0$, $4K_0$, and $8K_0$. We executed our experiment for one gigabyte of all introduced input types and for one to twenty threads.

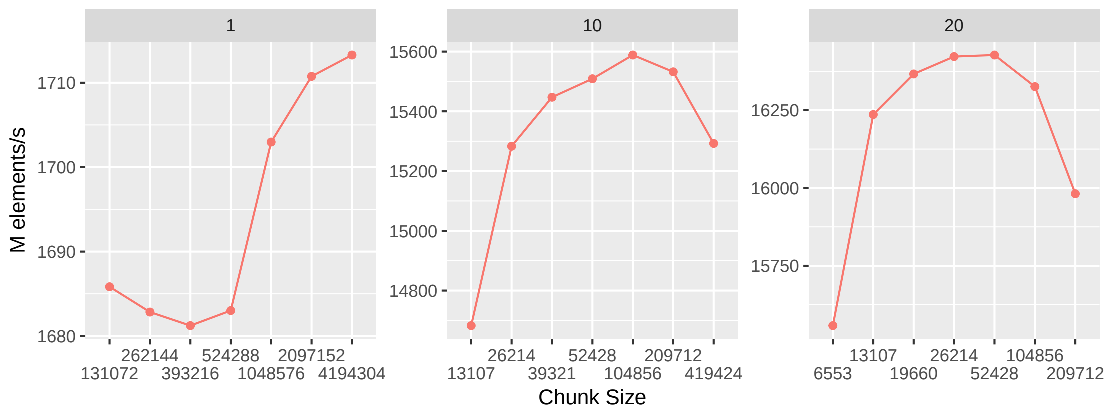


Figure 6.8: Chunk sizes experiment for 2^{30} elements of type uint8 and 1, 10, and 20 threads

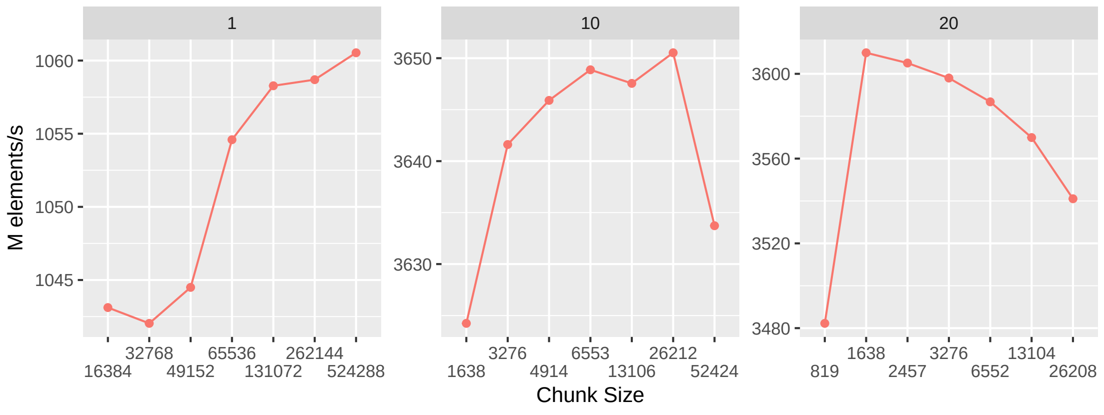


Figure 6.9: Chunk sizes experiment for 2^{27} elements of type uint64 and 1, 10, and 20 threads

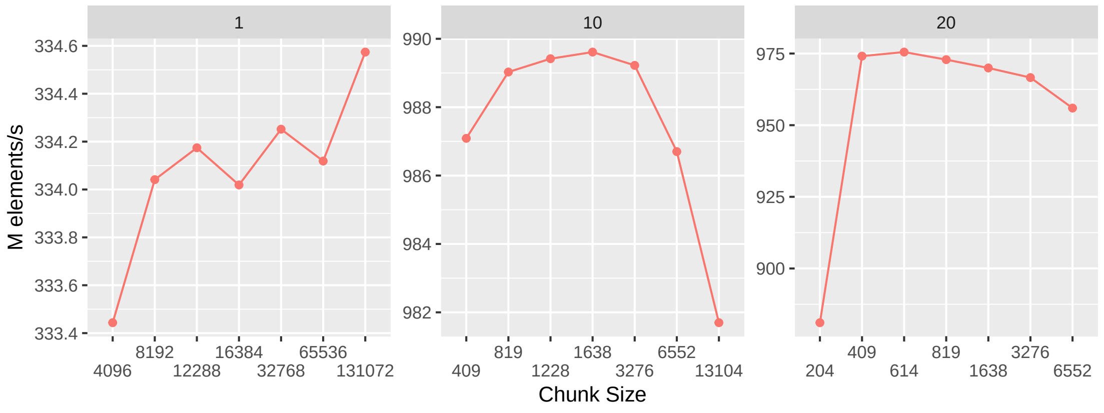


Figure 6.10: Chunk sizes experiment for 2^{25} elements of type record32 and 1, 10, and 20 threads

