FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Parallel Quicksort algorithm
Student:	Bc. Gabriel Hévr
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2024/2025

Instructions

1) Study in-place algorithms for sorting. Focus on the Quicksort algorithm and its optimization possibilities (including out-of-place). [1, 2]
2) Analyze the possibilities of combining Quicksort with other sorting algorithms. [3]
3) Investigate and analyze shared memory parallel implementations of Quicksort, exploring multi-threading. [4]
4) Based on the results of previous analyses, design and implement a parallel Quicksort algorithm using OpenMP combined with other sorting algorithms. [1, 2]
5) Based on the OpenMP implementation, develop a parallel Quicksort implementation leveraging C++ threads and relevant synchronization techniques, excluding over-standard C++ language extensions like OpenMP.
6) Conduct a comprehensive comparative analysis of the resulting implementations against each other and existing parallel Quicksort implementations in C++. [4, 5, 6, 7]

[1] Peters, Orson RL. "Pattern-defeating quicksort." arXiv preprint arXiv:2106.05123 (2021)
[2] Edelkamp, Stefan, and Armin Weiß. "Blockquicksort: Avoiding branch mispredictions in quicksort." Journal of Experimental Algorithmics (JEA) 24 (2019): 1-22.
[3] Musser, David R. "Introspective Sorting and Selection Algorithms". https://doi.org/10.1002/(SICI)1097-024X(199708)27:8%3C983::AID-SPE117%3E3.0.CO;2-%23
[4] LANGR, Daniel; TVRDIK, Pavel; SIMECEK, Ivan. AQsort: Scalable Multi-Array In-Place Sorting with OpenMP https://www.scpe.org/index.php/scpe/article/view/1207
[5] Bc. Klára Schovánková: Parallel Sorting in C++11, Master Thesis ČVUT FIT 2019
[6] Bc. Ondřej Voronecký: Efficient parallel multi-way Quicksort algorithm, Master Thesis

---

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 13 December 2023 in Prague.

ČVUT FIT 2023

[7] Singler, J.; Konsik, B.: The GNU Libstdc++ Parallel Mode: Software Engineering Considerations https://dl.acm.org/doi/10.1145/1370082.1370089

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 13 December 2023 in Prague.

Master's thesis

PARALLEL QUICKSORT ALGORITHM

Bc. Gabriel Hévr

Faculty of Information Technology
Department of Computer Systems
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.
May 8, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Hévr Gabriel. *Parallel Quicksort algorithm.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments viii

Declaration ix

Abstract x

Acronyms xi

Introduction 1

1 Sorting Algorithms 3
- 1.1 Classification . . . 3
- 1.2 Basic Sorting Algorithms . . . 5

2 Quicksort Algorithm 7
- 2.1 Design of Quicksort . . . 7
- 2.2 Combining with Other Sorting Algorithms . . . 10
- 2.3 Quicksort Optimizations . . . 12
- 2.4 Parallel Quicksort Algorithm . . . 19

3 Existing Implementations 25
- 3.1 Parallelization Methods . . . 25
- 3.2 Parallel Mode of Libstdc++ . . . 26
- 3.3 GCC's and Clang's PSTL . . . 26
- 3.4 CPP11Sort . . . 27
- 3.5 Other Parallel Sorting Algorithms . . . 27
- 3.6 pdqsort . . . 28

4 PPQSort (Parallel Pattern Quicksort) 31
- 4.1 Thread Balance . . . 31
- 4.2 Pivot Selection . . . 33
- 4.3 Parallel Partitioning . . . 34
- 4.4 Further Optimizations . . . 36
- 4.5 OpenMP Implementation . . . 37
- 4.6 C++ Threading Implementation . . . 38
- 4.7 Memory Usage . . . 44
- 4.8 Application Programming Interface (API) . . . 45
- 4.9 Testing Suite . . . 45

5 Comparative Analysis **49**
5.1 Testing Enviroments 49
5.2 Input Data 49
5.3 Finding Parameters 52
5.4 OpenMP vs. C++ Threading 54
5.5 Comparing with Other Implementations 60

6 Conclusion **81**

A More Measurements **83**

Contents of the Attached Archive **107**

List of Figures

2.1 Lomuto partitioning . . . 8
2.2 Hoare partitioning . . . 9
2.3 Block Partitioning . . . 14
2.4 Partition to the left . . . 17
2.5 Basic sorting networks . . . 17
2.6 Quicksort partition graph . . . 19
2.7 Parallel strided partition . . . 22
2.8 Parallel blocked partition . . . 22

4.1 End of the main stage . . . 34
4.2 Start of cleanup stage . . . 35
4.3 Swapping blocks in cleanup stage . . . 35
4.4 Final sequential partition . . . 36
4.5 Thread pool diagram . . . 40

5.1 Input data patterns . . . 51
5.2 Comparison of various block sizes in Hoare partition . . . 53
5.3 Comparison of various block sizes in branchless partition . . . 54
5.4 Comparison of various sequential thresholds . . . 55
5.5 Scalabilty of our implementations . . . 56
5.6 PPQSort performance with data patterns on STAR . . . 57
5.7 Comparison on int data patterns . . . 58
5.8 Comparison on string data patterns . . . 59
5.9 Comparison on different cardinalities . . . 59
5.10 Comparison on different sizes . . . 60
5.11 Comparison on matrices . . . 61
5.12 Comparison for int patterns data on STAR . . . 62
5.13 Comparison for low cardinality int data on STAR . . . 64
5.14 Comparison for small size int data on STAR . . . 65
5.15 Comparison for sparse matrices on STAR . . . 66
5.16 Comparison for int patterns data on Intel Cluster . . . 68
5.17 Comparison for low cardinality int data on Intel . . . 69
5.18 Comparison for smaller sizes int data on Intel . . . 70
5.19 Comparison for sparse matrices on Intel . . . 71
5.20 Comparison for int patterns data on ARM Cluster . . . 72
5.21 Comparison for low cardinality int data on ARM . . . 73
5.22 Comparison for smaller sizes int data on ARM . . . 74
5.23 Comparison for sparse matrices on ARM . . . 75

5.24 Comparison for int patterns data on RCI Cluster 77
5.25 Comparison for low cardinality int data on RCI 78
5.26 Comparison for smaller sizes int data on RCI 79
5.27 Comparison for sparse matrices on RCI 80

List of Tables

3.1 Comparison of existing implementations 29

5.1 Overview of testing environments 50
5.2 Overview of generated input data 52
5.3 Sparse matrices 53

A.1 All algorithms patterns short on STAR 84
A.2 All algorithms patterns int on STAR 84
A.3 All algorithms patterns double on STAR 85
A.4 All algorithms patterns string on STAR 85
A.5 All algorithms cardinalities int on STAR 86
A.6 All algorithms smaller size int on STAR 86
A.7 All algorithms matrices on STAR 87
A.8 All algorithms patterns short on Intel 87
A.9 All algorithms patterns int on Intel 88
A.10 All algorithms patterns double on Intel 88
A.11 All algorithms patterns string on Intel 89
A.12 All algorithms cardinalities int on Intel 89
A.13 All algorithms smaller size int on Intel 90
A.14 All algorithms matrices on Intel 90
A.15 All algorithms patterns short on ARM 91
A.16 All algorithms patterns int on ARM 91
A.17 All algorithms patterns double on ARM 92
A.18 All algorithms patterns double on ARM 92
A.19 All algorithms cardinalities int on ARM 93
A.20 All algorithms smaller size int on ARM 93
A.21 All algorithms matrices on ARM 94
A.22 All algorithms patterns short on RCI 94
A.23 All algorithms patterns int on RCI 95
A.24 All algorithms patterns double on RCI 95
A.25 All algorithms patterns double on RCI 96
A.26 All algorithms cardinalities int on RCI 96
A.27 All algorithms smaller size int on RCI 97
A.28 All algorithms matrices on RCI 97

List of Algorithms and Codes

2.1 QuickSort Algorithm 7
2.2 Lomuto Partition Scheme 8
2.3 Hoare Partition Scheme 9
2.4 QuickSort Algorithm recursion elimination 12
2.5 Block partitioning 13
2.6 Cyclic permutation 15
2.7 QuickSort: Parallel Recursion 20

4.1 Parallel Pattern Quicksort Algorithm 32
4.1 Task scheduler 41
4.2 Task stealing 42
4.3 Worker routine 44
4.4 Example of usage. 46
4.5 Use C++ threading 47
4.6 Cmake integration 47
4.7 Build and run scripts 48

I am grateful to my supervisor, doc. Ing. Ivan Šimeček, Ph.D., for providing invaluable guidance. Additionally, I extend my thanks to my friends and family for their unwavering support. The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is also gratefully acknowledged. I also thank M. Václavík of the Czech Technical University in Prague for providing access to university clusters.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 8, 2024

Abstract

This diploma thesis presents Parallel Pattern Quicksort (PPQSort), an innovative parallel quicksort algorithm designed to deliver exceptional performance and ease of use. Using C++ threads, PPQSort eliminates the need for external dependencies and allows seamless deployment across diverse computing environments. This thesis outlines the innovative quicksort optimizations utilized by PPQSort, such as branchless parallel partitioning and their efficient implementation. Furthermore, the study includes an extensive comparative analysis of PPQSort versus existing parallel quicksort algorithms on various machines and with different input data. The experimental evaluation results demonstrate that PPQSort is both speedy and robust, consistently outperforming the fastest existing parallel quicksort implementations on all inputs, often by an average of 50%.

Keywords parallel sorting, quicksort, C++20, thread pool, in-place sorting, parallel partitioning, PPQSort

Abstrakt

V této diplomové práci je představen PPQSort, inovativní paralelní Quicksort algoritmus, který poskytuje excelentní výkon a cílí na snadné použití. PPQSort je implementován pouze na základě vláken jazyka C++, tedy bez použití externích knihoven a nestandardních rozšíření. Díky tomu PPQSort umožňuje bezproblémové nasazení v různých výpočetních prostředích. Práce popisuje nové optimalizace Quicksortu, které PPQSort využívá, jako je paralelní rozdělování bez větvení a jejich efektivní implementaci. Práce dále prezentuje rozsáhlou srovnávací analýzu algoritmu PPQSort oproti stávajícím paralelním Quicksort algoritmům na různých strojích a s různými vstupními daty. Výsledky experimentálního vyhodnocení ukazují, že PPQSort je rychlý a robustní a trvale překonává nejrychlejší existující paralelní implementace Quicksortu na všech vstupech, často v průměru o 50%.

Klíčová slova paralelní řazení, quicksort, C++20, fond vláken, in-place řazení, paralelní rozdělování, PPQSort

Acronyms

ABI Application Binary Interface

AI Artificial Intelligence

AMD Advanced Micro Devices

AoS Array of Structures

API Application Programming Interface

ARM Advanced RISC Machines

CCCL CUDA C++ Core Libraries

COO Coordinate Storage Format

CPU Central Processing Unit

CSC Compressed Sparse Column

CSR Compressed Sparse Row

CUDA Compute Unified Device Architecture

GCC GNU Compiler Collection

GCC BQS GCC Balanced QuickSort

GNU GNU's Not Unix — an extensive collection of free software

GPU Graphics Processing Unit

gtest GoogleTest framework

HW Hardware

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

LLVM Low Level Virtual Machine

MCSTL The Multi-core Standard Template Library

MPQsort Multiway parallel Quicksort

MSVC Microsoft Visual C++

NUMA Non–Uniform Memory Access

NVCC NVIDIA Cuda Compiler

OMP OpenMP

oneTBB OneAPI Threading Building Blocks

OpenMP Open Multi-Processing

OS Operating System

pdqsort Pattern-defeating Quicksort

PPL Parallel Patterns Library

PPQSort Parallel Pattern Quicksort

RAM Random Access Memory

SoA Structure of Arrays

TBB Threading Building Blocks

Introduction

Sorting algorithms are an essential part of computer science. They are used in many programs and systems, and their efficiency and speed are crucial. Despite being one of the oldest problems in computer science, sorting is still a highly active area of research and development. Modern approach is to parallelize the sorting routines. This technique maximizes the utilization of multi-core CPUs, resulting in significant performance improvements compared to sequential methods.

Although C++ has introduced parallel algorithms starting from the C++17 standard, their implementation and compiler support remain limited. Major compilers like GCC and Clang offer minimal support. While compilers like MSVC and Intel OneAPI C++ provide parallel algorithm implementations, they rely on complex, vendor-specific libraries. This dependence can create portability and ease of use challenges.

Our research aims to provide an effective parallel quicksort implementation without external dependencies. We focus on parallel quicksort implementations in shared-memory environments. Existing solutions often rely on non-standard extensions like OpenMP or external libraries such as oneTBB. We identified only two implementations solely based on the standard library, but they lacked novel approaches and optimizations for quicksort. To bridge this gap, we leverage the power of the C++ standard library to introduce our fast parallel quicksort algorithm named Parallel Pattern Quicksort (PPQSort).

To establish the current state of the art, we provide a thoughtful analysis of novel and effective sequential quicksort optimizations and their efficient implementations. Building upon this foundation, we introduce our parallel implementation of these optimizations, resulting in a fast parallel quicksort algorithm. Next, we focus on existing parallel sorting algorithms, examining their properties and comparing them to PPQSort. We conduct a comprehensive comparative analysis where PPQSort is compared with established parallel quicksort implementations to solidify our findings.

Our thorough benchmarking process involves a broad range of data types, input distributions, input sizes, input cardinalities, and sparse matrices running on four different machines. With such a comprehensive dataset, we can confidently validate our results. In the appendix, we present complete benchmark results, which include outcomes for different parallel sorting algorithms (not limited to quicksort).

Chapter 1
Sorting Algorithms

This chapter begins by exploring the fundamental properties that define how sorting algorithms operate. These properties will enable us to classify and compare different sorting algorithms effectively. Following this, we present an overview of basic in-place sorting algorithms, a crucial category that performs sorting without requiring additional memory space.

The main objective of sorting is to arrange the data according to a specific criterion. In a formal sense, it involves taking a sequence of n elements $a_1, ..., a_n$ as input, along with a comparator that allows comparison of any two elements in the input. The output of the sorting algorithm is a permutation of the input data such that $\forall i \in \langle 0, n-1) : b_i \leq b_{i+1}$. The resulting order depends on the comparator, which is denoted by the symbol $<$ in this thesis [1].

1.1 Classification

We can use different properties to evaluate and categorize sorting algorithms. Doing so allows us to compare and determine which algorithm suits our specific needs. It is important to note that no algorithm is flawless and possesses all the ideal properties. Usually, if an algorithm excels in one property, it comes at the expense of another. As a result, contemporary ranking algorithms often combine multiple algorithms to leverage their respective strengths. Besides time and memory complexity, the properties of interest include stability, adaptivity, recursiveness, and whether the algorithm functions within the comparison model.

1.1.1 Time Complexity

Time complexity is one of the most basic and essential properties we investigate in sorting algorithms. The complexity depends on the input size, which we will refer to as n from now on. We represent the time complexity using asymptotic notation, allowing us to disregard any multiplicative and additive constants. We are concerned with the best, average, and worst-case scenarios when considering time complexity.

The best-case scenario illustrates the algorithm's performance under ideal circumstances. It represents the fastest possible case in terms of time complexity. For instance,

if the desired element is found immediately in the first position in a linear search, this would be considered the best case. Similarly, applying the insertion sort algorithm [2, p. 17] to an already sorted array represents a best-case scenario for the insertion sort algorithm. Although the best-case complexity is not typically the primary focus when evaluating algorithms, as we prioritize the average and worst-case scenarios, it can still provide valuable insights.

In contrast, the maximum time required for an algorithm to complete a specific input size, n, represents its worst-case scenario and is commonly used to evaluate its time complexity. Consider a linear search, for example, where the element being searched for is located at the very end. While the algorithm usually performs faster than its worst-case time complexity, it may slow down for a few exceptional inputs. In such cases, calculating the average case complexity may be more beneficial, which involves taking the arithmetic mean of the algorithm's time requirements across all inputs of a particular size [1, 2].

The RAM model [1, p. 52] states that the sorting algorithms have a minimum time complexity of $O(n \log(n))$. It is because any sorting algorithm requires at least $O(n \log(n))$ comparisons [2, p. 207]. Sorting algorithms such as Merge sort [2, p. 34], Quick sort [2, p. 182], and heap sort [2, p. 161] are well-known examples of algorithms that have this average time complexity.

On the contrary, there exist sorting algorithms, such as radix sort [2, p. 211], that have a lower time complexity but usually rely on specific conditions for the input data and are not applicable in the comparison model. Our research aims for complete flexibility and thus concentrates on comparison-based algorithms that can handle input data without special requirements.

1.1.2 Memory Usage

Memory complexity is a crucial aspect that determines the memory needs of an algorithm. It specifically refers to the amount of memory consumed based on the input size, n. The memory complexity of an algorithm only takes into account the additional memory used for auxiliary computations, excluding the size of the input data. Algorithms are categorized as either *in-place* or *out-of-place*, depending on their memory complexity.

In-place[1] algorithms are known for their low memory usage, as they only require constant extra memory to run, regardless of the input size. This results in a memory complexity of $O(1)$. On the other hand, out-of-place algorithms require additional memory allocation during execution, and the amount of memory needed is proportional to the input size, n. Merge Sort is a famous example of an out-of-place algorithm with a space complexity of $O(n)$.

1.1.3 Other Properties

Beyond the time complexity and memory usage, the following additional properties of sorting algorithms can still provide valuable insights into a given sorting algorithm:

[1] The Latin term "in situ" is also used in literature.

- Stability is another crucial attribute. A sorting algorithm is considered stable if it preserves the relative order of equal elements. In other words, if two items are equal according to the sort key, their order in the sorted output will be the same as in the input.
- Adaptibility is another interesting characteristic. An adaptable algorithm can modify its behavior in response to the specific structure of the input data. It allows it to achieve improved performance when dealing with partially sorted data or data that follows certain patterns. A prominent illustration of an adaptable sorting algorithm is the insertion sort. This algorithm showcases the ability to adjust its time complexity to almost $O(n)$ when presented with almost sorted input, resulting in highly efficient execution in such circumstances.
- Recursivness property differentiates between recursive and non-recursive algorithms. A recursive algorithm solves more minor instances of the problem by calling itself. Recursion is a fundamental concept in divide-and-conquer strategies, where the input is divided into smaller parts, each part is sorted independently, and then the sorted parts are merged or combined to generate the final sorted output.

1.2 Basic Sorting Algorithms

Sorting algorithms are a fundamental concept in computer science. Basic sorting algorithms are simple to implement, concise, and operate directly on the given sequence. However, the time complexity of these algorithms often increases quadratically. Bubble sort, selection sort, and insertion sort are examples of such non-recursive algorithms that operate in-place. They are suitable for sorting smaller sequences or as components in a recursive sorting algorithm. Among these, insertion sort is commonly preferred due to its efficiency when dealing with nearly sorted inputs. In our work, we will extensively use insertion sort and its variations, which Chapter 4 elaborates on in detail.

Apart from basic sorting algorithms, more intricate in-place sorting techniques exist that are efficient and extensively utilized for larger datasets. Heap sort stands out as a prominent example. This algorithm organizes the input data in a heap structure and repeatedly extracts the largest element from the heap. This element is then placed at the end of the sorted portion of the array. By reheapifying the remaining elements, heap sort ensures that the next largest element is always ready for movement. This process continues until all elements are sorted. Heap sort is notable for its guaranteed time complexity of $O(n \log(n))$, making it suitable for various applications.

Although many sorting algorithms are not in-place, like merge sort, our primary focus will be on in-place sorting methods. Among these, quicksort is a highly advanced in-place algorithm discussed in Chapter 2.

Chapter 2

Quicksort Algorithm

Quicksort was first proposed by Tony Hoare in 1961 [3]. Over the years, it has gained widespread popularity as one of the most efficient sorting algorithms for practical use. Quicksort is a recursive comparison-based and unstable sorting algorithm. Quicksort is considered in-place despite having a space complexity of $O(\log n)$ due to recursion calls. Regarding average-case scenarios, the optimized version of quicksort generally outperforms other sorting methods, such as merge sort or heapsort. That is why our work focuses primarily on the quicksort algorithm.

2.1 Design of Quicksort

QuickSort is a prime example of the divide-and-conquer strategy in algorithm design. Its main goal is to break down the task of sorting an entire array into smaller sub-problems. The process starts by selecting a pivot element, which is then used to partition the input sequence in-place into two subsequences. Once the partitioning is complete, all elements in the left subsequence are smaller than the pivot, while all elements in the right subsequence are greater than or equal. The algorithm is then recursively applied to both subsequences until an empty or single-element subsequence is reached, which is naturally sorted. The pseudocode in Code 2.1 demonstrates the complete concept. There are various partitioning methods, including using multiple pivots to divide the input sequence into multiple subsequences [4, 5]. However, our research mainly focuses on partitioning using a single pivot.

Algorithm 2.1 QuickSort Algorithm

```
procedure QUICKSORT(A, lo, hi)
    if lo < hi then
        pivotIndex ← PARTITION(A, lo, hi)
        QUICKSORT(A, lo, pivotIndex − 1)        ▷ Sort the left sub-array
        QUICKSORT(A, pivotIndex + 1, hi)        ▷ Sort the right sub-array
    end if
end procedure
```

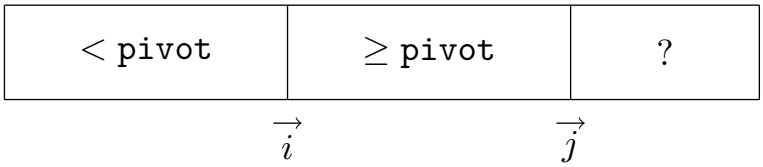


Figure 2.1 Lomuto partitioning

2.1.1 Partitioning Schemes

Partitioning is a crucial procedure in the quicksort algorithm, dividing the array into subsequences. The two commonly used partition schemes in quicksort are Lomuto [6] and Hoare [3].

The Lomuto partitioning technique involves two indices, i and j, incrementing from left to right through the array. The key difference is that i separates the elements smaller than the pivot from the larger ones. It acts as a pointer that indicates where the next smaller element than the pivot should be inserted. Meanwhile, the j index iterates through the array, examining each element sequentially. Whenever j comes across an element smaller than the pivot, that element is swapped with the element at index i, and i is incremented. This process continues until j traverses the entire array. At this point, the pivot is swapped with the element at the i-th position, effectively placing it in its correct sorted position. The code snippet 2.2 also exemplifies the primary idea, where the function `GetPivot` selects a pivot and positions it at the beginning of the input data. There are different techniques for selecting a pivot, and we discuss them in Chapter 2.1.2. A visual representation of the Lomuto partitioning process is in Figure 2.1.

Algorithm 2.2 Lomuto Partition Scheme

```
procedure LomutoPartition(A, lo, hi)
    pivot ← getPivot(A, lo, hi)        ▷ Place pivot at the begging and return copy
    i ← lo + 1
    for j ← lo + 1 to hi do
        if A[j] < pivot then
            SWAP(A[i], A[j])
            i ← i + 1
        end if
    end for
    SWAP(A[i], A[lo])                  ▷ Place pivot on correct position
    return i
end procedure
```

Conversely, the Hoare partitioning method, the original scheme devised by the quicksort creator, employs two indices that initiate from the opposite ends of the array and move towards each other. This method is equally effective, but it has the potential to be further optimized through parallelization, as elaborated in Chapter 2.4.2. The Code 2.3 demonstrates the main idea, while Figure 2.2 visually shows how the method works.

Algorithm 2.3 Hoare Partition Scheme

```
procedure HoarePartition(A, lo, hi)
    pivot ← getPivot(A, lo, hi)          ▷ Place pivot at the begging and return copy
    l ← lo + 1
    r ← hi
    while l < r do
        while A[l] < pivot and l < r do l++ end while
        while A[r] > pivot and l < r do r-- end while
        if l < r then
            SWAP(A[l], A[r])
            l ← l + 1
            r ← r − 1
        end if
    end while
    SWAP(A[lo], A[l])                    ▷ Place pivot on correct position
    return l
end procedure
```

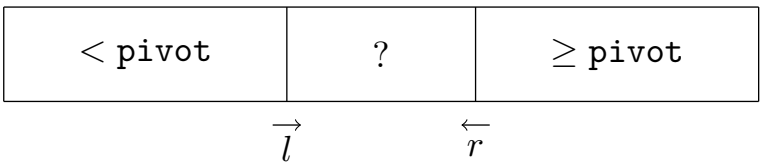


Figure 2.2 Hoare partitioning

2.1.2 Pivot Selection

Selecting the optimal pivot is a critical factor for the success of any quicksort implementation. By choosing a pivot close to the median, the algorithm can divide the sequence into two evenly distributed parts, resulting in high efficiency. However, the algorithm's time complexity can escalate to $O(n^2)$ if an unsuitable pivot is selected. The standard techniques for selecting a pivot are the following:

- **Fixed Element:** Typically choosing the first or last element of the array as the pivot. It is simple but can lead to poor performance if the array is already or nearly sorted.
- **Random Element:** The pivot is selected randomly, which helps prevent slow performance on sorted arrays and reduces the likelihood of encountering the worst-case time complexity on average. However, there is still a non-negligible chance that the chosen pivot will be unfavorable.
- **Median-of-n:** Typically, the median is selected by considering three or five elements. In the case of the Median-of-three approach, the pivot is chosen as the median value from the first, middle, and last elements. Similarly, the same principle applies when dealing with five or any number n of element samples. This approach enhances balance and demonstrates efficient performance across different input distributions.
- **Median-of-Medians:** To identify a suitable pivot in large arrays, a more complex but efficient approach is to use the median-of-medians technique. For instance, the median-of-three-medians technique is as follows: the first median is determined by calculating the median of the first, middle, and last elements. Then, the second median is determined by utilizing the elements at the indices first+1, middle+1, and last−1, and so forth. The pivot is then chosen as the median value from these calculated medians. By employing this algorithm, a pivot that is closer to the overall median can be obtained, resulting in improved balance. John Tukey was the first to describe this idea [7].

The pivot selection strategy can be customized based on the characteristics of the input data, such as the input size. Typically, the more sophisticated the method of selecting a pivot is, the more even the partitioning becomes; however, this also adds to the computational overhead involved in determining it. More sophisticated techniques exist, such as selecting a pivot as the median from a large number of samples. However, some studies [8] indicate that finding a "good" pivot may not necessarily enhance the performance of quicksort. Surprisingly, intentionally selecting a skewed pivot can improve performance. It is because, although the number of instructions decreases with a higher quality pivot, there is also an increased likelihood of branch mispredictions.

2.2 Combining with Other Sorting Algorithms

Combining quicksort with other sorting algorithms can exploit the strengths and mitigate the weaknesses of each sorting strategy. This hybrid approach often leads to improved performance, particularly for certain datasets or specific computational environments.

2.2.1 Insertion Sort

When dealing with small datasets or nearly sorted sequences, insertion sort is an excellent choice due to its simplicity and efficiency. Combining it with quicksort can be especially effective since quicksort can struggle with smaller partitions due to the overhead of recursive calls. The recommended approach is to use quicksort for larger partitions and switch to insertion sort when the partition size falls below a certain threshold, typically determined empirically. This combination can significantly reduce the overhead associated with quicksort's recursive calls for small arrays while benefiting from insertion sort's ability to efficiently complete the sorting process once the array is nearly sorted.

2.2.2 Heapsort

Chapter 1.2 explains the sorting algorithm known as heapsort. This method utilizes a heap data structure to sort the elements in the input sequence. First, the heap structure is built in place with a time complexity of $O(n)$. Then, the restructuring operation takes place after the root is removed and requires $O(\log n)$ time. This operation is performed $O(n)$ times, resulting in an overall time complexity of $O(n + n \log n) = O(n \log n)$ [9]. Due to its consistent time complexity in all scenarios, heapsort can be combined with quicksort, particularly in situations where the worst-case performance of quicksort is a concern. This approach is the main concept behind the hybrid sorting algorithm called introsort.

2.2.3 Introsort

In this hybrid approach, quicksort is utilized for its fast average-case performance. However, if the depth of recursion suggests that the worst-case scenario might materialize during execution, the algorithm can switch to heapsort to complete the sorting. This strategy provides a safety net for quicksort, ensuring that the sorting process does not degrade to time complexity $O(n^2)$ even in the worst case. Combining quicksort's divide-and-conquer efficiency with heapsort's robustness achieves a more reliable and uniformly efficient sorting process.

Quicksort is a highly efficient algorithm for real-world data, provided that the pivot is selected carefully to avoid the unlikely worst-case time complexity of $O(n^2)$. However, it is essential to note that quicksort can still be vulnerable to adversarial inputs intentionally designed to induce this worst-case scenario [10]. The past libc++ implementation encountered issues related to this vulnerability [11]. That is why introsort comes into play, emphasizing its significance.

A more reliable and uniformly efficient sorting process can be achieved by combining the divide-and-conquer efficiency of quicksort with the robustness of heapsort. However, determining the precise moment to switch from quicksort to heapsort can be a challenging task. If the switch is made too soon, the speed of quicksort may be lost, but if it is made too late, the worst-case scenario can significantly impact performance. The typical method is to monitor the recursion depth and transition to heapsort when the depth exceeds a certain threshold. This threshold is usually a logarithmic factor of the size of the array [12]. Nevertheless, there are other methods available that could provide increased accuracy and effectiveness, as outlined in Chapter 2.3.6.

Currently, the introsort algorithm is widely integrated into various standard library sorting functions. As of now, the three primary C++ implementations – libstdc++, libc++, and Microsoft Visual C++ – all utilize the introsort algorithm or its optimized variant as the `std::sort` algorithm in their respective libraries. We can confirm this by examining their source code [13, 14, 15].

2.3 Quicksort Optimizations

In this section, we will explore the potential optimizations for quicksort to minimize its time complexity. Our research will encompass innovative techniques, such as minimizing recursive overheads and fine-tuning the algorithm for specific data patterns.

2.3.1 Recursion Elimination

In quicksort, eliminating recursion involves converting the algorithm's recursive nature into an iterative one. This technique can decrease the call stack size and potentially enhance robustness, especially for larger datasets where deep recursion can lead to stack overflow. One common strategy is to remove a single recursive call, as shown in Code 2.4. Doing so can reduce the number of recursive calls from $O(n)$ to $O(\log n)$ if the pivot is the median or close to it. Another improvement is always performing recursion on the smaller subsequence, ensuring that the recursion stack stays bounded by $log_2(n)$ even in cases where the pivot is skewed [16].

Algorithm 2.4 QuickSort Algorithm recursion elimination

```
procedure QuickSort(A, lo, hi)
    while lo < hi do
        pivotIndex ← Partition(A, lo, hi)
        QuickSort(A, lo, pivotIndex − 1)
        lo ← pivotIndex + 1
    end while
end procedure
```

2.3.2 Block Partitioning

One way to enhance the efficiency of novel quicksort implementations is by eliminating expensive branch predictions during the partitioning stage. This optimization technique is known as block partitioning and was showcased in the BlockQuicksort algorithm [17].

Block partitioning is a technique that replaces if constructs with data-dependent moves to eliminate branch mispredictions. The process involves two stages: scanning and rearranging. Two buffer offsets, namely $\texttt{offsets}_{\texttt{L}}$, and $\texttt{offsets}_{\texttt{R}}$, are used to store pointers for misplaced elements. These buffers have a static size of B. The block partitioning process begins with two pointers, one on each side, moving towards the middle. Elements under the pointers are compared to the pivot, and instead of halting at the first element requiring swapping, only a pointer (offset) is stored in the offset buffer, and the pointer continues moving toward the middle. The misplaced elements

are rearranged after scanning an entire block of B elements. This involves processing offset buffers and swapping the elements they point to until one of them no longer contains offsets. The scanning stage is then restarted, and the empty buffer is refilled.

The algorithm proceeds in this manner until fewer elements than twice the block size remain. At that point, the algorithm scans the remaining elements as one or two final blocks, which may be smaller, and carries out a final rearranging stage. Following this stage, one buffer may be empty while the other may still contain elements. By passing through the non-empty buffer, all of the corresponding elements can be shifted to the left or right, similar to the process used in the Lomuto partitioning method, but without actually performing comparisons. Code 2.5 and Figure 2.3 show the whole concept.

