Robust Scalable Sorting

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Michael Axtmann

Tag der mündlichen Prüfung: 17. Mai 2021

1. Referent: Prof. Dr. Peter Sanders
Karlsruher Institut für Technologie
Deutschland

2. Referent: Prof. Guy E. Blelloch
Carnegie Mellon University
Vereinigte Staaten von Amerika

To my parents, my sister, and my wife.

Abstract

Sorting is one of the most important basic algorithmic problems. Thus, it is not a surprise that sorting algorithms are needed in a very large number of applications. These applications are executed on a wide range of different machines—from smartphones with energy-efficient multi-core processors to supercomputers with thousands of machines interconnected by a high-performance network. Since single-core performance has stagnated, parallel applications have become an indispensable part of our everyday lives. Efficient and scalable algorithms are key to take advantage of this immense availability of (parallel) computing power. In this thesis, we study sequential and parallel sorting algorithms aiming at robust performance for a diverse set of input sizes, input distributions, data types, and machines.

In the first part of this thesis, we study sequential sorting as well as parallel sorting on shared-memory machines. We propose *In-place Parallel Super Scalar Samplesort* (IPS^{4}o), a new comparison-based algorithm that is provably in-place, i.e., the amount of additional memory does not depend on the size of the input. An essential result is that the in-place property improves the performance of IPS^{4}o compared to similar non-in-place algorithms. Until now, the in-place feature has usually been associated with a loss of speed for most algorithms. IPS^{4}o is also provably cache-efficient and performs $\mathcal{O}(n/t \log n)$ work per thread when executed with t threads. Additionally, IPS^{4}o incorporates a branchless decision tree to minimize the number of branch mispredictions, takes advantage of memory locality, and handles many elements with equal keys—so-called *duplicated keys*—by separating them into "equality buckets". For the special case of sorting integer keys, we use the algorithmic framework of IPS^{4}o to implement *In-place Parallel Super Scalar Radix Sort* (IPS^{2}Ra).

We validate the performance of our algorithms in an extensive experimental study involving 21 state-of-the-art sorting algorithms, six data types, ten input distributions, four machines, four memory allocation strategies, and input sizes varying over seven orders of magnitude. On the one hand, the study shows that our algorithms have consistently good performance. On the other hand, it reveals that many competitors have large performance issues: With IPS^{4}o, we obtain a robust comparison-based sorting algorithm that outperforms other parallel in-place comparison-based sorting algorithms by almost a factor of three. In the large majority of the cases, IPS^{4}o is the fastest comparison-based algorithm. At this point, it is irrelevant whether we compare IPS^{4}o to sequential or parallel, in-place or non-in-place algorithms. IPS^{4}o even outperforms competing implementations of integer sorting algorithms in many cases. The remaining cases mainly include uniformly distributed inputs and inputs with keys containing only few bits. These inputs are usually "easy" for integer sorting algorithms. Our integer sorter IPS^{2}Ra outperforms other integer sorting algorithms for these inputs in the large majority of

the cases. Exceptions are some very small inputs for which most of the algorithms are very inefficient. However, algorithms dedicated for these input sizes are usually much slower for the remaining range of input sizes.

In the second part of this thesis, we study sorting algorithms for distributed systems and their robust scalability with regard to the number of processors, the input size, duplicated keys, and the distribution of input keys to processors. Our main contributions are four robust scalable sorting algorithms that allow us to cover the entire range of input sizes. Three of the four are new fast algorithms that implement low overhead mechanisms to make them scale robustly regardless of "difficult" inputs, e.g., inputs where the location of the input elements is correlated to the key values—so-called *skewed element distributions*—or inputs with duplicated keys. The fourth algorithm is simple and may be considered as folklore.

Previous algorithms for inputs of *medium* and *large* size have an unacceptably large communication volume or an unacceptably large number of message exchanges. For these input sizes, we describe a robust multi-level generalization of samplesort that represents a feasible compromise between a moderate communication volume and a moderate number of message exchanges. We overcome these previously incompatible goals with scalable approximate splitter selection and a new data routing algorithm. As an alternative, we present a generalization of mergesort with the advantage of perfect load balance. For *small* inputs, we design a variant of quicksort that overcomes the problem of skewed element distributions and duplicated keys with fast high-quality pivot selection, element randomization, and low overhead duplicate handling. Previous practical approaches with polylogarithmic latency either have at least a logarithmic factor more communication or only consider uniform input. For *very small* inputs, we propose a practical and fast, yet work-inefficient algorithm with logarithmic latency. For these inputs, previous efficient approaches are theoretical algorithms mostly with prohibitively large constant factors. For the *smallest* inputs, we recommend an algorithm that sorts the data while the input is routed to a single processor.

An important contribution of this thesis to the practical side of algorithm engineering is a communication library that we call *RangeBasedComm* (RBC). RBC allows an efficient implementation of recursive algorithms with sublinear running time by providing scalable and efficient communication primitives on processor subsets. The RBC library significantly speeds up the algorithms, e.g., one competitor even by more than two orders of magnitude.

We present an extensive experimental study involving two supercomputers with up to 262 144 cores, 11 algorithms, 10 input distributions, and input sizes varying over nine orders of magnitude. For all but the largest input sizes, we are the only ones performing benchmarks on these large machine instances. The study also shows that our algorithms have a robust performance and outperform competing implementations significantly. Whereas our algorithms provide consistent performance on all inputs, our competitors' performance breaks down on "difficult" inputs or they literally break.

Deutsche Zusammenfassung

Sortieren ist eines der wichtigsten algorithmischen Grundlagenprobleme. Es ist daher nicht verwunderlich, dass Sortieralgorithmen in einer Vielzahl von Anwendungen benötigt werden. Diese Anwendungen werden auf den unterschiedlichsten Geräten ausgeführt – angefangen bei Smartphones mit leistungseffizienten Multi-Core-Prozessoren bis hin zu Supercomputern mit Tausenden von Maschinen, die über ein Hochleistungsnetzwerk miteinander verbunden sind. Spätestens seitdem die Single-Core-Leistung nicht mehr signifikant steigt, sind parallele Anwendungen in unserem Alltag nicht mehr wegzudenken. Daher sind effiziente und skalierbare Algorithmen essentiell, um diese immense Verfügbarkeit von (paralleler) Rechenleistung auszunutzen. Diese Arbeit befasst sich damit, wie sequentielle und parallele Sortieralgorithmen auf möglichst robuste Art maximale Leistung erzielen können. Dabei betrachten wir einen großen Parameterbereich von Eingabegrößen, Eingabeverteilungen, Maschinen sowie Datentypen.

Im ersten Teil dieser Arbeit untersuchen wir sowohl sequentielles Sortieren als auch paralleles Sortieren auf Shared-Memory-Maschinen. Wir präsentieren *In-place Parallel Super Scalar Samplesort* (IPS^{4}o), einen neuen vergleichsbasierten Algorithmus, der mit beschränkt viel Zusatzspeicher auskommt (die sogenannte „in-place“ Eigenschaft). Eine wesentliche Erkenntnis ist, dass unsere in-place-Technik die Sortiergeschwindigkeit von IPS^{4}o im Vergleich zu ähnlichen Algorithmen ohne in-place-Eigenschaft verbessert. Bisher wurde die Eigenschaft, mit beschränkt viel Zusatzspeicher auszukommen, eher mit Leistungseinbußen verbunden. IPS^{4}o ist außerdem cache-effizient und führt $\mathcal{O}(n/t \log n)$ Arbeitsschritte pro Thread aus, um ein Array der Größe n mit t Threads zu sortieren. Zusätzlich berücksichtigt IPS^{4}o Speicherlokalität, nutzt einen Entscheidungsbaum ohne Sprungvorhersagen und verwendet spezielle Partitionen für Elemente mit gleichem Schlüssel. Für den Spezialfall, dass ausschließlich ganzzahlige Schlüssel sortiert werden sollen, haben wir das algorithmische Konzept von IPS^{4}o wiederverwendet, um *In-place Parallel Super Scalar Radix Sort* (IPS^{2}Ra) zu implementieren.

Wir bestätigen die Performance unserer Algorithmen in einer umfangreichen experimentellen Studie mit 21 State-of-the-Art-Sortieralgorithmen, sechs Datentypen, zehn Eingabeverteilungen, vier Maschinen, vier Speicherzuordnungsstrategien und Eingabegrößen, die über sieben Größenordnungen variieren. Einerseits zeigt die Studie die robuste Leistungsfähigkeit unserer Algorithmen. Andererseits deckt sie auf, dass viele konkurrierende Algorithmen Performance-Probleme haben: Mit IPS^{4}o erhalten wir einen robusten vergleichsbasierten Sortieralgorithmus, der andere parallele in-place vergleichsbasierte Sortieralgorithmen fast um den Faktor drei übertrifft. In der überwiegenden Mehrheit der Fälle ist IPS^{4}o der schnellste vergleichsbasierte Algorithmus. Dabei ist es nicht von Bedeutung, ob wir IPS^{4}o mit Algorithmen vergleichen,

die mit beschränkt viel Zusatzspeicher auskommen, Zusatzspeicher in der Größenordnung der Eingabe benötigen, und parallel oder sequentiell ausgeführt werden. IPS^{4}o übertrifft in vielen Fällen sogar konkurrierende Implementierungen von Integer-Sortieralgorithmen. Die verbleibenden Fälle umfassen hauptsächlich gleichmäßig verteilte Eingaben und Eingaben mit Schlüsseln, die nur wenige Bits enthalten. Diese Eingaben sind in der Regel „einfach" für Integer-Sortieralgorithmen. Unser Integer-Sorter IPS^{2}Ra übertrifft andere Integer-Sortieralgorithmen für diese Eingaben in der überwiegenden Mehrheit der Fälle. Ausnahmen sind einige sehr kleine Eingaben, für die die meisten Algorithmen sehr ineffizient sind. Allerdings sind Algorithmen, die auf diese Eingabegrößen abzielen, in der Regel für alle anderen Eingaben deutlich langsamer.

Im zweiten Teil dieser Arbeit untersuchen wir skalierbare Sortieralgorithmen für verteilte Systeme, welche robust in Hinblick auf die Eingabegröße, häufig vorkommende Sortierschlüssel, die Verteilung der Sortierschlüssel auf die Prozessoren und die Anzahl an Prozessoren sind. Das Resultat unserer Arbeit sind im Wesentlichen vier robuste skalierbare Sortieralgorithmen, mit denen wir den gesamten Bereich an Eingabegrößen abdecken können. Drei dieser vier Algorithmen sind neue, schnelle Algorithmen, welche so implementiert sind, dass sie nur einen geringen Zusatzaufwand benötigen und gleichzeitig unabhängig von „schwierigen" Eingaben robust skalieren. Es handelt sich z.B. um „schwierige" Eingaben, wenn viele gleiche Elemente vorkommen oder die Eingabeelemente in Hinblick auf ihre Sortierschlüssel ungünstig auf die Prozessoren verteilt sind.

Bisherige Algorithmen für *mittlere* und *größere* Eingabegrößen weisen ein unzumutbar großes Kommunikationsvolumen auf oder tauschen unverhältnismäßig oft Nachrichten aus. Für diese Eingabegrößen beschreiben wir eine robuste, mehrstufige Verallgemeinerung von Samplesort, die einen brauchbaren Kompromiss zwischen dem Kommunikationsvolumen und der Anzahl ausgetauschter Nachrichten darstellt. Wir überwinden diese bisher unvereinbaren Ziele mittels einer skalierbaren approximativen Splitterauswahl sowie eines neuen Datenumverteilungsalgorithmus. Als eine Alternative stellen wir eine Verallgemeinerung von Mergesort vor, welche den Vorteil von perfekt ausbalancierter Ausgabe hat. Für *kleine* Eingaben entwerfen wir eine Variante von Quicksort. Mit wenig Zusatzaufwand vermeidet sie das Problem ungünstiger Elementverteilungen und häufig vorkommender Sortierschlüssel, indem sie schnell qualitativ hochwertige Splitter auswählt, die Elemente zufällig den Prozessoren zuweist und einer Duplikat-Behandlung unterzieht. Bisherige praktische Ansätze mit polylogarithmischer Latenz haben entweder einen logarithmischen Faktor mehr Kommunikationsvolumen oder berücksichtigen nur gleichverteilte Eingaben ohne mehrfach vorkommende Sortierschlüssel. Für *sehr kleine* Eingaben schlagen wir einen einfachen sowie schnellen, jedoch arbeitsineffizienten Algorithmus mit logarithmischer Latenzzeit vor. Für diese Eingaben sind bisherige effiziente Ansätze nur theoretische Algorithmen, die meist unverhältnismäßig große konstante Faktoren haben. Für die *kleinsten* Eingaben empfehlen wir die Daten zu sortieren, während sie an einen einzelnen Prozessor geschickt werden.