As the results of our experiment did not significantly differ from each other, we provide only a representative excerpt from our results. From these results we see that the afore mentioned "sweet spot" indeed is around $K = K_0$. For higher thread counts and bigger sized input types we observe that the curve already falls slightly but the absolute loss from deviating from our proposed threshold is very low. In contrast we can see that bigger chunk sizes facilitate single-threaded execution. This is surprising, because this single thread has to fetch new chunks less often the bigger the chunks get and therefore our single-threaded parallel algorithm performs comparable to our sequential algorithm when treating the input as one chunk. For our further experiments we set the chunk size to our proposed threshold of $K = \frac{L_2}{2Tp}$.

6.2.2 Speedup

For both our parallel algorithms it is important to provide a good speedup in order to stay competitive. For this we measure the gained speedup of the respective parallel algorithm by dividing its running time by the time needed to execute parallel BF-Quicksort utilizing only one thread. We do this for each added logical thread. Furthermore, we also add an indicator for the speedup of our sequential algorithm relative to the single-threaded parallel algorithm. For convenience we split this experiment into two parts: One for our parallel partition and one for our parallel Quicksort algorithm.

Parallel Partition

For our parallel partitioning algorithm we fixed the input size to 2^{33} and the input types to uint8 and uint16. The graph for parallel BF-Partition is named bfParPar and the indicator for sequential BF-Partition is named bfSeqPar.

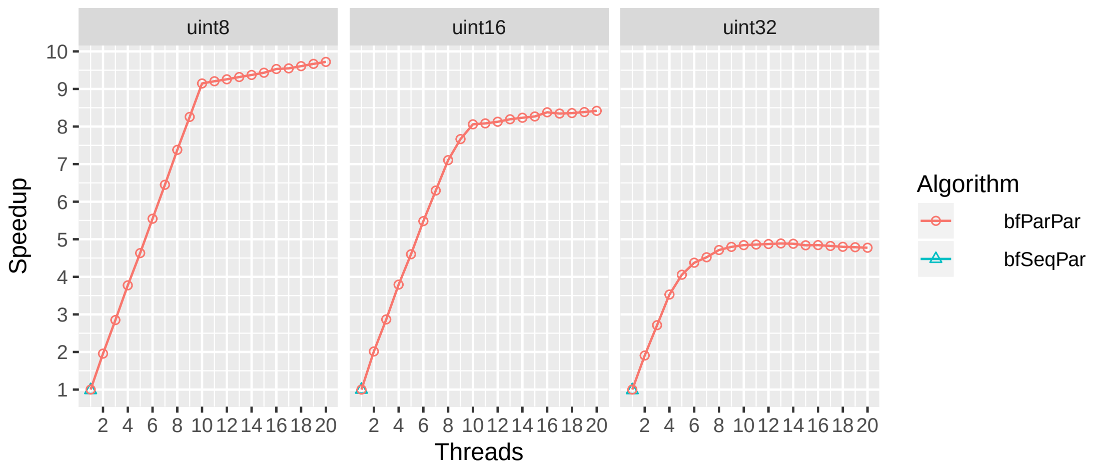


Figure 6.11: Scaling of parallel BF-Partition for 2^{32} uint8, uint16, and uint32 inputs

From Figure 6.11 we can see that our parallel partitioning algorithm scales nicely for uint8 inputs. For uint16 inputs we observe impeded scaling from 8 threads onwards and for uint32 inputs from 4 threads onwards. This is because BF-Partition has high memory bandwidth demand. We profiled this using Intel's VTune Amplifier [18]. Our results of one profiling run for 2^{32} uint16 elements and 20 threads are shown in the following figure:

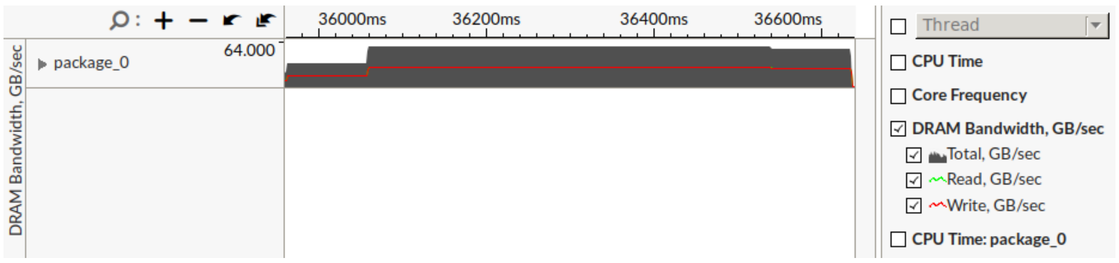


Figure 6.12: Memory bandwidth for BF-Partition using 2^{33} uint16 inputs and 20 threads

In Figure 6.12 we can see that parallel BF-Partition is most of its execution time memory bound for the presented configuration. This means that our algorithm processes data faster than it can request new inputs from main memory. From this follows that the observed scaling behavior is caused by limited memory bandwidth and therefore, better scaling would require better hardware.

Parallel Quicksort

For our Quicksort scaling experiment we fixed the input types to uint16 and uint32 and the input count to 2^{30}. The line plot for parallel BF-Quicksort is named bfParQs and the graph for sequential BF-Quicksort is named bfSeqQs. For this experiment we removed the all equal pattern detector from our algorithm.

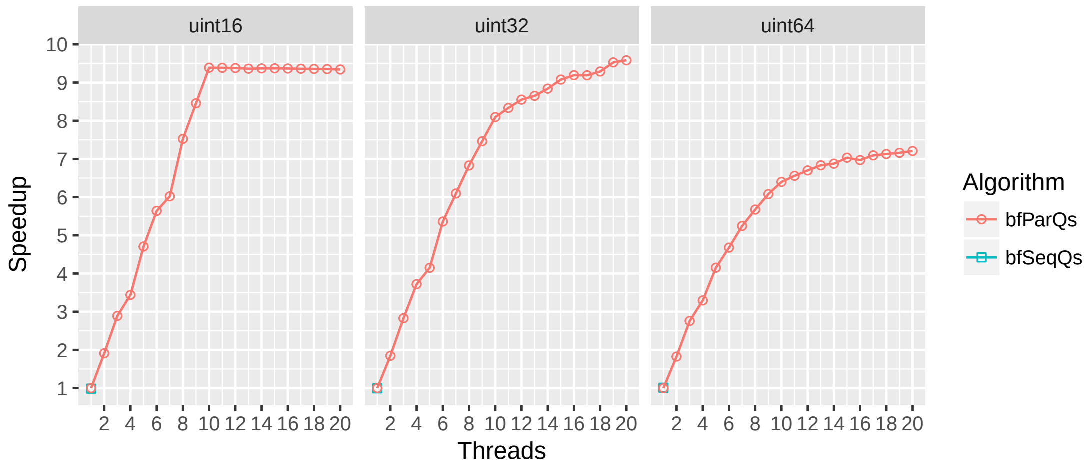


Figure 6.13: Scaling of parallel BF-Quicksort for 2^{30} uint16, uint32, and uint64 inputs

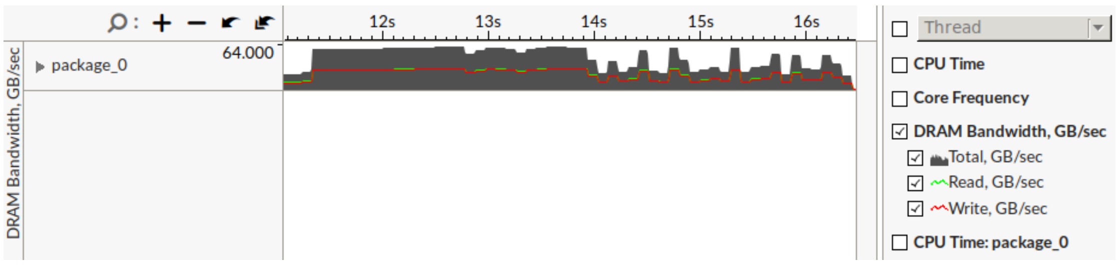


Figure 6.14: Memory bandwidth for BF-Quicksort using 2^{30} uint64 inputs and 20 threads