Algorithm 2.5 Block partitioning

```
procedure BlockPartition(A, lo, hi)
    pivot ← GetPivot(A, lo, hi)          ▷ Place pivot at the begging and return copy
    offsets_L[], offsets_R[], start_L, start_R, count_L, count_R ← 0
    while 2 * BlockSize > size ← hi − lo + 1 do
        if count_L == 0 then
            start_L ← 0
            for i ← 0 to blockSize do                    ▷ Scanning stage for left side
                offsets_L[count_L] ← i
                count_L ← count_L + (A[lo + i] ≥ pivot)
            end for
        end if
        if count_R == 0 then
            start_R ← 0
            for i ← 0 to blockSize do                    ▷ Scanning stage for right side
                offsets_R[count_R] ← i
                count_R ← count_R + (A[hi − i] < pivot)
            end for
        end if
        for i ← 0 to count ← min(count_L, count_R) do                ▷ Rearranging stage
            swap(A[lo + offsets_L[start_L+i]], A[hi - offsets_R[start_R + i]])
        end for
        count_L ← count_L − count; start_L ← start_L + count
        count_R ← count_R − count; start_R ← start_R + count
        if count_L == 0 then lo ← lo + BlockSize end if
        if count_R == 0 then hi ← hi − BlockSize end if
    end while
    PARTITION_REMAINING()    ▷ Process remaining elements using a similar concept
end procedure
```

This partition method effectively reduces the need for costly branch predictions. In contrast, the number of element accesses is doubled compared to the standard approach. However, if the block size B is chosen appropriately, the elements will still be stored in the L1 cache when swapped. Therefore, the extra scan has no impact on runtime at all. Edelkamp et al. [17, p.14] concluded, based on their experiments, that their final

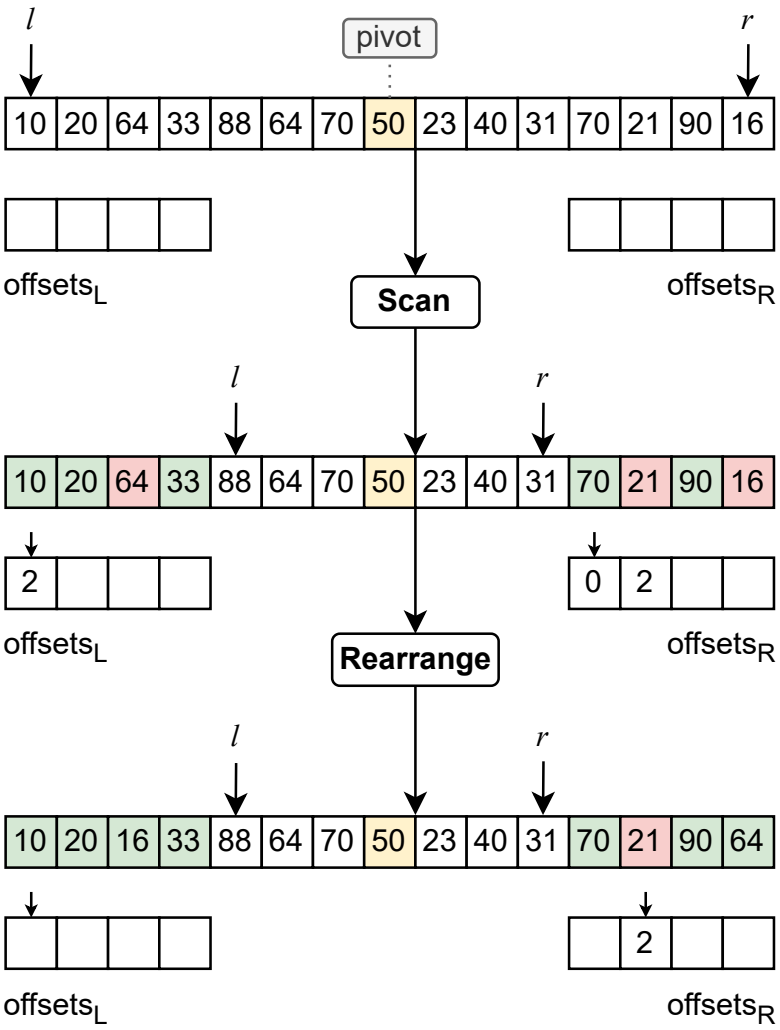


Figure 2.3 The figure above shows an example of the block partitioning process. The block size is $B = 4$, and the pivot has a value of 50. The left side of the block is first scanned for incorrectly positioned elements, with only one (64) being found and its offset stored in the $\texttt{offsets}_{\text{L}}$ buffer. The same scanning process is performed on the right side, revealing two incorrectly placed elements. The rearrangement stage involves swapping elements until one of the offset buffers is emptied. This buffer is then refilled, and the swapping process continues iteratively.

implementation of BlockQuicksort was 80% more efficient than GCC's `std::sort` at that time.

An additional method for optimization involves substituting swaps with cyclic permutations during the rearranging phase. Abhyankar et al. were the first to introduce this approach [18], which successfully eliminates the requirement for three swap codes and reduces the overall instruction code. Code 2.6 illustrates the optimized rearranging stage.

Algorithm 2.6 Cyclic permutation

Input: `A, lo, hi`: input array
Input: $\texttt{offsets}_\texttt{L}, \texttt{offsets}_\texttt{R}$: indices for misplaced elements
Input: `count`: number of elements to swap

```
temp ← A[lo + offsets_L[0]]
A[lo + offsets_L[0]] ← A[hi - offsets_R[0]]
for i ← 1 to count − 1 do
    A[hi - offsets_R[i - 1]] ← A[lo + offsets_L[i]]
    A[lo + offsets_L[i]] ← A[hi - offsets_R[i]]
end for
A[hi - offsets_R[count - 1]] ← temp
```

2.3.3 Quicksort Adaptivness

As mentioned in Chapter 1.1.3, adaptiveness refers to the ability to adjust to different permutations of input data and to identify familiar patterns more efficiently. However, detecting a pattern requires performing certain checks, which can be expensive and increase time complexity. Therefore, achieving a balance between the complexity of detection and the ability to adapt is crucial.

One of the most effective compromises is identifying a swap-free partition. Once the partitioning process is completed, we can verify if any elements were swapped. If no swaps occurred, we refer to this partition as *swapless*. This check can be performed simply by comparing pointers without significantly increasing the time complexity. If the partition is swapless, we can attempt a partial insertion sort on both partitions. This sorting algorithm evaluates the number of swaps required, and if it exceeds a small threshold, the regular sorting routine continues. However, if the sort successfully arranges the partitions with only minor corrections, we have a sorted sequence, and recursion is unnecessary. By appropriately selecting pivots, this approach can achieve linear-time complexity when sorting ascending or descending inputs, even when arbitrary elements are appended to the input sequence [19].

2.3.4 Detection of Duplicate Elements

The presence of duplicate elements can result in selecting the same pivot element as the previous one. As a result, an imbalanced partitioning occurs, leading to a higher recursion depth and eventually necessitating a switch to heapsort. Novel quicksort implementations address this inefficiency by detecting input data with many duplicate ele-

ments and taking advantage of it. In the following section, we will examine the methods utilized by BlockQuicksort [17] and pdqsort [19].

The BlockQuicksort algorithm incorporates a check to determine if the pivot occurs twice in the sample (in the case of a median of three) or if the partitioning is significantly unbalanced. In either case, it performs a scan after partitioning to identify duplicate elements that are equal to the pivot. The scan starts from the position of the pivot and examines the larger half of the input sequence. Any elements that are found to be equal to the pivot are moved to the side of the pivot. This scanning process continues as long as every fourth element equals the pivot. This ratio can be adjusted, but it ensures that the check stops quickly if there are only a few duplicates. After this check, any elements identified as equal to the pivot are kept in the middle of the array, effectively excluding them from future recursive calls [17, p.11].

The pdqsort algorithm functions differently. Before partitioning, it checks whether the pivot chosen is the same as the one chosen in the previous partitioning. This check only applies to subsequences, not on the input's far left. The method compares the current pivot to the element preceding the first element in the subsequence, which is the previous pivot. Pdqsort applies a slightly different partitioning technique to the subsequence if they are the same. This method is similar to the Hoare partition, except that identical elements are positioned in the left subsequence instead of the right. After partitioning, the left subsequence contains sorted elements identical to the pivot, eliminating the need for further recursion on that subsequence. This concept is illustrated in Figure 2.4. Orson demonstrated that this technique has a time complexity of $O(k \cdot n)$, where k is the number of distinct values in the input. It should be noted that this is an upper bound, and time complexity $O(n \cdot log(n))$ applies when k is large. Please see the original paper [19] for comprehensive proof.

2.3.5 Optimizing Small Sorts

When selecting a pivot as the median of medians of x elements, the algorithm relies on a quick sorting function for a small fixed number of elements. For example, when choosing a pivot as the median of 3 medians, the algorithm requires a fast sorting algorithm for three elements, which we will refer to as `sort3`. Sorting networks is a practical approach to implementing these small sorts. These networks consist of a fixed number of wires that transport values and comparator modules that link pairs of wires. The comparator modules exchange the values on the wires if they are not in the desired order [20].

Sorting networks can be evaluated using two metrics: size and depth. Size refers to the number of compare-and-swap operations required, while depth measures the number of parallel operation steps. Regarding visualization, compare-and-swap operations can be seen as wires in the network diagram, while depth corresponds to vertical layers. When sorting three elements, three compare-and-swap operations are needed. For four elements, five operations are required, and for five elements, nine operations are needed [21, pp. 219-247]. Figure 2.5 depicts the three essential sorting networks. For larger inputs, the complexity increases. The optimal size of the sorting network was proven for the input of a maximum of 12 elements [22]. The proof for bigger inputs remains an open question.

Google recently conducted groundbreaking research that focused on improving the

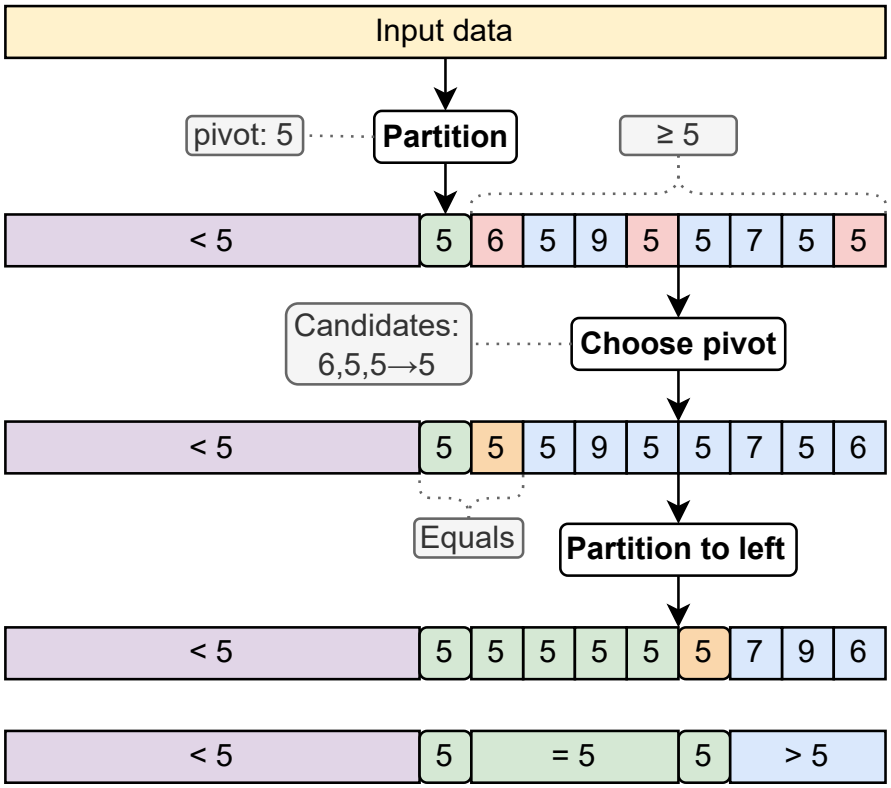


Figure 2.4 Partition to the left. First, the partitioning process follows the usual procedure of dividing the input data into three segments: those lower than the pivot, the pivot itself, and those greater than or equal to the pivot. During subsequent partitioning of the right side, the same pivot is selected. In that case, we resort to partitioning to the left instead of the classic partition. Elements that are equal to or lower than the pivot are placed to the left; however, no lower elements are present, as they were already swapped in the initial iteration. Consequently, after this partitioning step, the data will be segregated into a left segment that contains elements equal to the pivot, the pivot itself, and a right segment containing elements larger than the pivot. In the subsequent iteration, there is no need to recursively partition the left side, as it only consists of elements equal to the pivot.

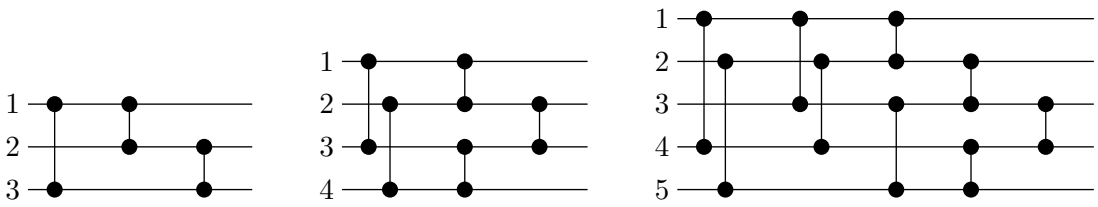


Figure 2.5 The optimal sorting networks for three, four, and five elements have sizes of three, five, and nine, and depths of three, three, and five, respectively.

efficiency of small sorting algorithms, such as `sort3`. Using their AlphaDev AI, they identified a faster `sort3` algorithm by minimizing the number of assembly instructions needed. AlphaDev is an extension of the renowned AlphaZero, which has already proven its superiority by defeating human world champions in various games, including chess and Go. The procedure followed by AlphaDev was uncommon. Instead of starting with C++ code and translating it into assembly instructions, they took a reverse approach. They iteratively explored various instructions to find the code with minimal assembly instructions. The resulting code was then translated into C++ and implemented in the LLVM framework [23, 24].

According to the authors of Google research [23], they stated that their `sort3` algorithm is both the fastest and the shortest, consisting of only 17 assembly instructions, and no other faster algorithm exists. However, subsequent studies have introduced shorter sorting algorithms (e.g., only 15 instructions) that claim to be slightly faster [25, 26]. Notably, these sorting algorithms enhance the time complexity by a mere few nanoseconds (which is understandable for such a small algorithm). However, they are usually implemented in assembly language, making it challenging to write equivalent C++ code independent of the processor architecture.

2.3.6 Switch to Heapsort

The conventional method for deciding when to transition to heapsort involves tracking the recursion depth. Nevertheless, an alternative method involves monitoring the sizes of the partitions. The pdqsort algorithm utilizes this technique [19, p.6].

Let us define a *bad partition* as any more unbalanced partition than $p = 0.125$, where p is a percentile of the pivot, e.g., $p = 0.5$ for perfect partition. If $p = 0.125$, the pivot position after partitioning is below the 12.5% percentile or above the 87.5% percentile. At first, we initialize a counter to $\log n$, and whenever we encounter a bad partition, we reduce the counter. If the counter reaches 0 at the beginning of a recursive call, we use heapsort to sort that subsequence. The $p = 0.125$ is not chosen arbitrarily. Our goal is to determine an optimal value for p that will enhance the acceleration of the process when transitioning to heapsort, ensuring that the switch to heapsort is not made too early. The author of pdqsort measured that heapsort takes approximately twice as long as quicksort when sorting randomly shuffled data. Therefore, it is preferred to switch to heapsort if the unbalanced partitions will cause quicksort to slow down by roughly twice as much. Suppose we imagine a situation in which the quicksort partition consistently places pn elements in the left partition and $(1-p)n$ in the right partition. In that case, we can characterize its runtime using the following recurrence relation:

$$T(n,p) = n + T(pn,p) + T((1-p)n,p)$$

The best-case scenario is when $p = 0.5$, while the worst-case scenario is when $p = 0$ or $p = 1$. The Akra-Bazzi theorem proves that for any p in the range of $(0,1)$, the function $T(n,p)$ has a behavior of $\Theta(n \log n)$ [27]. Moreover, by resolving the recurrence [28], we can explore the deceleration of quicksort in contrast to the optimal scenario:

$$\lim_{n\to\infty} \frac{T(n,p)}{T(n,\frac{1}{2})} = \frac{1}{H(p)}$$

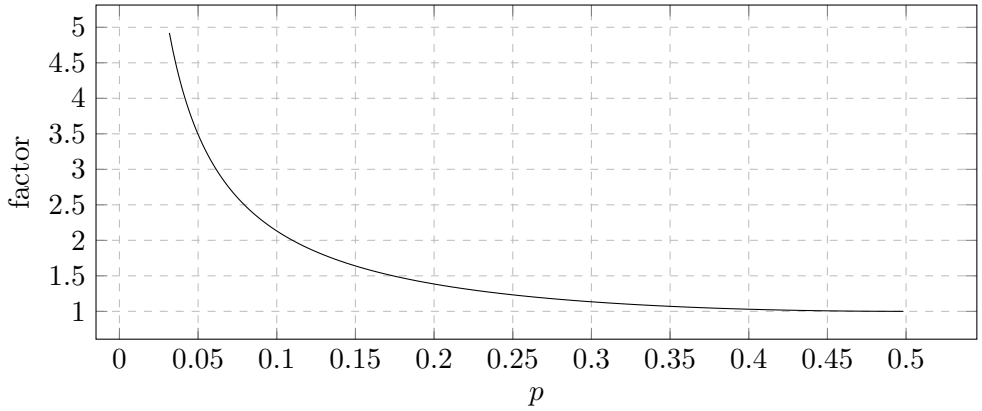


Figure 2.6 The graph shows the decrease factor in the time complexity of $T(n, p)$ for different values of p. It nicely illustrates the reason for the overall speed of quicksort. For example, when we partition the data into partitions with size ratios of 80/20, that means the $p = 0.2$, the execution speed is only 40% slower than the optimal scenario ($p = 0.5$). Moreover, it is essential to note that the difference between a high-quality pivot and an average one is slight, leading to only a small advantage gained from consistently choosing a superior pivot. Conversely, the contrast between an average and poor pivot is significant. This figure was taken from pdqsort paper. [19, p.7].

where H denotes Shannon's entropy function. Plotting this function offers a better understanding of the essential performance characteristics of quicksort, as shown in figure 2.6.

If we select a value p such that $H(p)^{-1} = 2$, a bad partition will result in poorer performance than heapsort. The choice of p can be adjusted accordingly for various system architectures or different worst-case sorting algorithms. The pdqsort algorithm selected $p = 0.125$ due to its proximity to being twice as slow and because only simple bit shifts are needed to calculate this value. This approach is more precise than introsort's fixed logarithmic limit on recursive calls to prevent worst-case scenarios. Peterson observed that introsort might encounter difficulties sorting inputs with unfavorable patterns, which will diminish after a few partition procedures. Subsequently, his method switches to utilizing the faster quicksort for the remaining sorting tasks, whereas introsort overly emphasizes the initial challenges and reverts to using heapsort [19].

Specific input patterns exhibit a self-repeating structure upon partitioning, leading to repeated selection of a similar pivot. Traditionally, quicksort algorithms address this issue by introducing randomness in pivot selection. However, pdqsort offers an alternative strategy. Following the partitioning step, it evaluates whether the partition is *bad*, and if so, it deterministically swaps future pivot candidates with other elements to disrupt specific input patterns.

2.4 Parallel Quicksort Algorithm

This section will explore the possibilities for parallelizing the quicksort algorithm. It will provide a concise overview of how quicksort can be parallelized and give an idea of incorporating and parallelizing the optimizations discussed in Chapter 2.3. Parallelization of

these optimizations is explored in detail in Chapter 4. Quicksort can be parallelized by parallelizing the recursive calls, the partitioning process, or ultimately by parallelizing both.

2.4.1 Parallel Recursive Calls

In the sequenced quicksort algorithm, the left subsequence is processed first, followed by the right subsequence, after the partition step. It is important to emphasize that sorting these subsequences is entirely independent. Therefore, they can be sorted in parallel without waiting for one another. Parallelizing the recursive call is straightforward, direct, and efficient. It can be easily accomplished in OpenMP using task parallelism. The code 2.7 demonstrates the idea. It is also needed to determine the appropriate threshold for transitioning to the sequenced quicksort through empirical experiments.

However, this parallelization approach does not fully exploit the capabilities of multi-core architecture. Initially, only one core is utilized, and as more parallel calls are made over time (following the binary tree structure), additional cores are engaged. However, the partition step, the most complex aspect of the quicksort algorithm, is not parallelized. Therefore, it is crucial to parallelize the partition process to fully exploit the power of multi-core CPUs [29].

Algorithm 2.7 QuickSort: Parallel Recursion

```
procedure PARALLEL_QUICKSORT(A, lo, hi)
    while lo < hi do
        if hi − lo + 1 < threshold then
            return QUICKSORT(A, lo, hi)
        end if
        pivotIndex ← PARTITION(A, lo, hi)
        Parallel call: PARALLEL_QUICKSORT(A, lo, pivotIndex − 1)
        lo ← pivotIndex + 1
    end while
end procedure
procedure QUICKSORT(A, lo, hi)
    while lo < hi do
        pivotIndex ← PARTITION(A, lo, hi)
        QUICKSORT(A, lo, pivotIndex − 1)
        lo ← pivotIndex + 1
    end while
end procedure
```

2.4.2 Parallel Partition

Both in-place parallel partitioning algorithms and out-place algorithms exists. However, a significant number of out-place algorithms are commonly derived from the initial method developed by Guy Blelloch [30], which relies on prefix scan[1] and requires a mini-

[1]Other terms such as cumulative sum or sum scan are also utilized in the literature.

mum of $O(n)$ additional memory. Moreover, these algorithms are primarily designed for GPU implementations [31], and therefore we will not go into their examination further.

On the contrary, in-place partitioning techniques strive to perform partitioning directly within the original data array without requiring auxiliary memory. The most known and used in-place parallel partitioning techniques are strided, F&A and technique from the The Multi-core Standard Template Library (MCSTL)[2]. The F&A parallel partition technique was introduced by Tsigas et al. [32] as an advance of the earlier strided parallel partition method proposed by Francis and Pannan [33]. Johannes Singler et al. [34] introduced the MCSTL method, similar to the F&A technique. All these parallel partition algorithms involve two stages: the main stage and the cleanup stage.

Strided and Blocked Partition

Strided partitioning involves dividing the input data into p parts, where p represents the number of threads. Subsequently, each thread processes elements at intervals of p, implying that the i th thread will handle elements at positions $i + 0$, $i + p$, $i + 2p$, and similarly. Once each thread sequentially partitions its elements, it returns a position that separates the elements for that thread (referred to as a pivot for those elements). Following this, these returned positions' minimum and maximum values determine the interval, which may be unsorted. Consequently, this unsorted segment (denoted as dirty) is sequentially partitioned. The time complexity of the main stage is $O(n/p)$, and the time complexity of the cleanup stage is $O(1)$ in the average case, but $O(n)$ in the worst case [33].

One drawback of this algorithm is the inefficient cache locality, as all threads tend to access the sequence over its entire length. The algorithm can be extended to work with blocks to enhance cache efficiency. Instead of assigning individual elements to each thread, coherent blocks of elements are statically assigned. The size of the block is represented by b, and if $b = 1$, the algorithm is called strided; otherwise, it is considered blocked [35]. Figure 2.7 illustrates the procedure of strided partition, while Figure 2.8 depicts the process of block partition.

F&A and MCSTL Partition

The F&A (Fetch and Add) algorithm is an improved parallel strided/blocked partitioning technique. In this approach, threads dynamically fetch new blocks of elements from both sides as needed rather than being statically assigned specific blocks of elements. Initially, each thread gets two blocks: one from the beginning and one from the end. The threads then perform the partitioning process on their blocks. Once a block is fully processed, meaning all its elements are either lower respective greater/equal to the pivot, the block is called *neutralized*. The thread will then fetch a new block from the side where the neutralized block was located. This algorithm is named fetch and add because it relies on the atomic fetch-and-add instruction. All threads share left- and right-hand pointers, which mark the boundary between the segment with assigned blocks and the segment with unprocessed blocks. When a thread needs to fetch a new block, it must read the current pointer value (indicating the start of the new block) and increment the pointer by

[2]This library is now outdated.

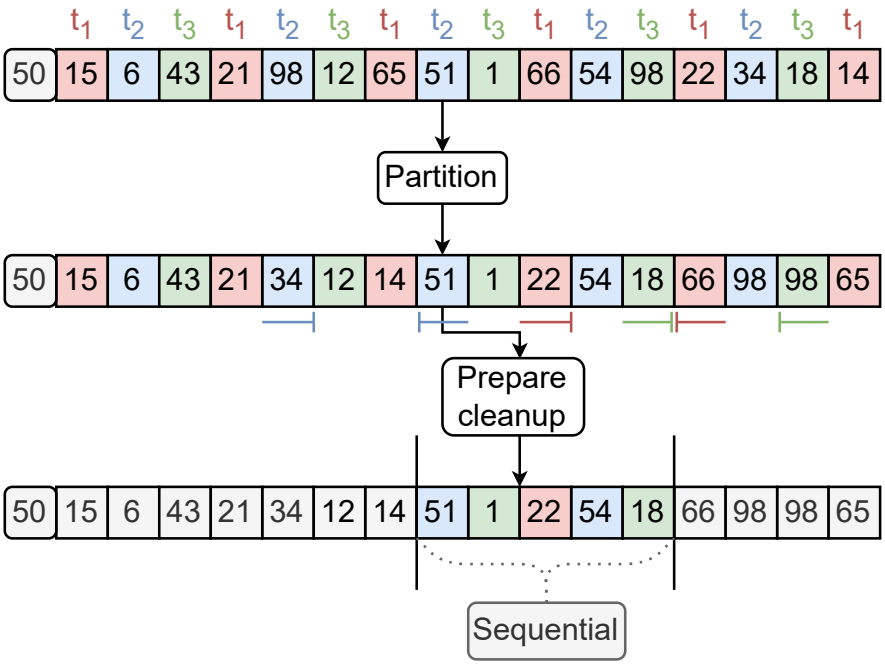


Figure 2.7 Parallel strided partition. We use three threads and choose 50 as the pivot, positioning it at the beginning. Following the main stage, the boundaries of the dirty segment are determined, allowing for subsequent sequential cleaning.

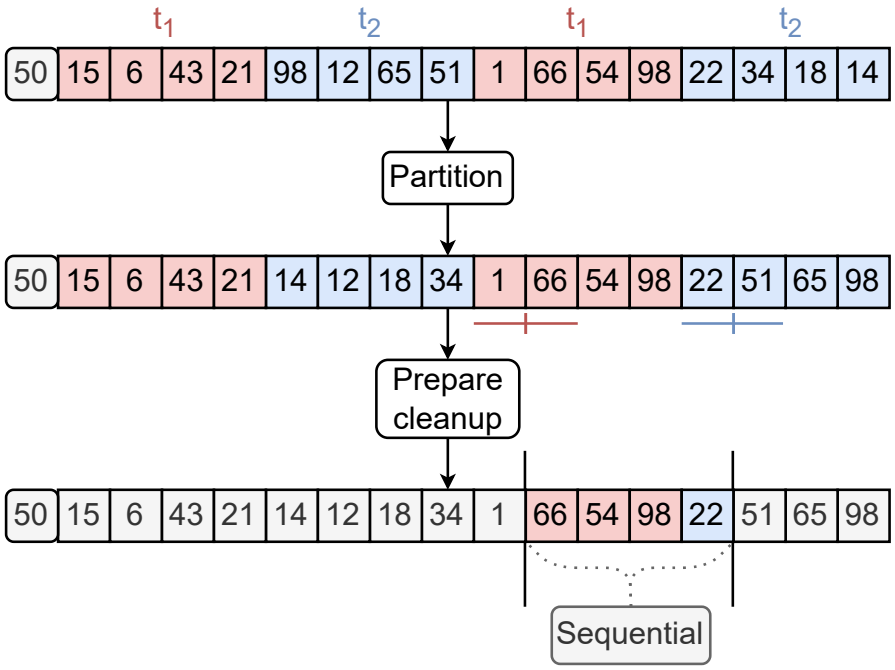


Figure 2.8 Parallel blocked partition. We use two threads, block size $b = 4$, and choose 50 as the pivot, positioning it at the beginning. Following the main stage, the boundaries of the dirty segment are determined, allowing for subsequent sequential cleaning.

the block size to identify the elements taken. The fetch-and-add instruction is perfectly suitable for this purpose [32].

Following the main stage, every thread may have one dirty block, leading to a potential maximum of p dirty blocks before the cleanup stage. These blocks are then swapped in the middle of the whole sequence, creating a new continuous sequence, which is then processed by a single thread.

The sole distinction between F&A and MCSTL lies in their approach to cleanup. While F&A sequentially conducts the cleanup, MCSTL recursively applies the parallel partition method to the new sequence with fewer threads. If there is only one block left or just one thread accessible, the remaining partitioning is executed sequentially. The time complexity of the main stage in the F&A and MCSTL algorithms is $O(n/p)$, while the cleanup stage requires $O(b \cdot p)$ for F&A and $O(b \cdot \log p)$ for MCSTL.

Cleanup Phase Improvements

The cleanup stage is a topic of further analysis and research, as all these partition algorithms disregard part of the completed work in the main stage, leading to redundant comparisons. L. Frias et al. [35] introduced a parallel partition algorithm that achieves minimal comparisons. Its improvement lies in the enhanced cleanup stage.

Let us define *Frontier* as a pointer that separates the processed part within a block from the unprocessed part. Let us also define *misplaced elements* as unprocessed elements that are not in the middle and processed elements that are in the middle. Therefore, a *misplaced block* is defined as a block that includes at least one misplaced element. Then m represents the total number of misplaced elements, and M represents the number of misplaced blocks. It is important to note that the maximum number of misplaced blocks is $2p$, since after the main stage, there could be a maximum of p dirty blocks, all of which may not be placed correctly, resulting in $2p$ misplaced blocks.

Following the main stage, a shared binary tree is created which is accessible to all threads and holds information about incorrectly positioned blocks. It includes the number of misplaced elements before the frontier, the number of misplaced elements after the frontier, the total count of elements before the frontier, and the total count after the frontier. The internal nodes accumulate information from the children. This shared tree can then be used to determine exactly which elements need to be swapped without making any extra comparisons.

This unique cleanup procedure reduces the parallel time of the F&A algorithm from $O(b \cdot \log p)$ to $O(\log^2 p)$ while minimizing the number of comparisons. However, the experimental findings by L. Frias et al. [35] indicate that the practical advantages of their cleanup method are limited. This is primarily due to fact, that usually the number of incorrectly positioned elements is minimal. Furthermore, other studies have noted that this particular cleanup stage has not resulted in acceleration [36, p. 26]. Thus, the optimal approach to conducting a cleanup stage without redundant comparisons and with improved time complexity remains an open question.

Chapter 3
Existing Implementations

Initially, Tsigas et al. [32] introduced a highly effective parallel quicksort algorithm that utilizes the F&A partition method. Subsequent algorithms have drawn inspiration from this approach. Nowadays, many parallel quicksort implementations in C++ exist. However, many of the implementations available in public repositories lack efficiency, often due to their utilization of sequential partitioning, resulting in low sorting performance and scalability. Moreover, many of these algorithms require non-standard C++ language extensions such as OpenMP [37] or external libraries like Threading Building Blocks (TBB) [38] or Boost [39]. We have discovered only two effective quicksort implementations that were developed without utilizing external libraries or non-standard extensions: cpp11sort [40] and poolstl's sort [41].

This section will outline the distinctions among various methods for parallelizing the code. Subsequently, we will examine different parallel quicksort implementations that have been identified, as well as explore specific sequential approaches that have the potential to be adapted into efficient parallel algorithms.

3.1 Parallelization Methods

Three primary options to parallelize an algorithm are: Open Multi-Processing (OpenMP), OneAPI Threading Building Blocks (oneTBB), and C++ threading. Each of these varies in their level of abstraction and specific implementation details.

- **OpenMP** is a non-standard extension for C/C++ and Fortran languages that facilitates the straightforward parallelization of algorithms through preprocessor pragmas. It allows for parallelization at a high level of abstraction, making it easy to implement without needing to work with low-level primitives. It also includes a library that offers additional techniques for precisely synchronizing threads and similar tasks [37].
- **oneTBB** is a third-party library created by Intel. It is built upon the deprecated Threading Building Blocks (TBB) and is largely compatible with its source code. Thus, software initially created using TBB typically works smoothly with oneTBB. This library offers a range of functions, interfaces, and classes for parallelizing code

and enhanced control over aspects such as thread creation and scheduling. Nonetheless, it is necessary to write code that is tailored for oneTBB constructs [38].

- **C++ threading** represents the third method. It allows for the parallelization of code solely utilizing features from the C++ standard library without depending on external resources. This choice provides extensive control over almost all multithreading aspects, including thread scheduling and atomic operations. Nevertheless, it demands considerable development effort and expertise. However, employing low-level primitive operations enables precise control over operations, while the library provides some decent abstraction. Additionally, it is possible to create custom classes and constructs to accomplish a high level of abstraction.

In summary, both OpenMP and oneTBB offer a straightforward approach to parallelizing code, albeit at the cost of relying on external extensions and libraries. Consequently, users are responsible for ensuring the installation of these dependencies to utilize the implementation. Conversely, parallelizing the implementation solely with C++ features demands extra work but presents the most convenient choice for end users.

3.2 Parallel Mode of Libstdc++

Libstdc++ implements the GNU C++ Standard Library, which includes a parallel mode [42] providing parallel algorithms, such as two variations of parallel quicksort. It should be noted that this parallel mode uses the Open Multi-Processing (OpenMP) extension. The quicksort implementation comes in two versions: the *unbalanced* and *balanced* variants. In the unbalanced version, a reasonable number of elements (usually 100 by default) are utilized to choose the pivot. Subsequently, the thread counts are evenly allocated to each subtask following the partitioning step. Conversely, the balanced quicksort algorithm only selects the pivot as the median of three elements. After partitioning, it assigns threads to the new subtasks proportionally to the lengths of their subsequences.

3.3 GCC's and Clang's PSTL

Per the C++ standard, versions C++17 and above must include support for parallel algorithms [43]. This entails the addition of new versions of standard library algorithms template functions that take an Execution Policy as their first argument. These policies dictate how the algorithm can be parallelized, such as enabling thread usage, vectorization, or potentially executing on a GPU. These policies can be supplied by the compiler or by linked third-party libraries. However, the level of compiler support differs. According to the information from cppreference [44], libstdc++ requires linking with TBB to activate parallel algorithms. Alternatively, the libc++ library offers limited support for parallel algorithms and requires linking with an experimental flag. We also performed tests using various setups and verified that libstdc++ supports parallel sorting algorithms when correctly linked with TBB. In contrast, libc++ does not compile without the experimental flag, and when used, the speed of `std::sort` with the parallel execution policy remains the same as with the sequential policy, regardless of whether it

is linked with TBB or OpenMP. Furthermore, the documentation states that no guarantees of API or ABI stability are given for experimental features. Our experiments employed Clang and libc++ version 17.0.1, indicating that parallel functionalities will likely be increasingly accessible in future releases.

The parallel algorithms in libstdc++ come from the intel library, which has been partially merged to the libstdc++ [45, 46]. We have examined the source code of libstdc++ to identify the section related to parallel sorting. The code reveals [47] that the concurrent `std::sort` utilizes a parallel stable sorting method, which involves generating a task tree and performing a parallel merge, as proposed by A. Robinson [48]. Therefore, it is not the quicksort algorithm.

3.4 CPP11Sort

CPP11sort is a highly effective implementation of the multithreaded quicksort algorithm. One of its key benefits is that it does not need external libraries or non-standard language/compiler extensions. The implementation is header-only and complies with the C++11 standard. Based on their benchmark findings, it offers better sorting performance than the main existing implementations of GNU, Intel, and Microsoft.