Ein wichtiger Beitrag dieser Arbeit zu der praktischen Seite von Algorithm Engineering ist die Kommunikationsbibliothek *RangeBasedComm* (RBC). Mit RBC ermöglichen wir eine effiziente Umsetzung von rekursiven Algorithmen mit sublinearer Laufzeit, indem sie skalierbare und effiziente Kommunikationsfunktionen für Teilmengen von Prozessoren bereitstellt.

Zuletzt präsentieren wir eine umfangreiche experimentelle Studie auf zwei Supercomputern mit bis zu 262 144 Prozessorkernen, elf Algorithmen, zehn Eingabeverteilungen und Eingabegrößen variierend über neun Größenordnungen. Mit Ausnahme von den größten Eingabegrößen ist diese Arbeit die einzige, die überhaupt Sortierexperimente auf Maschinen dieser Größe durchführt. Die RBC-Bibliothek beschleunigt die Algorithmen teilweise drastisch – einen konkurrierenden Algorithmus sogar um mehr als zwei Größenordnungen. Die Studie legt dar, dass unsere Algorithmen robust sind und gleichzeitig konkurrierende Implementierungen leistungsmäßig deutlich übertreffen. Die Konkurrenten, die man normalerweise betrachtet hätte, stürzen bei „schwierigen" Eingaben sogar ab.

Acknowledgments

First and foremost, I would like to thank my doctoral advisor, Peter Sanders, for his support in so many ways. I'm grateful for granting me a position in his research team, for giving me advice whenever needed, and for letting me pursue my research interests. Thank you, Peter Sanders and Guy Blelloch, for reviewing my thesis.

Special thanks go to my co-authors Timo Bingmann, Daniel Ferizovic, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Peter Sanders, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, Armin Wiebigke, and Sascha Witt for their collaboration. Many thanks also go to Marisa Kirsch, Darren Strash, and Sascha Witt for proof-reading my thesis. I thank Jesper Larsson Träff and Christian Schulz for inviting me to Vienna for a research visit. I would also like to thank Sascha Witt for sharing an office with me as well as Julian Arz, Tomáš Balyo, Daniel Funke, Demian Hespe, Lorenz Hübschle-Schneider, Sebastian Lamm, and Darren Strash for joint recreational climbing and swimming activities.

I would like to thank all previous and current co-workers for the great years together in this wonderful team—thank you Yaroslav Akhremtsev, Julian Arz, Tomáš Balyo, Veit Batz, Norbert Berger, Timo Bingmann, Anja Blancani, Daniel Funke, Simon Gog, Demian Hespe, Tobias Heuer, Lukas Hübner, Lorenz Hübschle-Schneider, Markus Iser, Moritz Kobitzsch, Sebastian Lamm, Tobias Maier, Vitaly Osipov, Dennis Schieferdecker, Sebastian Schlag, Dominik Schreiber, Christian Schulz, Daniel Seemaier, Jochen Speck, Darren Strash, Marvin Williams, and Sascha Witt.

I gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN [SD15] at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF).

I gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (LRZ, www.lrz.de). I also gratefully thank all authors who have made their implementations available to the public. Special thanks go to SAP AG, Ingo Mueller, and Sebastian Schlag for making their 1-factor algorithm [SM13] available. This research was partially supported by DFG project SA 933/11-1.

I would like to thank my dear friend Darren Strash—I very much appreciate the friendship that started on our first day working together at the Karlsruhe Institute of Technology. Finally, I send love to my dear sister, my dear parents, and my wonderful wife Marisa—only because of you I became who I am today. Thank you for being there for me when I needed you.

Table of Contents

1 Introduction and Overview **1**

1.1 Sequential and Shared-Memory Sorting Algorithms . . . 2
1.2 Distributed Sorting Algorithms . . . 3
1.3 Machine Models . . . 4
1.3.1 Random Access Machine (RAM) . . . 4
1.3.2 The Parallel Random Access Machine (PRAM) . . . 5
1.3.3 The Parallel External Memory (PEM) Model . . . 5
1.3.4 The Single-Ported Message Passing Model . . . 6
1.3.5 Bulk Synchronous Parallel (BSP) Model . . . 7
1.4 General Preliminaries . . . 8
1.5 Contributions . . . 9

I Sequential and Shared-Memory Sorting **15**

2 Overview of Sequential and Shared-Memory Sorting Algorithms **17**

2.1 Definitions and Preliminaries . . . 17
2.2 Quicksort . . . 20
2.3 Samplesort . . . 21
2.4 Radix Sort . . . 21
2.5 (Strictly) In-Place Mergesort . . . 22

3 Robust Scalable In-Place Sorting Algorithms **25**

3.1 In-Place Parallel Super Scalar Samplesort (IPS^{4}o) . . . 25
3.1.1 Sequential and Parallel Partitioning . . . 25
3.1.2 Task Scheduling . . . 31
3.2 In-Place Parallel Super Scalar Radix Sort (IPS^{2}Ra) . . . 37
3.3 Analysis of IPS^{4}o . . . 38
3.3.1 Additional Memory Requirement . . . 38
3.3.2 Parallel Complexity and I/O Complexity . . . 39
3.3.3 Branch Mispredictions and Local Work . . . 43

4 Experiments and Conclusion **49**

4.1 Implementation Details . . . 53
4.2 Statistical Evaluation . . . 54
4.3 Sequential Algorithms . . . 56
4.3.1 Comparison of Average Slowdowns . . . 56

4.3.2 Running Times for Uniform Input 60
4.3.3 Comparison of Performance Profiles 62
4.4 Influence of the Memory Allocation Policy 63
4.5 Evaluation of the Parallel Task Scheduler 64
4.6 Parallel Algorithms 67
4.6.1 Comparison of Average Slowdowns 67
4.6.2 Running Times for Uniform Input 69
4.6.3 Speedup Comparison and Strong Scaling 71
4.6.4 Input Distributions and Data Types 73
4.6.5 Comparison of Performance Profiles 76
4.6.6 Comparison to IMSDradix 76
4.7 Phases of Our Algorithms 78
4.8 Conclusion 80

II Engineering Distributed Sorting Algorithms 85

5 Overview of Distributed Sorting Algorithms 87
5.1 Definitions and Preliminaries 87
5.1.1 Communication Primitives 88
5.1.2 Multiway Merging and Partitioning 90
5.1.3 Multisequence Selection 91
5.2 Delivering a Bulk of Data 91
5.3 Sorting Algorithms from Tiny to Huge Inputs 92
5.3.1 (All)gather-Merge-Sort 94
5.3.2 Fast Work-Inefficient Rank 94
5.3.3 Bitonic Sort 95
5.3.4 Parallel Quicksort 95
5.3.5 Multiway Sorting Algorithms 96
5.4 More Related Work 98

6 Robust Scalable Distributed Sorting Algorithms 101
6.1 Robust Fast Work-Inefficient Ranking and Sorting (RFIS) 101
6.2 Building Blocks for Hypercube Quicksort 103
6.2.1 Bound for the Balls into Bins Problem 103
6.2.2 Randomized Shuffling on Hypercubes 104
6.2.3 Approximate Median Selection with a Single Reduction 105
6.3 Robust Quicksort on Hypercubes 107
6.4 Delivering k Data Partitions to k PE-Groups 113
6.5 Generalizing Multiway Mergesort (RLM-sort) 119
6.6 Adaptive Multi-Level Samplesort (AMS-sort) 120

7 Experiments and Conclusion 127
7.1 Implementation Details 129
7.1.1 The k-way Data Exchange 130

7.1.2 Parameter Tuning 131
7.2 Methodology 131
7.3 Input Size Analysis and Algorithm Comparison 132
7.4 Robustness Analysis 138
7.4.1 Robustness of Logarithmic Latency Algorithms 138
7.4.2 Robustness of RQuick 140
7.4.3 Robustness of Multiway Hypercube-Based Sorting 143
7.4.4 Robustness of AMS-sort 144
7.5 Efficiency in Scaling Experiments 148
7.6 Conclusion 149

A Sequential and Shared-Memory Sorting **155**
A.1 Limit the Number of Recursions 155
A.2 From In-Place to Strictly In-Place 157
A.3 More Measurements 161

B On the Performance of MPI Libraries **177**
B.1 Startup Overheads of Communication Patterns 177
B.2 Faster Sorting with MPI—Communicators and Collectives 180

List of Algorithms **185**

List of Figures **187**

List of Tables **189**

List of Theorems **191**

Publications and Supervised Theses **193**

Bibliography **195**

Chapter 1

Introduction and Overview

1

In this thesis, we examine how to efficiently sort data on shared-memory and distributed-memory machines. Our target is to robustly scale sorting up to the largest shared-memory machines and distributed systems with low overhead in theory as well as in practice—even for "difficult" worst case inputs.

Before we address these challenges in the following chapters, we present the reader to our motivation. Afterwards, we describe the machine models and general preliminaries used in this thesis. We conclude the chapter with an overview of our contributions.

"Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms."

— INTRODUCTION TO ALGORITHMS [Cor+09]

Roughly speaking, sorting is the task of bringing elements in a specific generally non-decreasing order. Sorting algorithms are very important for many applications. The applications may inherently need a subroutine to obtain sorted data. They may also take advantage of the property that data is sorted or that similar data is stored logically or physically close together. Therefore, it is not surprising that reference books on sequential and parallel algorithms often devote sorting a separate chapter [Cor+09; Kum+94; San+19]. These applications demand fast sorting algorithms for a diverse spectrum of machine architectures as well as for a wide range of inputs in regard to data type, input size, and distribution of key values and elements.

Nowadays, applications are executed on a wide range of different machine architectures. About fifteen years ago, the CPU frequency stopped increasing while Moore's law—the number of transistors on a processor increases exponentially—still applies. Since then, the performance of a system has mainly been improved by increasing the number of cores in CPUs, the number of CPUs in the machines, and the number of machines in supercomputers and clusters. For example, workstation CPUs have evolved from one core in 2015 to 64 cores in 2020 [AMD15; AMD20]. At the same time, the number of cores of the largest supercomputers has risen by almost two orders of magnitude to more than ten million cores assembled in the *Sunway TaihuLight* supercomputer [top05; top20]. Even in everyday life, we have access to smartphones with half a dozen cores [App20]. These improvements can satisfy the need to solve tasks faster and process data in larger amounts. Still, efficient and scalable parallel algorithms designed for these computer architectures are required. In this thesis, we study sequential sorting as well as parallel sorting on shared-memory machines in Part I, and sorting on machines with distributed memory in Part II.

1.1 Sequential and Shared-Memory Sorting Algorithms

In Part I of this thesis, we study the problem of sorting an array of n elements sequentially as well as in parallel on shared-memory machines with t threads. Since sorting is considered to be one of the most fundamental algorithmic problems, hundreds of publications have studied how to make sorting algorithms fast. As of today's date, more than four hundred technical reports in the field of computer science contain the term "sort" in the title on the open-access repository `arxiv.org`. At the first glance, this seems to be a surprise, since simple asymptotically optimal algorithms have been known for more than 50 years, i.e., mergesort [Knu73], which is optimal in the worst case, or quicksort [Hoa62], which is optimal with high probability for random pivots. However, this is only the beginning in the quest for the fastest sorting algorithm. Today's modern machines provide plenty of opportunities for optimization and many hardware features, such as the cache [SW04; BFV07; Fra04; BGS10; KDD17], instruction parallelism [SW04; Cor20; HWF15; PR14], branch prediction [SW04; KS06a; EW16; Yar10; Ska16a; BES17], multi-core [TZ03; SSP07; BGS10; Shu+12; TZ03; Rei07; Obe+19b; MG89; HNR90; FP92; HWF15; PR14; KDD17; BES17; PR14], and virtual memory [JM14; WS11], have been studied in the context of sorting.