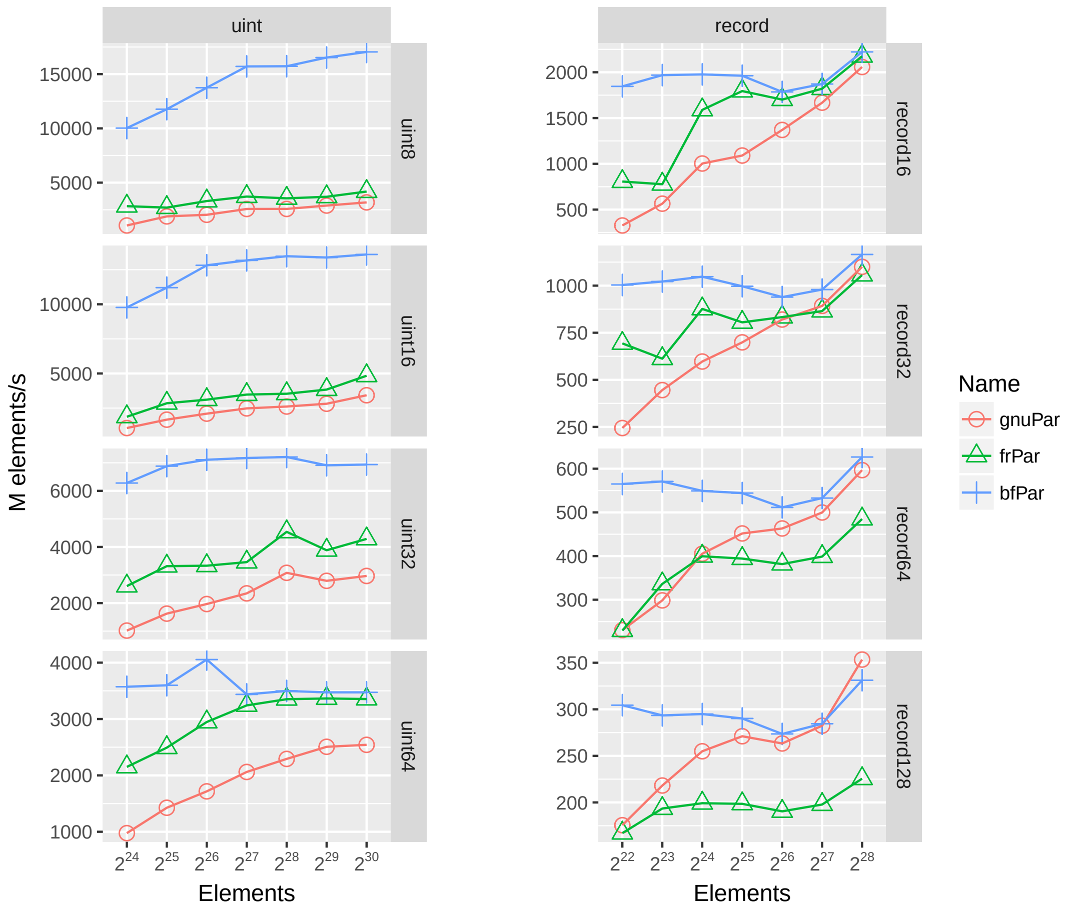
As we can see in Figure 6.13 the speedup of BF-Quicksort is decent for uint16 inputs and gets worse for larger input types. This is because of the in Subsection 6.2.2 discussed memory bandwidth limitations in our parallel partitioning procedure. As our measurements for bandwidth utilization indicate, we see this impeded scaling behavior because of dynamic load balancing we implemented accordingly to Tsigas et. al.'s proposed *Process Partition* phase: This process of proportionally assigning threads to partitions from where they start their recursion independently, leads to the fact that we constantly load data from memory in the shared L3-Cache during the first iterations of parallel BF-Quicksort. This is problematic, because all of our processors work wide spread across the input and therefore hardly work on the same data present in the L3-Cache, which in turn causes several requests to main memory. As we can see in Figure 6.14, this effect prevails persistently during the first half of the algorithm. In contrast, its second half works out way better as soon as most threads start their *Sequential Sorting with Helping* phase in which they can finally share L3-Cache data. Because of time constraints we sadly were not able to solve this algorithmic problem within the scope of this thesis, but we will provide a possible solution to it in our last chapter.