Even after conducting a comprehensive examination, we did not discover any comparable implementations that are as efficient as the top three existing implementations and do not depend on external third-party libraries or non-standard extensions. The implementations we discovered that did not rely on external libraries were ineffective. Although we did find effective implementations, they depended consistently on TBB or OpenMP or similar non-standard libraries/extensions. Consequently, the cpp11sort is a distinctive approach that combines portability and efficiency [40].

3.5 Other Parallel Sorting Algorithms

Several other well-known parallel sorting algorithms were discovered and examined. The most significant ones we found are as follows:

- **oneTBB:** library offers parallel sorting functionality [49], which can be accessed through a function template `tbb::parallel_sort`. Upon inspecting the source code [50], it becomes evident that `tbb::parallel_sort` internally employs parallel quicksort. Nevertheless, the quicksort algorithm depends on the specialized thread pool and task scheduling mechanisms provided by this external library.

- **poolSTL:** This compact library focuses on implementing parallel standard library algorithms conforming with the C++17 standard. While it does not provide an implementation for all functions, it is easy to integrate, has no external dependencies, and includes a parallel quicksort implementation for `std::sort` along with parallel partitioning [41].

- **Thrust:** Thrust is a C++ library created by NVIDIA. It is part of CUDA C++ Core Libraries (CCCL) and is commonly employed to facilitate performance portability between GPUs and multicore CPUs. Thrust also offers parallel versions of standard

algorithms such as `std::sort`. Although some adjustments may be necessary, it is a header-only library that should be compilable without requiring NVIDIA Cuda Compiler (NVCC). Also, it is crucial to note that for enabling parallel algorithms, some parallel programming framework is needed, such as OpenMP, TBB or CUDA [51].

- **Parallel Patterns Library (PPL):** Microsoft Visual C++ (MSVC) incorporates PPL, which implements parallel algorithms in accordance with the C++17 standard. Nevertheless, it is integrated with Microsoft's development tools and OS, limiting its use in other environments [52].

- **Boost:** Boost is an extensive collection of libraries designed for the C++ programming language, offering a wide range of functionalities, such as multithreading support. It also provides a parallel sorting implementation called `block_indirect_sort`. A small auxiliary memory of constant size is required for this merging algorithm [53]. However, the code is based on a multithreading framework within the Boost library.

- **AQsort:** AQsort is a parallel quicksort algorithm that allows the user to specify a function for swapping elements. This feature enables the sorting of multiple datasets, such as arrays, simultaneously [29]. Before C++23, this functionality was a notable advantage that was not available in other implementations. With the introduction of C++23, the `std::ranges::views::zip` can partly replace this capability.

- **MPQsort:** MPQsort is an innovative parallel quicksort algorithm that differs from traditional implementations by using multiple pivots for parallel partitioning. This unique approach sets MPQsort apart as the first algorithm of its kind [4]. The provided implementation is contained within a single header file, making it straightforward to integrate. Nevertheless, the parallelization is achieved using non-standard extensions OpenMP [54].

- **IPS^{4}o:** IPS^{4}o stands for in-place Superscalar Samplesort. This comparison-based algorithm incorporates numerous enhancements, such as dynamic load balancing and reducing branch mispredictions by using a branchless decision tree. It uses OpenMP if available. Otherwise, it uses C++ threads. However, in both cases, it still depends on a third-party library. It must be linked against TBB and against GCC's libatomic to enable 16-byte atomic compare-and-exchange instructions. Although it is not a quicksort algorithm, we included it in our extended benchmarks due to its reputation as one of the most efficient parallel comparison-based algorithms [55].

3.6 pdqsort

The Pattern-defeating Quicksort (pdqsort) is an optimized sequential quicksort algorithm. It uses a variant of block partitioning as described in Chapter 2.3.2. However, the implementation applies this specific partitioning technique exclusively when the input elements are native numeric types, and the comparison function is `std::less` or a similar function. Otherwise, a classic Hoare partitioning technique is used. Additional optimizations include the removal of recursion (Chapter 2.3.1), adaptiveness to input

Name	Algorithm	Memory usage	External dependencies	Highlight
PPQSort	quicksort	in-place	**None**	Parallel Pattern Quicksort
GCC `std::sort`	merging algorithm	out-place	oneTBB	Ported from oneTBB
Libstdc++ (unbalanced)	quicksort	in-place	OpenMP	evenly splitting threads
Libstdc++ (balanced)	quicksort	in-place	OpenMP	allocating threads proportionally to subtask sizes
CPP11Sort	quicksort	in-place	**None**	Header-only, C++11 compliant
oneTBB sort	quicksort	out-place	oneTBB	Splits input to small tasks
poolSTL	quicksort	in-place	**None**	Header-only, C++17 compliant
Thrust	k-way mergesort	out-place	CUDA, OpenMP, oneTBB (one required)	Thrust internal implementation
PPL	quicksort	in-place	MSVC	Microsoft Parallel Patterns Library
Boost block indirect sort	merging algorithm	out-place	Boost	Upper bounded small memory usage
AQsort	quicksort	in-place	OpenMP	Allows the sorting of multiple datasets at once
MPQsort	quicksort	in-place	OpenMP	Multiway parallel Quicksort algorithm
IPS^{4}o	Samplesort	in-place	oneTBB	Divides data into buckets and sort them recursively

Table 3.1 Comparison of existing implementations

permutations (Chapter 2.3.3), which involves the identification of numerous duplicate elements (Chapter 2.3.4) and also preventing the worst case by identifying bad partitions (Chapter 2.3.6).

The pdqsort has demonstrated notable speed and efficiency in various benchmark evaluations. We have also examined several well-known quicksort implementations by Igor van den Hoven, such as crumsort [56]. As their benchmarks indicate, he has created a diverse collection of rapid and intriguing sorting algorithms. Unfortunately, the code is written in C rather than C++, offering an old C API. Understanding the source code can sometimes be challenging, and technical papers for these sorting methods are unavailable. However, these algorithms have introduced some innovative and fascinating techniques, including a stable quicksort known as fluxsort. We observed that these algorithms perform well on small arrays (up to $1e5$ elements), but for larger arrays (up to $1e9$), pdqsort tends to outperform them. Exploring the parallelization of these algorithms in the future would be intriguing; however, for now, we have chosen to develop a quick and efficient parallel sorting algorithm called Parallel Pattern Quicksort (PPQSort), drawing inspiration from pdqsort.

Chapter 4

PPQSort (Parallel Pattern Quicksort)

The aim was to consolidate all the optimization methods identified in the study and develop an effective and fast parallel quicksort algorithm. This section introduces our approach – Parallel Pattern Quicksort (PPQSort), based on the introsort algorithm. The Parallel Pattern Quicksort draws inspiration from the sequential sorting technique of Pattern-defeating Quicksort and integrates various improvements from it. It also draws inspiration from CPP11sort in terms of parallelization. We developed two implementations in C++. One is based on the OpenMP framework, while the other can be compiled without external libraries, using only the standard C++20 library. This chapter will describe the design of the PPQSort algorithm and discuss both implementations. Code 4.1 illustrates the general structure of the algorithm.

4.1 Thread Balance

The implementation of PPQSort relies on a thread pool. Initially, it generates p threads, where p equals the number of CPU cores. Subsequently, the threads await tasks and are executed from the thread pool. Firstly, the primary thread initiates the algorithm and generates a new task. Subsequently, two threads are active, each creating another task, and so forth. The challenge lies in achieving parallel partitioning effectively, as we aim to maximize CPU usage, while an excessive number of concurrent threads could result in performance degradation.

The main concept is to ensure that we do not exceed the number of CPU cores with the threads that we use. Specifically, the goal is to utilize p threads consistently. For example, if CPU has 16 cores, initially, all 16 threads should be used for partitioning. However, in subsequent recursive calls, with two threads already executing the algorithm's main loop, the aim is to limit the partitioning to only eight threads for both subtasks. In our research, we have analyzed various implementations, such as AQsort, which balance thread usage based on the lengths of the subsequences generated by partitioning. In our experiments, we did not observe any speedup for our implementation when modifying thread distribution; in some cases, the opposite effect was noted. Hence,

Algorithm 4.1 Parallel Pattern Quicksort Algorithm

```
procedure PARALLEL_PPQSORT(A, lo, hi, bad_allowed, threads)
    while true do
        size ← hi − lo
        if size < seq_threshold then
            return SEQ_PPQSORT(A, lo, hi)
        end if
        CHOOSE_PIVOT(A, lo, hi)                    ▷ will put pivot at the begining
        if sorting right side and pivot same as previous then
            pivotIndex ← PARTITION_LEFT(A, lo, hi)
            lo ← pivotIndex + 1 and continue
        end if
        pivotIndex ← PARTITION_RIGHT(A, lo, hi)
        if swappless partitioning and CHECK_SORTED(A, lo, hi) then return end if
        if l_size < size/ratio or r_side < size/ratio then
            if bad_allowed == 0 then return HeapSort(A, lo, hi) end if
            bad_allowed ← bad_allowed − 1
            Shuffle
        end if
        threads ← threads/2
        Add task: PARALLEL_PPQSORT(A, lo, pivotIndex − 1, bad_allowed, threads)
        lo ← pivotIndex + 1
    end while
end procedure
procedure SEQ_PPQSORT(A, lo, hi, bad_allowed)
    while true do
        size ← hi − lo
        if size < insert_threshold then return INSERTION_SORT(A, lo, hi) end if
        CHOOSE_PIVOT(A, lo, hi)
        if sorting right side and pivot same as previous then
            pivotIndex ← PARTITION_LEFT(A, lo, hi)
            lo ← pivotIndex + 1 and continue
        end if
        pivotIndex ← PARTITION_RIGHT(A, lo, hi)
        if swappless partitioning and CHECK_SORTED(A, lo, hi) then return end if
        if l_size < size/ratio or r_side < size/ratio then
            if bad_allowed == 0 then return HeapSort(A, lo, hi) end if
            bad_allowed ← bad_allowed − 1
            Shuffle
        end if
        SEQ_PPQSORT(A, lo, pivotIndex − 1, bad_allowed)
        lo ← pivotIndex + 1
    end while
end procedure
```

PPQSort does not distribute threads fairly but instead allocates half of the threads to each created subsequence. The division by 2 can also be quickly calculated using bit shifting.

Sorting small subsequences concurrently, even though they could be sorted more effectively sequentially, is not preferred. Therefore, initially, we verify the size of the subsequences to ensure that they are not too small. We sort them using the sequential version if they are too small. The sequential threshold for transitioning to the sequential version must be determined thoughtfully to achieve an optimal balance in the distribution of thread workload. We could use this formula:

$$\texttt{seq\_thr} = \frac{n}{p}$$

However, in that scenario, the threshold is too large. In such cases, certain threads could hinder performance as they work on sorting a lengthy and intricate subsequence while others have finished their tasks and are just waiting for the slower thread to complete. The PPQSort employs an improved formula to determine the sequential threshold:

$$\texttt{seq\_thr} = \frac{n}{p} \cdot \frac{1}{k}$$

In this scenario, $k > 1$ represents a small value determined through empirical experiments. This strategy will result in a situation where there are many more tasks in the thread pool than threads, leading to a more evenly distributed workload among the threads.

4.2 Pivot Selection

Efficient and rapid selection of a pivot is crucial, requiring a balance between the quality of the pivot and the speed of its selection. Figure 2.6 illustrates that the distinction between a superior pivot and an average one is slight. Therefore, investing time in selecting the optimal pivot is not justified. Hence, the PPQSort algorithm selects pivots based on the median of three medians, with each median being determined from three elements. These medians are computed within the array itself: the first one is derived from the first, middle, and last elements, the second one from the second element, the one after the middle, the one before the last element, and so forth. In the end, the medians will be positioned in the middle; we will sort them and select the middle element as the pivot. If the input array is small, we will use the median of three elements (first, middle, and last).

Efficiently sorting three elements is crucial, as this operation is repeated frequently when selecting a pivot. We have developed two versions of the function `sort3`. One is based on the optimized `sort3` from Google AlphaDev, while the other is a custom implementation. The key distinction is that the first relies solely on conditional moves, eliminating the need for the compiler to generate branches in the code. However, it always performs three comparisons. On the other hand, custom `sort3` will result in branch code. However, in the best-case scenario, it only requires two comparisons, making it beneficial when the comparison function is expensive.

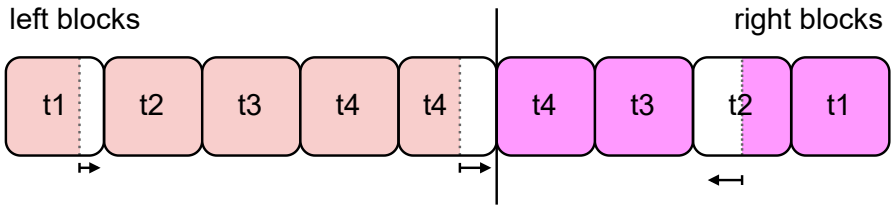


Figure 4.1 End of the main stage. The main stage of parallel partitioning has been completed. All threads have finished their assignments, and no more blocks are left. As a result, the algorithm moves on to the cleanup stage.

We also implemented a method for selecting the pivot as the median of five medians. Nevertheless, this strategy did not enhance efficiency, prompting us to use three medians instead. The decision of whether to use the branchy or branchless version of the sorting algorithm is made during compilation, although users also have the option to specify their preferences. Further details can be found in Chapter 4.8.

4.3 Parallel Partitioning

Similar to `sort3`, we have developed two variations of partitioning. The first one involves branching and is founded on the F&A algorithm. The second version is a parallelized branchless partitioning method adapted from the sequential block partitioning method discussed in Chapter 2.3.2.

4.3.1 Fetch and Add Method

We employ F&A partitioning as detailed in Chapter 2.4.2. Initially, we assess whether the input data are sufficiently large to enable parallel partitioning, ensuring a minimum of two blocks per thread. If this criterion is not met, we will transition to sequential partitioning. The optimal block size B was empirically determined in Chapter 5.3. Parallel partitioning starts by initially assigning two blocks to each thread to avoid collisions during startup. Following this, the threads will continue to process and fetch subsequent blocks until the entire input sequence is processed (see Figure 4.1).

Subsequently, there may be some dirty blocks present. Each thread will check if the blocks it last processed are dirty, which will determine the total number of dirty blocks on each side. This information allows us to identify the separators that divide the data into separate sections, specifically clean and dirty segments, as shown in Figure 4.2.

We manage an array called *reserved* that is used to reserve a spot in the dirty segment. Each thread containing at least one dirty block will verify whether its dirty blocks are positioned correctly, specifically within the middle dirty segment. If so, they will update the reserved array with this information. Subsequently, the reserved array enables threads with dirty blocks in clean segments to locate suitable clean blocks in dirty segments and exchange them. This operation is illustrated in Figure 4.3. Following this, we proceed to process the dirty blocks sequentially. We also considered enhancing the cleanup stage (Chapter 2.4.2), but this method introduces significantly more complexity

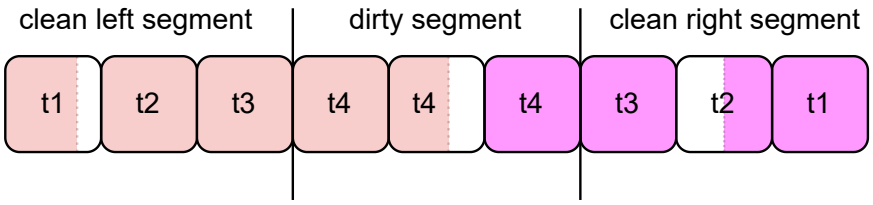


Figure 4.2 Start of cleanup stage. The cleanup stage begins by determining the number of dirty blocks on the left and right sides. Based on this information, the splitters that mark the borders of the dirty segment are identified. In this instance, two dirty blocks are on the left side and one on the right. As a result, the dirty segment will start two blocks to the left of the midpoint and continue one block to the right of the midpoint. The next step is to rearrange the blocks so that the dirty segment will contain all the dirty blocks.

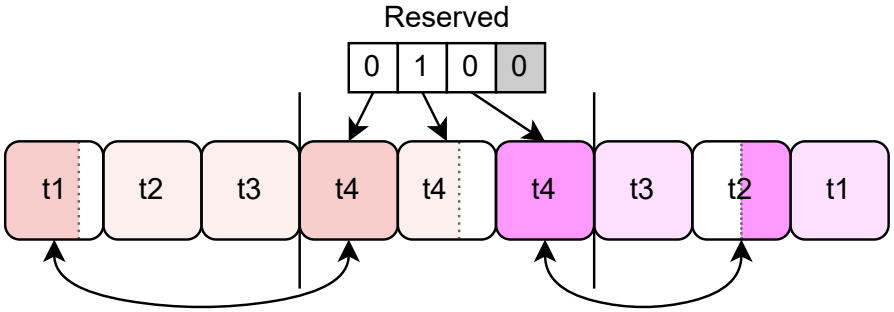


Figure 4.3 Swapping blocks in the cleanup stage. During the cleanup stage, blocks are being swapped. The reserved array indicates which blocks have a reserved spot in the dirty segment before swapping. Any blocks which did not reserved a spot will be subject to swapping. The size of the reserved array is determined by the number of threads, which corresponds to the maximum number of dirty blocks. In this scenario, only three blocks are considered dirty. While the second block from the left of thread t_4 has a reserved spot, the remaining two blocks do not have reserved spots in the dirty segment and will be swapped out by dirty blocks.

and, based on the benchmarks provided by the author, only offers a minimal speed-up. Therefore, we chose to go with a more straightforward sequential cleanup instead.

Recognizing the significance of correctly synchronizing threads is crucial to avoid data races and associated problems. It is necessary to utilize appropriate atomic constructs, such as atomic variables in C++ implementation and the atomic clause in OpenMP. In order to guarantee that all threads can access the same information, two barriers must be established. The first barrier is employed to determine the overall count of dirty blocks. The second barrier is required before swapping dirty blocks to the center. The implementation has been enhanced to monitor whether the partition was swapless (see Chapter 2.3.3). Each thread now tracks whether any elements were swapped, and we also check if any elements were swapped during the cleanup stage. If no elements were swapped at all, we identify a swapless partition, indicating that the input data is likely sorted, enabling us to adjust the algorithm.

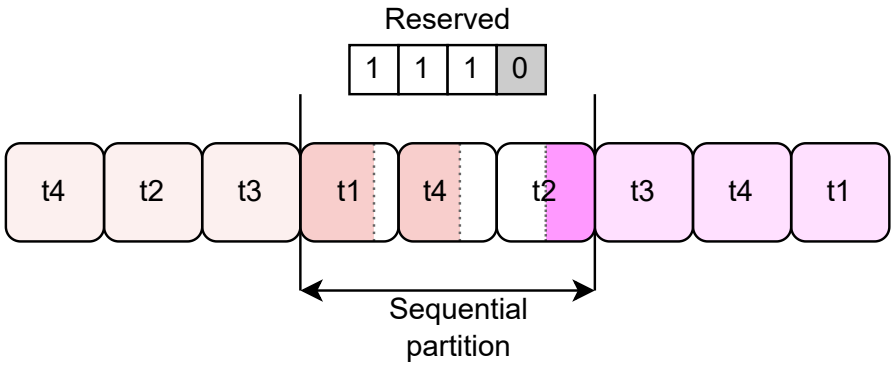


Figure 4.4 Final sequential partition. All dirty blocks have been moved to the middle. Subsequently, the middle segment is neutralized sequentially.

4.3.2 Branchless Parallel Partition

The parallel branchless partitioning implementation adopts a similar approach, utilizing an almost identical algorithm. The primary difference lies in the way threads manage block neutralization. Initially, the concept was for each thread to perform branchless partitioning within the block repeatedly. This process included acquiring a block, constantly filling offset buffers, exchanging elements, and obtaining a new block if necessary. However, this method would introduce significant complexity and more conditional code due to the need to verify if the block concludes during the scanning stage. We have opted to pursue an alternative approach. Consequently, when the thread encounters the next block, it will scan the entire block in one iteration. This allows us to accomplish this task using a single straightforward loop without the need for any conditional code. It is crucial, however, to carefully select the block size; a block that is too small may result in threads blocking and causing delays for each other. Conversely, a too large block may cause not all scanned elements to fit in the L1 cache. The branchless method is also used in sequential cleanup. Additionally, we have also integrated the identification of swapless partition.

4.4 Further Optimizations

In addition to the optimized partitioning, optimized pivot selection, and thread balancing as previously discussed, our PPQSort algorithm includes additional improvements. Moreover, we have parallelized certain sequential optimizations to increase effectiveness even more.

- **Recursion elimination** Instead of employing two recursive calls, we utilize a while loop and a single recursive call. This approach helps decrease the number of recursive calls to $O(\log n)$, thereby reducing the stack memory needed.
- **Adaptivness** As previously mentioned, we verify whether the partition is swapless. If so, we infer that the input data is already sorted and adjust our algorithm accordingly. In the scenario where only one thread is available, we attempt to sort the

partitioned data using a sequential partial insertion sort (Chapter 2.3.3). In cases where multiple threads are present, the input data is distributed among them, with each thread handling a segment that overlaps by one element and verifying if its assigned segment is sorted. If all threads confirm the sorted status of their segments, then the input data is considered sorted. Otherwise, we abandon this approach and revert to the previous quicksort algorithm.

- **Duplicate elements** We check whether the chosen pivot is identical to the pivot used in the previous partitioning. If they are the same, we alter the partitioning method by putting the elements equal to the pivot on the left side of it. Following this partitioning step, there is no need to recursively process the left side, as it would solely consist of elements that are equal to the pivot. This optimization can only be performed when partitioning a segment that is not the leftmost. The visual representation can be seen in Figure 2.4.

- **Insertion sort** When the length of the input sequence is less than a specific threshold, we opt for insertion sort as it is typically faster for a small number of elements. We establish a larger threshold for basic elements with the default comparison function as comparisons are less costly. Additionally, to manage the elimination of duplicate elements, we keep track of whether we are processing the leftmost partition. Exploiting this information, when sorting a partition that is not the leftmost, we can eliminate one boundary check and further improve the speed.

- **Unbalanced partition** We employed a technique to count bad partitions. If an bad partition is detected, we will rearrange the elements to introduce new candidates for the next partition's pivot. Otherwise, we will use heap sort if the number of bad partitions surpasses a specific limit. This strategy is inspired by the pdqsort algorithm, and further information can be found in Chapter 2.3.6.

4.5 OpenMP Implementation

Both of our implementations are written in C++, allowing us to have extensive control over memory and other low-level operations while still being able to utilize abstract and modular programming techniques. Our first implementation is based on OpenMP, simplifying the process of parallelizing code and incorporating synchronization constructs and other functionalities to support parallel programming. Our code works fully with OpenMP version 4.5. It can also use newer features from OpenMP version 5.1, but only if your compiler supports those features. Compilers that fully support OpenMP 4.5 include GCC version 11 or later and Clang version 6 or later. To use the optional features from OpenMP 5.1, you will need GCC version 12 or later or Clang version 17 or later.

OpenMP's task construct is a precious feature for our implementation. OpenMP provides an efficient task pool, which makes it straightforward to parallelize recursive calls within quicksort using task parallelism. We naturally used the task construct to parallelize our PPQSort algorithm effectively. Another key feature that we employed is nested parallelism. In OpenMP, nested parallelism allows threads within a parallel region to

create a limited number of subthreads and wait for their completion. This functionality was crucial in implementing parallel partitioning in our code. Our implementation also relies on OpenMP's barrier and atomic operations of OpenMP's, including atomic capture. We leverage the more efficient atomic compare capture construct when using OpenMP version 5.1 or later. We achieve the same functionality by employing a critical section if an older version of OpenMP is used (below 5.1).

4.6 C++ Threading Implementation

Our second implementation is based solely on standard C++ features and uses no external libraries or non-standard extensions. This makes our implementation portable and architecture-independent, which is a significant advantage over other parallel quicksort implementations that typically rely on third-party libraries. Our implementation complies with the C++20 standard.

4.6.1 Standard Library Features

The Concurrency support library was introduced in C++11, and C++20 has included some extra beneficial features to enhance this library, which we have utilized, namely:

- **Jthread:** In order to achieve parallelism, we use `std::jthread`, which is based on `std::thread` but incorporates additional functionalities like stopping the thread under specific conditions.
- **Memory order semantics:** We use memory order semantics to specify the order in which different threads see memory operations. This is essential to ensure correct and consistent behavior in a multi-threaded environment. Refer to Chapter 4.6.2 for detailed description.
- **Barriers:** We utilize `std::barrier` to synchronize threads at specific points in the algorithm. This ensures that all threads have completed their tasks before proceeding to the next stage. In some cases, `std::latch` is utilized, which is similar to `std::barrier`, except that it is not reusable; once it reaches zero after being decremented and all threads are unblocked, there is no option to reset the counter.
- **Semaphores:** We use `std::binary_semaphore` for managing access to shared resources and synchronizing threads. Instead of utilizing `std::condition_variable`, we opt for semaphores to obstruct and await signals. This approach offers a simpler implementation and, in some cases, better performance according to the cpp reference [57]: "semaphores can be considered alternatives to `std::condition_variables`, often with better performance."
- **Atomics and locks:** `std::atomic` or `std::atomic_ref` are employed for creating atomic variables. Further `std::mutex` and its wrappers, such as `std::unique_lock` or `std::lock_guard` are utilized for establishing locks and implementing diverse synchronization and communication mechanisms.

4.6.2 Memory Order Semantics

Modern CPUs optimize performance by potentially reordering instructions and memory accesses. The memory model of the programming language governs this behavior.

A memory model defines the expected order of memory operations from the perspective of a program. C++ allows specifying the memory order for each atomic operation, influencing how surrounding memory accesses are ordered. The default seq_cst (sequentially consistent) order ensures a safe and strict execution order but can be overly conservative.

C++ offers a variety of memory order semantics, but processors have varying levels of support. When the specified combination is not supported, the compiler inserts instructions with "stronger" semantics (limiting reordering possibilities). For example, x86-64 lacks relaxed-store semantics, so every write automatically becomes a release-store. On the contrary, relaxed-store is available on architectures like ARMv8.1.

In PPQSort, we carefully choose appropriate memory order semantics for atomic operations. Since most operations are independent, we use relaxed semantics for efficiency. However, in specific cases, stronger ordering is necessary. We typically use the release in combination with acquire semantics.

On strongly-ordered architectures, explicitly specifying memory order has a minor performance impact. However, specifying memory order on weakly-ordered models, such as ARM processors, allows the code to benefit from weaker ordering and potentially achieve faster execution.

4.6.3 Thread Pool

C++20 introduced several valuable features for concurrent programming, yet it still lacks a key element: a thread pool. This feature is required to parallelize the control flow of our quicksort algorithm, particularly the recursive calls. To achieve this, we have developed a custom thread pool.

The custom thread pool we developed draws inspiration from the concept presented by Sean Parent [58]. Instead of using a single atomic queue for thread tasks, which may become inefficient and cause thread blocking due to potential queue overload, we opt for a more effective strategy. Our approach involves employing multiple queues to enhance the throughput of the thread pool, with each thread being assigned its queue. This technique is illustrated in Figure 4.5.

Task Scheduler

When a new task is created, the scheduler determines the appropriate queue for the task. The aim is to reduce instances where a thread must wait to acquire a lock to improve the efficiency of CPU utilization. We have developed a thread-safe queue that uses a lock to achieve this goal. Nevertheless, this queue includes unique functions, namely `try_push` and `try_pop`. Suppose one of these functions is invoked while another thread currently holds the lock. In that case, the function will immediately return a failure indication instead of waiting for the lock to be released. This approach allows us to iterate through the queues and assign the task to one that is not busy without blocking. If all queues are occupied, we can iterate through the queues K times. We do not know

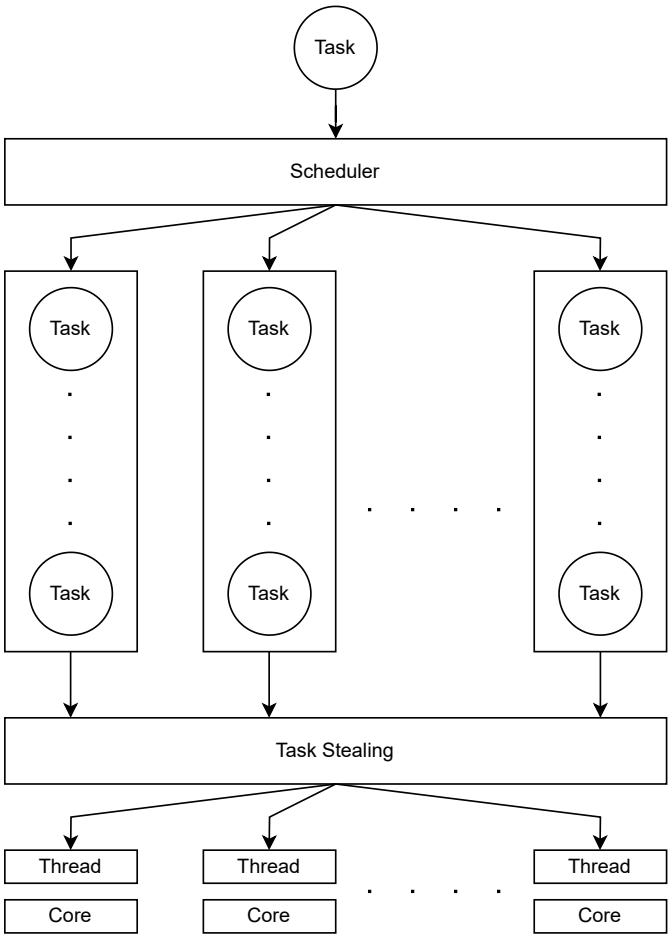


Figure 4.5 Thread pool diagram. Every thread possesses its queue. The scheduler distributes new tasks to the thread's queues, and task stealing prevents thread starvation.

Code listing 4.1 Task scheduler. The following code snippet demonstrates how our task scheduler adds new tasks to the queues. Initially, we attempt to add the task without pausing to acquire the lock and prevent blocking. However, if unsuccessful for K attempts, we add a task to the queue indicated by our index while waiting for the lock to be obtained.

```
void push_task(taskType&& task) {
    total_tasks_.fetch_add(1, std::memory_order_release);

    // spin around queues to find non-busy queue
    const std::size_t i = idx_.fetch_add(1, std::memory_order_relaxed);
    constexpr unsigned int K = 2;
    for (std::size_t n = 0; n < t_ * K; ++n) {
        if (q_[(i + n) % t_].try_push(std::forward<taskType>(task))) {
            pending_tasks_.fetch_add(1, std::memory_order_release);
            pending_tasks_.notify_all();
            return;
        }
    }

  // if all queues busy, wait for the first one
    q_[i % t_].push(std::forward<taskType>(task));
    pending_tasks_.fetch_add(1, std::memory_order_release);
    pending_tasks_.notify_all();
}
```

which queue will become free first, so this approach is logical instead of waiting for one queue. However, if all the queues are still busy after K repeats, we can utilize the `push` function, which will wait until the lock is acquired. In addition, we maintain an atomic index to determine the starting point for iterating through the queues. This index is incremented each time a task is assigned to ensure a more balanced distribution of tasks. This technique is demonstrated in the Code 4.1.

Task Stealing

Another exciting optimization within our thread pool concerns the distribution of tasks from queues to threads. If each thread were to retrieve tasks exclusively from its queue, it would result in a significant load imbalance issue. Therefore, when a thread seeks new tasks, it adopts a strategy similar to when tasks are added to the queues. This strategy involves traversing the queues using the `try_pop` function, which immediately reports a failure if the queue is busy or empty. Consequently, threads can acquire tasks from various queues if their queue is busy or empty. When all queues are occupied, the `pop` method is employed for our queue, which patiently waits for the lock before returning a task or signaling that the queue is empty. We call this approach task stealing and it enhances the load distribution and speeds up the processing rate, as demonstrated in Code 4.2.

Code listing 4.2 Task stealing. The code snippet below illustrates how our threads retrieve tasks from queues. At first, we try to obtain the task without stopping to acquire the lock and avoid blocking. We cycle through all queues, allowing us to take tasks from other queues, which is why we call it task stealing. Nevertheless, if this attempt is unsuccessful, we fetch a task from our queue, even if it requires waiting for the lock.

```
bool get_next_task(const unsigned int id) {
    bool found = false;

    // spin around all queues to find a task, start with my queue
    for (std::size_t n = 0; n < threads_count_; ++n) {
        if (auto task = queues_[(id + n) % threads_count_].try_pop()) {
            run_task(std::forward<taskType>(task.value()));
            found = true;
            break;
        }
    }

    // if all queues busy or empty, check our queue if empty
    if (!found) {
        if (auto task = queues_[id].pop()) {
            run_task(std::forward<taskType>(task.value()));
            found = true;
        }
    }

    return found;
}
```

Thread-Safe Queue Implementation Considerations

In selecting a container for our thread-safe queue implementation, we initially considered using `std::deque`, known for its constant-time insertion and removal at both ends. However, the standard implementation of `std::deque` typically involves individually allocated fixed-size arrays with additional bookkeeping, leading to two-pointer dereferences for indexed access and noncontiguous storage of elements. While expanding a deque is fast due to the absence of element copying, it comes with a high minimum memory overhead. Consequently, we decided to utilize `std::vector` instead, which offers quicker element access. Although expanding a vector is slower, we proactively reserve sufficient memory to prevent reallocations. However, `std::vector` provides constant-time insertion and deletion operations only at the end, making it behave more like a thread-safe stack than a queue in this scenario. However, since the task processing order is irrelevant in our scenario, we can leverage the performance benefits of the vector container.

Worker Thread Wait Strategy

We aim to avoid having our threads engage in spinlock and continuously monitoring for pending tasks. Therefore, we have incorporated a basic sleep mechanism for cases with no tasks awaiting processing. The common practice involves using a mutex with `std::condition_variable` and waiting for the condition variable to be signaled. We decided to utilize the latest features of C++20, which are seen as an alternative approach and better suited for our situation. A thread-safe queue is already in place, and the number of pending tasks is monitored. Consequently, we utilized the `std::atomic::wait` function. This method is more straightforward and efficient than a condition variable as it eliminates the need for an extra mutex. At the start of the loop, the working thread verifies if there are any tasks (i.e. if the atomic counter is not 0). However, if there are no tasks, the thread sleeps and waits for signals to wake up. As a result, when another thread adds a new task to the queue, it notifies all the waiting threads. Notifying all of them is preferable to expedite gathering new tasks.