On the one hand, it is somewhat surprising that quicksort is the algorithm of choice in most libraries, e.g., in standard libraries of programming languages such as Java and C++, or in parallel programming libraries such as TBB [Rei07]. On the other hand, even simple implementations of quicksort are somewhat cache-efficient (it still reads each element a logarithmic number of times), have an asymptotically optimal running time with high probability, and are in-place—an important feature for machines with limited memory. Furthermore, quicksort can be parallelized [SSP07; Rei07] and sophisticated implementations can reduce branch mispredictions significantly [KS06b; EW16]. Although we have not found an implementation of quicksort that provides all of these features, it seems difficult to beat quicksort on a similar or larger feature set. For example, mergesort can take advantage of inputs with presorted sequences. In the remaining cases, it turns out that mergesort is slower in practice. This is in particular the case for in-place mergesort. Samplesort either exploits instruction parallelism and minimizes branch mispredictions [SW04] or runs in parallel [Ble+96; BGS10; Shu+12]. Radix sorting algorithms can exploit fast element classification, e.g., for (nearly) uniform distributed input. However, these algorithms pay the price for accepting imbalanced buckets in the remaining cases and only work for a limited set of data types. That said, we did not find an algorithm that outperforms quicksort in all circumstances.

With this thesis, we contribute the cache-efficient *In-place Parallel Super Scalar Samplesort* (IPS^{4}o) algorithm which exploits instruction parallelism and simultaneously reduces branch mispredictions significantly. Even though we focus our study on the more general case of comparison-based sorting, we also contribute *In-place Parallel Super Scalar Radix Sort* (IPS^{2}Ra) and compare ourselves to state-of-the-art radix sort algorithms.

1.2 Distributed Sorting Algorithms

In Part II and Appendix B of this thesis, we study the problem of sorting n input elements evenly distributed among t processors of a distributed-memory machine. On large machines, the network bottleneck becomes a significant issue. For the analysis, we thus have to consider message startup latencies and communication volume on the critical execution path. On real-world machines, the time for a message startup is significantly larger than the transfer time for a machine word. It turns out that sorting algorithms trade off message startups and communication volume differently. As a result, each algorithm is only suitable for a specific range of input sizes.

Although hundreds of papers on parallel sorting have been published, there is only a small number of practical studies that consider the largest machines with many thousands of processors (PEs). The most studied distributed sorting algorithms are single-level algorithms [Ble+96; Var+91; KK93; SSP07; SK10; SMB13; HKS19]. Single-level algorithms select $t - 1$ splitters, send them to all processors, partition the local data into t pieces, and redistribute them using an all-to-all data exchange. Since every PE receives at least the $t - 1$ splitters, these algorithms require a total input of $\Omega\left(t^2/\log t\right)$ elements to be efficient. In this case, each processor may exchange $\Omega(t/\log t)$ messages when the data exchange algorithm sends the pieces directly to the target processors. Thus, the constant factor of the term $\Omega\left(t^2/\log t\right)$ may become very large. This makes single-level algorithms only efficient for very large inputs.

For smaller input sizes, we need faster sorting algorithms since t-way sorting is not efficient and scalable enough. The task to bring "similar" data together is important for many applications on distributed systems. For example, sorting on space filling curves [Bad13] is one approach to partition computational domains and can be much more efficient than graph-based partitioning algorithms [Den03]. Sorting may then become the limiting factor when scaling to thousands of processors, e.g., when a dynamic scheduler of simulations uses space filling curves for frequent load rebalancing. For these applications, the number of simulation steps per time step is the important performance metric and the input per processor can be very small [LBH07]. Thus, fast sorting algorithms for small inputs are required—even when scalability is beyond question. A special case for sorting small inputs is the distributed collective operation `MPI_Comm_Split` which basically means sorting inputs with one element per processor [SW11].

Asymptotically optimal algorithms have been intensively studied in theory. For example, Cole's mergesort [Col88] algorithm sorts inputs with one element per PE in time $\mathcal{O}(\log t)$. However, this comes with large constant factors in practice. Simple algorithms with polylogarithmic message startup latency are variants of hypercube quicksort [Wag87; LM92; SMB13] and bitonic sort [SMB13]. However, quicksort is fast on only a narrow range of small input sizes since it exchanges the data a logarithmic number of times. Bitonic sort even exchanges data $\mathcal{O}\left(\log^2 t\right)$ times.

So-called r-level sorting algorithms [KK93; GV94; Goo99; SMB13; HKS19] try to close the gap between algorithms for very large inputs with $\mathcal{O}(t)$ message exchanges and algorithms for small inputs with polylogarithmic startup latency but $\Omega(\log t)$ data exchanges. The lower bound of r-level sorting algorithms is $\Omega\left(rn/t + rt^{1/r}\right)$. However, a straightforward implementation has $\mathcal{O}(rn/t)$ communication volume but $\Omega(t)$ message startups in the worst case [KK93]. Implementations that restrict data exchanges to communication partners sharing the same

hypercube edge can guarantee $\mathcal{O}(rt^{1/r})$ message startups. However, these algorithms require t to be a power of two and the communication volume increases to $\mathcal{O}(rn/t^{1/2})$ in the worst case—even when the splitter elements partition the data into buckets of equal size. As a consequence, most of these algorithms are slow or crash for “difficult” inputs, i.e., when many elements have equal keys (duplicated keys) and when the location of the input elements is correlated to the key values (skewed element distributions). Unfortunately, these disadvantages also apply to variants of hypercube quicksort. As a consequence, these algorithms are mostly studied for random inputs.

In Part II of this thesis, we consider practical massively parallel sorting algorithms for large inputs down to inputs where the number of elements is smaller than the number of processors. We propose four robust sorting algorithms that cover the entire range of input sizes. We consider *robustness* in regard to scalability, i.e., its running time dependent on the input size and the number of processors, and in regard to the presence of duplicated keys as well as skewed input distributions.

1.3 Machine Models

In this section, we describe the well-known sequential machine model *random access machine* (RAM) [vNeu45; SS63; CR72] as well as several parallel machine models. Parallel machine models are further divided into models of shared-memory machines and distributed-memory machines. The simplest shared-memory machine model used in this work is the *parallel random access machine* (PRAM) [FW78]. The *parallel external memory* (PEM) model [Arg+08] is an extension of the PRAM to two memory levels. The *single-ported message passing model* describes a distributed-memory machine with point-to-point message exchanges. The *bulk synchronous parallel* (BSP) model [Val90] is a more abstract model that exchanges messages bulk synchronously. Sanders et al. [San+19] describe and discuss these models in detail.

1.3.1 Random Access Machine (RAM)

The *RAM* is a sequential machine with a single *processor* and *uniform main memory*. The main memory is a storage with an infinite number of available *machine words*. The processor has direct access to a small number of *registers*. The registers store machine words that are input and output of basic operations executed by the processor. A machine word stores an integer value whose absolute value is bounded by a polynomial in the input size of a program. This allows a machine word to be used to index main memory that is polynomial to the input size. The processor accesses the main memory with load and store operations. A random access machine executes a program that is basically a sequence of *basic operations*, i.e., arithmetic operations, comparison operations, logical operations, bitwise operations, (un)conditional branches, and load respectively store operations. For each executed operation, we account a single *time step*. The *running time*, often just called “time”, of a program is the number of operations required until the program terminates.

1.3.2 The Parallel Random Access Machine (PRAM)

The *PRAM* is the equivalent of the RAM for shared-memory parallel computing. It consists of t PEs sharing a *common main memory*. Each PE executes the same program in *synchronized time steps*. The execution order of the operations executed by the PEs may vary caused by conditional branches depending on the data, their rank, and the total number of PEs. The RAM model is divided into several submodels. The EREW-PRAM allows *exclusive reads* from and *exclusive writes* to the main memory. The CREW-PRAM allows *concurrent reads* from and *exclusive writes* to the main memory. In the real world, machines support something similar to concurrent reads, although the read value may not be the globally last written value. Exclusive accesses to the same memory location must be coordinated, e.g., by serialization in $\mathcal{O}(t)$ time or by combining the machine words in $\mathcal{O}(\log t)$ time. The CRCW-PRAM is the most powerful machine since it allows both, *concurrent reads* as well as *concurrent writes*. Conflict resolution rules define the semantics of concurrent writes. For example, the resolution rule "arbitrary" only allows writes to the same cell if all PEs writing to that cell write the same value. Another resolution rule is the "common" semantic that writes the value of a random PE among all PEs attempting to write to the same cell at the same time. The time measure of the PRAM model is the asymptotic number of time steps of the PE that finishes the program at last. We call the sequence of instructions executed by this PE the critical path.

In real world machines, access to the main memory happens in a rather asynchronous way. This makes it difficult to coordinate concurrent memory accesses, e.g., when a single memory access is split into several execution stages or when a series of operations shall exclusively operate on a part of the memory without interference with other PEs. *Atomic operations* overcome this problem for commonly used short series of operations. An atomic operation is executed at a stretch. This guarantees that the required values in the main memory are accessed exclusively by this operation, without interference.

1.3.3 The Parallel External Memory (PEM) Model

The *PEM model* is a cache-aware extension of the parallel random-access machine. This model is used to analyze parallel algorithms if the main issue is the number of accesses to the main memory. In the PEM model, each of the t PEs has a private cache of size M and access to main memory happens in *memory blocks* of size B. The *I/O complexity* of an algorithm is the number of parallel memory block transfers (I/Os) between the main memory and the private caches on the critical path. In this work, we use the term *cache-efficient* as a synonym for I/O-efficient when we want to emphasize that we consider memory block transfers between the main memory and the private cache. The PEM model supports the same access policies to the main memory as the PRAM model. Namely, we get the submodels EREW-PEM, CREW-PEM, and CRCW-PEM. Concurrent writes can use the conflict resolution rules as proposed for the CRCW-PRAM model. Similarly, exclusive accesses to the same memory location must be coordinated, e.g., by serialization, which accounts to $\mathcal{O}(t)$ I/Os, or by combining blocks, which accounts to $\mathcal{O}(t)$ I/Os.

For a given machine instance of the PEM model, the parameters M and B are constants. Most of the time, however, we treat M and B as variables in our asymptotic analysis in order to expose the effects of block size and cache limitations.

We adopt an asynchronous variant of the PEM model based on the asynchronous CRCW PRAM by Sanders et al. [San+19, p. 38-40] (also see [Gib89]). The asynchronous variant allows atomic instructions such as *fetch-and-add* on variables that are shared with other threads. In our model, we charge t I/Os if a thread atomically accesses a shared variable. We avoid additional delays due to *false sharing* by allocating at most one shared variable to each memory block.

1.3.4 The Single-Ported Message Passing Model

A common abstraction of communication in supercomputers is the *single-ported message passing model*. This model describes a distributed-memory machine consisting of t random access machines connected by a communication network. Each PE i executes the same program and knows its rank $i \in [0 .. t)$ as well as the total number of PEs. The PEs exclusively communicate with *point-to-point communication*. It takes time $\alpha + m\beta$ to send a message of size m machine words. The parameter α defines the *message startup overhead* of the communication. The parameter β defines the *time to communicate* one machine word. Multiple messages can be transferred at the same time. However, each PE can send and receive at most one message at the same time. For a given machine instance of the message passing model, the parameters α and β are constants. Most of the time, however, we treat α and β as variables in our asymptotic analysis in order to expose the effects of latency and bandwidth limitations. For simplicity, the single-ported message passing model ignores network hierarchies and assumes that $\alpha \gg \beta \gg 1$ where our unit is the time for executing a simple machine instruction. We also assume that the size of a machine word is equivalent to the size of a data element.

The cost measure of this message passing model is the asymptotic number of time steps executed by the PE that terminates last—the so-called *critical path*. The critical path consists of the following measures: (1) The "local work" executed by the PE, (2) its time for the actual message transfers, and (3) the time it spends waiting until its sending and receiving communication partners have posted the communication requests.