6.2.3 Comparison

In this section we measure the performance for our two parallel implementations against prevailing competitors. For convenience we again split this experiment into two parts: The first part is about parallel partitioning and the second part is about parallel sorting.

Parallel Partition

There are not many parallel libraries containing standalone parallel partitioning algorithms, one of the most famous implementation is the parallel partitioning algorithm from GNU's libstdc++ parallel mode (gnuPar), which is a collection of parallellized algorithms provided by the C++ Standard Library [29]. Furthermore, there is one implementation proposed by Frias and Petit [13] (frPar), which is particularly interesting, because it also uses the parallelization approach of Tsigas and Zhang [30] which they enhanced by introducing a tree like data structure that aims for enabling parallelism in the *Sequential Partition of the Data* phase.



(a) Comparison in million elements per second for uint inputs

(b) Comparison in million elements per second for for record inputs

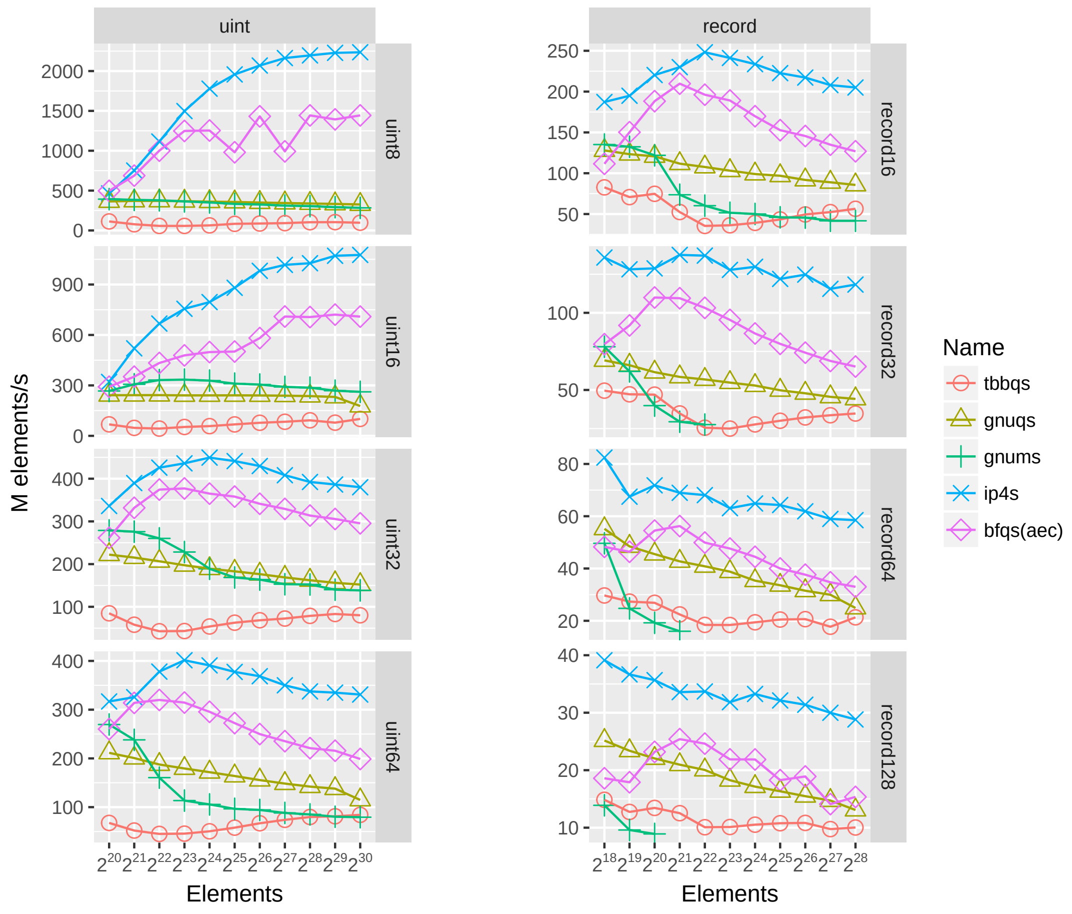
Figure 6.15: Comparison of different parallel partitioning algorithms

As we can see from Figure 6.15, BF-Partition is in almost all cases the best parallel partitioning algorithm. Furthermore, it beats the partitioning procedure of Frias et. al. as long as it is not memory bound by up to a factor of 6.2 for 2^{26} uint8 inputs. We can also observe that as soon as all algorithms start beginning to get memory bound (e.g. for 2^{27} record16 inputs), BF-Partition is still superior in almost all cases as the performance of the algorithms converge. We suspect that the drop in Frias et. al.'s algorithm for larger input types is because of their tree building phase, which needs more time for those inputs and therefore impedes their scaling.