It is essential to mention that a sleeping thread can spontaneously awaken, but it will verify whether the atomic variable has been modified. If not, it will return to a sleeping state. Hence, we can be confident that the thread will be unblocked only when the variable has been modified. The documentation also mentions that transient changes from an old value to another value and then back to the old value may not be captured due to the ABA problem, and the thread will not unblock [59]. The ABA issue may occur when the atomic counter transitions from 0 to 1 and then back to 0 rapidly. In such a scenario, certain threads that are in a sleep state might not detect this rapid change. However, in our situation, this is not a concern because the reversion of the variable to 0 signals that another thread has already taken over the task. Therefore, the sleeping threads can sleep undisturbed.

Thread Pool Shutdown

When the master thread is waiting for the thread pool to complete, it verifies the existence of any pending or currently executing tasks. If there are none, the thread pool can be shut down. However, constantly checking for pending tasks would only be inef-

Code listing 4.3 Worker routine. The following code snippet demonstrates the procedure for the active thread. Initially, it enters a sleep state if there are no pending tasks. Upon awakening, it verifies whether it should terminate its operation, as the thread pool has completed its tasks. Subsequently, it retrieves its assigned tasks or obtains tasks from alternative queues. If no tasks are acquired, it assesses whether there are no remaining tasks and, if so, sends a signal.

```
void worker(const std::stop_token& stop_token, const unsigned int id) {
    while (true) {
        // sleep until there are any tasks in queues
        pending_tasks_.wait(0, std::memory_order_acquire);
        if (stop_token.stop_requested())
            break;

        // while there are tasks, execute them (mine or stolen)
        while (get_next_task(id));

        // no tasks are in queues or handled --> signal that our work
        ↪  is done
        if (total_tasks_.load(std::memory_order_acquire) == 0)
            threads_done_semaphore_.release();
    }
}
```

ficient. In this scenario, utilizing the wait function on an atomic variable is impossible because it only checks if the variable transitioned from an old state to a new one rather than a specific state (e.g., 0). Therefore, we have employed a semaphore, specifically `std::binary_semaphore`. The master thread will wait on this semaphore, and once the final thread completes its work, it will signal through the binary semaphore that all threads have finished their work.

Alternative Approaches

An alternative option to our solution is the implementation of a lock-free queue. However, implementing the lock-free queue in a model with multiple producers and consumers is quite complex. Therefore, we stay with our initial approach, a great hybrid implementation that balances speed and simplicity. It is crucial to mention that the scenario in which all queues are occupied occurs rarely, resulting in threads barely having to wait for locks.

4.7 Memory Usage

Our implementation leverages two auxiliary arrays for storing offsets, but these arrays are crafted to be small and constant. Each array occupies a mere 2048 bytes (2 KB), resulting in a negligible memory footprint.

A key element of our strategy involves using a thread pool and a method that balances

the number of running threads. These techniques control a set number of active threads, which helps prevent excessive memory consumption caused by running too many threads. The thread count (p) stays constant, ensuring that the total memory usage follows the equation: $4KB * p$.

As a result, the memory usage is a small constant value independent of the input size. This characteristic aligns with the definition of in-place algorithms in Big O notation. Constant factors are disregarded when expressing Big O complexity. Even with the auxiliary arrays, the algorithm's memory usage remains $O(1)$, effectively classifying it as in-place.

4.8 Application Programming Interface (API)

Our implementation is designed to be header-only, simplifying the integration process. Additionally, we provide a compact CMake file along with our implementation, allowing users already utilizing CMake to incorporate the implementation into their projects seamlessly. The API we offer adheres to the C++ standard and is similar to the `std::sort` API. Users can utilize execution policies to define the characteristics of an algorithm. These politics are predefined in `ppqsort::execution`. If no execution policies are explicitly defined, the implementation will proceed sequentially. The choice of the branchless version is determined by the type of element and the comparison function used. When the element type satisfies `std::is_arithmetic` and the default comparison function like `std::less` is employed, the branchless variant is chosen. In other cases, the algorithm will utilize the non-branchless version, but the user can enforce the branchless version. Possible usages are presented in code 4.4.

If the implementation is linked with OpenMP, it utilizes OpenMP. Otherwise, it resorts to C++ threads. The user can enforce the use of C++ threads, even if the program is linked with OpenMP, as shown in the code 4.5. Within the `ppqsort::parameters` namespace, programmers can adjust specific parameters to suit their preferences. These default values have been established through empirical testing.

4.9 Testing Suite

The repository we have published is a comprehensive suite that includes the implementation itself and other used components, such as automated tests, benchmarks, and documentation. The suite utilizes the modernCppStarter template [60], which effectively automates and integrates everything using cmake files. Another benefit is that the suite can be effortlessly integrated into IDEs such as CLion or Visual Studio, which natively support cmake files.

The tests are implemented using the GoogleTest framework (gtest), a popular and comprehensive C++ testing framework [61]. The gtest enables the creation of well-structured unit tests that promote clean code and maintainability. Some essential aspects of gtest include fixtures, tests that values and types can parameterize, and a wide range of assertions. Fixtures are specialized classes designed to establish and dismantle the necessary test environment for each test case. This approach minimizes redundant code and guarantees that tests are executed independently. The gtest framework enables the

Code listing 4.4 Example of usage. An example of usage is presented below, demonstrating the API and the various ways it can be invoked.

```
#include <vector>
#include <string>
#include <ppqsort.h>

int main() {
    std::vector<int> input{3, 4, 5, 5, 6, 2, 5};
    std::vector<std::string> input_str{"fdf", "fdgdf", "hytre",
    ↪  "wrea"};
    auto cmp = [](int a, int b) { return b - a < a - b; };

    // possible overloads
    ppqsort::sort(input.begin(), input.end());
    ppqsort::sort(input.begin(), input.end(), cmp);

    // use parallel version
    ppqsort::sort(ppqsort::execution::par, input.begin(), input.end());
    ppqsort::sort(ppqsort::execution::par, input.begin(), input.end(),
    ↪  cmp);

    // specify number of threads
    ppqsort::sort(ppqsort::execution::par, input.begin(), input.end(),
    ↪  16);

    // force parallel branchless version on strings
    ppqsort::sort(ppqsort::execution::par_force_branchless,
    ↪  input_str.begin(), input_str.end());

}
```

Code listing 4.5 Use C++ threading. The following code exemplifies how a programmer can mandate the utilization of C++ threading in the implementation, even when the program is linked with OpenMP.

```
#include <vector>
#include <string>

// enforce cpp threads
#define FORCE_CPP
#include <ppqsort.h>

int main() {
    std::vector<int> input{3, 4, 5, 5, 6, 2, 5};

    // call as usual
    ppqsort::sort(ppqsort::execution::par, input.begin(), input.end());
}
```

Code listing 4.6 Cmake integration. This code demonstrates how users can conveniently integrate this project into their CMake projects using CPM.cmake. Alternatively, they can use other approaches, such as FetchContent, or directly download the implementation directory.

```
include(cmake/CPM.cmake)
CPMAddPackage(
  NAME PPQSort
  GITHUB_REPOSITORY GabTux/PPQSort
  VERSION 1.0.3 # change this to latest commit or release tag
)
target_link_libraries(YOUR_TARGET PPQSort::PPQSort)
```

Code listing 4.7 Build and run scripts. Users can utilize these scripts to execute or build specific components conveniently. Permitted commands include all, standalone, tests, benchmark, docs, and clean.

```
@cf-frontend03-prod:~/PPQSort$ scripts/build.sh all
...
@cf-frontend03-prod:~/PPQSort$ scripts/run.sh standalone
...
```

creation of value- and type-parameterized tests, allowing them to be executed multiple times with varying input values or types. Moreover, it also includes a comprehensive collection of assertion macros for conveniently verifying conditions.

We have integrated Codecov into our GitHub repository [62] to ensure comprehensive code coverage and identify potential testing gaps. Codecov is a valuable tool that analyzes and reports on the effectiveness of the test suite. It provides detailed information on the percentage of code lines covered by tests, pinpointing areas where tests might be lacking. In our situation, Codecov typically indicates a notable $99 - 100\%$ code coverage, with occasional minor differences due to thread synchronization. It demonstrates that our test suite effectively tests every line of code. This high coverage level instills confidence in the code's quality and functionality.

For our benchmarks, we used Google's benchmark framework [63], which is widely used and shares a close API resemblance with gtest. We made use of fixtures and template macros, which helped us avoid redundant code and resulted in cleaner and more straightforward code. Lastly, for the documentation, we used Doxygen, which is a widely-used documentation generator tool [64]. We have also created bash scripts for easy use, allowing users to build or run all projects or specific parts. The script usage is demonstrated in Code 4.7.

We have released the whole suite including the implementation on GitHub [62] and published it to the general public.

Chapter 5
Comparative Analysis

Thorough testing is critical to the development of the PPQSort algorithm. We verified the correctness of the implementation, evaluated its efficiency, and compared it with other popular implementations.

In this chapter, we will first present the utilized testing environments. Subsequently, we will examine the preparation of input data. Following that, the succeeding section will elaborate on determining the optimal parameter values for our algorithm. Finally, we will compare the performance of our optimized implementation with other existing parallel quicksort implementations.

5.1 Testing Enviroments

We utilized four different setups to evaluate and compare our implementation, ensuring it is not optimized for a particular system architecture. Details about these environments can be found in Table 5.1. Each time we conduct tests and benchmarks on these clusters, we use a task scheduler to queue our executions. The scheduler ensures that the execution of tasks does not interfere with the execution of other tasks. Consequently, only one task is run on a single back-end node to ensure our results remain unaffected.

5.2 Input Data

In order to thoroughly evaluate and analyze efficiency, we have created two categories of data. The initial category consists of synthetic pseudorandomly generated data, while the second category comprises data generated through actual natural processes or applications.

We produce the synthetic data randomly in parallel with the help of OpenMP. A static seed guarantees consistent random data across all algorithms tested to ensure fair comparisons. Following data generation, we further process the data to create specific input patterns (e.g., ascending, descending). Details of all synthetically generated data sets are provided in Table 5.2.

We used the following basic types for our benchmarks:

- **short** Half-size integer (2 bytes)

	Cluster STAR	ClusterFIT Intel	ClusterFIT ARM	RCI Cluster
CPU architecture	amd64/x86_64	amd64/x86_64	aarch64	amd64/x86_64
CPU	Intel Xeon E5-2630 v4 2.20GHz	Intel Xeon Gold 6254 3.10GHz	Fujitsu A64FX	AMD EPYC 7543
HW architecture	NUMA	NUMA	NUMA	NUMA
Cores	2CPU · 10cores == 20	2CPU · 18cores == 36	4CPU · 12cores == 48	2CPU · 32cores == 64
L1 cache	32KB (x20)	32KB (x36)	64KB (x48)	32KB (x64)
L2 cache	256KB (x20)	1024KB (x36)	8MB (x4)	512KB (x64)
L3 cache	25600KB (x2)	25344KB (x1)	–	256MB (x2)
RAM	64GB	64GB	32GB	1TB
OS	CentOS 7.9.2009	Ubuntu 22.04	Rocky Linux 8.9	Rocky Linux 8.8
Compiler	GCC 11.2.1	GCC 11.4.0	GCC 13.1.1	GCC 13.2.0

Table 5.1 Overview of testing environments

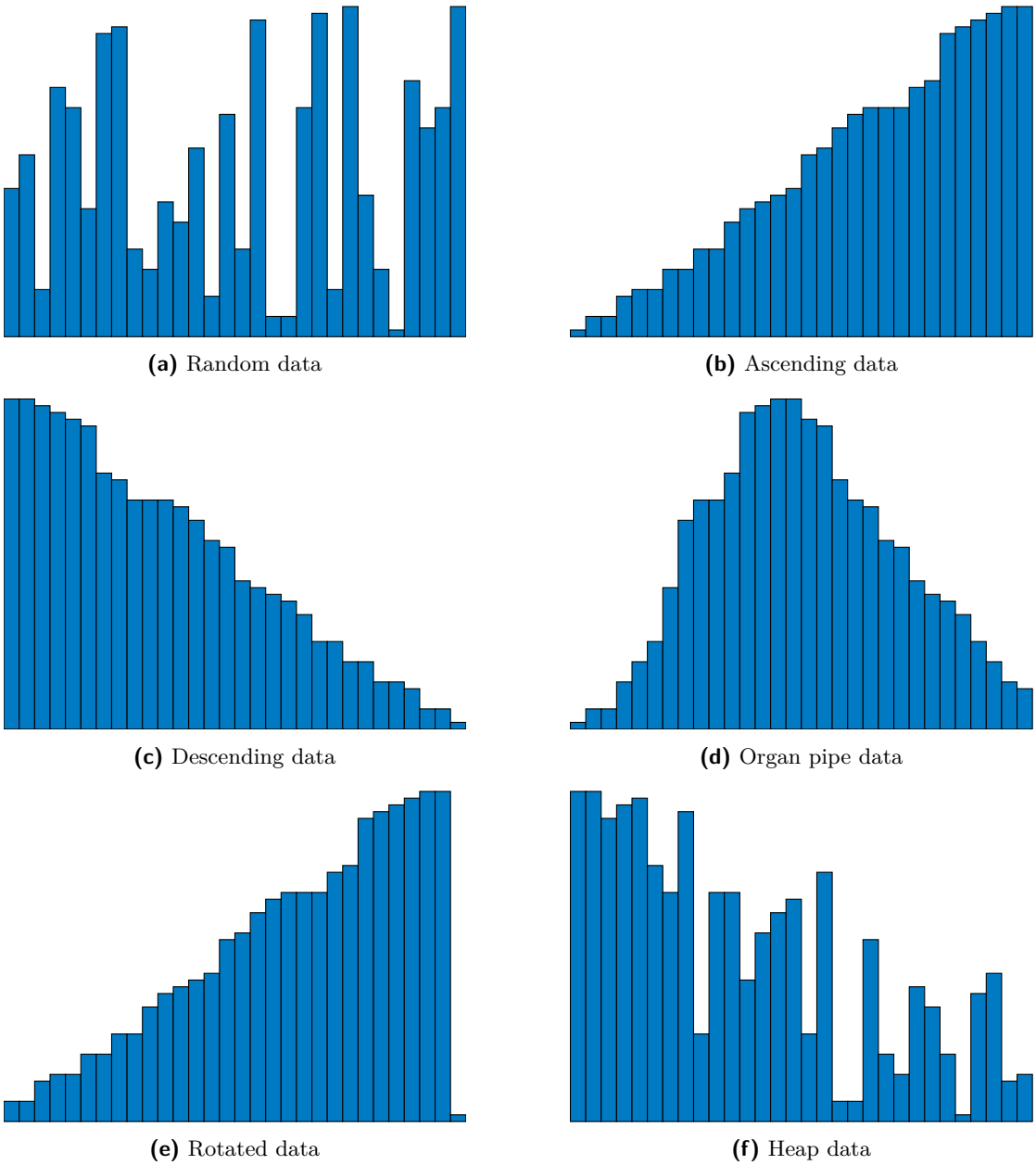
- **int** Integer (4 bytes)
- **double** Decimal numbers with double precision according to IEEE 754 (8 bytes)
- **string** A `std::string` containing 1001 elements, where the first 1000 elements are set to 0, and the last element is assigned a random value. This setup allows us to mimic elements with an expensive comparison function.

To encompass data commonly encountered in real-world applications and natural processes, we incorporated sparse matrices into our testing from SuiteSparse Matrix Collection [65]. Details about these matrices can be found in Table 5.3.

The sparse matrices were stored in the Coordinate Storage Format (COO) format [66, 67]. COO stores non-zero elements in three arrays: row indices, column indices, and values. This format requires no specific element order, making it ideal for assembling sparse matrices as generated non-zero elements are appended.

However, the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats are often preferred for efficient computations. Converting from COO to CSR requires a multi-array sorting of the non-zero elements based on row and column indices. This underscores the significance of efficiently sorting sparse matrices.

Among the sorting algorithms considered, only AQsort is capable of natively handling multi-array sorting. Other algorithms require transforming from the Structure of Arrays (SoA) to the Array of Structures (AoS) format. While C++23 offers the



(a) Random data

(b) Ascending data

(c) Descending data

(d) Organ pipe data

(e) Rotated data

(f) Heap data

Figure 5.1 Visualization of specific input data patterns for 30 elements.

Name	Types	Description
Random	short, int, double, string	truly random data
Ascending	short, int, double, string	sorted according to the $<$ operator
Descending	short, int, double, string	sorted and reversed
OrganPipe	short, int, double, string	first half is ascending, second half is descending
Rotated	short, int, double, string	sorted and then shifted by one to the left
Heap	short, int, double, string	in-place transformed into heap from random data
Small cardinality	int	random data, the cardinality (range) is limited
Small size	int	random data, smaller size

Table 5.2 Overview of generated input data

`std::views::zip` functionality for simplifying this task, we focused on C++20 for broader compatibility.

Therefore, we leveraged AQsort for direct multi-array sorting of COO matrices. For other algorithms, we performed the SoA-to-AoS transformation before sorting. Transformation time was excluded from benchmarks as it can vary significantly based on implementation details.

To facilitate efficient and memory-conscious loading of the matrices, we used the well-tested Fast Matrix Market (FMM) library [68]. By utilizing its enhanced parallel features, FMM enabled the effective loading of matrices into memory.

5.3 Finding Parameters

Several tunable parameters that control its behavior can influence the performance of the PPQSort algorithm. Finding the optimal configuration often requires careful consideration and experimentation. During development, we conducted extensive empirical testing on the STAR cluster (details in Table 5.1) to identify the most critical parameters: block size and sequential threshold. This chapter shows the impact of these parameters and presents results from the STAR cluster.

We evaluated PPQSort using various block sizes for both Hoare partitioning (see Figure 5.2) and branchless partitioning (see Figure 5.3). This analysis revealed that a block size of 2^{14} elements yielded optimal performance for Hoare partitioning. The block size $2^{10} = 1024$ for storing offsets indices was ideal for branchless partitioning.

We also analyzed the effect of the constant k (described in Chapter 4.1) on performance. This constant determines where PPQSort transitions to a sequential sorting algorithm for smaller subarrays. Based on the results presented in Figure 5.4, we identified an optimal k-value of 8. The choice of this value likely involves a trade-off between

	mawi_ 201512020330	uk-2005	dielFilterV3clx	Queen_4147
Rows	226, 196, 185	39, 459, 925	420, 408	4, 147, 110
Columns	226, 196, 185	39, 459, 925	420, 408	4, 147, 110
Nonzeros	480, 047, 894	936, 364, 282	32, 886208	316, 548, 962
Symmetric	Yes	No	Yes	Yes
Type	Integer	Binary	Complex	Real
Description	Packet trace data from the WIDE backbone	crawl of the .uk domain	analysis of a 4th-pole dielectric resonator	3D discretization by isoparametric hexahedral Finite Elements

Table 5.3 Sparse matrices. Overview of sparse matrices used in benchmarks.

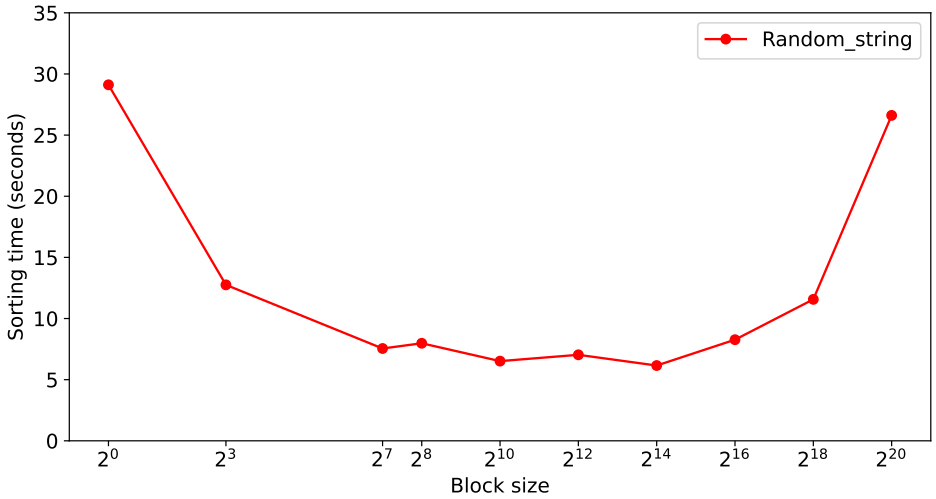


Figure 5.2 Comparison of various block sizes in Hoare partition. The PPQSort algorithm was executed using various block sizes on $n = 2e7$ prepended random strings (chapter 5.2). These tests were conducted with the OpenMP variant, although the outcomes for the C++ version showed similar results.

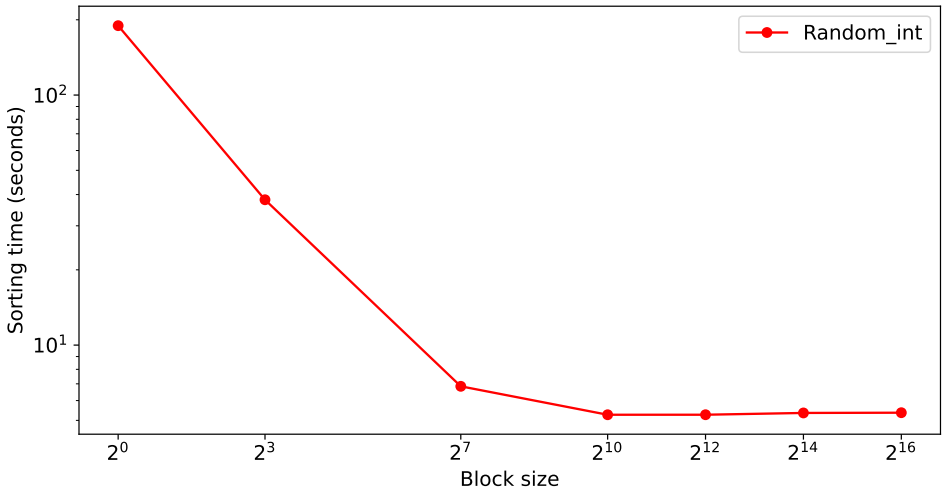


Figure 5.3 Comparison of various block sizes in branchless partition. The PPQSort algorithm was executed with different block sizes on a set of $n = 1e9$ random integers. These experiments were conducted with the C++ version, while the OpenMP version yielded comparable outcomes.

parallelization overhead for very small sub-arrays and the benefits of parallel processing for larger ones.

5.4 OpenMP vs. C++ Threading

This chapter evaluates the performance characteristics of our implementations, focusing on both time complexity and scalability. We achieved this by running the implementations on various input data. Our experiments were carried out on the STAR and Intel clusters, as outlined in Table 5.1. Unless explicitly mentioned otherwise, the input data size for each experiment was set to 2 billion elements (2e9).

5.4.1 Scalability

Our initial experiments focused on understanding the impact of thread count on performance. The Intel cluster we used uses a NUMA architecture, where each CPU possesses 18 cores. To account for this NUMA architecture, our benchmarks were designed such that when using less than 18 threads, all threads were assigned to a CPU. The results presented in Figure 5.5 demonstrate the favorable scaling characteristics of our implementation. Additionally, the figure highlights a fundamental property of NUMA architectures: the increased overhead associated with thread synchronization across multiple CPUs compared to a single CPU.

5.4.2 Data Patterns

To further evaluate the performance of our implementations, we performed tests on the STAR cluster using various basic input data patterns (depicted in Figure 5.1). The

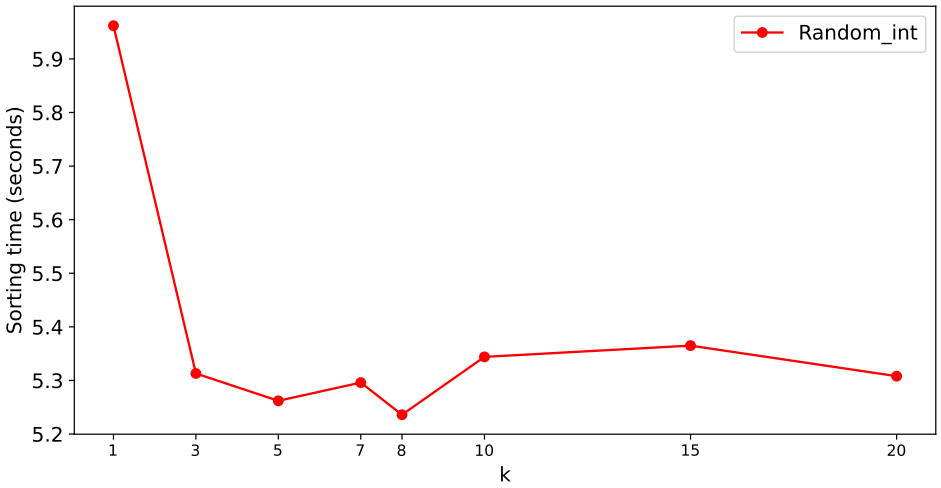


Figure 5.4 Comparison of various sequential thresholds. We experimented with various values of the constant k, which directly impacts the point at which the Parallel Pattern Quicksort (PPQSort) transitions to the sequential variant.

experiments were conducted using the basic data type `short`, `int`, `double` and the more intricate type `std::string` with $1,000$ zeros added at the beginning to increase the duration of the comparison. In particular, for these tests, we used the maximum available cores, resulting in a 1:1 thread-to-core mapping and a total of 20 threads.

Random data presents a complex scenario due to the absence of exploitable patterns. This requires a large number of comparisons and swaps for the algorithm to achieve sorting.

In the case of pre-sorted ascending data, the algorithm efficiently recognizes this state and avoids unnecessary swaps after the initial partition. It effectively detects a ”swapless partition,” indicating that the data is already sorted.

Descending pre-sorted data incurs a slightly higher complexity compared to ascending data. Nevertheless, the algorithm remains efficient because of its ability to perform a significant number of symmetrical swaps.

The organ pipe pattern can hinder certain algorithms due to its inherent symmetry and the possibility of selecting an unfavorable pivot. However, our implementation effectively addresses this issue by detecting ”bad partitions” and strategically shuffling future pivot candidates to eliminate symmetry-induced slowdowns.

Other data patterns, such as rotated and heaped data, can challenge some algorithms, but adaptive algorithms can exploit these patterns to improve efficiency. Our implementation demonstrates this by performing well on rotated data and exhibiting average performance on heap data.

Figure 5.6 offers a comprehensive comparison of the performance of our implementations across various types and data patterns. This figure provides valuable insight into how each implementation handles different data structures.

Figures 5.7 and 5.8 presents the performance comparison for specific data types. Figure 5.7 focuses on the execution time of both implementations when sorting `int` data

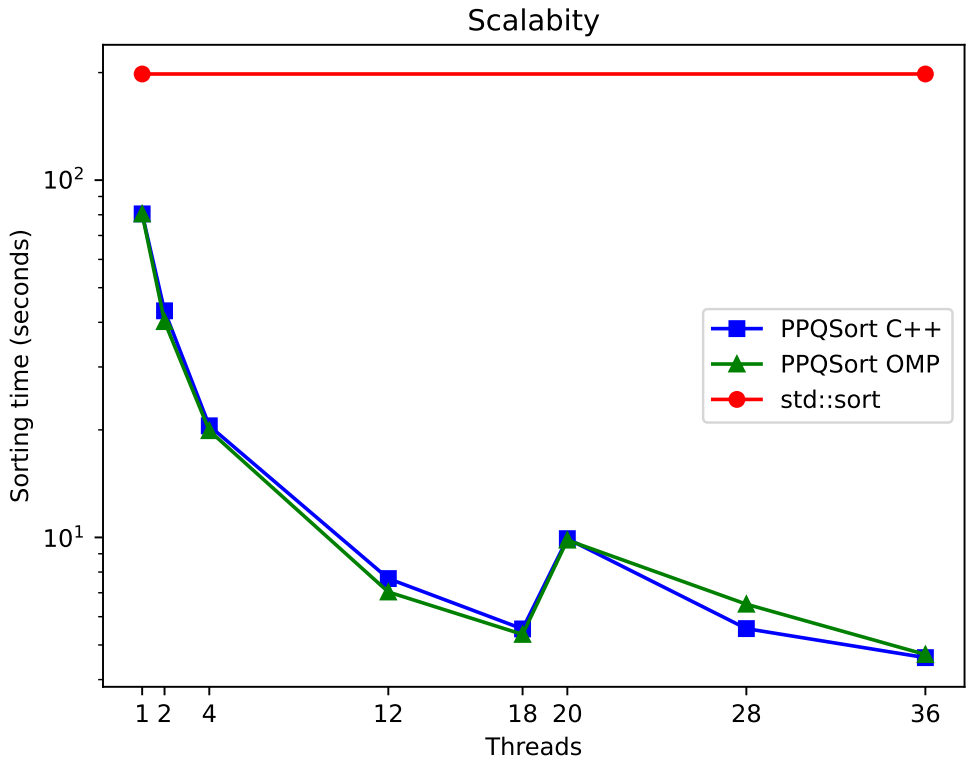


Figure 5.5 The figure shows the scalability of C++ and OpenMP implementation on an Intel cluster with a NUMA architecture. The experiment was conducted on random data with a size of $n = 2e9$. Note that the x-axis utilizes a logarithmic scale to enhance visibility. The findings demonstrate that our implementation exhibits good scalability, with the sequential version being over twice as fast as `std::sort`. It is noteworthy that with only 18 threads allocated, the implementations perform better than when 28 threads are allocated. This behavior is characteristic of NUMA architecture. The synchronization overhead between multiple CPUs (with more than 18 threads) is higher compared to a single CPU.

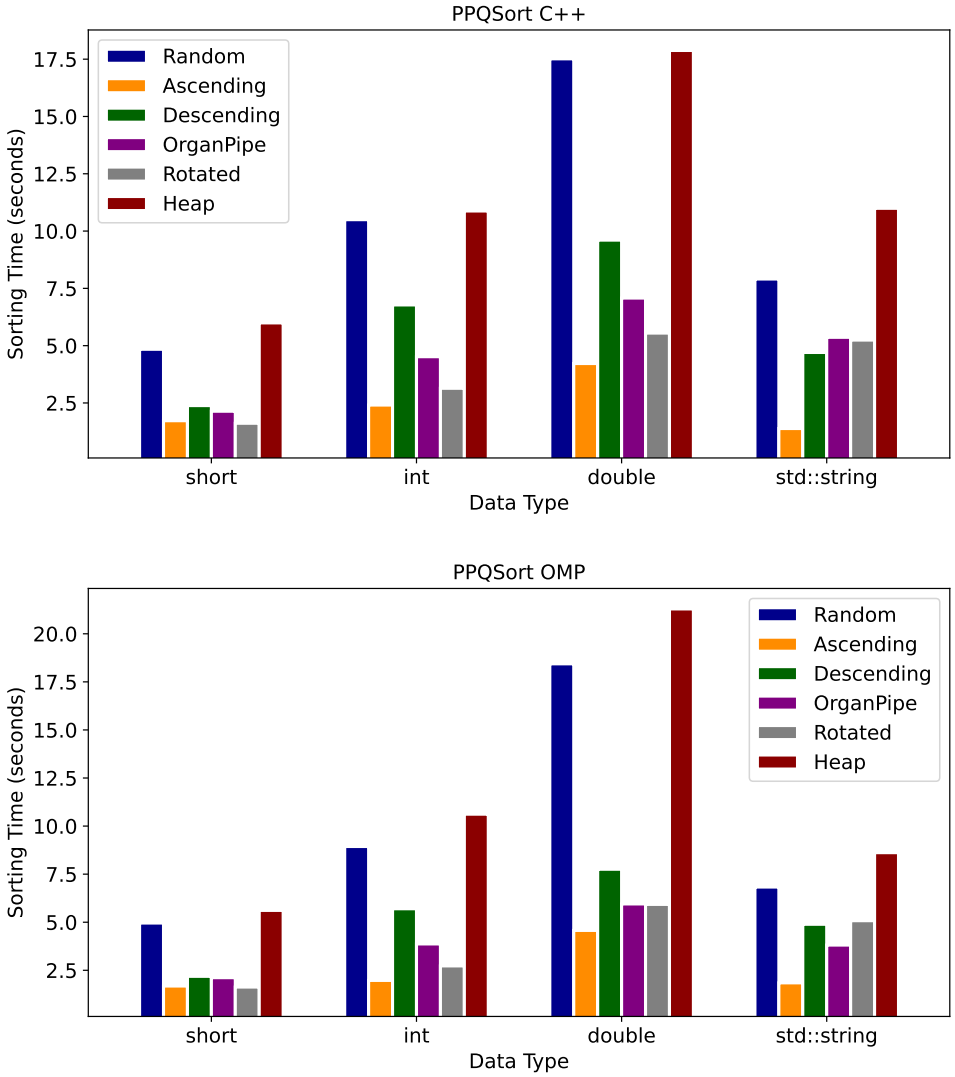


Figure 5.6 Evaluation of the performance of PPQSort C++ and OMP implementations on different data patterns. For the `short`, `int`, and `double` data types, we employed a size of $n = 2e9$, while for the `std::string` data type, the size was $n = 2e7$. Tests were conducted on the STAR cluster.