In reality, the time of a message transfer may also depend on many other parameters, e.g., the network hierarchy of the communication partners relative to each other and the communication protocols used by the communication library. For example, the Intel MPI library 2020 provides two protocols: The *eager protocol* has a small latency since it sends the message immediately independent of the receiver status. This protocol is usually used for small messages since the receiver has to provide interim buffers. The *rendezvous protocol* only starts the actual message transfer after the receiver has acknowledged an available receive buffer. Thus, this protocol has a large latency but can be used for arbitrary message sizes. We evaluate the performance of the Intel MPI library 2020 on the SuperMUC-NG [Lei18] supercomputer. The computation nodes of SuperMUC-NG are bundled into so-called *islands*. Within an island, the nodes are connected by a fat tree network. At the level of islands, the overall communication bandwidth is pruned by a factor of four. Looking at Figure 1.1 we see the running times of message transfers with different message lengths. We obtained the measurements by performing ping-pong message transfers between two communication partners for two seconds. Before the actual

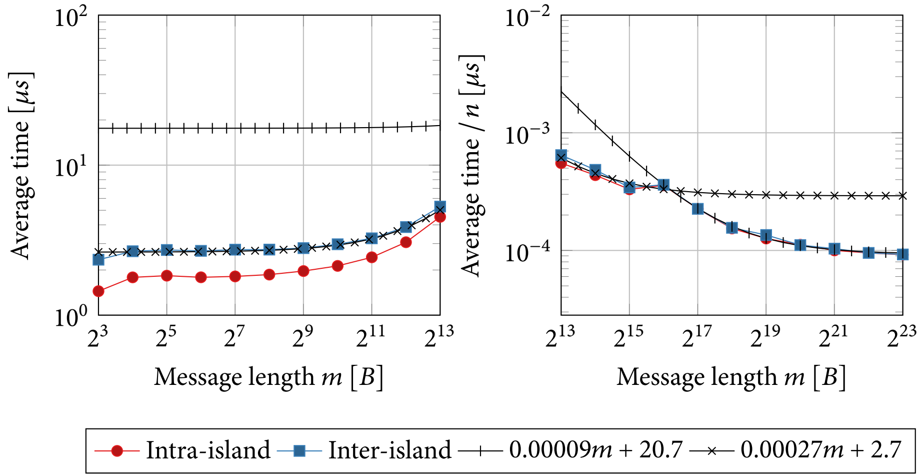


Figure 1.1: Running time of ping-pong benchmarks between two fixed PEs on SuperMUC-NG.

measurements, we executed the same message transfers as a warmup. Both, the intra-island communication as well as the inter-island communication use the eager protocol for messages up to 32 kB and the rendezvous protocol otherwise. The two black lines in Figure 1.1 fit the inter-island communication cost to the single-ported message passing model for both protocols separately with linear regression. According to the fitting, the eager protocol has a latency of 2.71 μs and a bandwidth of 3.6 GiB/s whereas the rendezvous protocol has a latency of 20.73 μs and a bandwidth of 11.1 GiB/s. The theoretical peak bandwidth is 12.5 GiB/s. Inter-island as well as intra-island communication has about the same running time with the rendezvous protocol. However, when we consider the eager protocol, we see that intra-island communication has an 86.90 % smaller latency than inter-island communication. The fittings of the two protocols to the single-ported message passing model predict the cost of the message transfers shown in Figure 1.1 very precisely. For example, the two models of inter-island communication have an error of less than 12.50 % (eager protocol) respectively 2.96 % (rendezvous protocol). Unfortunately, a single model for inter-island communication is much less accurate, e.g., it has an error of up to 55.45 %. When we also include intra-island communication, the single-ported message passing model fits even less to our measurements. We also observed high fluctuations in running time when we simultaneously exchanged data with a lot of different communication partners on various supercomputers. For more information, we refer to Appendix B.1.

1.3.5 Bulk Synchronous Parallel (BSP) Model

Many algorithms are *bulk synchronous*. Such algorithms are often described in the framework of the *BSP model*. Algorithms in the BSP model execute rounds of *local computation* each followed by a *synchronized bulk data transfer*. In the local computation phase, PEs can post

send requests. In the data transfer phase, the posted messages are then delivered to their target PEs. This communication is typically considered as *one-sided communication*, meaning that the receiving PEs do not need to post receive requests. Let $\mathrm{BSP}(h) = l + hg$ denote the time needed to execute a data exchange step where no PE receives or sends more than h words in total. The gap g defines the number of computation steps for transferring a single word. The term l represents the startup overhead for the data exchange algorithm and the time for its synchronization.

1.4 General Preliminaries

We use the notation $[a\,..\,b]$ as a shorthand for the ordered set $\{a,\ldots,b\}$, and $[a\,..\,b)$ for $\{a,\ldots,b-1\}$. Similarly, $[a_i\,..\,a_j]$ is a shorthand for $\{a_i,\ldots,a_j\}$, and $[a_i\,..\,a_j)$ is a shorthand for $\{a_i,\ldots,a_{j-1}\}$. We also use $\log x$ for $\log_2 x$ and $\ln x$ for $\log_e x$. The symbol $\mathbb{N}$ stands for the natural numbers including zero. We use the notation $\mathrm{P}[\cdot]$ to express probabilities.

Pseudocode

We use pseudocode in the description of algorithms rather than a normal programming language. Our pseudocode is an abstraction of imperative programming languages such as C++, Java, and Pascal. We use mathematical formulas and symbols to present calculations in a compact and legible way. Sometimes we simplify algorithms so much that we describe function calls textually. For example, we may write “select a random sample of size n from set A”.

Parallel algorithms are described in the single-program multiple-data programming model. In this model, the same pseudocode is executed in parallel by multiple PEs. Each PE has access to its index and to the total number of PEs. When we describe shared-memory algorithms, the PEs may access shared data structures, and distributed algorithms communicate via message passing. To increase readability, we also allow distributed algorithms to access non-local memory. For example, PE i may execute the command

$$\sum_{j\in[0\,..\,i)} a \,@\, \mathrm{PE}\; j$$

that adds up the values of variable a stored on PE $[0\,..\,i)$. We use this notation only when these commands can be executed efficiently with a single collective operation.

Chernoff Bound

A *Bernoulli experiment* is a random $\{0,1\}$-variable X with “success” probability $\mathrm{P}[X=1]=t$ and “failure” probability $\mathrm{P}[X=0]=1-t$. In its most generic form, the *Chernoff inequality* bounds the probability that a sequence of independent Bernoulli experiments deviates from the expected number of successes. Here, we consider a specific Chernoff bound [MU05] for Bernoulli trials. A *Bernoulli trial* is a set of independent random variables $[X_1\,..\,X_n]$ each having the same success probability $\mathrm{P}[X_i=1]=t$. The Chernoff inequality bounds the probability

that the Bernoulli trial deviates from the expected number of successes nt by a factor of at least $1 + \delta$ for any $\delta > 0$:

$$\mathrm{P}[\sum X_i \geq (1+\delta)tn] \leq e^{-\frac{\min(\delta,\delta^2)tn}{3}} \quad .$$

Parallel Algorithm Analysis

Let $T(n,t)$ be the asymptotic running time of a parallel algorithm A with an input of size n executed by t PEs. Also, let $T(n)$ be the running time of the fastest sequential algorithm B that solves the same problem. Then, the *absolute speedup* (speedup) of A is defined by

$$S(n,t) = \frac{T(n)}{T(n,t)}$$

and the *efficiency* of A is defined by

$$E(n,t) = \frac{S(n,t)}{t} \quad .$$

Furthermore, we denote the *total overhead function* of algorithm A as

$$T_o(t,n) = tT(n,t) - T(n) \quad .$$

The *isoefficiency function* [Kum+94] of an algorithm A describes the relation of the input size to the number of PEs such that the efficiency of A remains constant when the number of PEs increases. To determine the isoefficiency function of A, we require

$$T_o(n,t) \in T(n) \quad .$$

Roughly speaking, we want that the total overhead of A does not dominate the asymptotic running time of B. The Isoefficiency function $I(t)$ then describes n as a function of t such that this requirement is fulfilled, i.e., we need $T_o(I(t),t) \in T(I(t))$. Thus, when we use t' instead of t PEs and the speedup of A shall increase by a factor of t'/t, we must increase the input size by a factor of $I(t')/I(t)$.

1.5 Contributions

In this thesis, we present contributions to the field of sequential, shared-memory, and distributed sorting algorithms. The following list provides the main contributions to sequential algorithms and algorithms for shared-memory systems. The contributions base on the conference article Ref. [Axt+17c] and the technical report Ref. [Axt+20].

- **Algorithm IPS^{4}o.** The comparison-based sorting algorithm ***In****-place* ***P****arallel* ***S****uper* ***S****calar* ***S****ample****so****rt* provides the following features: IPS^{4}o works sequentially as well as in parallel, is provably in-place, cache-efficient, and has work $\mathcal{O}(n/t \log n)$ per thread. The algorithm avoids branch mispredictions and is robust against a large number of equal keys with low overhead.

- **Algorithm IPS²Ra.** The sorting algorithm ***I**n-Place **P**arallel **S**uper **S**calar **R**adix **S**ort* is a proof-of-concept implementation of *most-significant-digit (MSD) radix sort* [Fri56] using the in-place partitioning framework we developed for IPS⁴o.

- **Experiments.** We validate the performance in extensive experiments considering a large part of the cross product of 21 state-of-the-art sorting codes, 6 data types, 10 input distributions, 4 machines, 4 memory allocation strategies, and input sizes varying over 7 orders of magnitude.

- **Experimental results of IPS⁴o.** The algorithm outperforms all competing implementations in most cases. In many of these cases, the performance difference is large. For example, IPS⁴o outperforms its fastest parallel non-in-place and in-place comparison-based competitor on average by a factor of 2.00 and 2.53 respectively for 64-bit floating-point inputs of medium and large size. When IPS⁴o is slower than a competitor, this is only by a small percentage in the overwhelming majority of cases.

- **Experimental results of IPS²Ra.** The radix sort algorithm complements the results obtained for IPS⁴o by being even faster in the sequential case and in some cases involving few processor cores and small, uniformly distributed keys. IPS²Ra also outperforms other radix sorters in many cases. For example, IPS²Ra outperforms its fastest parallel sequential and parallel radix sort competitors on average by a factor of 1.15 and 1.54 respectively for 64-bit integer inputs of medium and large size.

The following list provides the main contributions to algorithms for distributed systems. The contributions base on the publications Ref. [Axt+15a; AS17; AWS18].

- **Data redistribution DMA [Axt+15a].** Let $t = k^r$ with $k, r \in \mathbb{N}$ be the number of PEs. Assume that each PE has partitioned its n/t local elements into k pieces and assume that the total size of pieces with number i is n/k. We implement the data redistribution algorithm ***D**eterministic **M**essage **A**ssignment* (DMA) which moves pieces with number i to PEs $[it/k .. (i+1)t/k)$. The calculation of the communication partners takes time $\mathcal{O}(\alpha \log t + \beta k)$. During the actual data exchange step, each PE sends and receives n/t elements and $\mathcal{O}(r)$ messages.

- **Algorithm AMS-sort [Axt+15a].** The implementation of the r-level algorithm ***A**daptive **M**ulti-level **S**ample**sort*** works for arbitrary t, guarantees that the PEs always have less than $(1+\epsilon)n/t$ local elements with tuning parameter ϵ, and has isoefficiency function $t^{1+1/r}/\log t$ assuming α, β, r and ϵ to be constants. In total, AMS-sort sends and receives at most $r(1+\epsilon)n/t$ elements and $\mathcal{O}\left(rt^{1/r}\right)$ messages per PE in the data redistribution and requires only $r^2 t^{1/r}/\epsilon$ splitter candidates. For the case that perfectly balanced output is required, we offer ***R**ecurse **L**ast **M**ultiway **M**erge**sort*** (RLM-sort) with isoefficiency function $t^{1+1/r}\log t$.