Parallel Sort

In our final experiment we compare parallel BF-Quicksort to other parallel sorting algorithms. For this we selected the following candidates:

- *Intel Thread Building Blocks' Parallel Sort* (tbbQs): This Quicksort algorithm of Intel is part of Intel's Thread Building Blocks library [19] which contains several parallel implementations of general purpose algorithms and standalone utilities to parallelize other programs.
- *GNU's Parallel Quicksort* (gnuQs): As already mentioned in the previous experiment, GNU's parallel mode implements many different algorithms of the C++ Standard Library. One of these implementations is a parallel version of Quicksort, which implements the same strategy for parallelizing Quicksort as proposed by Tsigas et. al. [30] and discussed in Section 4.2.
- *GNU's Parallel Mergesort* (gnuMs): Another parallel algorithm provided by GNU's parallel mode is a parallel implementation of Mergesort. This algorithm is the only one that does not work in-place. This also provides the reason why we did not execute the algorithm for all input permutations: Its memory demand was simply too high for some input combinations.
- *Parallel In-Place Super Scalar Samplesort* (ip4s): The in [4] proposed Samplesort implementation of Axtmann also features a parallel version, which also in its parallel mode is one of the best performing freely available in-place sorting algorithms to date.
- *Parallel BF-Quicksort with all equal check* (bfQs(aec)): For this experiment we enable our proposed all equal pattern detector (see Subsection 4.1.2).



(a) Comparison in million elements per second for uint inputs

(b) Comparison in million elements per second for for record inputs

Figure 6.16: Comparison of different parallel sorting algorithms

As we can see from this experiment our proposed parallel algorithm works particularly well. Apart from ip4s we provide the fastest sorting algorithm and even more, the fastest parallel single pivot Quicksort algorithm in our benchmark being factor 1.94 faster for 2^{30} uint32 inputs and still faster for suitably large counts of all other input types. Although we can again observe the already mentioned performance drops for bigger than 64 Byte values, BF-Quicksort is still performing very well for at least 2^{20} elements. Also for uint32 and uint64 we come quite close to the performance of the highly tuned ip4s, which is a great success.

7 Conclusion and Further Research

7.1 Conclusion and Open Questions

In this thesis we introduced BF-Quicksort which is a novel approach to realize branch-free in-place Quicksort. We also applied a parallelization strategy proposed by Tsigas and Zhang [30] to this algorithm in order to execute it concurrently on several processors of modern multi-core systems. Their work on parallel Quicksort not only provides the frame for our parallel algorithm, it also inspired us to develop BF-Quicksort's sequential version. After we introduced the necessary foundations, we showed that it takes a well performing sequential partition algorithm in order to enable high performance Quicksort. This is why we improved classical Hoare partitioning by applying branch avoiding techniques to it so that it avoids hardly predictable branches in its main loop. Our experiments show that these improvements significantly increase the performance of partition and therefore also the overall performance of Quicksort. Our sequential partitioning algorithm outperforms GCC's standard partition by more than a factor of 3 for integer inputs. The sequential implementation of BF-Quicksort is the fastest single pivot Quicksort algorithm we measured for almost all input types. Only a very recent version of Samplesort [4] was able to beat our algorithm in our benchmarks. In our experiments we also found that all analyzed branch-free and buffer based partitioning and Quicksort algorithms suffer a huge loss in performance for inputs larger than 64 Byte.

After applying parallelization to BF-Quicksort we discovered that the proposed approach does work very well for the partitioning procedure. The resulting algorithm scales nicely and outperforms its closest parallel partitioning competitors by up to a factor of 6.15 as long as our algorithm is not memory bandwidth bound. Interestingly, BF-Partition suffers this impediment already for relatively small input types of 16 Byte utilizing eight or more processors. Also our results for the resulting parallel implementation of BF-Quicksort are very decent: We outperform all other parallel single pivot Quicksort algorithms for almost all input configurations in our benchmark of uniformly at random chosen elements by a factor of at least 1.94 for reasonably large uint32 inputs.