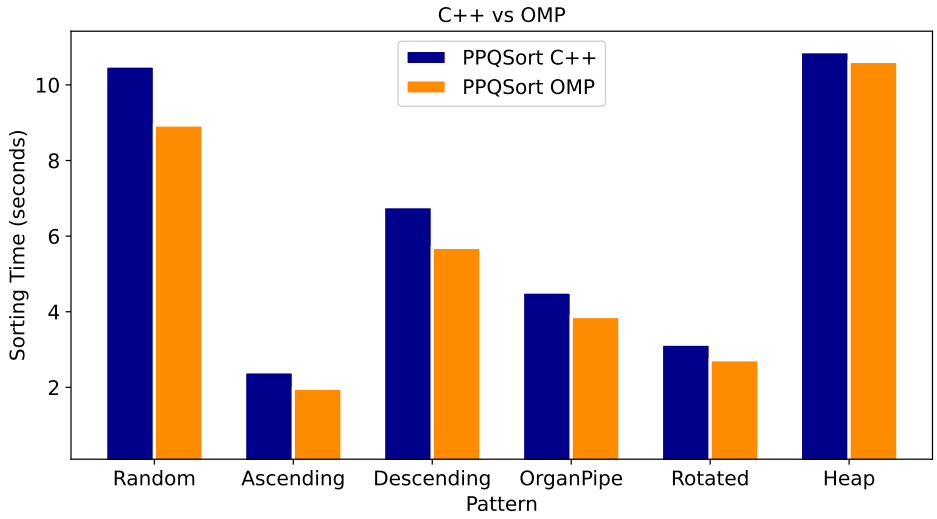


Figure 5.7 Evaluation of the performance of our implementations on different data patterns. The tests were conducted using $n = 2e9$ **int** elements.

across the various patterns. Similarly, Figure 5.8 compares the performance of implementations for `std::string` data patterns. Analyzing these figures along with Figure 5.6 allows for a granular examination of how the data type and pattern interplay influence the efficiency of our implementations. Both implementations demonstrate excellent performance and consistency across various data patterns. This versatility suggests their suitability for handling diverse sorting tasks.

Although the benchmarks in this chapter indicate a slight edge for the OpenMP implementation, Chapter 5.5 explores performance on different machines, where the C++ implementation often excels. These findings highlight the importance of considering the target hardware and application requirements when choosing between the two.

5.4.3 Different Cardinality and Size

To comprehensively assess the performance of our implementations, we performed additional tests on the STAR cluster. The tests examined random data with different attributes, including data cardinality (the number of unique elements) and data size. We investigated how efficiently the implementations handle data with low cardinality (all elements might be the same) and a range of data sizes to ensure they excel at sorting small and large datasets. The results in Figures 5.9 and 5.10 demonstrate that both implementations exhibit excellent performance and remarkably similar behavior in these data variations.

5.4.4 Matrices

To assess the effectiveness of our implementations beyond synthetic data, we conducted benchmarks using sparse matrices generated from real-world applications or processes.

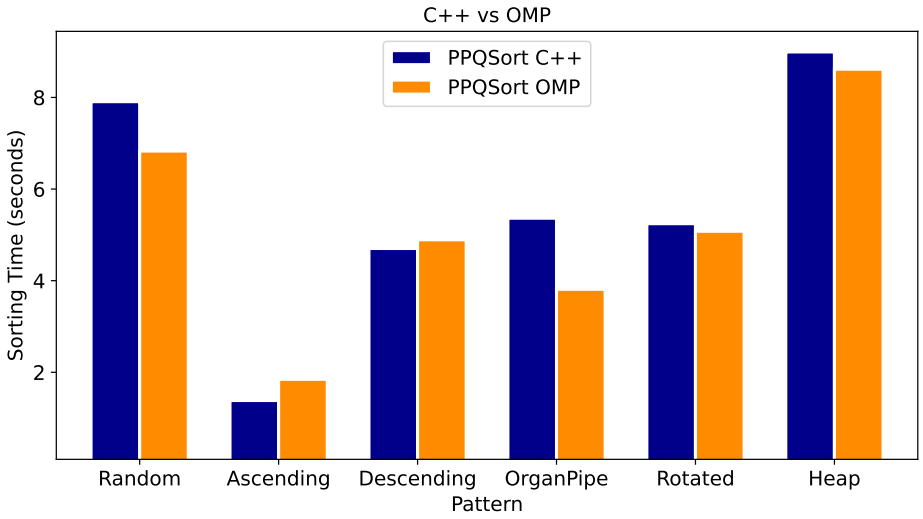


Figure 5.8 Evaluation of the performance of our implementations on different data patterns. The tests were conducted using $n = 2e7$ prepended `std::string` elements.

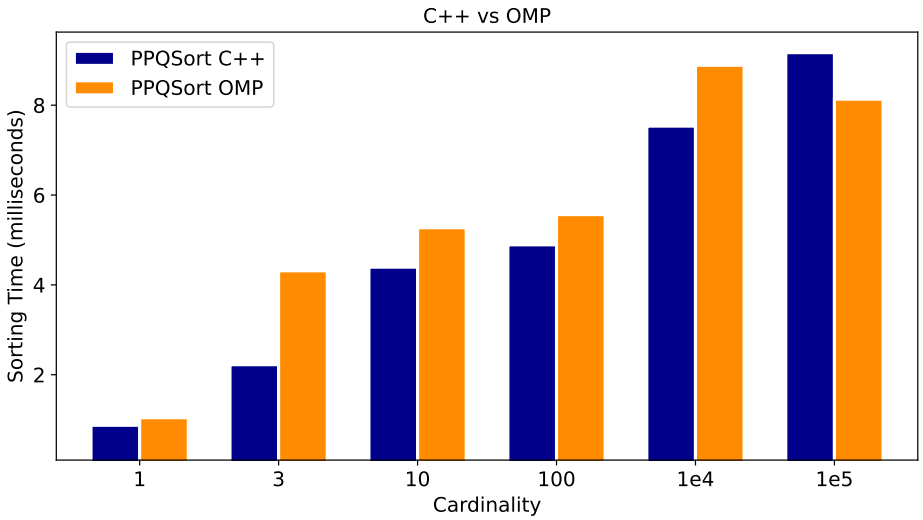


Figure 5.9 This figure illustrates the performance comparison of our implementations for sorting random integer (int) data with a size of 2 billion elements ($n = 2e9$). Cardinality, which signifies the number of unique elements, is varied in this experiment to examine its influence on sorting time.

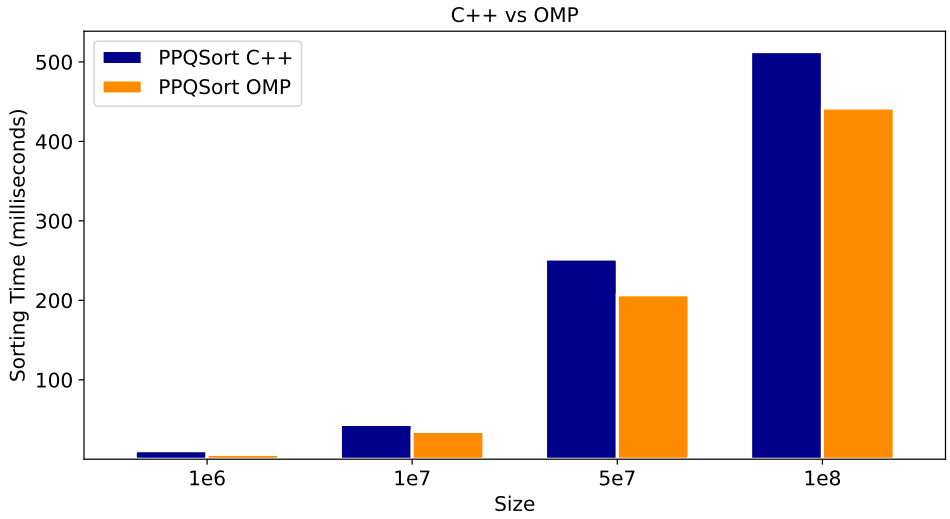


Figure 5.10 This figure showcases the scalability of our implementations by comparing their performance on sorting random integer (int) data with varying input sizes. While the data cardinality remains constant at int maximum ($n \approx$ 4e9), the figure explores how sorting time changes as the overall data size increases.

Figure 5.11 compares our implementations' performance when sorting sparse matrices from the real world. The results are encouraging, demonstrating that both implementations achieve excellent sorting times. In particular, the figure highlights the remarkable similarity in performance between the two approaches.

For detailed benchmark results, refer to Appendix A).

5.5 Comparing with Other Implementations

We conducted benchmarks against various parallel quicksort algorithms to assess our implementation comprehensively. In our benchmarks, `GCC BQS` denotes the balanced libstdc++ quicksort, while `GCC QS` refers to the unbalanced libstdc++ quicksort (see Table 3.1 for further details). In the attachments (Appendix A), we provide a comparison with additional high-performance parallel sorting algorithms, not limited to parallel quicksort algorithms. We conducted performance tests on four clusters to thoroughly assess our implementation. We always used the maximum available cores for these benchmarks, resulting in a 1:1 thread-to-core mapping.

5.5.1 STAR Cluster

On the STAR cluster (refer to Table 5.1 for more details about the testing environment), our PPQSort implementation exhibits exceptional performance. It consistently outperforms the state-of-the-art parallel quicksort implementations from GCC in nearly all test cases, sometimes by significant margins.

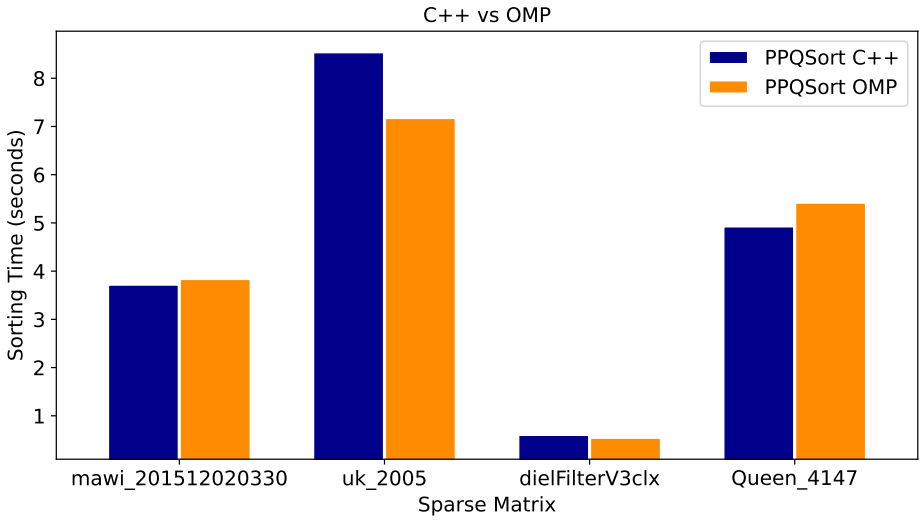


Figure 5.11 Comparison of our implementation's performance on various sparse matrices. These matrices are detailed in Table 5.3.

Data patterns

Figure 5.12 presents the results of sorting differently distributed input data using various algorithms. The graph shows that our PPQSort algorithm exhibits superior performance on all patterns. For randomly distributed data, it outperforms GCC BQS, by a factor of approximately 2.26.

The results for the ascending data distribution show that the oneTBB parallel sort is superior. However, this advantage stems from a pre-sorting check it performs before any actual sorting operation. While this check benefits already-sorted data, it introduces overhead for other distributions, potentially slowing down the algorithm. In contrast, cpp11sort shows a noticeable slowdown of approximately 1.62x for ascending data. On the other hand, our PPQSort implementation maintains impressive performance even without a pre-sorting check. It outperforms GCC implementations by approximately 2.43x.

Although our PPQSort implementation remains the fastest for descending data distributions, the performance gap compared to other fast implementations narrows in this case. However, PPQSort still maintains its lead, demonstrating its effectiveness on the descending data distribution.

Rotated data distribution is exciting in the context of adaptive algorithms. Although it presents an opportunity for these algorithms to exploit data characteristics and improve performance potentially, it can also lead to significant slowdowns. Our benchmarks demonstrate this trade-off. The rotated distribution results in a noticeable slowdown for the GCC algorithms. However, the MPQsort algorithm exhibits excellent speed when handling this data distribution. Despite this, the PPQSort OMP still proves to be significantly faster (approximately 1.88 times) compared to MPQsort, even when dealing with rotated distributions.

The OrganPipe distribution, known for its inherent symmetry, presents another in-

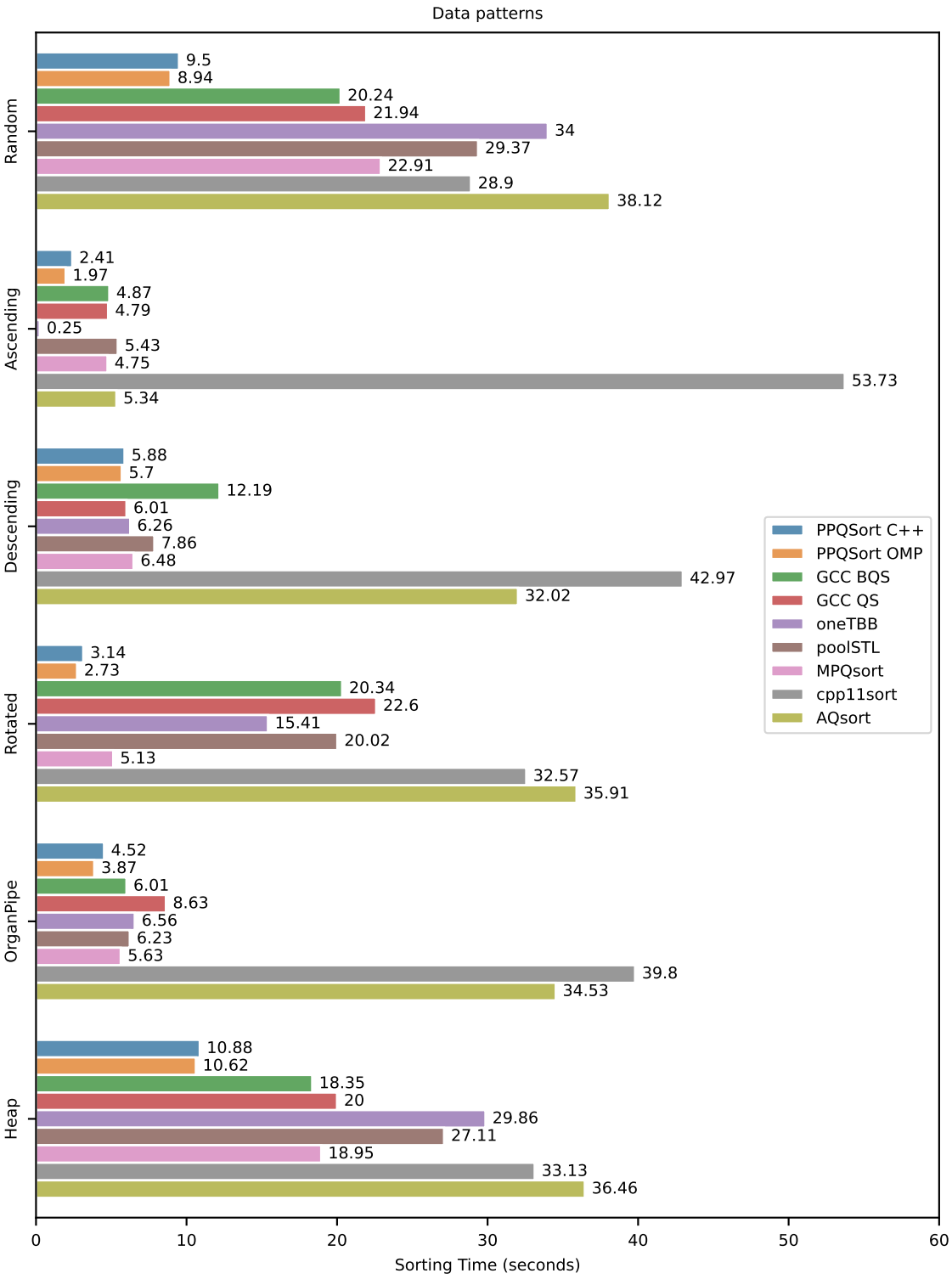


Figure 5.12 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing differently distributed input data in a massive dataset of **2 billion** (2e9) **integers**, running on **STAR** cluster.

teresting test case. Although the sorting times for various algorithms are similar, our PPQSort implementation still emerges as the fastest. We hypothesize that PPQSort ability to effectively detect bad partition and shuffle future pivot candidates plays a crucial role in its advantage. This capability allows PPQSort to disrupt the symmetrical nature of OrganPipe data, potentially leading to more efficient partitioning and sorting in this specific scenario.

The final data pattern we evaluated involved elements prearranged in a heap structure (refer to Chapter 5.2 for details). Interestingly, all the sorting algorithms exhibited a slowdown when handling these data. This behavior can be attributed to the inherent symmetry present in the heap structures. Breaking this symmetry to achieve efficient sorting is more complex than other patterns, such as organ-pipe distribution. Despite this slowdown, our implementations remained the fastest overall. This highlights the robustness of our implementations, even under challenging data conditions.

Low Cardinality and Small Size

We have also compared the algorithm's performance when sorting randomly generated data with low cardinality (few unique values). In low-cardinality data, many elements are identical. Here, an algorithm's ability to recognize and adapt to such data becomes crucial to avoid slowdowns. Our algorithm excels in this scenario due to its duplicate pivot check. This check significantly improves the sorting speed for low-cardinality data. Although oneTBB sort might be faster for inputs with cardinality 1 (all elements are identical) due to its presort check, our algorithm remains highly competitive for most cases of low cardinality. Furthermore, AQsort also shows excellent speed for low-cardinality arrays (Figure 5.13).

We compared the algorithm's performance on smaller, randomly generated arrays to verify their ability to switch to a more efficient sequential sort at the right time. The benchmark suite repeats these tests multiple times to account for potential variations – typically $1,000$ times for the smallest size and around 10 times for larger sizes. As shown in Figure 5.14, while the differences are minor, our implementation consistently outperforms others due to its optimized sequential version.

Matrices

Finally, we compare algorithms for sorting sparse matrices (see Table 5.3 for additional information). Figure 5.15 reveals some intriguing observations. Most algorithms yield similar results. Typically, our implementations are the fastest; when they are not, the differences are negligible. Our performance consistently delivers impressive speeds, frequently exceeding those of GCC implementations.

5.5.2 ClusterFIT Intel

We ran the benchmarks on ClusterFIT Intel, a cluster with more cores (36) and faster CPUs (details in Table 5.1). As expected, all algorithms exhibited performance improvements compared to the STAR cluster (see Figure 5.16). Our implementations maintained their dominance for randomly distributed data. Interestingly, cpp11sort demonstrated a dramatic speedup, highlighting its ability to take advantage of additional cores. However,

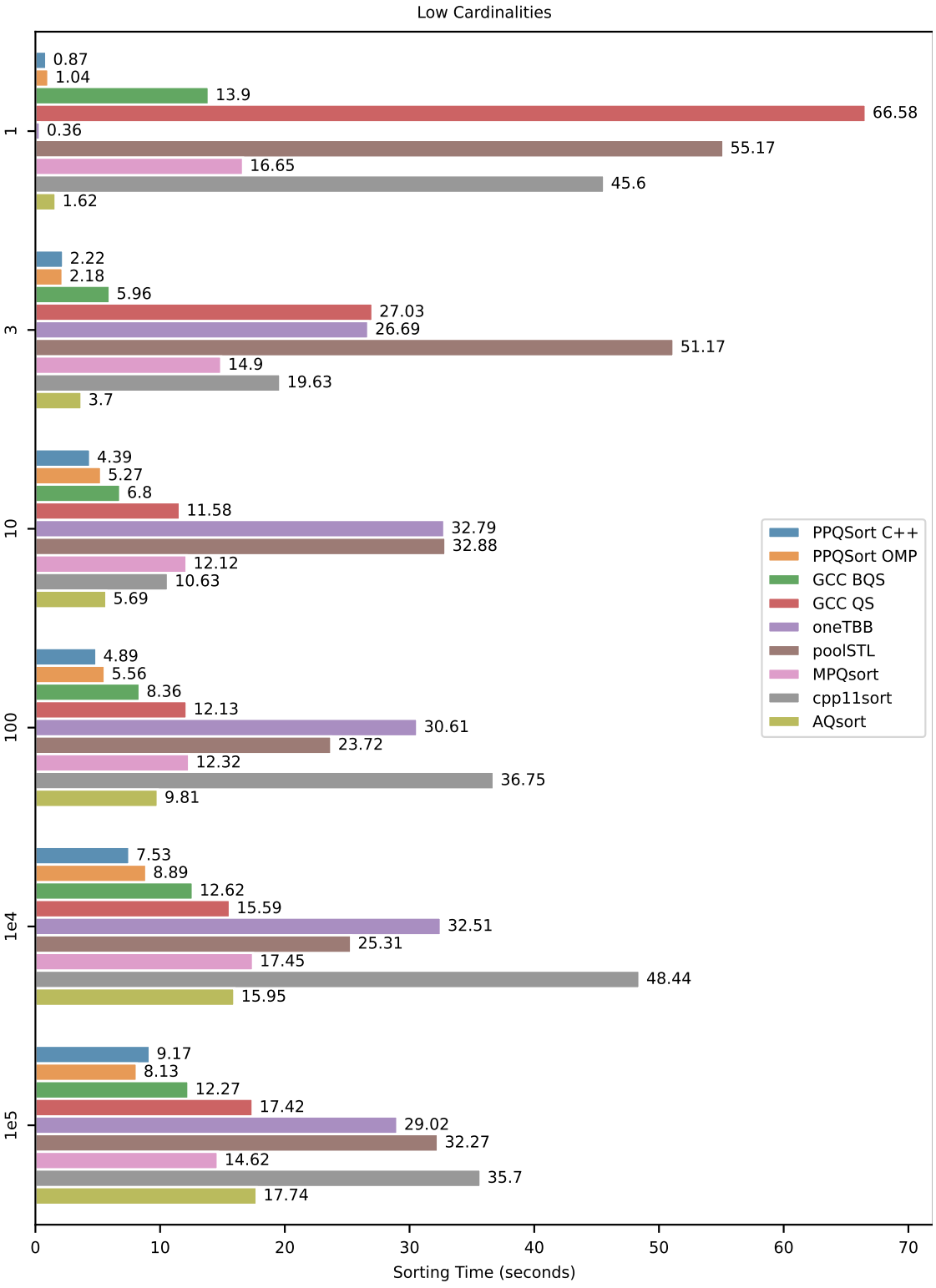


Figure 5.13 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing dataset of **2 billion** (2e9) **integers** with **low cardinality** and distributed **randomly**. Benchmarks were conducted on the **STAR** cluster.

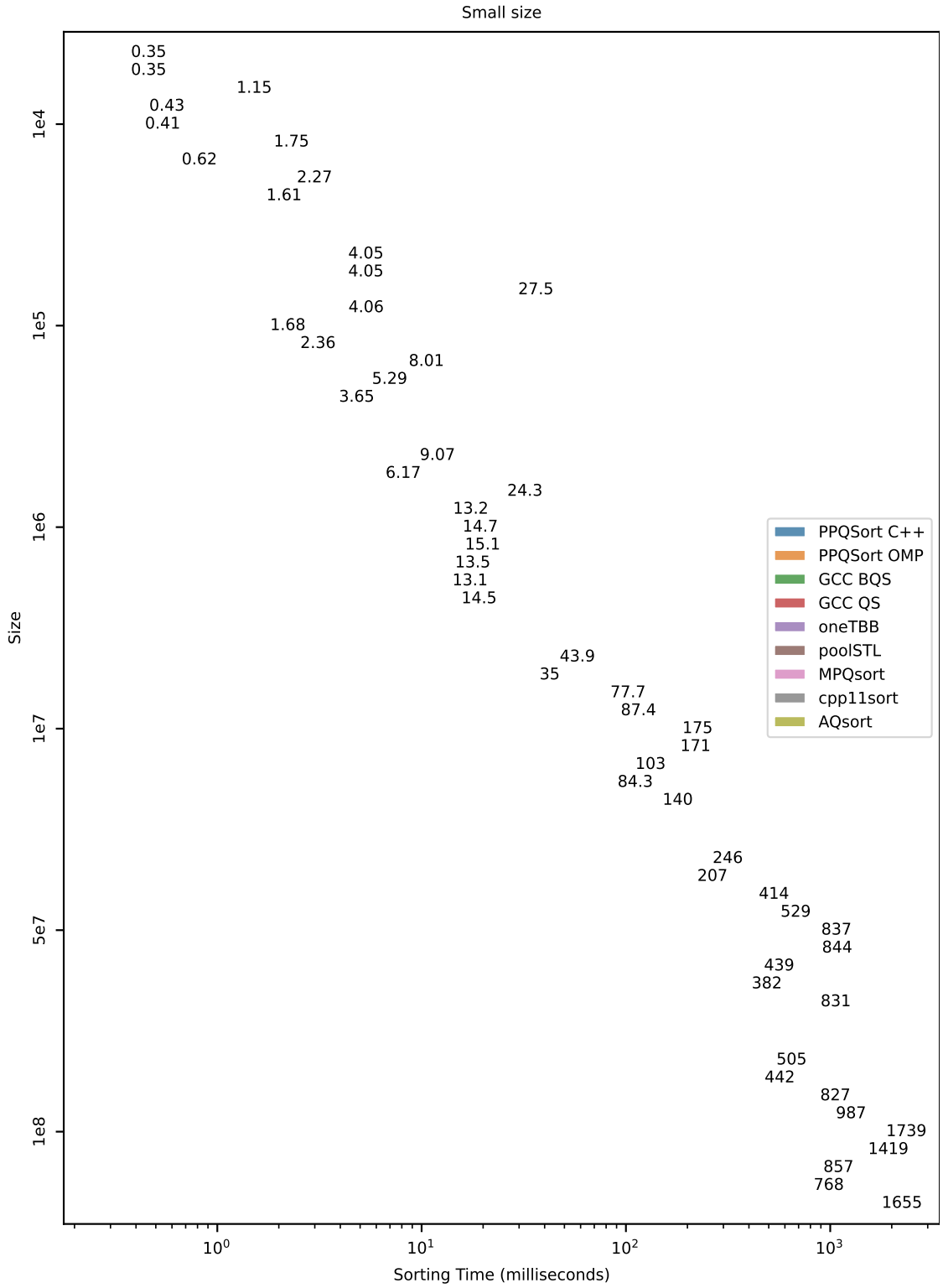


Figure 5.14 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing smaller datasets with **randomly** distributed **integers**. Benchmarks were conducted on the **STAR** cluster.

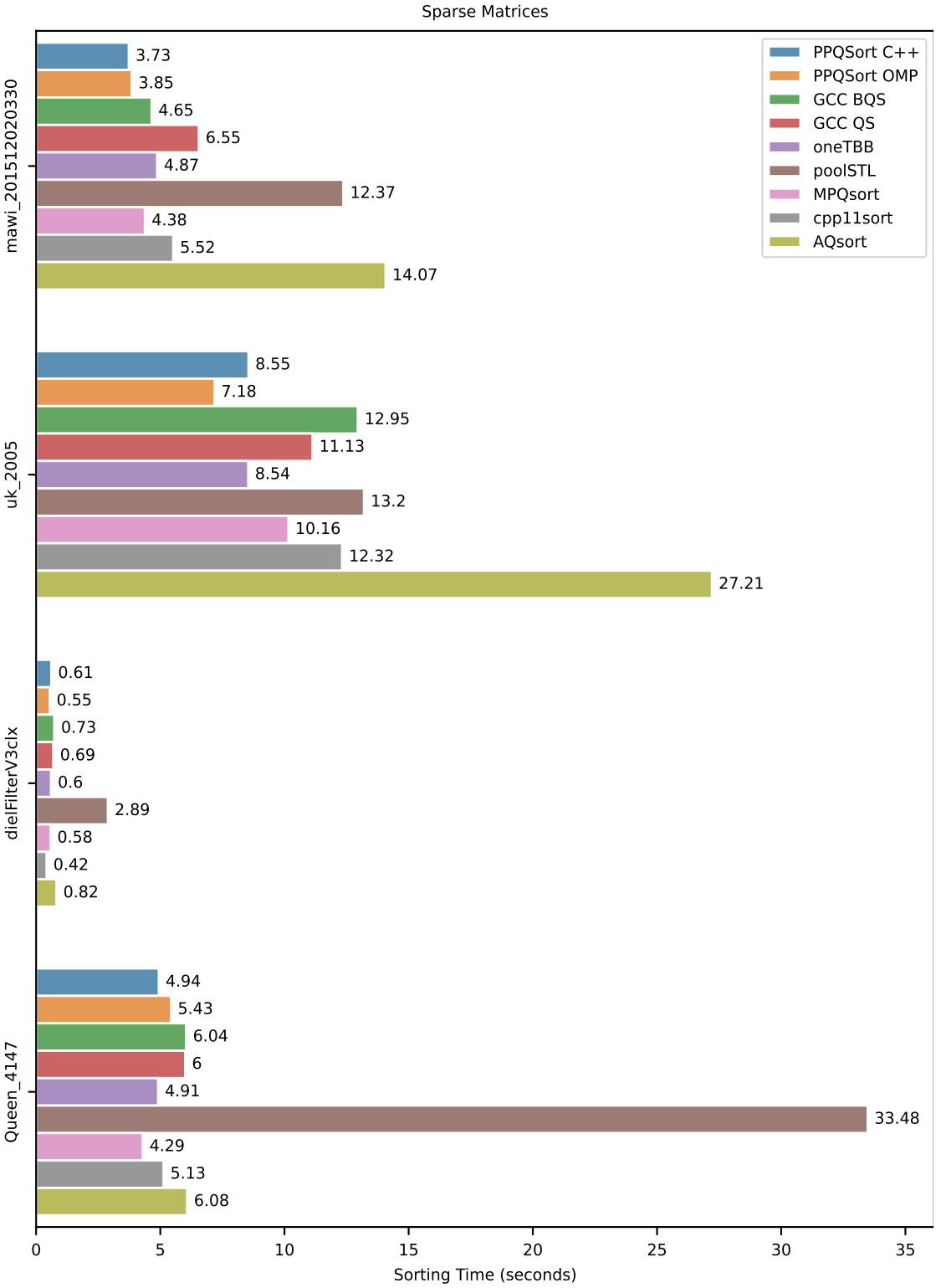


Figure 5.15 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing different sparse matrices. Benchmarks were conducted on the **STAR** cluster.

the C++ implementation of our PPQSort exhibited remarkable robustness by providing excellent sorting speeds, outperforming the second-fastest cpp11sort by a factor of 1.56x on randomly distributed data even when running on this different hardware setup.

For ascendingly distributed data, our PPQSort C++ variant still exhibited excellent speed, and the OMP variant was slightly slower but still maintained great performance. All algorithms performed as expected for descendingly distributed data except cpp11sort, which showed remarkable speed and was the fastest.

PPQSort dominance trend extends to organ pipe and heap data, as well as for smaller input sizes and low cardinality data (see Figures 5.17 and 5.18). Our implementations consistently rank among the fastest, often taking the lead. While cpp11sort again shows remarkable sorting times, other algorithms exhibit varying performance, some experiencing slowdowns and others showing speedups.

Although the sorting times for the sparse matrices exhibit some variation (see Figure 5.19), our implementations continue demonstrating impressive performance. They consistently rank among the fastest algorithms, particularly for large and complex matrices like mawi_201512020330 and uk-2005. The performance difference is minimal for smaller matrices like dielFilterV3clx, where cpp11sort takes the lead. Our OMP implementation remains the second-fastest, and our C++ implementation still ranks within the top five. Cpp11sort emerges as the fastest for the Queen_4147 matrix. However, our C++ implementation maintains a close third position, trailing only a few hundred milliseconds.

In summary, our implementations showcase exceptional robustness. Although not always the absolute leader, they consistently rank among the fastest algorithms, offering excellent performance across a wide range of input data and showcasing great hardware adaptability. Refer to Appendix A for detailed measurements.

5.5.3 ARM Cluster

We conducted performance tests on the ARM Cluster (see Table 5.1). The ARM architecture is notable for its utilization of a weakly ordered memory model, unlike AMD, which uses a more ”strong” ordered model (see Chapter 4.6.2).

On the ARM cluster, the performance appears comparable to that on other hardware, with the PPQSort C++ consistently proving to be the fastest implementation. This is likely attributed to the effective utilization of explicit memory semantics. These semantics allow PPQSort to optimize memory access patterns and potentially benefit from weaker memory ordering of the ARM architecture, leading to performance gains.

Our implementations maintained consistent leadership across various data distributions (see Figure 5.20), except for ascending data where oneTBB took the lead. For both smaller sizes and varying cardinalities (Figures 5.21 and 5.22), our implementations consistently achieved the fastest sorting times.

When considering sparse matrices, cpp11sort exhibited competitive performance, even slightly outperforming our implementations on the mawi_201512020330 matrix. However, our implementations remained dominant across the other matrices tested.

Our implementations consistently ranked among the fastest sorting algorithms on this hardware setup. In particular, the C++ implementation of PPQSort demonstrated a significant speedup due to its effective utilization of memory order semantics.