- **Algorithm RQuick [AS17].** ***R**obust Hypercube **Quick**sort* is an efficient and robust variant of hypercube quicksort that incorporates a fast and accurate splitter selection

algorithm, a low overhead measure to handle many equal keys, and input randomization to robustly sort skewed element distributions. We conjecture an isoefficiency function $t \log t$ and running time

$$\mathcal{O}\left(\alpha \log^2 t + \beta \log^2 t + \frac{n}{t} \log n\right) .$$

***R**obust Hypercube **Quick**sort*$^+$ (RQuick$^+$), a modification of RQuick, provides a splitter selection that is more complex, but offers splitter quality guarantees in return. For inputs without duplicates, RQuick$^+$ runs in time

$$\mathcal{O}\left(\alpha \log^2 t + \beta \log^2 t + \frac{n}{t} \log n\right)$$

with high probability.

- **Algorithm RFIS [AS17].** ***R**obust **F**ast **W**ork-**I**nefficient **S**orting* is a simple sorting algorithm a with low overheard measure to handle many equal keys and has running time

$$\mathcal{O}\left(\alpha \log t + \beta \tfrac{n}{t^{1/2}} + \tfrac{n}{t} \log \tfrac{n}{t}\right) .$$

- **RBC library [AWS18].** For an efficient implementation of sublinear time algorithms, we propose the lightweight library ***R**ange**B**ased**C**omm* (RBC) based on MPI. RBC provides asymptotically time optimal implementations of collective communication primitives on range-based communicators and communicator creation in constant time without synchronization and communication.

- **Speedups with RBC.** We use RBC to implement our algorithm. We also applied RBC to our competitor algorithms: RBC improves the performance of hypercube quicksort from Sundar et al. [SMB13] by more than one order of magnitude. RBC also improved the multi-level algorithms HykSort [SMB13] and HSS [HKS19] by a factor of more than 5 and more than two orders of magnitude respectively.

- **Experiments [AS17].** We compare our algorithms directly with several state of the art implementations considered in recent studies [SW11; SMB13; HKS19], on two supercomputers with up to 262 144 core, ten input distributions, and input sizes varying over 9 orders of magnitude.

- **Input size evaluation [AS17].** The experimental results confirm our hypothesis that four sorting algorithms for different input sizes can be used to span the entire parameter space: AMS-sort for medium-sized and large inputs, RQuick with polylogarithmic latency for small inputs, RFIS with logarithmic latency for very small inputs and inputs with less elements than PEs, and a simple algorithm that sorts while data is routed to a single PE to sort even less elements.

- **Robustness evaluation and algorithm comparison [AS17].** Experiments with ten input distributions validate that our algorithms are robust against skewed input distributions and repeatedly occurring keys.

  RQuick is for small inputs faster than hypercube quicksort from Sundar et al. [SMB13] by more than one order of magnitude. Classical bitonic sorting [Bat68; Joh84] is somewhat competitive only for a rather narrow range of input sizes.

  AMS-sort is the overall "winner" for medium-sized and large inputs: For some large inputs, HykSort [SMB13] is slightly faster than AMS-sort as well as HSS from Harsh et al. [HKS19] is competitive. However, HykSort and HSS are less robust, i.e., they are much slower than AMS-sort or even crash for skewed inputs and inputs with duplicates.

  The data redistribution algorithm **DMA** speeds up AMS-sort by up to three orders of magnitude for "hard" inputs compared to previous data redistribution algorithms proposed for multi-level sorting algorithms that also work with arbitrary t [KK93]. Classical samplesort and related algorithms that communicate the data only once are very slow even for rather large n/t.

I

Sequential and Shared-Memory Sorting

In this part of the thesis, we study sequential and shared-memory sorting algorithms and present two new cache-efficient in-place algorithms that outperform competing implementations significantly in the majority of cases.
Chapter 2 introduces basic tools in Section 2.1 and discusses related work in Sections 2.2–2.5. Afterwards, we describe our new samplesort algorithm IPS^{4}o in Section 3.1 of Chapter 3 and our radix sort algorithm IPS^{2}Ra in Section 3.2. Chapter 3 concludes with an in-place, local-work, and I/O complexity analysis in Section 3.3. We then turn to an extensive experimental evaluation in Chapter 4. We first give implementation details of IPS^{4}o and IPS^{2}Ra in Section 4.1 and describe our statistical evaluation measures in Section 4.2. Subsequently, we discuss the results of sequential and shared-memory experiments in Sections 4.3–4.7. A final conclusion and an outlook on possible future work is given in Section 4.8. Appendix A gives additional proofs, and provides further experimental data.

References and Attributions. This part of the thesis is based on a conference article [Axt+17c] and a technical report [Axt+20] jointly published with Peter Sanders, Sascha Witt, and Daniel Ferizovic. Most parts of the publications were written by the author of this thesis. He supervised the development of the sorting algorithms presented in this thesis and he supervised Daniel Ferizovic who was at that time a student employed at our institute. Daniel Ferizovic provided an initial implementation of IPS^{4}o which was later revised by Sascha Witt as well as extended and tuned by the author of this thesis. Daniel Ferizovic also obtained the measurements presented in the conference article. The extensive experimental evaluation presented in this part of the thesis is exclusive work of the author of this thesis. He also independently implemented the algorithm IPS^{2}Ra (and the reference implementation PS^{4}o). Peter Sanders provided the basic idea on how to make IPS^{4}o in-place as well as important advice for the development of IPS^{4}o, but he was not involved in the actual implementation process. Peter Sanders and Sascha Witt were involved in the editing of the publications and provided helpful remarks and comments to improve the presentation of the results. They also provided several notable parts of the publications. The technical report [Axt+20] is a consolidation of the work. Part I and Appendix A is a copy of most parts of this technical report. The report has also been submitted to the journal *ACM Transactions on Parallel Computing* and is currently under revision.

Chapter 2

2 Overview of Sequential and Shared-Memory Sorting Algorithms

This chapter provides important definitions, preliminary information, and related work for sequential and shared-memory sorting algorithms. Section 2.1 describes the *sorting problem*, defines the *in-place* property of algorithms in the sequential and parallel setting, and provides a summary of notations used throughout Part I of the thesis. We also give a description of samplesort and its non-in-place implementation *Super Scalar Samplesort*, which was the starting point for the development of IPS^{4}o and IPS^{2}Ra. In Sections 2.2–2.5, we describe the relevant related work of sequential as well as shared-memory sorting algorithms.

2.1 Definitions and Preliminaries

The input of a sorting algorithm is an array $A[0 .. n-1]$ of n elements, sorted by t PEs. Table 2.1 gives an overview of the most important notation used in this chapter. We expect that the output of a sorting algorithm is stored in the input array.

In algorithm theory, a sequential algorithm works **in-place** if it uses only constant space in addition to its input. We use the term *strictly in-place* for this case. In algorithm engineering, one is sometimes satisfied if the additional space is logarithmic in the input size. In this case, we use the term *in-place*. In the context of in-place algorithms, we count machine words and equate the machine word size with the size of a data element to be sorted. Note that other space complexity measures may count the number of used bits. For the purpose of **parallel in-place algorithms**, we interpret "constant" as "independent of the input size" but allow additional space to depend on parameters of the machine model like the number of PEs. Recently, models for parallel in-place algorithms were proposed [GOS21]. There, even in the strong variant, a logarithmic number of machine words are allowed to be allocated on the stack by each PE. As far as we understand it, our algorithms could be cast within that framework without resorting to the tricks we use to become strictly in-place.

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

Table 2.1: Summary of notations

Symbol	Meaning
A	input array
n	number of elements
t	# of PEs
M	fast memory size of the PEM model
B	block size of PEM model
b	block size within IPS^{4}o
k	distribution degree
b_i	i-th bucket
w_i	write pointer of bucket i
r_i	read pointer of bucket i
$T[\ell, r]$	the task to sort $A[l..r-1]$

(Super Scalar) Samplesort.

The k-way S^{4}o algorithm [SW04] starts with allocating two temporary arrays of size n—one data array to store the buckets, and one *oracle array* to store the bucket indices where input elements are placed. The partitioning routine contains three phases and is executed recursively. The sampling phase sorts $\alpha k - 1$ randomly sampled input elements where the *oversampling factor* α is a tuning parameter. The splitters $S = [s_0 .. s_{k-2}]$ are then picked equidistantly from the sorted sample. The classification phase classifies each input element, stores its target bucket in the oracle array, and increases the size of its bucket. Element e goes to bucket b_i if $s_{i-1} < e \leq s_i$ (with $s_{-1} = -\infty$ and $s_{k-1} = \infty$). Then, a prefix sum is used to calculate the bucket boundaries. The distribution phase uses the oracle array and the bucket boundaries to copy the elements from the input array into their buckets in the temporary data array.

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

Bingmann et al. [BES17] apply the branchless decision tree to parallel string sample sorting (StringPS^{4}o) and add additional buckets for elements identical to a splitter. After the decision tree of S^{4}o has assigned element e to bucket b_i, StringPS^{4}o updates the bucket to introduce additional equality buckets for elements corresponding to a splitter: Element e goes to bucket

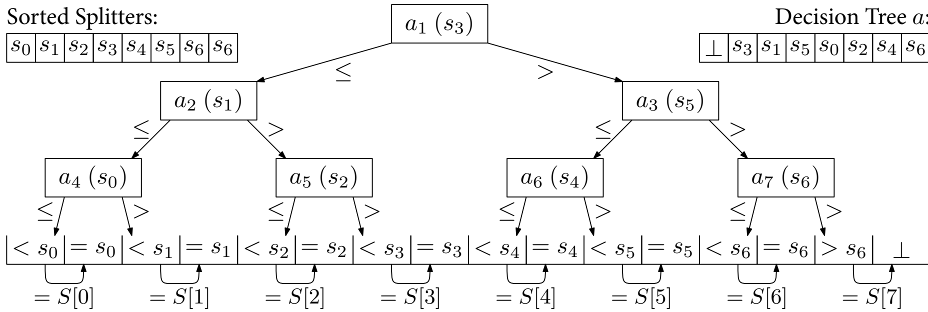


Figure 2.1: Branchless decision tree with 7 splitters and 15 buckets, including 7 equality buckets. The first entry of the decision tree array stores a dummy to allow tree navigation. The last splitter in the sorted splitter array is duplicated to avoid a case distinction.

$b_{2i+\mathbb{1}_{e=s_i}}$ if $i < k-1$, otherwise e goes to bucket b_{2i}.[1] The case distinction $i < k-1$ is necessary as a_{k-1} is undefined.

For IPS⁴o, we adopt (and refine) the approach of element classification but change the organization of buckets in order to make S⁴o in-place and parallel. Our element classification works as follows: Beginning by the source node $i = 1$ of the decision tree, the next node is calculated by $i \leftarrow 2i + \mathbb{1}_{a_i<e}$. When the leaves of the tree are reached, we update i once more $i \leftarrow 2i + \mathbb{1}_{a_i<e} - k$. For now, we know for e that $s_{i-1} < e \leq s_i$ if we assume that $s_{-1} = -\infty$ and that $s_{k-1} = \infty$. Finally, the bucket of e is $2i + 1 - \mathbb{1}_{e<s_i}$. Note that we do not use the comparison $\mathbb{1}_{e=s_i}$, from *StringPS⁴o* to calculate the final bucket. The reason is that our algorithm accepts a compare function < and StringPS⁴o compares radices. Instead, we use $1 - \mathbb{1}_{e<s_i}$ that is identical to $\mathbb{1}_{e=s_i}$ since we already know that $e \leq s_i$. Also, note that we avoid the case distinction $i < k-1$ from the classification of StringPS⁴o that may potentially cause a branch misprediction. Instead, we set $s_{k-1} = s_{k-2}$. Compared to S⁴o and StringPS⁴o, we support values of k that are no powers of two, i.e., when we had removed splitter duplicates in our algorithm. In these cases, we round up k to the next power of two and pad the splitter array S with the largest splitter. We note that this does not increase the depth of the decision tree. Figure 2.1 depicts our refined decision tree and Algorithm 1 classifies elements using the refined decision tree. Algorithm 1 classifies a chunk of elements in one step. A single instruction of the decision tree traversal is executed on multiple elements before the next operation is executed. We use loops to execute each instruction on a constant number of elements. It turned out that recent compilers automatically unroll these loops and remove the instructions of the loops for code optimization.