In this context we found that using a low overhead all equal pattern detector can bring huge performance increases for tightly skewed inputs compared to more sophisticated approaches that detect duplicate elements more fine grained. The received speedup from this improvements often is a multiple of the performance of non-pattern detecting approaches.

From our overall results we see that branch-free Quicksort works well for up to 64 Byte input types and suitably large input counts and that it is worth inducing additional instructions to avoid branches in its main loop. Because of time constraints there remain several unanswered questions:

- We could not deduce the actual reason for why large input types impede branch-free partition and therefore branch-free Quicksort this much by merely analyzing our profiling results. These results did not even change as we switched our compiler. We recommend to go into more depth in this area e.g. by using the Intel Architecture Code Analyzer [14].
- As implemented according to the parallel sorting algorithm proposed by Tsigas et. al., processors get divided relatively after each partitioning step. We concluded in Section 6.2.2, that this dynamic load balancing is further facilitating the bottleneck of our memory bound partitioner. We suggest to try a different approach to recurse into the algorithm by assigning all threads to one partition as long as the partition is larger than L3-Cache. From there on we can proceed as initially proposed utilizing dynamic workload balancing. This way we minimize main memory accesses and facilitate sharing of already loaded L3-Cache data.

7.2 Further Research

Besides the obligatory suggestion to further tune our proposed constants and formulas there is still room for more good theses in Quicksort [26]. Some of the ideas we came across when writing this thesis can provide good starting points for new theses:

- From our experiments in Chapter 6 we see that In-Place Super Scalar Samplesort performs exceptionally well. Since Samplesort can be considered as generalization to Quicksort by selecting a vast amount of pivots we conclude that a multi-pivot version of Quicksort e.g. Yaroslavsky's Dual Pivot Quicksort [34] would also greatly benefit form applying such branch avoiding techniques.
- To further tune our parallel implementation of BF-Partition it would be possible to apply the findings of Frias and Petit [13]. In their paper they propose an approach to reduce the duration of the *Sequential Partition of the Data* phase by adding a binary tree data structure which avoids redundant comparisons and enables parallel swapping of mispositioned elements. Although their findings

are interesting, the phase in which they build this tree impedes scaling of their implementation and as our experiments show, their algorithm particularly suffers performance for large input types.

- As the algorithm Pattern Defeating Quicksort implemented by Orson [22] demonstrates, it greatly makes sense to apply pattern detecting techniques to Quicksort. They apply those techniques on top of *BlockQuicksort* proposed by Edelkamp and Weiss [10]. Similar to our all equal check they use imbalances in the size of the partition to detect whether the input follows a certain pattern. Their proposed algorithm implements linear running time for ascending and descending sorted inputs, ascending and descending inputs that are followed by one "out-of-place" element (push back/push front inputs), and inputs with many duplicates. It would be interesting to academically investigate further into this theme to expand this list by applying lightweight pattern detection for even more input patterns found in theory and practice.

Overall we can say that our results indicate that it is worth further investigating into the field of branch-free partition based sorting and that it is worthwhile revisiting already established algorithms from this comparatively new perspective.

List of Figures

List of Tables

List of Code

Bibliography

[1] D Abhyankar and M. Ingle. “Engineering of a quicksort partitioning algorithm.” In: *International Journal of Global Research in Computer Science* 2.2 (2011), pp. 17–23.

[2] *ARM® Cortex®-A Series Programmer’s Guide for ARMv8-A*. Order Number: 248966-033. ARM. 2015.

[3] Martin Aumüller and Martin Dietzfelbinger. “Optimal Partitioning for Dual-Pivot Quicksort.” In: *ACM Trans. Algorithms* 12.2 (Nov. 2015), 18:1–18:36. ISSN: 1549-6325. DOI: 10.1145/2743020.

[4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. “In-place Parallel Super Scalar Samplesort (IPSSSSo).” In: *CoRR* abs/1705.02257 (2017). arXiv: 1705.02257.

[5] Sara Baase. *Computer algorithms: introduction to design and analysis*. Pearson Education India, 2009.

[6] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. “Proving JDK’s Dual Pivot Quicksort Correct.” In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Andrei Paskevich and Thomas Wies. Cham: Springer International Publishing, 2017, pp. 35–48.

[7] Jon Bentley. “Programming pearls: How to sort.” In: *Communications of the ACM* 27.4 (1984), 287–ff.

[8] Jon L. Bentley and M. Douglas McIlroy. “Engineering a sort function.” In: *Software: Practice and Experience* 23.11 (1993), pp. 1249–1265.