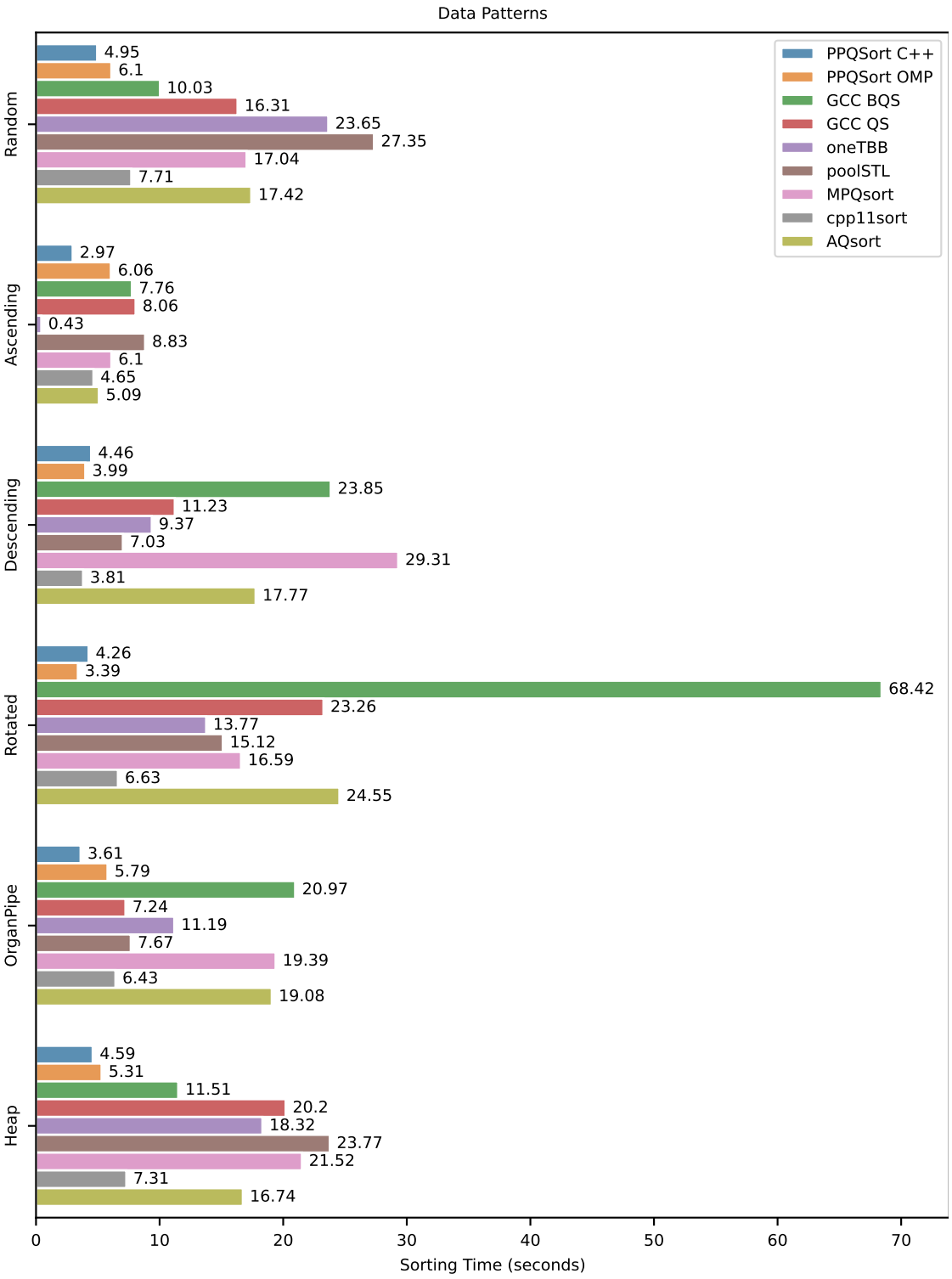


Figure 5.16 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing differently distributed input data in a massive dataset of **2 billion** (2e9) **integers**, running on **Intel** cluster.

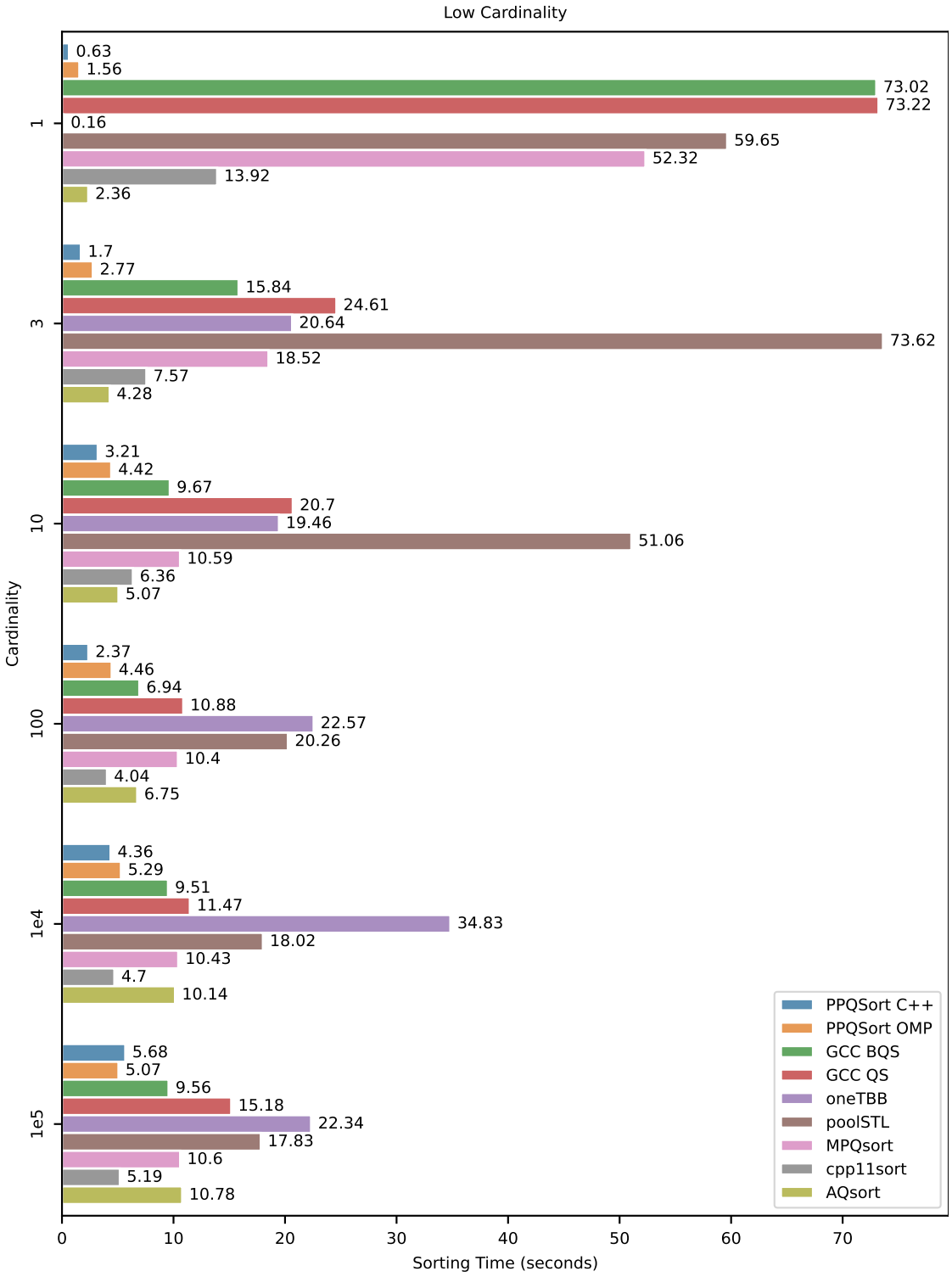


Figure 5.17 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing dataset of **2 billion (2e9) integers** distributed **randomly**. These random integers have a low cardinality (number of unique elements). Benchmarks were conducted on the **Intel** cluster.

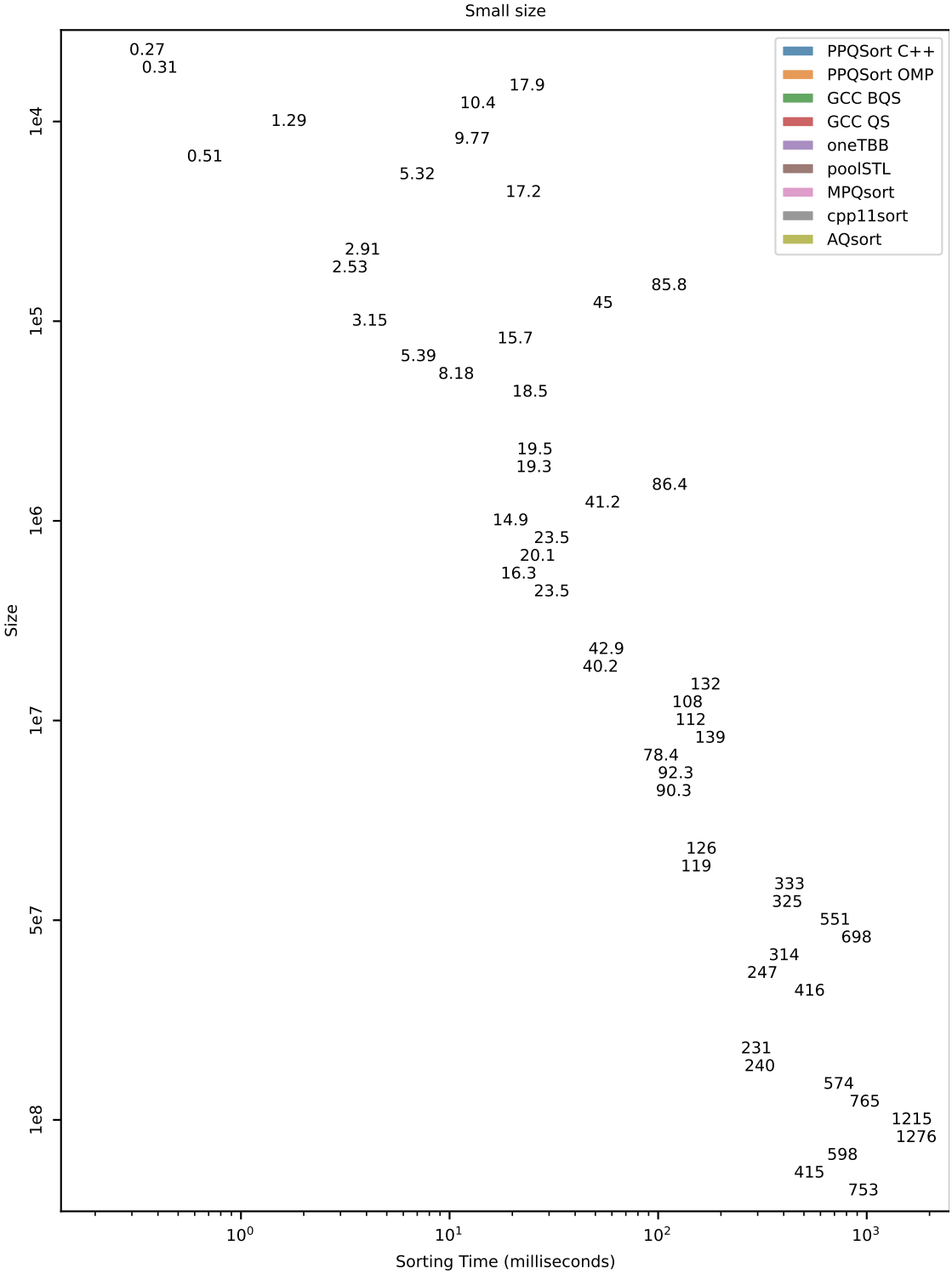


Figure 5.18 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing smaller datasets with **randomly** distributed **integers**. Benchmarks were conducted on the **Intel** cluster.

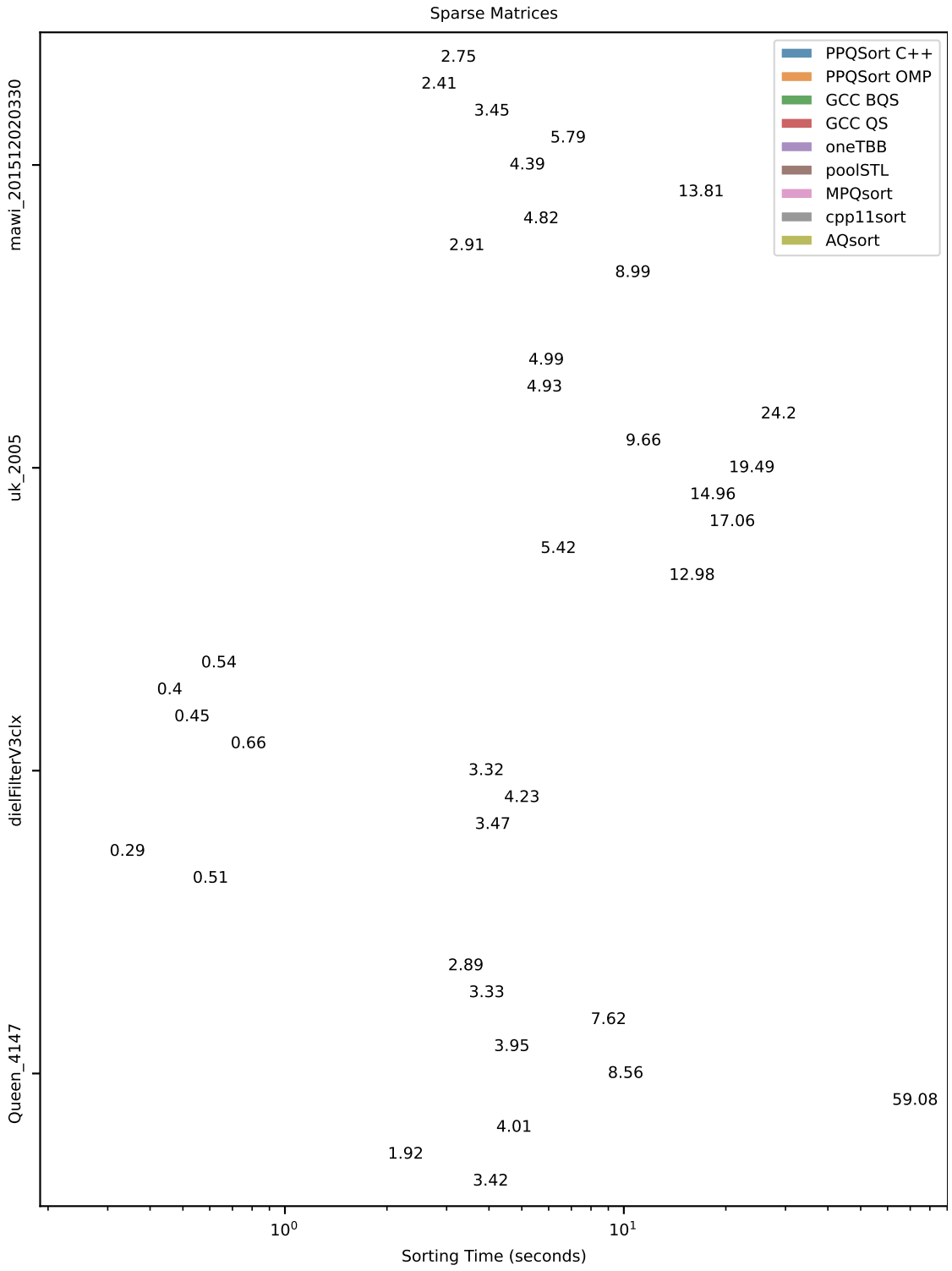


Figure 5.19 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing different sparse matrices. Benchmarks were conducted on the **Intel** cluster.

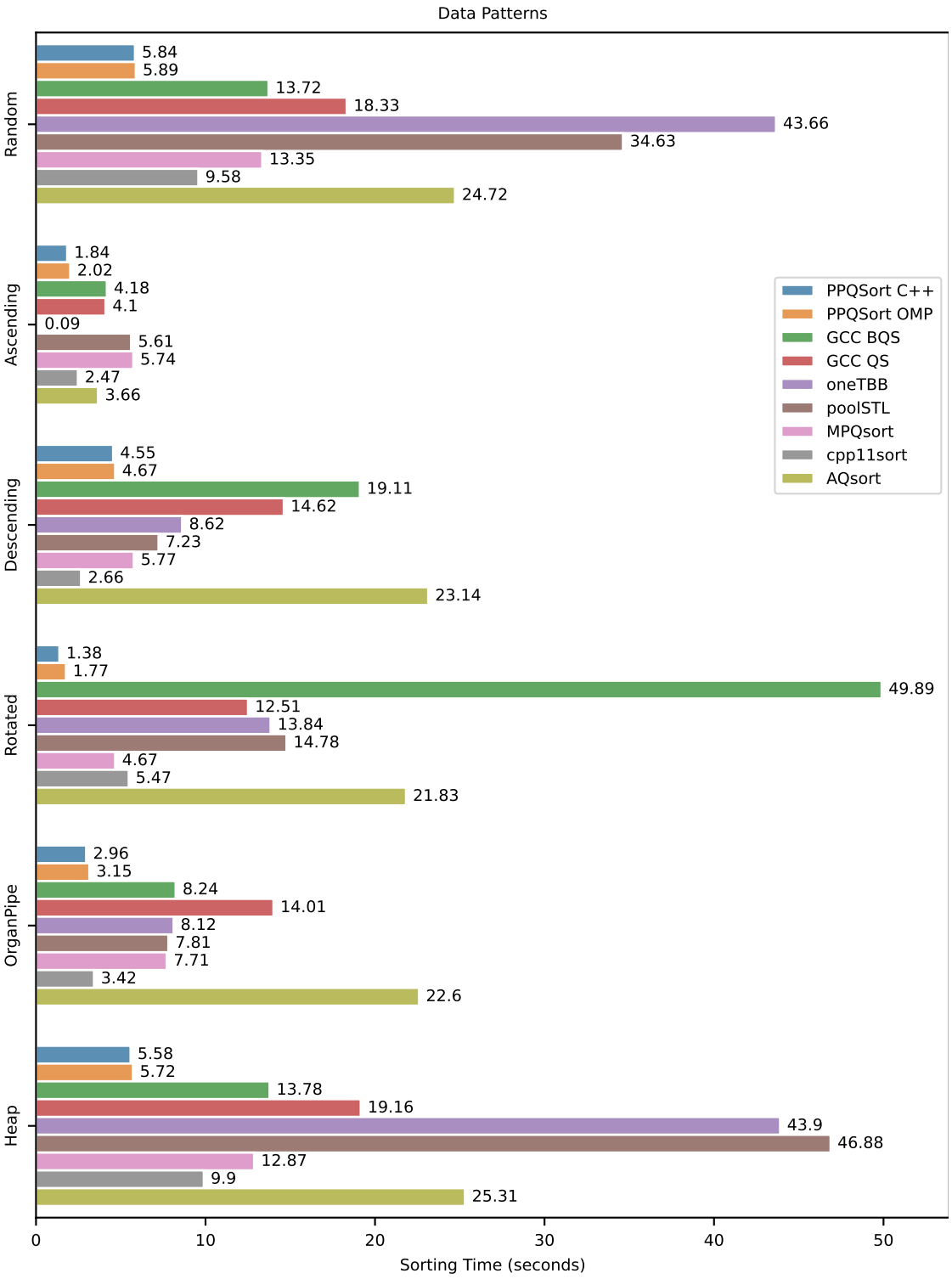


Figure 5.20 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing differently distributed input data in a massive dataset of **2 billion** (2e9) **integers**, running on **ARM** cluster.

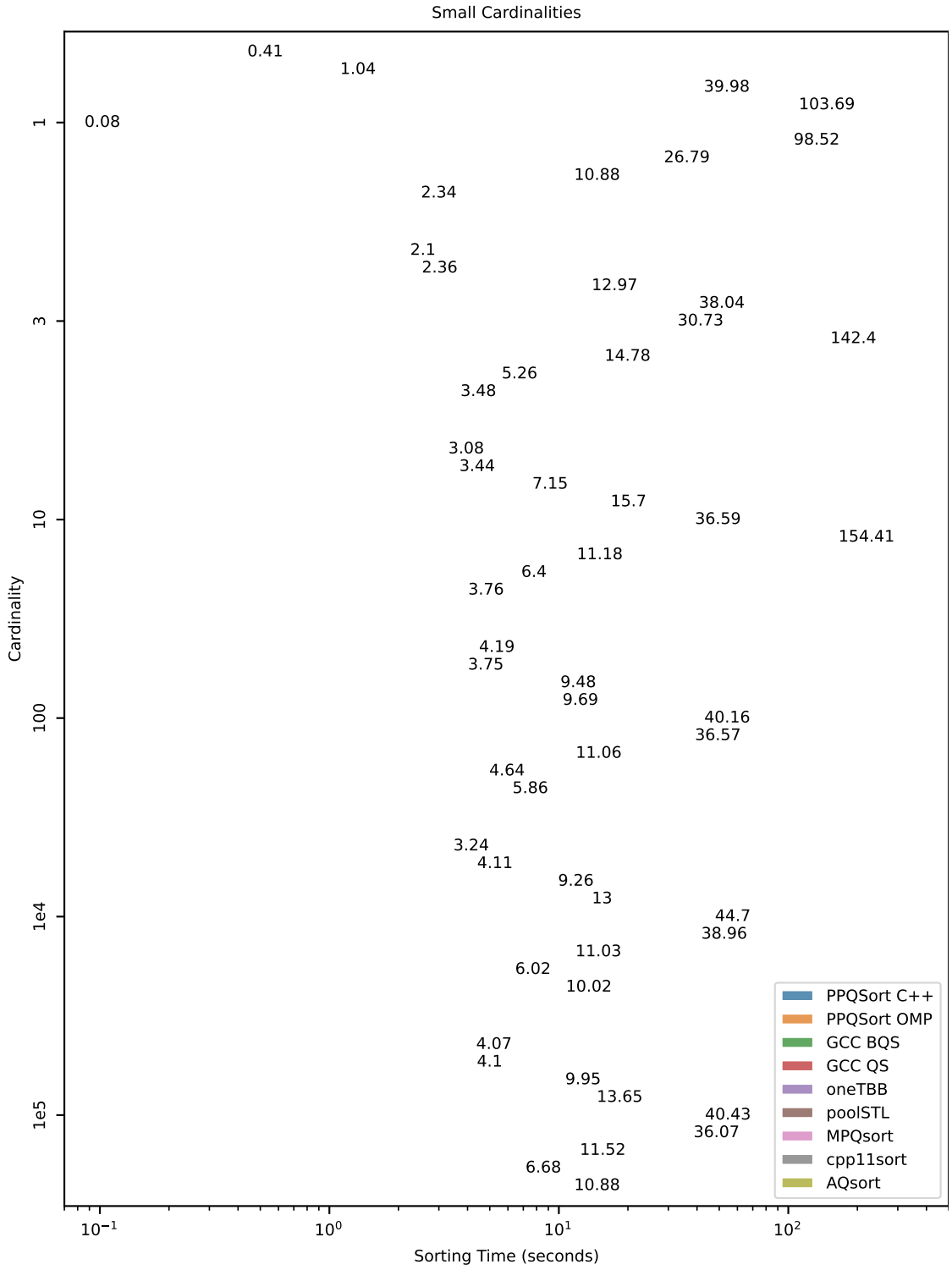


Figure 5.21 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing dataset of **2 billion (2e9) integers** distributed **randomly**. These random integers have a low cardinality (number of unique elements). Benchmarks were conducted on the **ARM** cluster.

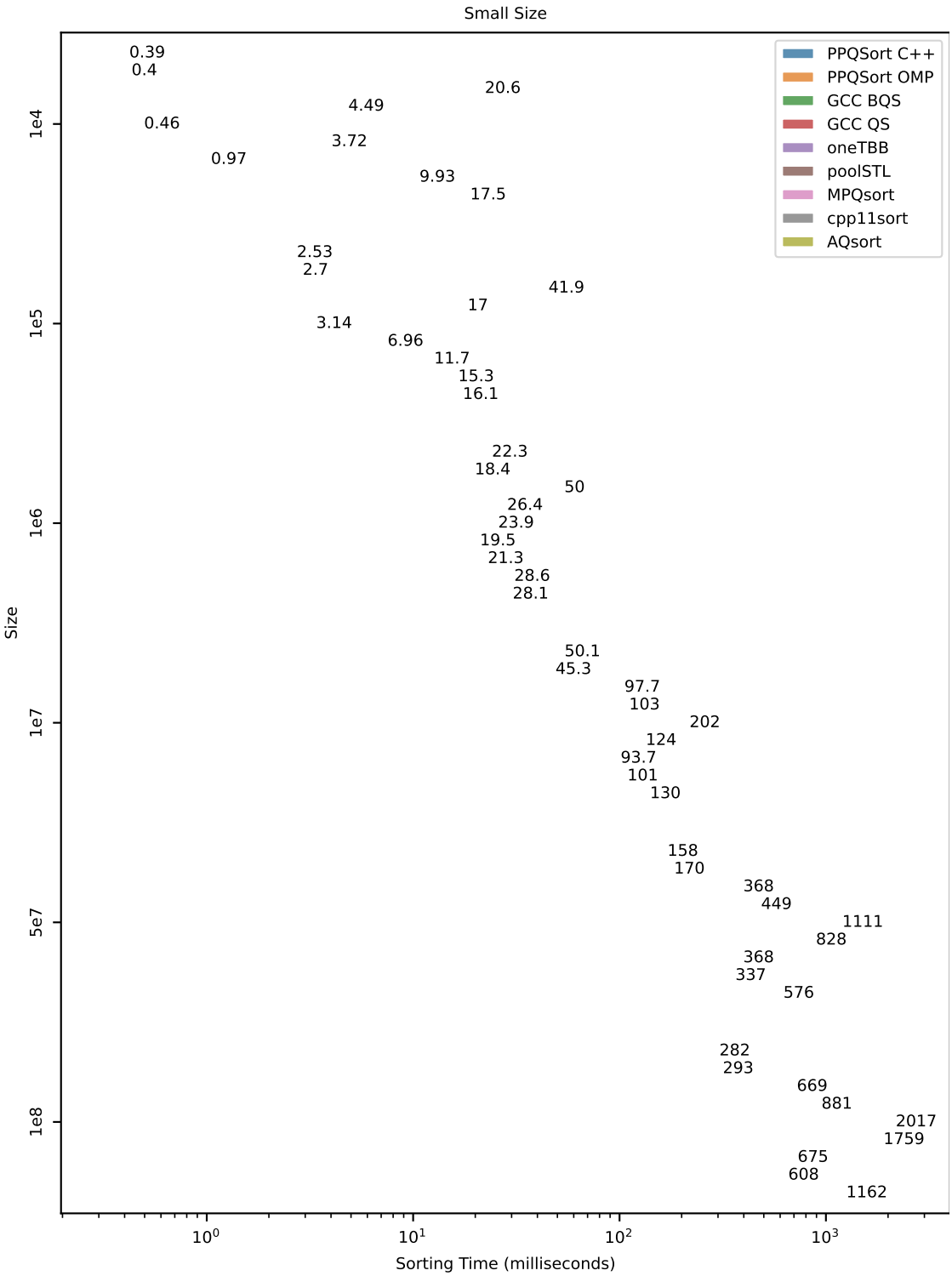


Figure 5.22 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing smaller datasets with **randomly** distributed **integers**. Benchmarks were conducted on the **ARM** cluster.

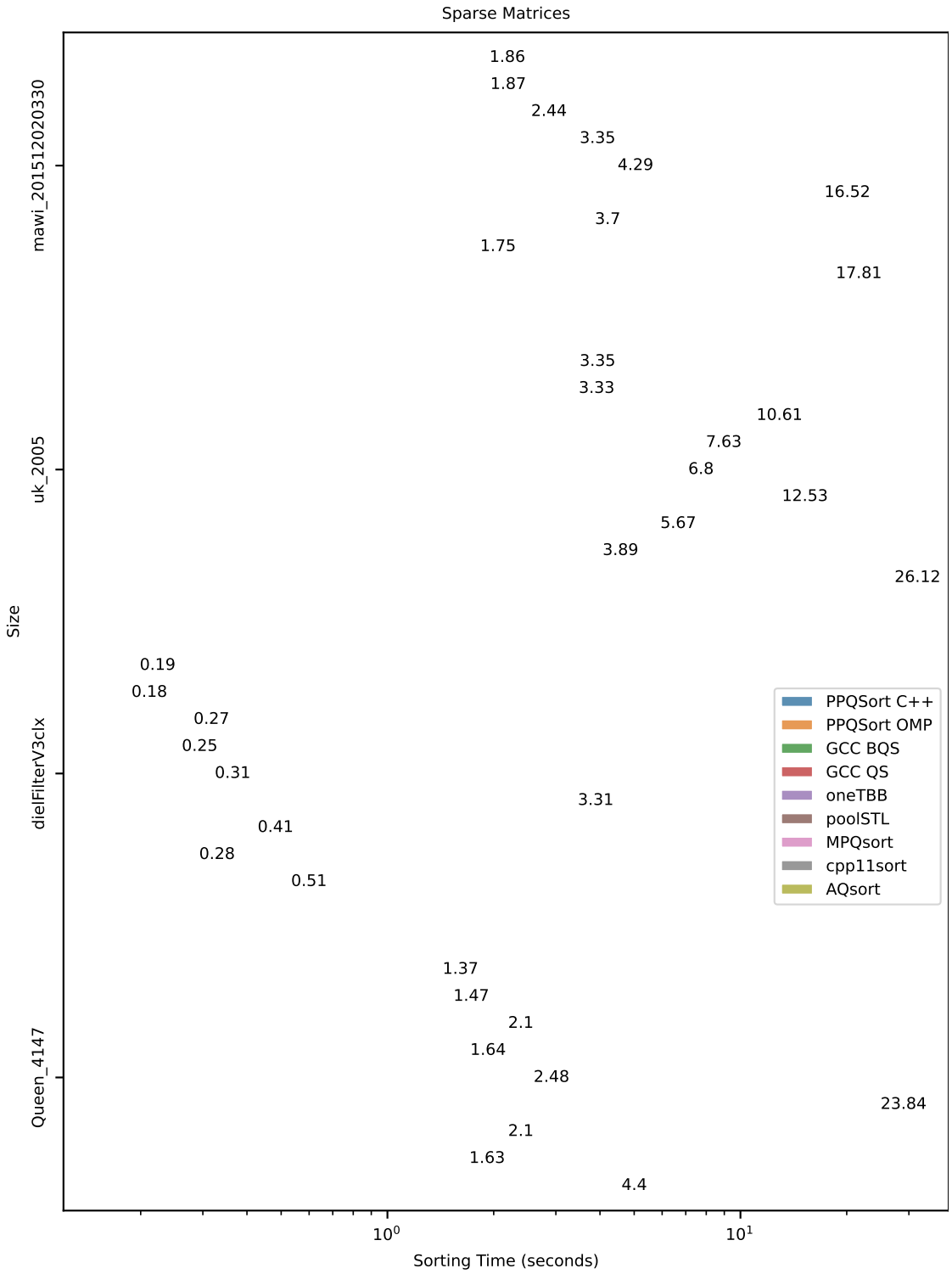


Figure 5.23 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing different sparse matrices. Benchmarks were conducted on the **ARM** cluster.

5.5.4 RCI Cluster

The RCI cluster has multiple computing nodes containing CPUs that have 64 cores. Despite the capability to run 126 threads simultaneously, our benchmarks were specifically carried out using 64 threads on 64 cores. This approach consistently mapped one thread to one CPU core.

The RCI cluster benchmarks echo the trends observed on other hardware configurations. Our PPQSort implementations consistently demonstrate strong performance, frequently ranking as the fastest among the evaluated parallel quicksort algorithms (see Figures 5.24, 5.25, 5.26 and 5.27).

Although the cpp11sort algorithm exhibits exceptional performance on certain input data, often achieving speeds comparable to our implementations, PPQSort remains a leader. This highlights the competitiveness of the sorting landscape.

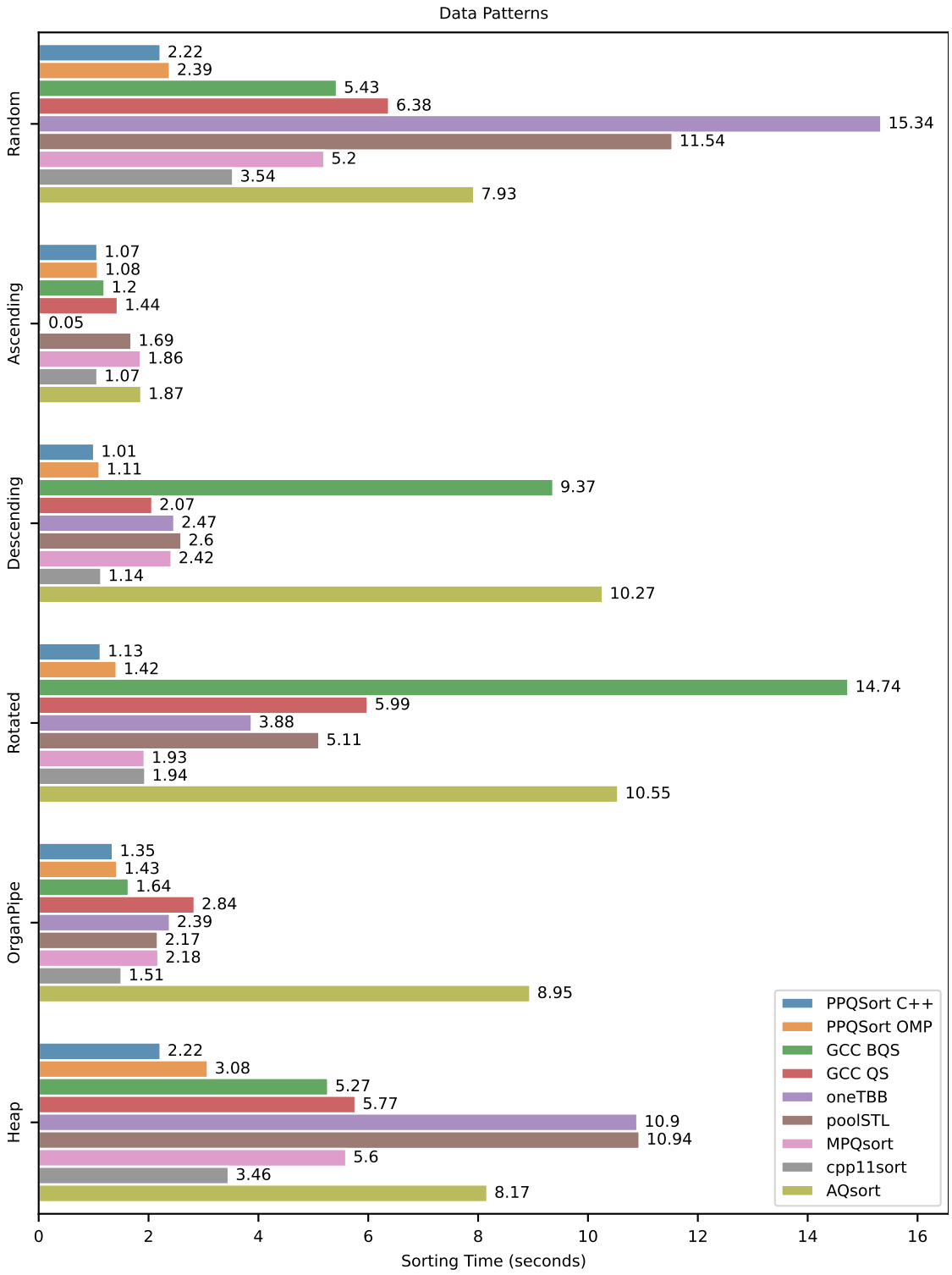


Figure 5.24 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing differently distributed input data in a massive dataset of **2 billion** (2e9) **integers**, running on **RCI** cluster.