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

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

Template parameters: s number of splitters, u unroll factor, *equalBuckets* Boolean value that indicates the use of equality buckets
Input: $A[0\mathinner{..}n-1]$ an array of n input elements
$tree[1\mathinner{..}s]$ decision tree, splitters stored in left-to-right breadth-first order
$splitter[0\mathinner{..}s]$ sorted array of s splitters, last splitter is duplicated
COMPARE(e_l, e_r) a comparator function that returns 0 or 1
OUTPUT(e, t) an output function that gets an element e and its target bucket t

```
l ← log_2(s + 1)        ▷ Log. number of buckets (equality buckets are excluded)
k ← 2^(l+1)                                          ▷ Number of buckets
b[0 .. u − 1]             ▷ Array to store current position in the decision tree
for j ← 0 in steps of u to n − u do               ▷ Loop over elements in blocks of u
    for i ← 0 to u − 1 do
        b[i] ← 1                                      ▷ Set position to the tree root
    for r ← 0 to l do          ▷ Unrolled by most compilers as l and u are constants
        for i ← 0 to u − 1 do
            b[i] ← 2 · b[i]+ COMPARE(tree[b[i]], a[j + i])      ▷ Navigate through the tree
    if equalBuckets then
        for i ← 0 to u − 1 do▷ Assign elements identical to the splitter to its equality bucket
            b[i] ← 2 · b[i] + 1−COMPARE(a[j + i], splitter[b[i] − k/2])
    for i ← 0 to u − 1 do
        OUTPUT(b[i] − k, a[j + i])
```

2.2 Quicksort

Variants of Hoare's quicksort [Hoa62; Mus97] are generally considered some of the most efficient general-purpose sorting algorithms. Quicksort works by selecting a *pivot* element and partitioning the array such that all elements smaller than the pivot are in the left part and all elements larger than the pivot are in the right part. The subproblems are solved recursively. Quicksort (with recursion on the smaller subproblem first) needs logarithmic additional space for the recursion stack. Strictly in-place variants [Dur86; BK86; Weg87] of quicksort avoid recursion, process the array from left to right, and use a careful placement of the pivots to find the end of the leftmost partition. A variant of quicksort (with a fallback to heapsort to avoid worst-case scenarios) is currently used in the C++ standard library of GCC [Mus97].

Some variants of quicksort use two or three pivots [Yar10; Kus+14] and achieve improvements of around 20 % in running time over the single-pivot case. The basic principle of quicksort remains, but elements are partitioned into three or four subproblems instead of two.

Quicksort can be parallelized in a scalable way by parallelizing both partitioning and recursion [MG89; HNR90; FP92]. Tsigas and Zhang [TZ03] show in practice how to do this in-place. Their algorithm scans the input from left to right and from right to left until the scanning positions meet—as in most sequential implementations. The crucial adaptation is to do this in a blockwise fashion such that each thread works at one block from each scanning direction

at a time. When a thread finishes a block from one scanning direction, it acquires a new one using an atomic fetch-and-add operation on a shared pointer. This process terminates when all blocks are acquired. The remaining unfinished blocks are resolved in a sequential cleanup phase. Our IPS^{4}o algorithm can be considered as a generalization of this approach to k pivots. This saves a factor $\Theta(\log k)$ of passes through the data. We also parallelize the cleanup process.

2.3 Samplesort

Samplesort [FM70; Ble+96; BGS10] can be considered as a generalization of quicksort that uses $k - 1$ splitters to partition the input into k subproblems (from now on called *buckets*) of about equal size. Unlike single- and dual-pivot quicksort, samplesort is usually not in-place, but it is well-suited for parallelization and more cache-efficient than quicksort.

The algorithm S^{4}o [SW04] improves samplesort by avoiding inherently hard-to-predict conditional branches linked to element comparisons. Branch mispredictions are very expensive because they disrupt the pipelined and instruction-parallel operation of modern processors. Traditional quicksort variants suffer massively from branch mispredictions [KS06a]. By replacing conditional branches with conditionally executed machine instructions, branch mispredictions can be largely avoided. This is done automatically by modern compilers if only a few instructions depend on a condition. As a result, S^{4}o is up to two times as fast as quicksort (`std::sort`), at the cost of $\mathcal{O}(n)$ additional space. BlockQuicksort [EW16] applies similar ideas to single-pivot quicksort, resulting in a very fast in-place sorting algorithm with performance similar to S^{4}o.

For IPS^{4}o, we used a refined version of the branchless decision tree from S^{4}o. S^{4}o has also been adapted for efficient parallel string sorting [BES17]. We apply their approach of handling identical keys to our decision tree.

2.4 Radix Sort

As for samplesort, the core of radix sort is a k-way data partitioning routine that is recursively executed. In its simplest way, all elements are classified once to determine the bucket sizes and then a second time to distribute the elements. Most partitioning routines are applicable to samplesort as well as to radix sort. Samplesort classifies an element with $\Theta(\log k)$ invocations of the comparator function while radix sort just extracts a digit of the key in constant time. In-place k-way data partitioning is often done element by element, e.g., in the sequential in-place radix sorters American Flag [MBM93] and SkaSort [Ska16a]. However, these approaches have two drawbacks. First, they perform the element classification twice. This is a particular problem when we apply this approach to samplesort as the comparator function is more expensive. Second, a naive parallelization where the threads use the same pointers and acquire single elements suffers from read/write dependencies.

In 2014, Orestis and Ross [PR14] outlined a parallel in-place radix sorter that moves blocks of elements in its k-way data partitioning routine. We use the same general approach for IPS^{4}o. However, the paper [PR14] leaves open how the basic idea can be turned into a correct in-place

algorithm. The published prototypical implementation uses 20 % additional memory, and does not work for small inputs or a number of threads different from 64.

In 2015, Minsik et al. published PARADIS [Cho+15], a parallel in-place radix sorter. The partitioning routine of PARADIS classifies the elements to get bucket boundaries and each thread gets a subsequence of unpartitioned elements from each bucket. The threads then try to move the elements within their subsequences so that the elements are placed in the subsequence of their target bucket. This takes time $\mathcal{O}(n/t)$. Depending on the data distribution, elements may still be in the wrong bucket. In this case, the threads repeat the procedure on the unpartitioned elements. Depending on the key distribution, the load of the threads in the partitioning routine differs significantly. No bound better than $\mathcal{O}(n)$ is known for this partitioning routine [Obe+19b].

In 2019, Shun et al. [Obe+19b] proposed an in-place k-way data partitioning routine for the radix sorter RegionSort. This algorithm builds a graph that models the relationships between element regions and their target buckets. Then, the algorithm performs multiple rounds where the threads swap regions into their buckets.

To the best of our knowledge, the initial version of IPS^{4}o [Axt+17c], published in 2017, is the first parallel k-way partitioning algorithm that moves elements in blocks, works fully in-place, and gives adequate performance guarantees. Our algorithm IPS^{4}o is more general than RegionSort in the sense that it is comparison-based. To demonstrate the advantages of our approach, we also propose the radix sorter IPS^{2}Ra that adapts our in-place partitioning routine.

2.5 (Strictly) In-Place Mergesort

There is a considerable amount of theory work on strictly in-place sorting (e.g., [Fra04; FG05; GL91]). However, there are few—mostly negative—results of transferring the theory work into practice. Implementations of non-stable in-place mergesort [KPT96; EKS12; EW19] are reported to be slower than quicksort from the C++ standard library. Katajainen and Teuhola report that their implementation [KPT96] is even slower than heapsort, which is quite slow for big inputs due to its cache-inefficiency. The fastest non-stable in-place mergesort implementation we have found is QuickMergesort (QMSort) from Edelkamp et al. [EW19]. Relevant implementations of stable in-place mergesort are WikiSort (derived from [KK08]) and GrailSort (derived from [HL92]). However, Edelkamp et al. [EW19] report that WikiSort is a factor of more than 1.5 slower than QMSort for large inputs and that GrailSort performs similar to WikiSort. Edelkamp et al. also state that non-in-place mergesort is considerably faster than in-place mergesort. There is previous theoretical work on sequential (strictly) in-place multiway merging [GG10]. However, this approach needs to allocate very large blocks to become efficient. In contrast, the block size of IPS^{4}o does not depend on the input size. Gu et al. [GOS21] propose a method to convert parallel non-in-place algorithms meeting the *Decomposable Property* into algorithms that only require $\mathcal{O}(n^{1-\epsilon})$ additional memory for $0 < \epsilon < 1$. They apply their method to merging, scan, filter, random permutation, list contraction, and tree contraction and provide implementations for the latter five algorithms. The best practical shared-memory mergesort algorithm we found is the non-in-place multiway mergesort algo-

rithm (MCSTLmwm) from the MCSTL library [SSP07]. We did not find any practical parallel in-place mergesort implementation.

Chapter 3

3 Robust Scalable In-Place Sorting Algorithms

In Sections 3.1 and 3.2, we propose the comparison-based sorting algorithm IPS^{4}o as well as the integer sorting algorithm IPS^{2}Ra. Both algorithms work in-place and can be executed sequentially as well as in parallel. In Section 3.3, we prove that the algorithms work in-place (Section 3.3.1). Additionally, we prove that IPS^{4}o is cache-efficient (Section 3.3.2) and that it performs $\mathcal{O}(n/t \log n)$ work per thread if some constraints apply in regard to the input size, the number of threads, and the machine parameters (Section 3.3.3).

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

IPS^{4}o is a recursive algorithm. Each recursion level divides the input into k buckets (*partitioning step*), such that each element of bucket b_i is smaller than all elements of b_{i+1}. Partitioning steps operate on the input array in-place and are executed with one or more PEs, depending on their size. If a bucket is smaller than a certain base case size, we invoke a base case algorithm on the bucket (*base case*) to sort small inputs fast. A scheduling algorithm determines at which time a base case or partitioning step is executed and which PEs are involved. We describe the partitioning steps in Section 3.1.1 and the scheduling algorithm in Section 3.1.2.

3.1.1 Sequential and Parallel Partitioning

A partitioning step consists of four phases, executed sequentially or by a (sub)set of the input PEs. **Sampling** determines the bucket boundaries. **Classification** groups the input into blocks such that all elements in a block belong to the same bucket. **(Block) permutation** brings the blocks into the globally correct order. Finally, we clean up blocks that cross bucket boundaries or remained partially filled in the **cleanup** phase. Figure 3.1 depicts an overview of a parallel partitioning step. The following paragraphs will explain each of these phases in more detail.

3.1.1.a) Sampling.

Similar to the sampling in S^{4}o, the sampling phase of IPS^{4}o creates a branchless decision tree—the tree follows the description of the decision tree proposed by Sanders and Winkel, extended by equality buckets[1]. For a description of the decision tree used in S^{4}o including our

[1] The authors describe a similar technique for handling duplicates, but have not implemented the approach for their experiments.

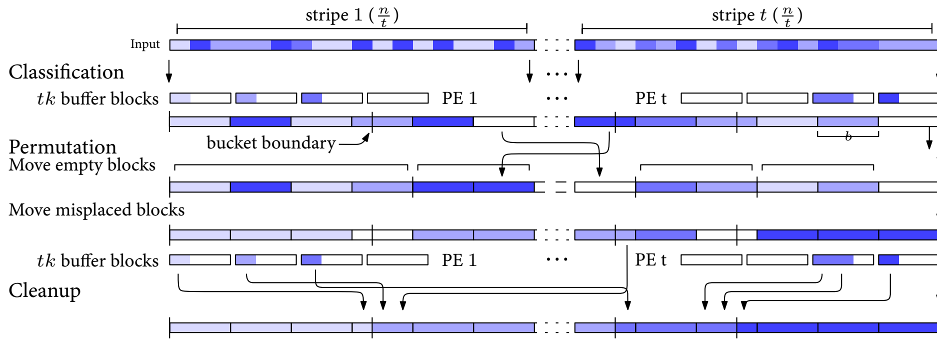


Figure 3.1: Overview of a parallel k-way partitioning step ($k = 4$) with t PEs and blocks of size three. Elements with the same color belong into the same bucket. The brighter the color, the smaller the bucket index. This figure depicts the first and last stripe of the input array, containing n/t elements each. In the classification phase, PE i classifies the elements of stripe i, moves elements of bucket j into its buffer block j, and flushes the buffer block back into its stripe in case of an overflow. In the permutation phase, the bucket boundaries are calculated and the blocks belonging into bucket j are placed in the blocks after bucket boundary j in two steps: First, the empty blocks are moved to the end of the bucket. Then, the misplaced blocks are moved into its bucket. The cleanup phase moves elements that remained in the buffer blocks and elements that overlap into the next bucket to their final positions.

refinements, we refer to Section 2.1. In IPS^{4}o, the decision tree is used in the classification phase to assign elements to buckets.