[9] Oracle Corporation. *Java® Platform, Standard Edition & Java Development Kit Version 9 API Specification Class Arrays*. 2017. URL: https://docs.oracle.com/javase/9/docs/api/java/util/Arrays.html#sort-int:A- (visited on 03/09/2018).

[10] Stefan Edelkamp and Armin Weiss. “BlockQuicksort: Avoiding Branch Mispredictions in Quicksort.” In: *24th Annual European Symposium on Algorithms (ESA 2016)*. Vol. 57. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 38:1–38:16. ISBN: 978-3-95977-015-6.

[11] Amr Elmasry and Jyrki Katajainen. "Lean programs, branch mispredictions, and sorting." In: *International Conference on Fun with Algorithms*. Springer. 2012, pp. 119–130.

[12] Maarten H van Emden. "Algorithms 402: Increasing the efficiency of quicksort." In: *Communications of the ACM* 13.11 (1970), pp. 693–694.

[13] Leonor Frias and Jordi Petit. "Parallel partition revisited." In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 142–153.

[14] Israel Hirsh and Gideon S. *Intel® Architecture Code Analyzer*. 2017. URL: `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer` (visited on 03/11/2018).

[15] Charles AR Hoare. "Quicksort." In: *The Computer Journal* 5.1 (1962), pp. 10–16.

[16] Eric Huss. "The C library reference guide." In: *Webmonkeys: A Special Interest* (1997).

[17] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Order Number: 248966-033. Intel Corporation. 2016.

[18] *Intel® Architecture Code Analyzer™ Amplifier*. 2017. URL: `https://software.intel.com/en-us/intel-vtune-amplifier-xe` (visited on 03/11/2018).

[19] *Intel® Thread Building Blocks (Intel® TBB)*. Intel Corporation. 2018. URL: `https://www.threadingbuildingblocks.org` (visited on 03/14/2018).

[20] Kanela Kaligosi and Peter Sanders. "How branch mispredictions affect quicksort." In: *European Symposium on Algorithms*. Springer. 2006, pp. 780–791.

[21] Jyrki Katajainen. "Sorting programs executing fewer branches." In: *CPH STL Report 2263887503, Department of Computer Science, University of Copenhagen* (2014).

[22] Orson Peters. *Pattern Defeating Quicksort*. 2015. URL: `https://github.com/orlp/pdqsort` (visited on 03/10/2018).

[23] *Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort*. URL: `http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-September/002633.html` (visited on 03/10/2018).

[24] Peter Sanders and Sebastian Winkel. "Super scalar sample sort." In: *European Symposium on Algorithms*. Springer. 2004, pp. 784–796.

[25] Robert Sedgewick. "Implementing quicksort programs." In: *Communications of the ACM* 21.10 (1978), pp. 847–857.

[26] Robert Sedgewick. "Quicksort." PhD thesis. Stanford University, 1975.

[27] Robert Sedgewick. "The analysis of Quicksort programs." In: *Acta Informatica* 7.4 (Dec. 1977), pp. 327–355. ISSN: 1432-0525. DOI: `10.1007/BF00289467`.

[28] T. Singh, D. K. Srivastava, and A. Aggarwal. "A novel approach for CPU utilization on a multicore paradigm using parallel quicksort." In: *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*. Feb. 2017, pp. 1–6. DOI: `10.1109/CIACT.2017.7977382`.

[29] *The GNU C++ Library*. Free Software Foundation. 2018. URL: `https://gcc.gnu.org/onlinedocs/libstdc++/index.html` (visited on 03/09/2018).

[30] Philippas Tsigas and Yi Zhang. "A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000." In: *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*. IEEE. 2003, pp. 372–381.

[31] John W. Tukey. "The ninther, a technique for low-effort robust (resistant) location in large samples." In: *Contributions to Survey Sampling and Applied Statistics*. Elsevier, 1978, pp. 251–257.

[32] Theo Ungerer. *Mikrocontroller und Mikroprozessoren*. Springer-Verlag, 2010.

[33] *Valgrind™ User Manual Cachegrind: a cache and branch-prediction profiler*. Valgrind Developers. 2017. URL: `http://valgrind.org/docs/manual/cg-manual.html` (visited on 03/10/2018).

[34] Vladimir Yaroslavskiy. *Dual-pivot quicksort algorithm*. 2009. URL: `http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf` (visited on 03/09/2018).