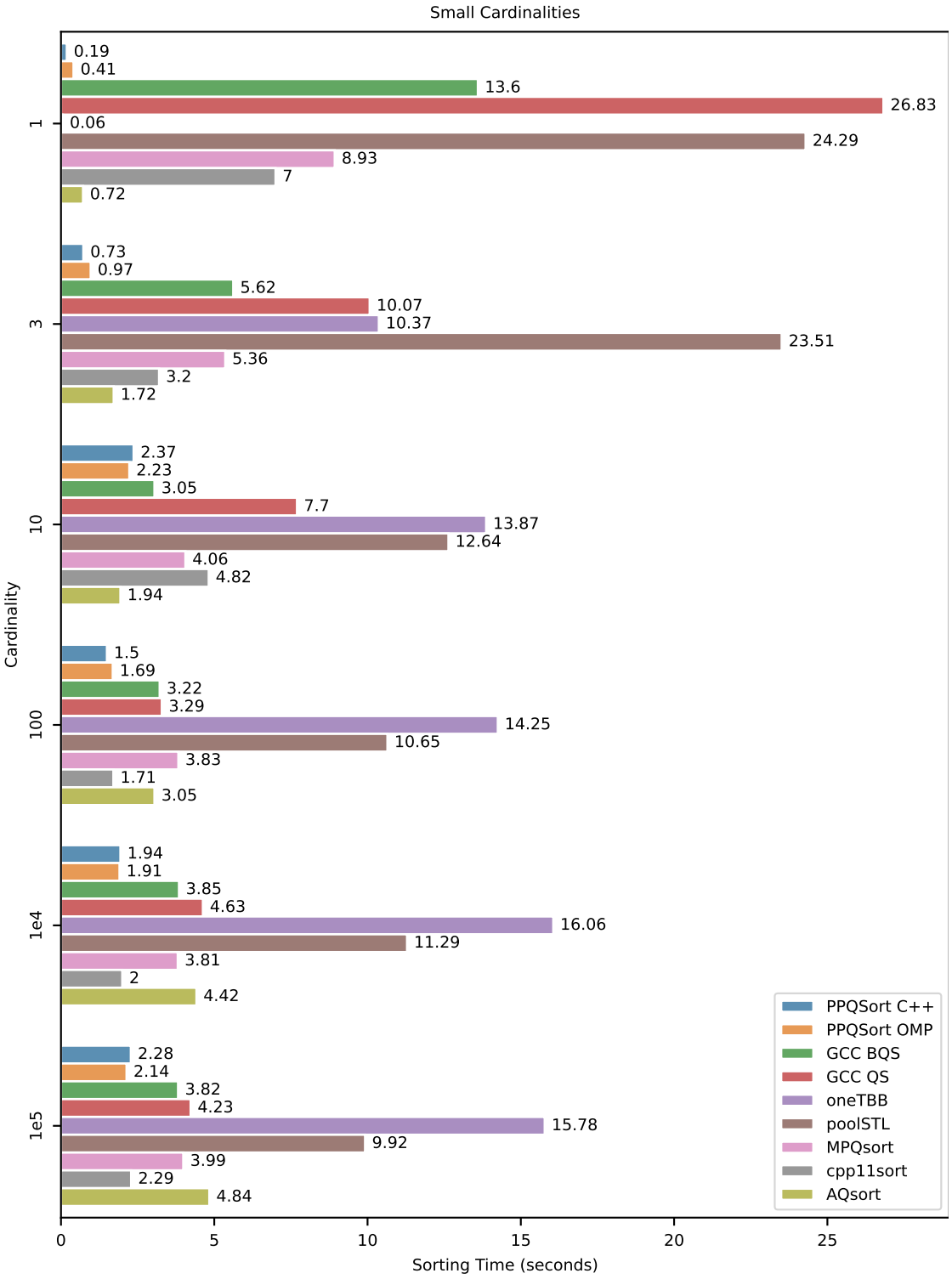


Figure 5.25 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing dataset of **2 billion** (2e9) **integers** distributed **randomly**. These random integers have a low cardinality (number of unique elements). Benchmarks were conducted on the **RCI** cluster.

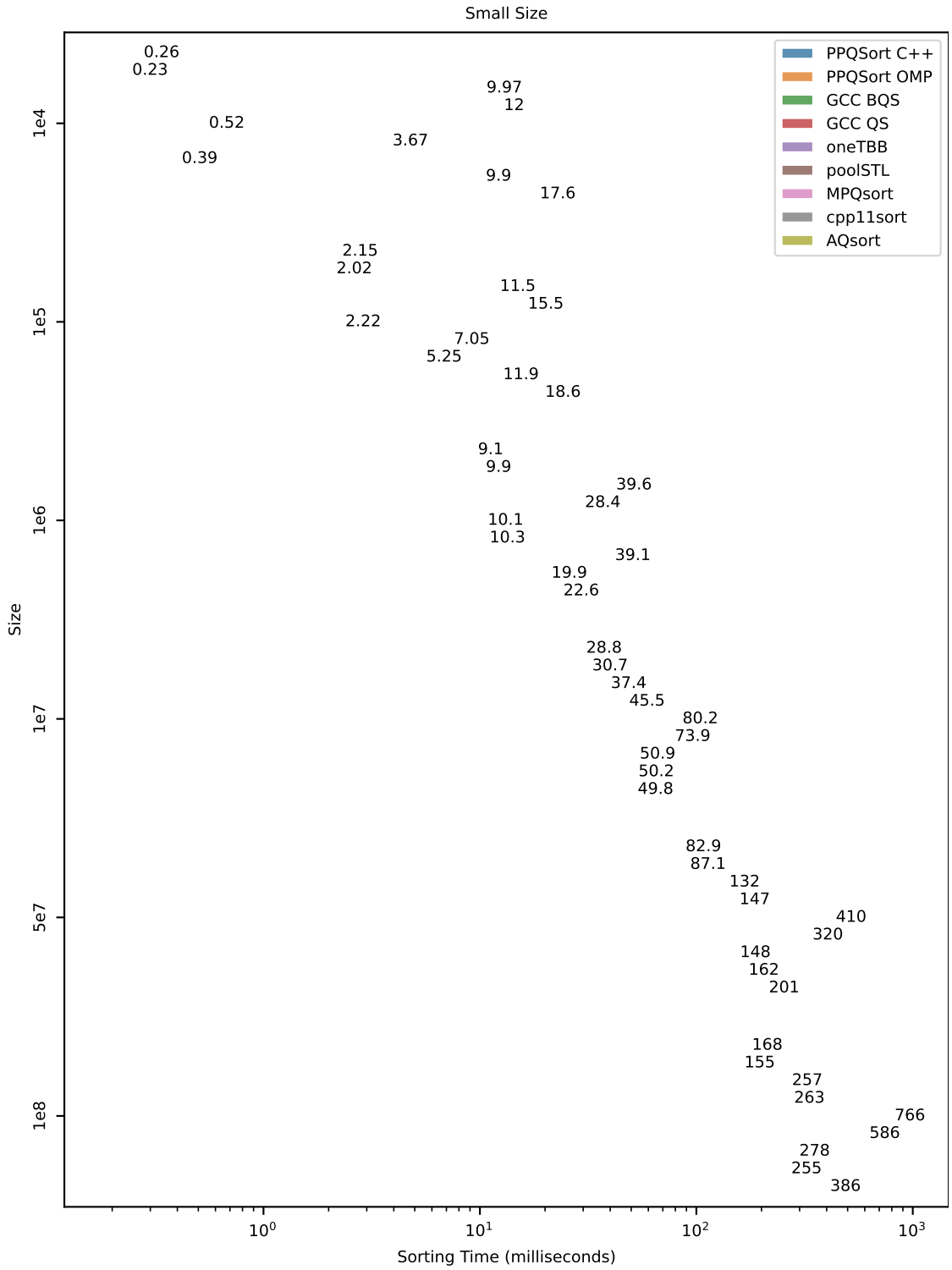


Figure 5.26 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing smaller datasets with **randomly** distributed **integers**. Benchmarks were conducted on the **RCI** cluster.

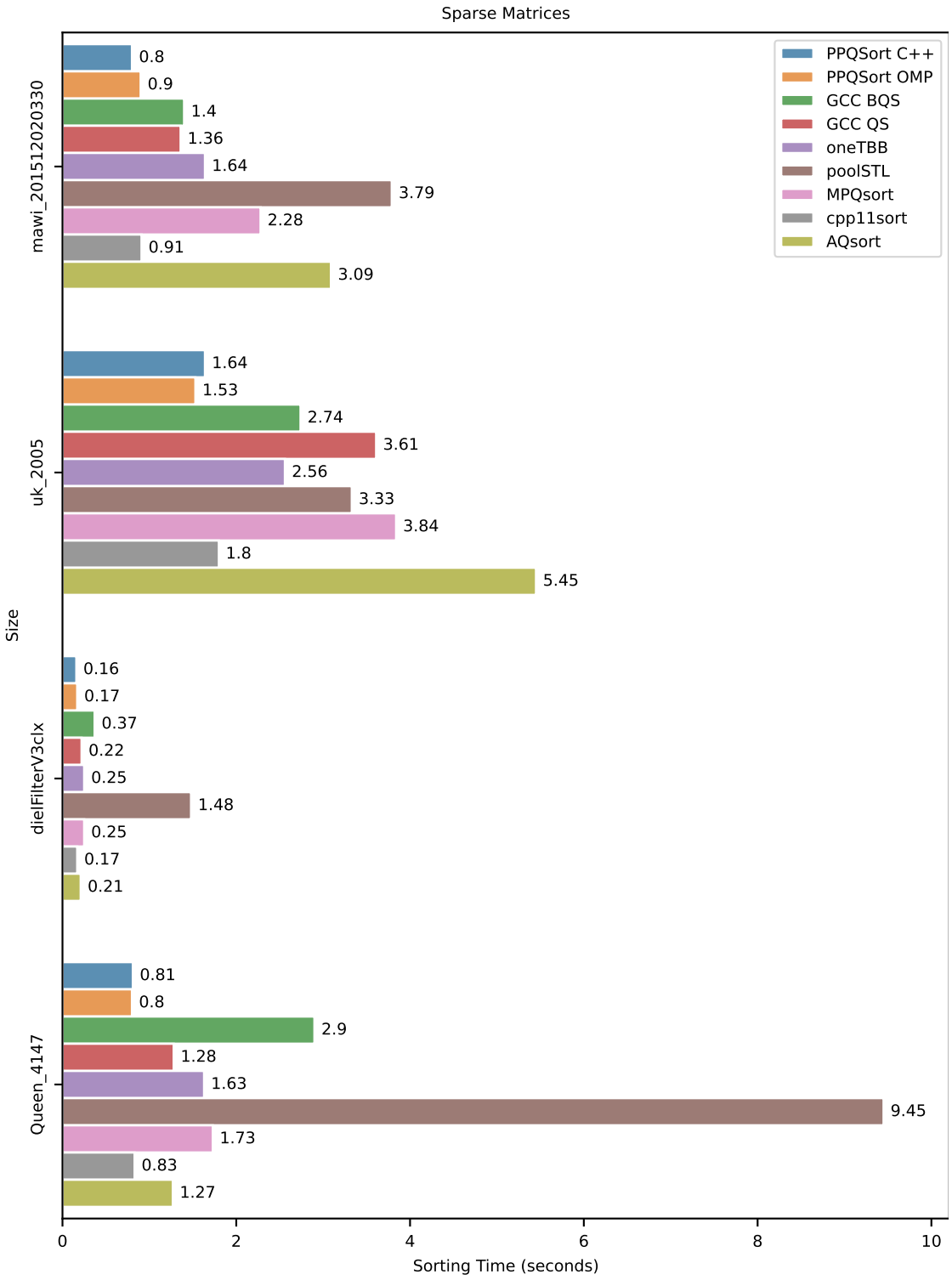


Figure 5.27 Comparison of sorting times for all evaluated parallel Quicksort algorithms when processing different sparse matrices. Benchmarks were conducted on the **RCI** cluster.

Chapter 6
Conclusion

In this paper, we have focused on in-place sorting algorithms and implemented a fast parallel quicksort algorithm.

Initially, in the research section, we have focused on the quicksort algorithm and its potential optimization opportunities. We have also analyzed the potential of combining quicksort with other sorting algorithms. Moreover, we have investigated and analyzed numerous shared memory parallel implementations of quicksort.

Based on the results of the research section, we have designed an effective parallel quicksort algorithm named PPQSort. We have implemented PPQSort initially using OpenMP and later developed a PPQSort implementation using C++ threads. Ultimately, we merged these implementations, resulting in header-only implementation, which can use OpenMP if available or C++ threads, as the user needs. We published the implementation alongside the testing and benchmarking suite.

We ran extensive benchmarks on four different machines. The benchmark results demonstrate that PPQSort is exceptionally fast, outperforming state-of-the-art parallel quicksort implementations on almost all inputs and all machines by a significant margin. For example, on an ARM cluster, PPQSort is approximately 1.6 times faster on random data than the second-fastest parallel quicksort implementation, cpp11sort.

PPQSort also performed exceptionally well in benchmarking against other parallel algorithms, not limited to quicksorts. IPS^{4}o was the only algorithm that surpassed PPQSort in many cases, but it has external dependencies that make it less versatile. Notably, on ARM cluster, PPQSort consistently outperformed all other algorithms including IPS^{4}o.

In future works, exploring the sample sort algorithm and devising a new hybrid sorting algorithm that combines optimizations from PPQsort and IPS^{4}o would be intriguing.

Appendix A

More Measurements

This Appendix presents an in-depth analysis of benchmark results for the hardware configurations outlined in Table 5.1. In many cases, the IPS^{4}o algorithm outperformed our PPQSort implementation. However, it is essential to note that IPS^{4}o relies on external dependencies such as TBB and is tied with GCC's libatomic library. On the other hand, PPQSort utilizes C++20 standard features, making it self-contained and potentially easier to integrate into existing projects without external dependencies. Notably, PPQSort emerged as the fastest implementation on the ARM cluster.

Note that we omitted the Thrust sorting algorithm from the ARM cluster benchmarks as it has shown a tendency to consume all available RAM when working with more complex data types, such as doubles and strings. Additionally, the MPQsort could not sort specific string inputs, which resulted in a segfault. We marked these run results as "X".

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	4.84s	1.73s	2.38s	**1.61s**	2.14s	5.99s	18.69s	3
PPQSort OMP	4.96s	1.68s	2.19s	1.63s	**2.12s**	5.62s	18.2s	2
GCC BQS	11.19s	5.61s	6.63s	18.02s	6.3s	11.8s	59.55s	7
GCC QS	16.52s	2.96s	3.17s	15.65s	2.92s	12.71s	53.93s	6
oneTBB	28.74s	**0.13s**	6.14s	5.36s	5.29s	25.39s	71.05s	9
poolSTL	25.96s	3.69s	4.96s	16.74s	4.62s	23.5s	79.47s	10
MPQsort	13.91s	4.83s	6.04s	4.83s	5.39s	12.9s	47.9s	5
cpp11sort	34.4s	62.58s	63.1s	61.68s	54.08s	50.13s	325.97s	12
AQsort	16.53s	2.42s	30.79s	28.39s	27.53s	17.03s	122.69s	11
Boost	14.95s	1.35s	**1.86s**	4.21s	3.83s	10.16s	36.36s	4
Thrust	18.91s	6.46s	6.04s	7.47s	6.35s	16.58s	61.81s	8
IPS^{4}o	**2.71s**	0.13s	2.42s	2.67s	2.51s	**2.52s**	**12.96s**	1

Table A.1 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **STAR** cluster. The input size was 2e9 and the data type used was **short**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	9.5s	2.41s	5.88s	3.14s	4.52s	10.88s	36.33s	3
PPQSort OMP	8.94s	1.97s	5.7s	**2.73s**	**3.87s**	10.62s	33.83s	2
GCC BQS	20.24s	4.87s	12.19s	20.34s	6.01s	18.35s	82.0s	6
GCC QS	21.94s	4.79s	6.01s	22.6s	8.63s	20.0s	83.97s	7
oneTBB	34.0s	**0.25s**	6.26s	15.41s	6.56s	29.86s	92.34s	8
poolSTL	29.37s	5.43s	7.86s	20.02s	6.23s	27.11s	96.02s	9
MPQsort	22.91s	4.75s	6.48s	5.13s	5.63s	18.95s	63.85s	5
cpp11sort	28.9s	53.73s	42.97s	32.57s	39.8s	33.13s	231.1s	12
AQsort	38.12s	5.34s	32.02s	35.91s	34.53s	36.46s	182.38s	11
Boost	13.78s	1.92s	**2.53s**	5.48s	4.89s	13.42s	42.02s	4
Thrust	44.69s	16.41s	14.06s	16.28s	14.59s	32.38s	138.41s	10
IPS^{4}o	**6.48s**	0.32s	6.01s	4.91s	5.62s	**6.5s**	**29.84s**	1

Table A.2 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **STAR** cluster. The input size was 2e9 and the data type used was **int**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	17.52s	4.22s	9.61s	**5.55s**	7.08s	17.88s	61.86s	2
PPQSort OMP	18.44s	4.57s	7.75s	5.93s	**5.95s**	21.29s	63.93s	3
GCC BQS	23.68s	7.49s	15.66s	32.53s	19.5s	24.63s	123.49s	6
GCC QS	37.45s	8.5s	15.38s	38.67s	10.27s	27.86s	138.13s	8
oneTBB	43.53s	**0.52s**	9.09s	29.34s	10.14s	36.02s	128.64s	7
poolSTL	44.54s	10.96s	14.3s	35.73s	10.57s	38.56s	154.66s	9
MPQsort	25.97s	17.43s	11.28s	9.21s	10.11s	29.19s	103.19s	5
cpp11sort	37.7s	52.49s	38.74s	30.78s	40.91s	38.57s	239.19s	10
AQsort	43.27s	8.17s	44.02s	53.68s	51.54s	38.65s	239.33s	11
Boost	19.9s	3.71s	**4.35s**	9.4s	8.78s	20.82s	66.96s	4
Thrust	69.23s	47.33s	43.77s	42.91s	42.28s	58.19s	303.71s	12
IPS^{4}o	**8.78s**	0.82s	7.89s	7.01s	7.99s	**9.61s**	**42.1s**	1

Table A.3 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **STAR** cluster. The input size was 2e9 and the data type used was **double**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	7.91s	1.38s	4.7s	5.24s	5.36s	10.99s	35.58s	3
PPQSort OMP	6.83s	1.85s	4.89s	5.08s	3.81s	8.62s	31.08s	2
GCC BQS	5.76s	6.56s	13.63s	42.01s	10.7s	8.45s	87.11s	9
GCC QS	13.72s	12.05s	11.95s	16.01s	12.09s	20.94s	86.76s	8
oneTBB	16.26s	**0.8s**	15.88s	15.96s	16.87s	20.45s	86.22s	7
poolSTL	23.59s	14.99s	18.26s	12.97s	15.69s	36.45s	121.95s	11
MPQsort	30.21s	X	X	X	X	X	X	X
cpp11sort	12.86s	12.12s	11.85s	11.89s	11.8s	17.71s	78.23s	6
AQsort	7.45s	5.22s	5.04s	4.54s	5.94s	8.99s	37.18s	4
Boost	8.15s	5.61s	5.69s	11.86s	4.84s	7.46s	43.61s	5
Thrust	16.09s	14.9s	14.57s	14.25s	15.22s	24.73s	99.76s	10
IPS^{4}o	**2.71s**	0.82s	**1.38s**	**1.28s**	**1.58s**	**1.53s**	**9.3s**	1

Table A.4 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **STAR** cluster. The input size was 2e7 and the data type used was **std::string**. Some algorithms faced errors during the sorting process, which resulted in missing results for certain patterns.

Algorithm	Cardinality 1	Cardinality 3	Cardinality 10	Cardinality 100	Cardinality 1e4	Cardinality 1e5	Total	Rank
PPQSort C++	0.87s	2.22s	4.39s	4.89s	7.53s	9.17s	29.07s	2
PPQSort OMP	1.04s	**2.18s**	5.27s	5.56s	8.89s	8.13s	31.07s	3
GCC BQS	13.9s	5.96s	6.8s	8.36s	12.62s	12.27s	59.91s	6
GCC QS	66.58s	27.03s	11.58s	12.13s	15.59s	17.42s	150.33s	9
oneTBB	**0.36s**	26.69s	32.79s	30.61s	32.51s	29.02s	151.98s	10
poolSTL	55.17s	51.17s	32.88s	23.72s	25.31s	32.27s	220.52s	12
MPQsort	16.65s	14.9s	12.12s	12.32s	17.45s	14.62s	88.06s	8
cpp11sort	45.6s	19.63s	10.63s	36.75s	48.44s	35.7s	196.75s	11
AQsort	1.62s	3.7s	5.69s	9.81s	15.95s	17.74s	54.51s	5
Boost	1.43s	4.57s	5.57s	8.27s	9.18s	15.61s	44.63s	4
Thrust	12.13s	13.01s	15.6s	13.69s	16.16s	15.9s	86.49s	7
IPS^{4}o	0.37s	3.31s	**2.23s**	**1.88s**	**3.41s**	**3.6s**	**14.8s**	1

Table A.5 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **STAR** cluster. The input size was 2e9 and the data type we used was **int**.

Algorithm	Size 1e4	Size 1e5	Size 1e6	Size 1e7	Size 5e7	Size 1e8	Total	Rank
PPQSort C++	**0.35ms**	4.05ms	9.07ms	43.9ms	246.0ms	505.0ms	808.37ms	3
PPQSort OMP	0.35ms	4.05ms	6.17ms	**35.0ms**	**207.0ms**	442.0ms	694.57ms	2
GCC BQS	1.15ms	27.5ms	24.3ms	77.7ms	414.0ms	827.0ms	1371.65ms	6
GCC QS	0.43ms	4.06ms	13.2ms	87.4ms	529.0ms	987.0ms	1621.09ms	8
oneTBB	0.41ms	1.68ms	14.7ms	175.0ms	837.0ms	1739.0ms	2767.79ms	11
poolSTL	1.75ms	2.36ms	15.1ms	171.0ms	844.0ms	1419.0ms	2453.21ms	9
MPQsort	0.62ms	8.01ms	13.5ms	103.0ms	439.0ms	857.0ms	1421.13ms	7
cpp11sort	2.27ms	5.29ms	13.1ms	84.3ms	382.0ms	768.0ms	1254.96ms	5
AQsort	1.61ms	3.65ms	14.5ms	140.0ms	831.0ms	1655.0ms	2645.76ms	10
Boost	0.35ms	4.1ms	24.4ms	50.7ms	283.0ms	590.0ms	952.55ms	4
Thrust	75.1ms	70.8ms	86.1ms	307.0ms	987.0ms	1912.0ms	3438.0ms	12
IPS^{4}o	0.38ms	**1.28ms**	**5.7ms**	37.9ms	295.0ms	**350.0ms**	**690.26ms**	1

Table A.6 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **STAR** cluster. The data type we used **int**.

Algorithm	mawi_201512020330	uk-2005	dielFilterV3clx	Queen_4147	Total	Rank
PPQSort C++	**3.73s**	8.55s	0.61s	4.94s	17.83s	2
PPQSort OMP	3.85s	**7.18s**	0.55s	5.43s	**17.01s**	1
GCC BQS	4.65s	12.95s	0.73s	6.04s	24.37s	8
GCC QS	6.55s	11.13s	0.69s	6.0s	24.37s	9
oneTBB	4.87s	8.54s	0.6s	4.91s	18.92s	4
poolSTL	12.37s	13.2s	2.89s	33.48s	61.94s	11
MPQsort	4.38s	10.16s	0.58s	4.29s	19.41s	5
cpp11sort	5.52s	12.32s	**0.42s**	5.13s	23.39s	7
AQsort	14.07s	27.21s	0.82s	6.08s	48.18s	10
Boost	5.29s	12.29s	0.48s	**2.98s**	21.04s	6
Thrust	21.39s	45.78s	1.87s	15.9s	84.94s	12
IPS^{4}o	3.93s	10.59s	0.45s	3.07s	18.04s	3

Table A.7 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on sparse matrices, conducted on the **STAR** cluster.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	2.66s	1.34s	1.94s	3.96s	1.54s	3.24s	14.68s	2
PPQSort OMP	3.1s	3.92s	2.02s	1.77s	3.57s	3.76s	18.14s	3
GCC BQS	7.72s	13.64s	13.6s	33.59s	13.04s	8.97s	90.56s	11
GCC QS	11.19s	6.08s	6.97s	12.62s	8.27s	11.78s	56.91s	7
oneTBB	18.11s	**0.28s**	5.9s	13.86s	6.31s	18.09s	62.55s	8
poolSTL	21.16s	4.7s	4.62s	10.52s	4.47s	20.19s	65.66s	9
MPQsort	12.52s	4.59s	7.94s	5.65s	12.01s	9.14s	51.85s	5
cpp11sort	5.15s	1.93s	2.74s	2.1s	2.46s	4.78s	19.16s	4
AQsort	8.36s	2.51s	19.36s	15.49s	14.14s	8.74s	68.6s	10
Boost	24.82s	1.84s	1.61s	6.46s	4.76s	14.06s	53.55s	6
Thrust	25.48s	18.4s	20.06s	17.5s	19.18s	23.52s	124.14s	12
IPS^{4}o	**1.35s**	0.46s	**1.14s**	**1.14s**	**1.32s**	**2.38s**	**7.79s**	1

Table A.8 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **Intel** cluster. The input size was 2e9 and the data type used was **short**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	4.95s	2.97s	4.46s	4.26s	3.61s	4.59s	24.84s	2
PPQSort OMP	6.1s	6.06s	3.99s	3.39s	5.79s	5.31s	30.64s	3
GCC BQS	10.03s	7.76s	23.85s	68.42s	20.97s	11.51s	142.54s	11
GCC QS	16.31s	8.06s	11.23s	23.26s	7.24s	20.2s	86.3s	7
oneTBB	23.65s	0.43s	9.37s	13.77s	11.19s	18.32s	76.73s	6
poolSTL	27.35s	8.83s	7.03s	15.12s	7.67s	23.77s	89.77s	8
MPQsort	17.04s	6.1s	29.31s	16.59s	19.39s	21.52s	109.95s	10
cpp11sort	7.71s	4.65s	3.81s	6.63s	6.43s	7.31s	36.54s	5
AQsort	17.42s	5.09s	17.77s	24.55s	19.08s	16.74s	100.65s	9
Boost	7.53s	2.45s	3.25s	7.04s	6.23s	7.45s	33.95s	4
Thrust	52.92s	46.37s	40.54s	43.79s	36.79s	44.76s	265.17s	12
IPS^{4}o	**3.01s**	**0.35s**	**2.56s**	**2.09s**	**2.74s**	**2.59s**	**13.34s**	1

Table A.9 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **Intel** cluster. The input size was 2e9 and the data type used was **int**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	9.63s	8.15s	6.42s	6.67s	7.27s	9.56s	47.7s	3
PPQSort OMP	10.8s	8.8s	6.51s	**6.59s**	7.94s	9.81s	50.45s	4
GCC BQS	17.25s	12.39s	30.46s	75.82s	18.75s	13.44s	168.11s	11
GCC QS	22.04s	29.14s	11.62s	34.66s	13.48s	22.26s	133.2s	8
oneTBB	29.93s	0.68s	16.54s	26.8s	15.16s	20.53s	109.64s	6
poolSTL	42.12s	11.9s	16.94s	24.38s	13.7s	37.77s	146.81s	10
MPQsort	18.79s	14.95s	29.09s	20.38s	14.65s	33.54s	131.4s	7
cpp11sort	10.71s	5.6s	4.62s	8.12s	5.64s	10.72s	45.41s	2
AQsort	22.76s	7.78s	28.41s	32.85s	28.9s	23.9s	144.6s	9
Boost	10.56s	4.95s	4.48s	19.66s	11.41s	11.09s	62.15s	5
Thrust	99.41s	85.42s	86.69s	87.04s	86.81s	88.57s	533.94s	12
IPS^{4}o	**4.34s**	**0.58s**	**3.73s**	14.17s	**3.84s**	**4.42s**	**31.08s**	1

Table A.10 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **Intel** cluster. The input size was 2e9 and the data type used was **double**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	3.05s	1.3s	3.7s	5.12s	3.24s	6.78s	23.19s	3
PPQSort OMP	4.08s	1.63s	3.84s	4.49s	2.16s	6.79s	22.99s	2
GCC BQS	8.6s	15.12s	33.45s	56.71s	22.36s	10.52s	146.76s	10
GCC QS	11.63s	13.09s	11.34s	14.78s	14.89s	10.51s	76.24s	7
oneTBB	21.0s	**0.52s**	18.1s	21.46s	26.59s	25.5s	113.17s	8
poolSTL	30.02s	17.17s	27.57s	17.88s	17.04s	26.33s	136.01s	9
MPQsort	21.06s	X	X	X	X	X	X	X
cpp11sort	5.65s	6.2s	5.88s	6.24s	6.11s	8.15s	38.23s	4
AQsort	8.88s	4.95s	5.54s	5.98s	6.05s	9.18s	40.58s	5
Boost	6.63s	7.09s	9.1s	15.01s	5.74s	8.31s	51.88s	6
Thrust	340.12s	285.0s	255.41s	285.59s	272.12s	354.2s	1792.44s	11
IPS^{4}o	**1.71s**	1.04s	**1.66s**	**1.76s**	**1.6s**	**1.35s**	**9.12s**	1

Table A.11 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **Intel** cluster. The input size was 2e7 and the data type used was **std::string**. Some algorithms faced errors during the sorting process, which resulted in missing results for certain patterns.

Algorithm	Cardinality 1	Cardinality 3	Cardinality 10	Cardinality 100	Cardinality 1e4	Cardinality 1e5	Total	Rank
PPQSort C++	0.63s	1.7s	3.21s	2.37s	4.36s	5.68s	17.95s	2
PPQSort OMP	1.56s	2.77s	4.42s	4.46s	5.29s	5.07s	23.57s	3
GCC BQS	73.02s	15.84s	9.67s	6.94s	9.51s	9.56s	124.54s	9
GCC QS	73.22s	24.61s	20.7s	10.88s	11.47s	15.18s	156.06s	10
oneTBB	**0.16s**	20.64s	19.46s	22.57s	34.83s	22.34s	120.0s	8
poolSTL	59.65s	73.62s	51.06s	20.26s	18.02s	17.83s	240.44s	12
MPQsort	52.32s	18.52s	10.59s	10.4s	10.43s	10.6s	112.86s	7
cpp11sort	13.92s	7.57s	6.36s	4.04s	4.7s	5.19s	41.78s	5
AQsort	2.36s	4.28s	5.07s	6.75s	10.14s	10.78s	39.38s	4
Boost	1.33s	4.68s	5.11s	6.7s	9.26s	16.69s	43.77s	6
Thrust	34.77s	42.9s	38.38s	40.7s	37.22s	38.13s	232.1s	11
IPS^{4}o	0.38s	**1.24s**	**0.81s**	**1.78s**	**2.12s**	**2.02s**	**8.35s**	1

Table A.12 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **Intel** cluster. The input size was 2e9 and the data type we used was **int**.

Algorithm	Size 1e4	Size 1e5	Size 1e6	Size 1e7	Size 5e7	Size 1e8	Total	Rank
PPQSort C++	**231.0ms**	126.0ms	42.9ms	19.5ms	2.91ms	0.27ms	422.58ms	2
PPQSort OMP	240.0ms	**119.0ms**	**40.2ms**	19.3ms	**2.53ms**	0.31ms	**421.34ms**	1
GCC BQS	574.0ms	333.0ms	132.0ms	86.4ms	85.8ms	17.9ms	1229.1ms	7
GCC QS	765.0ms	325.0ms	108.0ms	41.2ms	45.0ms	10.4ms	1294.6ms	8
oneTBB	1215.0ms	551.0ms	112.0ms	14.9ms	3.15ms	1.29ms	1897.34ms	10
poolSTL	1276.0ms	698.0ms	139.0ms	23.5ms	15.7ms	9.77ms	2161.97ms	11
MPQsort	598.0ms	314.0ms	78.4ms	20.1ms	5.39ms	0.51ms	1016.4ms	6
cpp11sort	415.0ms	247.0ms	92.3ms	16.3ms	8.18ms	5.32ms	784.1ms	5
AQsort	753.0ms	416.0ms	90.3ms	23.5ms	18.5ms	17.2ms	1318.5ms	9
Boost	328.0ms	214.0ms	53.6ms	20.7ms	3.12ms	0.3ms	619.72ms	4
Thrust	1878.0ms	1161.0ms	409.0ms	300.0ms	289.0ms	339.0ms	4376.0ms	12
IPS^{4}o	257.0ms	168.0ms	60.1ms	**9.87ms**	3.91ms	**0.24ms**	499.12ms	3

Table A.13 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **Intel** cluster. The data type we used **int**.