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

3.1.1.b) Classification

The input array A is viewed as an array of blocks each containing b elements (except possibly for the last one). For parallel processing, we divide the blocks of A into t stripes of equal size—one

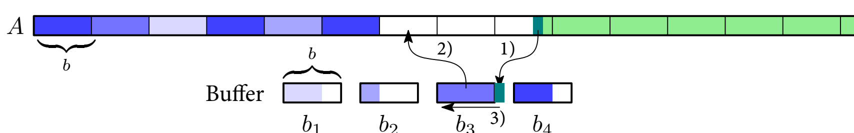


Figure 3.2: Classification. Blue elements have already been classified, with different shades indicating different buckets. Unprocessed elements are green. We perform three steps in this example. 1) The next element (in dark green) is determined to belong to bucket b_3. 2) As that buffer block is already full, we first have to write it into the array A. 3) Then, we can write the new element into the now empty buffer.

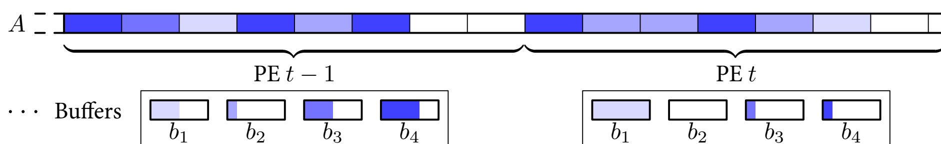


Figure 3.3: Input array and block buffers of the last two PEs after classification.

for each PE. Each PE works with a local array of k *buffer blocks*—one for each bucket. A PE then scans its stripe. Using the search tree created in the sampling phase, each element in the stripe is classified into one of the k buckets and then moved into the corresponding local buffer block. If this buffer block is already full, it is first written back into the local stripe, starting at the front. It is clear that there is enough space to write b elements into the local stripe, since at least b more elements have been scanned from the stripe than have been written back—otherwise, no full buffer could exist.

In this way, each PE creates blocks of b elements belonging to the same bucket. Figure 3.2 shows a typical situation during this phase. To achieve the in-place property, we do not track which bucket each block belongs to. However, we count how many elements are classified into each bucket, since we need this information in the following phases. This information can be obtained almost for free as a side effect of maintaining the buffer blocks. Figure 3.3 depicts the input array after classification. Each stripe contains full blocks, followed by empty blocks. The remaining elements are still contained in the buffer blocks.

3.1.1.c) Block Permutation

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

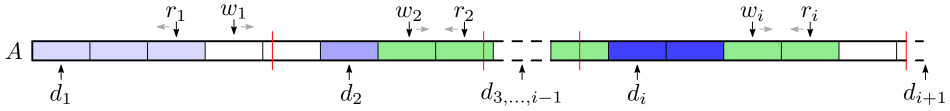


Figure 3.4: Invariant during block permutation. In each bucket b_i, blocks in $[d_i, w_i)$ are already correct (blue), blocks in $[w_i, r_i]$ are unprocessed (green), and blocks in $[\max(w_i, r_i + 1), d_{i+1})$ are empty (white).

maintaining atomic read and write pointers for each bucket. Costs for updating these pointers are amortized by making blocks sufficiently large.

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

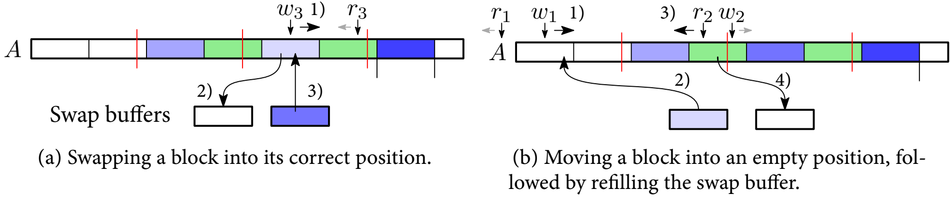
For each b_i, a write pointer w_i and a read pointer r_i are introduced; these will be set such that all unprocessed blocks, i.e., blocks that still need to be moved into the correct bucket, are found between w_i and r_i. At the beginning of the permutation phase, w_i is set to d_i and r_i is set to d_{i+1}. During the block permutation, we maintain the following invariant for each bucket b_i, visualized in Figure 3.4:

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

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

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

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



(a) Swapping a block into its correct position.

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

Figure 3.5: Block permutation examples. The numbers indicate the order of the operations.

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

Once it has a block, each PE classifies the first element of that block to find its destination bucket b_{dest}. There are now two possible cases, visualized in Figure 3.5:

- As long as $w_{\text{dest}} \leq r_{\text{dest}}$, write pointer w_{dest} still points to an unprocessed block in bucket b_{dest}. In this case, the PE increases w_{dest}, reads the unprocessed block into its empty swap buffer, and writes the other one into its place.

- If $w_{\text{dest}} > r_{\text{dest}}$, no unprocessed block remains in bucket b_{dest} but w_{dest} now points to an empty block. In this case, the PE increases w_{dest}, writes its swap buffer to the empty block, and then reads a new unprocessed block from its primary bucket.

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

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

When a PE fetches a new unprocessed block, it reads and modifies either w_i or r_i. The PE also needs to read the other pointer for the case distinctions. These operations are performed simultaneously to ensure a consistent view of both pointers for all PEs. Figure 3.6 depicts an example of the permutation phase with three PEs and four buckets.

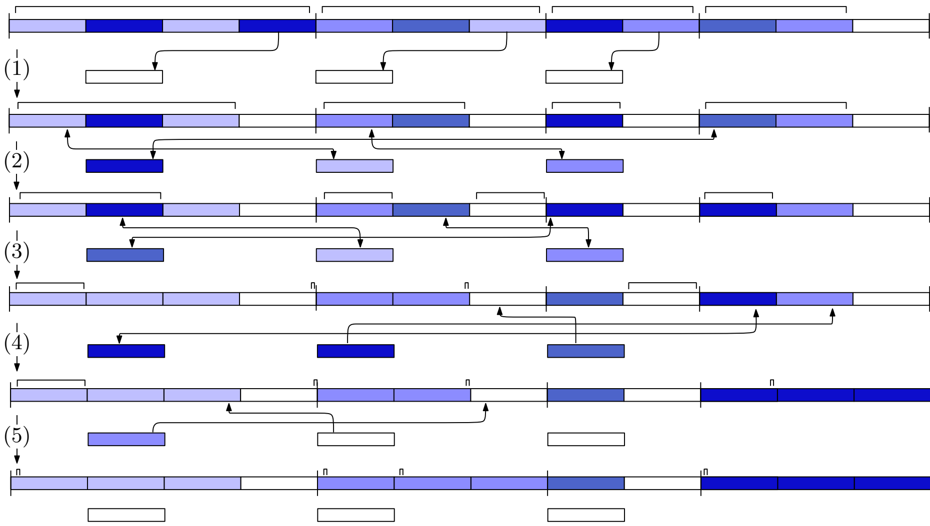


Figure 3.6: An example of a permutation phase with $k = 4$ buckets and $t = 3$ PEs. The brackets above the buckets mark the unprocessed blocks. After five permutation steps, the blocks were moved into their target buckets. (1) The buffer blocks are filled with blocks. (2-3) Swap buffer block with the leftmost unprocessed block of the buffer block's buckets. (4) PE 0 and 1 have a buffer block for the last bucket. They increase the write pointer of this bucket concurrently. PE 0 executes the fetch-and-add operation first, the PE swaps its buffer block with the second block (unprocessed block) of the last bucket. PE 1 writes its buffer block into the third block (empty block) of the last bucket. After step four, PEs 1 and 2 finished a permutation chain, i.e., flushed their buffer block into an empty block. (5) PE 0 flushes its buffer block into an empty block. PE 1 classifies the last unprocessed block of the first bucket but this block is already in its target bucket.

3.1.1.d) Cleanup

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

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

(i) Elements may be in the head of b_{i+1} if the last block belonging into bucket b_i overlaps into bucket b_{i+1}.

(ii) Elements may be in the partially filled buffers from the classification phase.

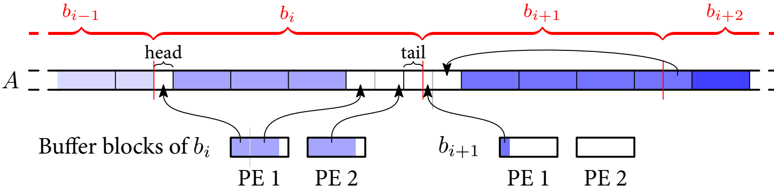


Figure 3.7: An example of the steps performed during cleanup.

(iii) Elements of the last bucket (in this PE's area) may be in the swap buffer.

(iv) Elements of one bucket may be in the overflow buffer.

Empty array entries consist of the head of b_i and any (empty) blocks to the right of w_i (inclusive). Although the concept is relatively straightforward, the implementation is somewhat involved, due to the many parts that have to be brought together. Figure 3.7 shows an example of the steps performed during the cleanup phase. Afterwards, all elements are back in the input array and correctly partitioned, ready for recursion.

3.1.2 Task Scheduling

In this section, we describe the task scheduling of IPS^{4}o. We also establish basic properties of the task scheduler. The properties are used to understand how the task scheduler works. The properties are also used later on in Section 3.3 to analyze the parallel I/O complexity and the local work of IPS^{4}o.

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

Before we describe the scheduling algorithm, we introduce some definitions and derive some properties from these definitions to understand how the task scheduler works. A task $T[l,r)$ either partitions the (sub)array $A[l,r-1]$ with a partitioning step (*partitioning task*) or sorts the base case $A[l,r-1]$ with a base case sorting algorithm (*base case task*). A partitioning step performed by a group of PEs (a *PE group*) is a *parallel partitioning task* (parallel task) and a partitioning step with one PE is a *sequential partitioning task*. Each PE has a *local stack* to store *sequential tasks*, i.e., sequential partitioning tasks and base cases. Additionally, each PE i stores a handle G_i to its current PE-group and has access to the handles stored by the other PEs of the PE-group.

The initial task is executed by all PEs. We now consider a subtask $T[l,r)$ of a parallel task that had been processed by some PEs $[\underline{t}\,..\,\overline{t})$. To decide whether $T[l,r)$ is a parallel task or a sequential task, we denote $\underline{t}'$ as $\lfloor lt/n \rfloor$ and $\overline{t}'$ as $\lfloor rt/n \rfloor$. $T[l,r)$ is a parallel task when $\overline{t}' - \underline{t}' > 1$.

In this case, the task is executed by PE group $[\underline{t}' .. \overline{t}')$. Otherwise, the task is a sequential task and will be processed by PE $\lfloor \min(\underline{t}', \overline{t} - 1) \rfloor$.

We use a base case threshold n_0 to determine whether a sequential task is a sequential partitioning task or a base case task: Buckets with at most $2n_0$ elements as well as buckets of a task with at most kn_0 elements are considered as base cases. Otherwise, it is a sequential partitioning task. We use an adaptive number of buckets k for partitioning steps with less than $k^2 n_0$ elements, such that the expected size of the base cases is between $0.5n_0$ and n_0 while the expected number of buckets remains equal or larger than $\sqrt{k}$. To this end, for partitioning steps with $kn_0 < n' < k^2 n_0$ elements, we adjust k to $2^{\lceil (\log_2(n'/n_0)+1)/2 \rceil}$. When $n_0 < n' \le kn_0$, we set $k = 2^{\lceil \log_2(n'/n_0) \rceil}$. This adaption of k is important for a robust running time of IPS^{4}o. In the analysis of IPS^{4}o, the adaption of k will allow us to amortize the sorting of samples. In practice, our experiments have shown that for fixed k, the running time per element oscillates with maxima around $k^i n_0$.