Algorithm	mawi_201512020330	uk-2005	dielFilterV3clx	Queen_4147	Total	Rank
PPQSort C++	2.75s	4.99s	0.54s	2.89s	11.17s	4
PPQSort OMP	2.41s	4.93s	0.4s	3.33s	11.07s	3
GCC BQS	3.45s	24.2s	0.45s	7.62s	35.72s	9
GCC QS	5.79s	9.66s	0.66s	3.95s	20.06s	6
oneTBB	4.39s	19.49s	3.32s	8.56s	35.76s	10
poolSTL	13.81s	14.96s	4.23s	59.08s	92.08s	11
MPQsort	4.82s	17.06s	3.47s	4.01s	29.36s	8
cpp11sort	2.91s	5.42s	0.29s	1.92s	10.54s	2
AQsort	8.99s	12.98s	0.51s	3.42s	25.9s	7
Boost	3.64s	6.94s	0.41s	1.95s	12.94s	5
Thrust	30.48s	63.63s	3.79s	25.23s	123.13s	12
IPS^{4}o	**1.57s**	**3.6s**	**0.21s**	**1.42s**	**6.8s**	1

Table A.14 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on sparse matrices, conducted on the **Intel** cluster.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	3.11s	1.72s	**2.1s**	1.69s	**1.72s**	2.79s	**13.13s**	1
PPQSort OMP	3.37s	1.88s	2.12s	**1.67s**	2.04s	2.98s	14.06s	3
GCC BQS	9.6s	16.54s	16.9s	40.98s	17.07s	9.37s	110.46s	10
GCC QS	11.62s	2.89s	3.87s	12.81s	8.92s	10.98s	51.09s	6
oneTBB	42.53s	**0.1s**	8.96s	7.27s	7.84s	40.71s	107.41s	9
poolSTL	34.64s	4.84s	8.41s	20.39s	7.72s	42.13s	118.13s	11
MPQsort	12.46s	7.1s	6.46s	6.26s	6.94s	10.94s	50.16s	5
cpp11sort	6.67s	2.41s	2.18s	2.12s	2.16s	6.6s	22.14s	4
AQsort	11.74s	2.31s	21.01s	19.63s	20.77s	11.37s	86.83s	8
Boost	32.81s	2.73s	3.2s	8.36s	6.35s	23.05s	76.5s	7
IPS^{4}o	**1.9s**	0.64s	2.83s	2.83s	2.81s	**2.32s**	13.33s	2

Table A.15 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **ARM** cluster. The input size was 2e9 and the data type used was **short**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	5.84s	1.84s	4.55s	**1.38s**	**2.96s**	5.58s	**22.15s**	1
PPQSort OMP	5.89s	2.02s	4.67s	1.77s	3.15s	5.72s	23.22s	2
GCC BQS	13.72s	4.18s	19.11s	49.89s	8.24s	13.78s	108.92s	8
GCC QS	18.33s	4.1s	14.62s	12.51s	14.01s	19.16s	82.73s	7
oneTBB	43.66s	**0.09s**	8.62s	13.84s	8.12s	43.9s	118.23s	10
poolSTL	34.63s	5.61s	7.23s	14.78s	7.81s	46.88s	116.94s	9
MPQsort	13.35s	5.74s	5.77s	4.67s	7.71s	12.87s	50.11s	6
cpp11sort	9.58s	2.47s	**2.66s**	5.47s	3.42s	9.9s	33.5s	4
AQsort	24.72s	3.66s	23.14s	21.83s	22.6s	25.31s	121.26s	11
Boost	8.2s	3.0s	4.26s	13.96s	6.97s	7.92s	44.31s	5
IPS^{4}o	**4.8s**	0.19s	5.97s	5.21s	5.59s	**4.91s**	26.67s	3

Table A.16 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **ARM** cluster. The input size was 2e9 and the data type used was **int**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	**6.12s**	1.95s	4.76s	**1.72s**	**3.16s**	**6.09s**	**23.8s**	1
PPQSort OMP	6.38s	2.45s	5.25s	1.98s	4.04s	6.16s	26.26s	2
GCC BQS	14.47s	4.38s	18.53s	44.96s	17.32s	14.95s	114.61s	8
GCC QS	18.99s	3.43s	14.09s	13.43s	9.31s	17.7s	76.95s	7
oneTBB	47.07s	**0.13s**	9.42s	15.5s	10.14s	48.66s	130.92s	10
poolSTL	41.14s	5.41s	8.98s	14.05s	8.34s	42.58s	120.5s	9
MPQsort	21.18s	6.65s	6.81s	6.07s	7.4s	15.21s	63.32s	6
cpp11sort	11.77s	2.33s	**2.85s**	7.25s	3.73s	11.21s	39.14s	4
AQsort	29.46s	3.97s	31.92s	29.19s	31.44s	28.65s	154.63s	11
Boost	9.64s	3.98s	4.54s	15.67s	7.29s	9.23s	50.35s	5
IPS^{4}o	7.87s	0.17s	7.5s	6.91s	7.23s	7.15s	36.83s	3

Table A.17 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **ARM** cluster. The input size was 2e9 and the data type used was **double**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	5.53s	3.09s	3.24s	3.39s	3.15s	10.31s	28.71s	2
PPQSort OMP	5.38s	2.92s	3.08s	3.48s	3.65s	11.32s	29.83s	3
GCC BQS	12.0s	22.15s	31.85s	77.86s	17.14s	6.83s	167.83s	9
GCC QS	14.2s	13.41s	13.91s	17.18s	13.57s	18.09s	90.36s	7
oneTBB	33.29s	**0.26s**	22.04s	21.69s	24.65s	42.06s	143.99s	8
poolSTL	32.18s	22.9s	42.4s	23.45s	35.88s	66.69s	223.5s	10
MPQsort	25.38s	X	X	X	X	X	X	X
cpp11sort	5.87s	5.7s	5.45s	5.38s	5.86s	9.6s	37.86s	4
AQsort	5.89s	5.38s	6.08s	5.27s	5.0s	40.49s	68.11s	5
Boost	7.5s	11.46s	11.21s	24.68s	7.87s	6.54s	69.26s	6
IPS^{4}o	**0.92s**	0.47s	**1.68s**	**1.51s**	**1.88s**	**1.5s**	**7.96s**	1

Table A.18 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **ARM** cluster. The input size was 2e9 and the data type used was **string**.

Algorithm	Cardinality 1	Cardinality 3	Cardinality 10	Cardinality 100	Cardinality 1e4	Cardinality 1e5	Total	Rank
PPQSort C++	0.41s	**2.1s**	3.08s	4.19s	3.24s	4.07s	**17.09s**	1
PPQSort OMP	1.04s	2.36s	3.44s	3.75s	4.11s	4.1s	18.8s	2
GCC BQS	39.98s	12.97s	7.15s	9.48s	9.26s	9.95s	88.79s	8
GCC QS	103.69s	38.04s	15.7s	9.69s	13.0s	13.65s	193.77s	10
oneTBB	**0.08s**	30.73s	36.59s	40.16s	44.7s	40.43s	192.69s	9
poolSTL	98.52s	142.4s	154.41s	36.57s	38.96s	36.07s	506.93s	11
MPQsort	26.79s	14.78s	11.18s	11.06s	11.03s	11.52s	86.36s	7
cpp11sort	10.88s	5.26s	6.4s	4.64s	6.02s	6.68s	39.88s	5
AQsort	2.34s	3.48s	3.76s	5.86s	10.02s	10.88s	36.34s	4
Boost	2.84s	3.44s	3.35s	4.08s	13.5s	20.72s	47.93s	6
IPS^{4}o	12.84s	5.04s	**2.93s**	**1.81s**	**2.69s**	**3.11s**	28.42s	3

Table A.19 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **ARM** cluster. The input size was 2e9 and the data type we used was **int**.

Algorithm	Size 1e4	Size 1e5	Size 1e6	Size 1e7	Size 5e7	Size 1e8	Total	Rank
PPQSort C++	50.1ms	**0.39ms**	**2.53ms**	22.3ms	158.0ms	**282.0ms**	**515.32ms**	1
PPQSort OMP	45.3ms	0.4ms	2.7ms	**18.4ms**	170.0ms	293.0ms	529.8ms	2
GCC BQS	97.7ms	20.6ms	41.9ms	50.0ms	368.0ms	669.0ms	1247.2ms	7
GCC QS	103.0ms	4.49ms	17.0ms	26.4ms	449.0ms	881.0ms	1480.89ms	8
oneTBB	202.0ms	0.46ms	3.14ms	23.9ms	1111.0ms	2017.0ms	3357.5ms	11
poolSTL	124.0ms	3.72ms	6.96ms	19.5ms	828.0ms	1759.0ms	2741.18ms	10
MPQsort	93.7ms	0.97ms	11.7ms	21.3ms	368.0ms	675.0ms	1170.67ms	6
cpp11sort	101.0ms	9.93ms	15.3ms	28.6ms	337.0ms	608.0ms	1099.83ms	5
AQsort	130.0ms	17.5ms	16.1ms	28.1ms	576.0ms	1162.0ms	1929.7ms	9
Boost	76.4ms	0.6ms	6.7ms	30.6ms	227.0ms	395.0ms	736.3ms	4
IPS^{4}o	**36.4ms**	0.58ms	6.1ms	28.0ms	**132.0ms**	333.0ms	536.08ms	3

Table A.20 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **ARM** cluster. The data type we used **int**.

Algorithm	mawi_20151202030	uk-2005	dielFilterV3clx	Queen_4147	Total	Rank
PPQSort C++	1.86s	3.35s	0.19s	**1.37s**	**6.77s**	1
PPQSort OMP	1.87s	**3.33s**	**0.18s**	1.47s	6.85s	2
GCC BQS	2.44s	10.61s	0.27s	2.1s	15.42s	8
GCC QS	3.35s	7.63s	0.25s	1.64s	12.87s	6
oneTBB	4.29s	6.8s	0.31s	2.48s	13.88s	7
poolSTL	16.52s	12.53s	3.31s	23.84s	56.2s	11
MPQsort	3.7s	5.67s	0.41s	2.1s	11.88s	5
cpp11sort	**1.75s**	3.89s	0.28s	1.63s	7.55s	3
AQsort	17.81s	26.12s	0.51s	4.4s	48.84s	10
Boost	2.87s	4.08s	0.26s	1.75s	8.96s	4
IPS^{4}o	3.52s	8.88s	0.52s	2.53s	15.45s	9

Table A.21 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on sparse matrices, conducted on the **ARM** cluster.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	1.55s	0.63s	0.75s	**0.62s**	**0.68s**	1.39s	5.62s	2
PPQSort OMP	1.62s	0.98s	1.1s	0.99s	1.07s	1.78s	7.54s	3
GCC BQS	3.92s	6.6s	8.07s	17.53s	6.77s	3.85s	46.74s	11
GCC QS	4.38s	0.98s	1.34s	4.05s	1.41s	3.81s	15.97s	6
oneTBB	15.88s	**0.04s**	3.21s	2.74s	2.99s	11.46s	36.32s	10
poolSTL	13.59s	1.38s	2.39s	3.93s	2.3s	10.25s	33.84s	8
MPQsort	3.86s	1.87s	2.31s	1.84s	2.06s	3.9s	15.84s	5
cpp11sort	1.98s	0.81s	0.94s	0.83s	0.92s	2.07s	7.55s	4
AQsort	4.89s	1.12s	8.62s	8.16s	8.05s	4.89s	35.73s	9
Boost	7.26s	0.58s	0.8s	3.42s	4.08s	10.21s	26.35s	7
IPS^{4}o	**0.59s**	0.04s	**0.68s**	0.63s	0.68s	**0.7s**	**3.32s**	1

Table A.22 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **RCI** cluster. The input size was 2e9 and the data type used was **short**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	2.22s	1.07s	1.01s	**1.13s**	1.35s	2.22s	9.0s	2
PPQSort OMP	2.39s	1.08s	1.11s	1.42s	1.43s	3.08s	10.51s	3
GCC BQS	5.43s	1.2s	9.37s	14.74s	1.64s	5.27s	37.65s	10
GCC QS	6.38s	1.44s	2.07s	5.99s	2.84s	5.77s	24.49s	7
oneTBB	15.34s	**0.05s**	2.47s	3.88s	2.39s	10.9s	35.03s	9
poolSTL	11.54s	1.69s	2.6s	5.11s	2.17s	10.94s	34.05s	8
MPQsort	5.2s	1.86s	2.42s	1.93s	2.18s	5.6s	19.19s	6
cpp11sort	3.54s	1.07s	1.14s	1.94s	1.51s	3.46s	12.66s	4
AQsort	7.93s	1.87s	10.27s	10.55s	8.95s	8.17s	47.74s	11
Boost	3.47s	0.63s	**0.9s**	3.27s	3.66s	3.18s	15.11s	5
IPS^{4}o	**1.29s**	0.15s	1.41s	1.13s	**1.29s**	**1.5s**	**6.77s**	1

Table A.23 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **RCI** cluster. The input size was 2e9 and the data type used was **int**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	3.53s	1.95s	2.08s	2.11s	1.97s	3.73s	15.37s	2
PPQSort OMP	3.67s	1.95s	2.28s	2.36s	2.35s	3.59s	16.2s	3
GCC BQS	6.04s	1.99s	12.3s	20.69s	2.33s	6.36s	49.71s	9
GCC QS	8.96s	2.45s	2.73s	8.94s	3.24s	7.66s	33.98s	7
oneTBB	20.47s	**0.1s**	4.0s	4.75s	3.44s	14.47s	47.23s	8
poolSTL	21.84s	2.5s	4.42s	8.06s	3.32s	13.51s	53.65s	10
MPQsort	8.48s	2.74s	3.27s	2.89s	3.1s	8.05s	28.53s	6
cpp11sort	5.48s	1.65s	1.81s	3.13s	2.2s	5.43s	19.7s	5
AQsort	10.79s	2.14s	12.36s	15.13s	12.55s	10.67s	63.64s	11
Boost	4.58s	1.19s	**1.36s**	3.82s	3.23s	4.62s	18.8s	4
IPS^{4}o	**1.94s**	0.11s	1.76s	**1.39s**	**1.59s**	**2.02s**	**8.81s**	1

Table A.24 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **RCI** cluster. The input size was 2e9 and the data type used was **double**.

Algorithm	Random	Ascending	Descending	Rotated	OrganPipe	Heap	Total	Rank
PPQSort C++	1.85s	1.14s	1.16s	1.08s	0.95s	2.93s	9.11s	3
PPQSort OMP	1.81s	1.29s	1.27s	1.09s	1.14s	2.44s	9.04s	2
GCC BQS	2.42s	6.13s	14.66s	34.28s	8.51s	3.61s	69.61s	10
GCC QS	2.85s	2.93s	2.86s	4.7s	2.64s	3.55s	19.53s	6
oneTBB	5.84s	0.18s	6.83s	6.31s	6.65s	12.21s	38.02s	8
poolSTL	10.53s	4.76s	4.87s	4.64s	5.33s	20.78s	50.91s	9
MPQsort	5.91s	105.56s	4.39s	X	4.56s	X	X	X
cpp11sort	2.37s	2.29s	1.91s	2.0s	2.15s	3.11s	13.83s	5
AQsort	1.44s	1.46s	1.5s	1.93s	2.09s	3.44s	11.86s	4
Boost	1.79s	2.93s	2.58s	8.26s	3.21s	1.76s	20.53s	7
IPS^{4}o	**0.27s**	**0.17s**	**0.34s**	**0.37s**	**0.34s**	**0.3s**	**1.79s**	1

Table A.25 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on different data patterns, conducted on the **RCI** cluster. The input size was 2e9 and the data type used was **string**.

Algorithm	Cardinality 1	Cardinality 3	Cardinality 10	Cardinality 100	Cardinality 1e4	Cardinality 1e5	Total	Rank
PPQSort C++	0.19s	0.73s	2.37s	1.5s	1.94s	2.29s	9.02s	2
PPQSort OMP	0.41s	0.97s	2.23s	1.69s	1.91s	2.14s	9.35s	3
GCC BQS	13.6s	5.62s	3.05s	3.22s	3.85s	3.82s	33.16s	8
GCC QS	26.83s	10.07s	7.7s	3.29s	4.63s	4.23s	56.75s	9
oneTBB	**0.06s**	10.37s	13.87s	14.25s	16.06s	15.78s	70.39s	10
poolSTL	24.29s	23.51s	12.64s	10.65s	11.29s	9.92s	92.3s	11
MPQsort	8.93s	5.36s	4.06s	3.83s	3.81s	3.99s	29.98s	6
cpp11sort	7.0s	3.2s	4.82s	1.71s	2.0s	2.29s	21.02s	5
AQsort	0.72s	1.72s	1.94s	3.05s	4.42s	4.84s	16.69s	4
Boost	0.64s	1.56s	1.51s	1.48s	8.33s	16.59s	30.11s	7
IPS^{4}o	0.06s	**0.35s**	**0.36s**	**0.42s**	**0.68s**	**0.76s**	**2.63s**	1

Table A.26 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **RCI** cluster. The input size was 2e9 and the data type we used was **int**.

Algorithm	Size 1e4	Size 1e5	Size 1e6	Size 1e7	Size 5e7	Size 1e8	Total	Rank
PPQSort C++	0.26ms	2.15ms	9.1ms	28.8ms	82.9ms	168.0ms	291.21ms	3
PPQSort OMP	0.23ms	**2.02ms**	9.9ms	30.7ms	87.1ms	155.0ms	284.95ms	2
GCC BQS	9.97ms	11.5ms	39.6ms	37.4ms	132.0ms	257.0ms	487.47ms	5
GCC QS	12.0ms	15.5ms	28.4ms	45.5ms	147.0ms	263.0ms	511.4ms	7
oneTBB	0.52ms	2.22ms	10.1ms	80.2ms	410.0ms	766.0ms	1269.04ms	11
poolSTL	3.67ms	7.05ms	10.3ms	73.9ms	320.0ms	586.0ms	1000.92ms	10
MPQsort	0.39ms	5.25ms	39.1ms	50.9ms	148.0ms	278.0ms	521.64ms	8
cpp11sort	9.9ms	11.9ms	19.9ms	50.2ms	162.0ms	255.0ms	508.9ms	6
AQsort	17.6ms	18.6ms	22.6ms	49.8ms	201.0ms	386.0ms	695.6ms	9
Boost	0.23ms	3.02ms	14.9ms	38.7ms	92.0ms	170.0ms	318.85ms	4
IPS^{4}o	**0.2ms**	2.1ms	**3.97ms**	**11.2ms**	**35.4ms**	**69.4ms**	**122.27ms**	1

Table A.27 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on random data patterns with different cardinalities, conducted on the **RCI** cluster. The data type we used **int**.

Algorithm	mawi_201512020330	uk-2005	dielFilterV3clx	Queen_4147	Total	Rank
PPQSort C++	0.8s	1.64s	0.16s	0.81s	3.41s	3
PPQSort OMP	0.9s	**1.53s**	0.17s	0.8s	3.4s	2
GCC BQS	1.4s	2.74s	0.37s	2.9s	7.41s	8
GCC QS	1.36s	3.61s	0.22s	1.28s	6.47s	7
oneTBB	1.64s	2.56s	0.25s	1.63s	6.08s	6
poolSTL	3.79s	3.33s	1.48s	9.45s	18.05s	11
MPQsort	2.28s	3.84s	0.25s	1.73s	8.1s	9
cpp11sort	0.91s	1.8s	0.17s	0.83s	3.71s	4
AQsort	3.09s	5.45s	0.21s	1.27s	10.02s	10
Boost	1.14s	2.28s	0.13s	0.64s	4.19s	5
IPS^{4}o	**0.68s**	1.58s	**0.08s**	**0.6s**	**2.94s**	1

Table A.28 A thorough comparison of all assessed implementations (not limited to quicksort algorithms) on sparse matrices, conducted on the **RCI** cluster.

Declaration of Generative AI and AI-assisted technologies

During the preparation of this work, the author used Grammarly AI tool to enhance the academic tone and precision of certain sections of the text. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

Bibliography

1. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů* [online]. 2. vyd. Praha: CZ.NIC, 2022 [visited on 2024-02-02]. ISBN 978-80-88168-66-9. Available from: `https://pruvodce.ucw.cz/static/pruvodce.pdf`. Errata up to 2023-12-08.
2. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms* [online]. 4th ed. Cambridge, Massachusetts: The MIT Press, 2022 [visited on 2024-02-02]. ISBN 9780262046305. Available from: `https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf`.
3. HOARE, C. A. R. Algorithm 64: Quicksort. *Commun. ACM*. 1961, vol. 4, no. 7, p. 321. ISSN 0001-0782. Available from DOI: `10.1145/366622.366644`.
4. VORONECKÝ, Ondřej. *Efficient parallel multi-way Quicksort algorithm*. 2023. masterthesis. Czech Technical University in Prague.
5. WILD, Sebastian. *Dual-Pivot Quicksort and Beyond: Analysis of Multiway Partitioning and Its Practical Potential*. 2016. PhD thesis. Technischen Universität Kaiserslautern.
6. BENTLEY, Jon. *Programming Pearls*. Addison-Wesley Professional, 1999.
7. TUKEY, John W. The Ninther, a Technique for Low-Effort Robust (Resistant) Location in Large Samples. In: DAVID, H.A. (ed.). *Contributions to Survey Sampling and Applied Statistics*. Academic Press, 1978, pp. 251–257. ISBN 978-0-12-204750-3. Available from DOI: `https://doi.org/10.1016/B978-0-12-204750-3.50024-1`.
8. KALIGOSI, Kanela; SANDERS, Peter. How Branch Mispredictions Affect Quicksort. In: AZAR, Yossi; ERLEBACH, Thomas (eds.). *Algorithms – ESA 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 780–791. ISBN 978-3-540-38876-0.
9. FORSYTHE, G. E. Algorithms. *Commun. ACM*. 1964, vol. 7, no. 6, pp. 347–349. ISSN 0001-0782. Available from DOI: `10.1145/512274.512284`.
10. MCILROY, M. D. A killer adversary for quicksort. *Softw. Pract. Exper.* 1999, vol. 29, no. 4, pp. 341–344. ISSN 0038-0644.

11. PETERS, Orson. *20837 – libc++ std::sort has O(n^2) worst case, standard mandates O(N log(N))* [online]. Bugzilla, 2014 [visited on 2024-02-23]. Available from: https://bugs.llvm.org/show_bug.cgi?id=20837.
12. MUSSER, DAVID R. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience*. 1997, vol. 27, no. 8, pp. 983–993. Available from DOI: https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-\#.
13. PROJECT, GNU. *GCC* [online]. GitHub, 2024 [visited on 2024-04-15]. Available from: https://github.com/gcc-mirror/gcc/blob/8ca585e56c1d6837f96ddd88c13ed1e815c74f93/libstdc%2B%2B-v3/include/bits/stl_algo.h#L1899.
14. GROUP, LLVM Developer. *llvm-project* [online]. GitHub, 2024 [visited on 2024-04-15]. Available from: https://github.com/llvm/llvm-project/blob/f958ad3b89c38be84dcf263ef9f9508a5cd3a6e3/libcxx/include/__algorithm/sort.h#L750.
15. MICROSOFT. *STL* [online]. GitHub, 2024 [visited on 2024-04-15]. Available from: https://github.com/microsoft/STL/blob/1bc5ca60fcb41c2ed87a721d6ca2f77844cf6dc6/stl/inc/algorithm#L8001.
16. SEDGEWICK, Robert. Implementing Quicksort programs. *Commun. ACM*. 1978, vol. 21, no. 10, pp. 847–857. ISSN 0001-0782. Available from DOI: 10.1145/359619.359631.
17. EDELKAMP, Stefan; WEISS, Armin. BlockQuicksort: Avoiding Branch Mispredictions in Quicksort. *ACM J. Exp. Algorithmics*. 2019, vol. 24. ISSN 1084-6654. Available from DOI: 10.1145/3274660.
18. ABHYANKAR, D.; INGLE, Maya. Engineering of a Quicksort Partitioning Algorithm. *Journal of Global Research in Computer Sciences*. 2011, vol. 2, pp. 17–23. Available also from: https://api.semanticscholar.org/CorpusID:53919163.
19. PETERS, Orson R. L. Pattern-defeating Quicksort. *CoRR*. 2021, vol. abs/2106.05123. Available from arXiv: 2106.05123.
20. O'CONNOR, Daniel G.; NELSON, Raymond J. *Sorting system with nu-line sorting switch*. Inventor: Daniel G. O'CONNOR; Raymond J. NELSON. Publ.: 1962-10. Patent issued April 10, 1962. US Patent US3029413A. [Visited on 2024-02-12]. Available from: https://patents.google.com/patent/US3029413A/en.
21. KNUTH, Donald E. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN 0201896850.
22. HARDER, Jannis. An Answer to the Bose-Nelson Sorting Problem for 11 and 12 Channels. *CoRR*. 2020, vol. abs/2012.04400. Available from arXiv: 2012.04400.
23. MANKOWITZ, Daniel J. et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*. 2023, vol. 618, no. 7964, pp. 257–263. ISSN 1476-4687. Available from DOI: 10.1038/s41586-023-06004-9.
24. GELMI, Marco. *D118029 Introduce branchless sorting functions for sort3, sort4 and sort5.* [online]. LLVM Phabricator archive, 2022 [visited on 2024-02-14]. Available from: https://reviews.llvm.org/D118029.

25. AG, Mimicry. *Faster Sorting Beyond DeepMind's AlphaDev* [online]. 2023-09. [visited on 2024-02-15]. Tech. rep. Available from: https://www.mimicry.ai/faster-sorting-beyond-deepminds-alphadev.

26. NERI, Cassio. *Shorter and faster than Sort3AlphaDev*. 2023. Available from eprint: arXiv:2307.14503.

27. AKRA, Mohamad; BAZZI, Louay. On the Solution of Linear Recurrence Equations. *Computational Optimization and Applications*. 1998, vol. 10, no. 2, pp. 195–210. ISSN 1573-2894. Available from DOI: 10.1023/A:1018373005182.

28. FILMUS, Yuval. *Solving recurrence relation with two recursive calls* [Computer Science Stack Exchange]. [N.d.]. [visited on 2024-02-24]. Available from eprint: https://cs.stackexchange.com/q/31930.

29. LANGR, Daniel; TVRDÍK, Pavel; ŠIMEČEK, Ivan. AQsort: Scalable Multi-Array In-Place Sorting with OpenMP. *Scalable Comput. Pract. Exp.* 2016, vol. 17, pp. 369–391. Available also from: https://api.semanticscholar.org/CorpusID:17262177.

30. BLELLOCH, Guy E. Vector Models for Data-Parallel Computing. In: 1990. Available also from: https://api.semanticscholar.org/CorpusID:1610858.

31. CEDERMAN, Daniel; TSIGAS, Philippas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *ACM J. Exp. Algorithmics*. 2010, vol. 14. ISSN 1084-6654. Available from DOI: 10.1145/1498698.1564500.

32. TSIGAS, Philippas; ZHANG, Yi. A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN Enterprise 10000. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.* 2003, pp. 372–381. Available also from: https://api.semanticscholar.org/CorpusID:39733.

33. FRANCIS, R.S; PANNAN, L.J.H. A parallel partition for enhanced parallel QuickSort. *Parallel Computing*. 1992, vol. 18, no. 5, pp. 543–550. ISSN 0167-8191. Available from DOI: https://doi.org/10.1016/0167-8191(92)90089-P.

34. SINGLER, Johannes; SANDERS, Peter; PUTZE, Felix. MCSTL: The Multi-core Standard Template Library. In: KERMARREC, Anne-Marie; BOUGÉ, Luc; PRIOL, Thierry (eds.). *Euro-Par 2007 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 682–694. ISBN 978-3-540-74466-5.

35. FRIAS, Leonor; PETIT, Jordi. Parallel Partition Revisited. In: MCGEOCH, Catherine C. (ed.). *Experimental Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 142–153. ISBN 978-3-540-68552-4.

36. SCHOVÁNKOVÁ, Klára. *Parallel Sorting in C++11*. 2019. masterthesis. Czech Technical University in Prague.

37. DAGUM, Leonardo; MENON, Ramesh. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*. 1998, vol. 5, no. 1, pp. 46–55.

38. PHEATT, Chuck. Intel® threading building blocks. *J. Comput. Sci. Coll.* 2008, vol. 23, no. 4, p. 298. ISSN 1937-4771.

39. GURTOVOY, Aleksey; ABRAHAMS, David. The Boost C++ metaprogramming library. 2002.

40. LANGR, Daniel; SCHOVÁNKOVÁ, Klára. CPP11sort: A parallel quicksort based on C++ threading. *Concurrency and Computation: Practice and Experience*. 2022, vol. 34, no. 4, e6606. Available from DOI: https://doi.org/10.1002/cpe.6606.

41. LUGOWSKI, Adam. *poolSTL* [online]. GitHub, 2024 [visited on 2024-02-22]. Available from: https://github.com/alugowski/poolSTL/releases/tag/v0.3.5.

42. SINGLER, Johannes; KONSIK, Benjamin. The GNU libstdc++ parallel mode: software engineering considerations. In: *Proceedings of the 1st International Workshop on Multicore Software Engineering*. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 15–22. IWMSE '08. ISBN 9781605580319. Available from DOI: 10.1145/1370082.1370089.

43. ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. 2017. Available also from: https://www.iso.org/standard/68564.html. Draft: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

44. CPPREFERENCE. *Compiler support for C++17* [online]. 2023. [visited on 2024-02-20]. Available from: https://en.cppreference.com/w/cpp/compiler_support/17#C.2B.2B17_library_features. Snapshot: https://web.archive.org/web/20240220132425/https://en.cppreference.com/w/cpp/compiler_support/17.

45. KUKANOV, Alexey. *Proposal to contribute Intel's implementation of C++17 parallel algorithms* [online]. 2017. [visited on 2024-02-21]. Available from: https://gcc.gnu.org/legacy-ml/libstdc++/2017-11/msg00112.html.

46. WAKELY, Jonathan. *Integrate C++17 parallel algorithms · gcc-mirror/gcc@061f457* [online]. 2019. [visited on 2024-02-21]. Available from: https://github.com/gcc-mirror/gcc/commit/061f457868281238db43ef784aa12269cc866adb.

47. PROJECT, GNU. *GCC* [online]. GitHub, 2024 [visited on 2024-04-16]. Available from: https://github.com/gcc-mirror/gcc/blob/61ab046a3277c256867f596e73ce5b5ee9041a9d/libstdc%2B%2B-v3/include/pstl/parallel_backend_tbb.h#L1161.

48. ROBINSON, Arch D. *A Parallel Stable Sort Using C++11 for TBB, Cilk Plus, and OpenMP* [online]. Intel, 2014 [visited on 2024-02-21]. Available from: https://web.archive.org/web/20161203085708/https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp.

49. ALEXEY, Kukanov et al. *parallel_sort — oneAPI Specification 1.3-rev-1 documentation* [online]. 2022. [visited on 2024-02-21]. Available from: https://spec.oneapi.io/versions/latest/elements/oneTBB/source/algorithms/functions/parallel_sort_func.html#parallel-sort. publisher: Intel Corporation.

50. INTEL. *oneTBB* [online]. GitHub, 2024 [visited on 2024-04-16]. Available from: https://github.com/oneapi-src/oneTBB/blob/f0d4aba544ab15d7d3d5937aad314b538dd90672/include/oneapi/tbb/parallel_sort.h#L244.

51. CCCL DEVELOPMENT TEAM. *CCCL: CUDA C++ Core Libraries* [online]. GitHub, 2023 [visited on 2024-02-22]. Available from: https://github.com/NVIDIA/cccl/releases/tag/v2.3.1.

52. WHITNEY, Tyler et al. *Parallel Patterns Library (PPL)* [online]. 2021. [visited on 2024-02-22]. Available from: https://learn.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170.

53. TAPIA, Francisco Jose. BLOCK INDIRECT A new parallel sorting algorithm. *Boost docs*. 2016. Available also from: https://www.boost.org/doc/libs/1_84_0/libs/sort/doc/papers/block_indirect_sort_en.pdf.

54. VORONECKÝ, Ondřej. *MPQsort* [online]. GitHub, 2024 [visited on 2024-02-22]. Available from: https://github.com/voronond/MPQsort.

55. AXTMANN, Michael; WITT, Sascha; FERIZOVIC, Daniel; SANDERS, Peter. *Engineering In-place (Shared-memory) Sorting Algorithms*. 2021. Available from arXiv: 2009.13569 [cs.DC].

56. HOVEN, Igor van den. *crumsort* [online]. GitHub, 2024 [visited on 2024-02-24]. Available from: https://github.com/scandum/crumsort.

57. CPPREFERENCE. *std::counting_semaphore, std::binary_semaphore – cppreference.com* [online]. 2023. [visited on 2024-03-09]. Available from: https://en.cppreference.com/w/cpp/thread/counting_semaphore.

58. PARENT, Sean. Better Code: Concurrency. In: *NDC Conferences* [online]. 2017 [visited on 2024-03-07]. Available from: https://youtu.be/zULU6Hhp42w?si=0NXQ3l0cRN-H72Ek&t=1695.

59. CPPREFERENCE. *std::atomic¡T¿::wait – cppreference.com* [online]. 2020. [visited on 2024-03-10]. Available from: https://en.cppreference.com/w/cpp/atomic/atomic/wait.

60. MELCHIOR, Lars. *ModernCppStarter* [online]. GitHub, 2024 [visited on 2024-03-11]. Available from: https://github.com/TheLartians/ModernCppStarter.

61. GOOGLE. *GoogleTest* [online]. GitHub, 2024 [visited on 2024-03-12]. Available from: https://github.com/google/googletest.

62. HÉVR, Gabriel. *PPQSort* [online]. GitHub, 2024 [visited on 2024-04-15]. Available from: https://github.com/GabTux/PPQSort.

63. GOOGLE. *Benchmark* [online]. GitHub, 2024 [visited on 2024-03-12]. Available from: https://github.com/google/benchmark.

64. HEESCH, Dimitri van. *Doxygen* [online]. GitHub, 2024 [visited on 2024-03-12]. Available from: https://github.com/doxygen/doxygen.

65. DAVIS, Timothy A.; HU, Yifan. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*. 2011, vol. 38, no. 1, Article 1. Available from DOI: 10.1145/2049662.2049663.

66. BARRETT, Richard; BERRY, Michael; CHAN, Tony; DEMMEL, James; DONATO, June; DONGARRA, Jack; EIJKHOUT, Victor; POZO, Roldan; ROMINE, Chris; VAN DER VORST, Henk. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. *Mathematics of Computation*. 1996, vol. 64. Available from DOI: 10.2307/2153507.

67. SAAD, Yousef. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003. Available from DOI: 10.1137/1.9780898718003.

68. LUGOWSKI, Adam. *fast_matrix_market: Fast and Full-Featured Matrix Market I/O Library*. 2023-01. Available from DOI: 10.5281/zenodo.10223767.

Contents of the Attached Archive

- `src/` ..... source codes
  - `PPQSort-1.0.3.zip` ..... source code of PPQSort suite v1.0.3 [62]
- `text/` ..... thesis text sources
  - `latex-src.zip` ..... LaTeX source codes
  - `hevrgabr-thesis.pdf` ..... this thesis in PDF format