From these definitions, Lemmas 3.1–3.4 follow. The lemmas allow a simple scheduling of parallel tasks and PE-groups.

Lemma 3.1 (Relationship between array position and PE index)

The parallel task $T[l, r)$ covers position $(i+1)n/t - 1, i \in [0 .. t)$ of the input array if and only if PE i executes the parallel task $T[l, r)$.

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

$$i = \lfloor ((i+1)n/t - 1)t/n \rfloor \ge \lfloor l_s t/n \rfloor = \underline{t}_s$$
$$i = \lfloor ((i+1)n/t - 1)t/n \rfloor < \lfloor r_s t/n \rfloor = \overline{t}_s$$

follows that PE i executes the parallel task $T[l, r)$. For the "≥" and the "<", we use that task $T[l, r)$ covers position w of the input array, i.e., $l \le w < r$.

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

$$(i+1)n/t - 1 \ge (\underline{t} + 1)n/t - 1 \ge l_s$$
$$(i+1)n/t - 1 < \overline{t}n/t \le r_s$$

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

Lemma 3.2 (Relationship between PE of sequential task and parallel parent task)

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

Proof. Let the parallel task $T[l, r)$ be processed by PEs $[\underline{t} .. \overline{t})$. Task $T[l_s, r_s)$ is processed by PE $i = \min(\lfloor l_s t/n \rfloor, \overline{t} - 1)$. We have to show that $\underline{t} \le i < \overline{t}$ holds. Indeed, inequality $i =$

$\min(\lfloor l_s t/n \rfloor, \bar{t}-1) < \bar{t}$ holds. The inequality $i \geq \underline{t}$ is only wrong if $\lfloor l_s t/n \rfloor < \underline{t}$ or if $\bar{t}-1 < \underline{t}$. However, we have $l \leq l_s$ as task $T[l_s, r_s)$ is a bucket of its parent task $T[l,r)$. Thus, $\lfloor l_s t/n \rfloor \geq \lfloor lt/n \rfloor = \underline{t}$ holds as the first PE $\underline{t}$ of $T[l,r)$ is defined as $\lfloor lt/n \rfloor$. Also, we have $\bar{t}-1 > \underline{t}$ as task $T[l,r)$ is a parallel task with at least two PEs, i.e., $\bar{t}-\underline{t} > 1$. □

Lemma 3.3 (Parallel subtasks are processed by subgroup of PEs)

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

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

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

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

As the parallel task $T[l_s, r_s)$ is processed by PE i, $T[l_s, r_s)$ covers position $(i+1)n/t-1$ of the input array (see Lemma 3.1). As task $T[l_s, r_s)$ is a bucket of $T[l,r)$, the parallel task $T[l,r)$ also covers the position $(i+1)n/t-1$. From Lemma 3.1 follows that the parallel task $T[l,r)$ is processed by PE i. □

Lemma 3.4 (One parallel task per recursion level and PE)

On each recursion level, PE i works on at most one parallel task.

Proof. Let $S_i^j, i \in [0\,..\,t)$ be the set of parallel tasks on recursion level j that cover the position $(i+1)n/t-1$ of the input array. From Lemma 3.1 follows that PE i processes on recursion level j only the tasks S_i^j. The set S_i^j contains at most one task as tasks on the same recursion level are disjoint. □

3.1.2.a) Static Scheduling

We start the description of the task scheduler by describing the static scheduling part. The idea behind the static scheduling is that each PE executes its tasks in depth-first search order tracing parallel tasks first. From Lemmas 3.3 and 3.4 follows, keeping the order of execution in mind, that each PE first executes all of its parallel tasks before it starts to execute its sequential tasks.

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

(i) A parallel partitioning step is invoked on $A[l, r-1]$.

(ii) The buckets of the partitioning step induce a set of subtasks S.

(iii) If subtask $T[l_s, r_s) \in S$ is a sequential task, PE $i = \lfloor \min(l_s t/n, \bar{t}-1) \rfloor$ adds the subtask to its local stack. From Lemma 3.2, we know that PE i is actually also processing the current task $T[l,r)$. This allows PEs to add sequential tasks exclusively to their own local stack, so no concurrent stacks are required.

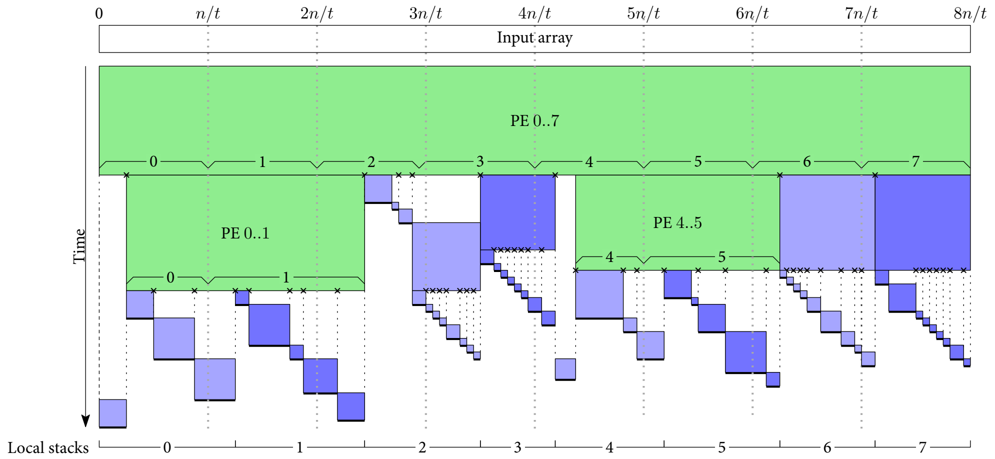


Figure 3.8: Example schedule of a task execution in IPS^{4}o with 8 PEs where partitioning steps split tasks into 8 buckets. Each rectangle represents a task in execution. The height of a task is defined by the size of the task divided by the number of PEs assigned to this task. For parallel tasks (green), the PEs processing that task are shown in the rectangles. The sequential partitioning tasks (blue) are covered by the local stack that stores the task until processing. Base case tasks are omitted for the sake of simplicity. The crosses at the bottom of a rectangle indicate bucket boundaries. The brackets pointing downwards are used to decide in which local stack the sequential subtasks are inserted. Tasks stored in local stack i are executed by PE i.

(iv) Each PE $i \in [\underline{t} .. \overline{t})$ extracts the subtask $T_s = T[l_s, r_s)$ from S that covers position $\lfloor (i+1)n/t \rfloor - 1$ of the input array A. Also, PE i calculates $\underline{t}_s = \lfloor l_s t/n \rfloor$ as well as $\overline{t}_s = \lfloor r_s t/n \rfloor$ and continues with the case distinction $\overline{t}_s - \underline{t}_s \leq 1$ and $\overline{t}_s - \underline{t}_s > 1$.

If $\overline{t}_s - \underline{t}_s \leq 1$ or if T_s is an equality bucket, PE i once synchronizes with G and starts processing the sequential tasks on its local stack.

Otherwise, T_s is actually a parallel task that has to be processed by the PEs $[\underline{t}_s .. \overline{t}_s)$. From Lemmas 3.1 and 3.3 follows that the PEs $[\underline{t}_s .. \overline{t}_s)$ are currently all processing $T[l, s)$ and exactly these PEs selected the same task T_s. This allows setting up the PEs $[\underline{t}_s .. \overline{t}_s)$ for the next parallel task T_s without keeping the PEs waiting: The first PE $\underline{t}_s$ of task T_s creates the data structure representing the task's new PE-group $G' = [\underline{t}_s .. \overline{t}_s)$ and updates the PE-group handles $[G_{\underline{t}_s} .. G_{\overline{t}_s})$ of the PEs $[\underline{t}_s .. \overline{t}_s)$ to the new data structure. Afterwards, all PEs of $[\underline{t}_s .. \overline{t}_s)$ synchronize with PE-group G and access their new PE-group G' using the updated PE-group handles. Finally, the PEs $[\underline{t}_s .. \overline{t}_s)$ start processing task T_s with PE-group G'.

If a PE no longer processes another parallel task, it starts processing the sequential tasks of its stack until the stack is empty. Base cases are sorted right away. When the next task $T[l, r)$ is

Algorithm 2 Task Scheduler

Input: $A[0\,..\,n-1]$ array of n input elements, t number of PEs, i current PE
$T[l,r) \leftarrow T[0,n)$ ▷ Current task, initialized with $A[1\,..\,n]$
$G_i[\underline{t},\overline{t}) = G[0,t)$ ▷ Initialize PE-group containing PE $\underline{t}=0$ to $\overline{t}=t$ (excl.)
$D \leftarrow \varnothing$ ▷ Empty local stack
if $\overline{t}-\underline{t}=1$ **then** D.pushFront($T[l,r)$) ▷ Initial task is a sequential, go to sequential phase
else
  while true do ▷ Execute current parallel task
    $[b_0\,..\,b_{k-1}] \leftarrow$ partitionParallel($A[l,r-1],G_i$) ▷ Partitioning step; returns buckets
    for $[l_s\,..\,r_s) \leftarrow b_{k-1}$ **to** b_0 **do** ▷ Handle the buckets
      if $(i+1)n/t-1 \in [l_s,r_s)$ **then** ▷ Update current task
        $T[l,r) \leftarrow T[l_s,r_s)$ ▷ It might be i's next parallel task
      if $\lceil r_s t/n \rceil - \lceil l_s t/n \rceil \leq 1$ **and** $i=\min(l_s t/n, \overline{t}-1)$ **then**
        D.push($\{T[l_s,r_s),l,r\}$) ▷ PE i adds sequential task to its local stack
    $\underline{t} \leftarrow l \cdot t/n$; $\overline{t} \leftarrow r \cdot t/n$ ▷ Range of PEs used by current task
    if $\overline{t}-\underline{t} \leq 1$ **then break** ▷ Go to sequential phase as $T[l,r)$ is not a parallel task
    if $i=\underline{t}$ **then** ▷ PE i creates the PE subgroup as it is the first PE
      $G_i \leftarrow$ createPEGroup($[\underline{t}\,..\,\overline{t}]$)
      for $j \leftarrow \underline{t}$ **to** $\overline{t}-1$ **do** ▷ Set subgroup for all subgroup PEs
        $G_j \leftarrow$ ReferenceOf(G_i)
    WaitFor($\underline{t}$) ▷ Wait until PE subgroup is created
    JoinPEGroup(G_i) ▷ Join shared data structures
while notEmpty(D) **do** ▷ Execute sequential tasks
  $\{T[l_s,r_s),l,r\} \leftarrow$ pop(D)
  if $r_s-l_s \leq 2n_0$ **or** $r-l \leq kn_0$ **then**
    processBaseCase($A[l_s,r_s-1]$)
  else
    $[b_0\,..\,b_{k-1}] \leftarrow$ partitionSequential($A[l_s,r_s-1]$) ▷ Partitioning step—returns buckets
    for $b \leftarrow b_{k-1}$ **to** b_0 **do**
      D.push($\{T[$begin(b), end(b)$),l_s,r_s\}$) ▷ Add seq. subtasks

a sequential partitioning task, three steps are performed. First, a sequential partitioning step is executed on $A[l,r-1]$. Second, a new sequential subtask is created for each bucket. Finally, the PE adds these subtasks to its local stack in sorted order. Algorithm 2 shows the steps of the task scheduling algorithm in detail. The scheduling algorithm is executed by all PEs simultaneously. Figure 3.8 shows an example schedule of a task execution in IPS⁴o.

From Lemmas 3.5 and 3.6 follows that the workload of sequential tasks and parallel tasks is evenly divided between the PEs. This property is used in Section 3.3 to analyze the parallel I/O complexity and the local work.

Lemma 3.5 (Relationship between subarray and PE-group)
Let $T[l, r)$ be a parallel task with PE-group $[\underline{t} .. \overline{t})$ and $t' = \overline{t} - \underline{t}$ PEs. Then, $T[l, r)$ processes a consecutive sequence of elements that starts at position $l \in [\underline{t}n/t .. (\underline{t}+1)n/t - 1]$ and ends at position $r \in [\overline{t}n/t - 1 .. (\overline{t}+1)n/t - 1]$ of the input array. This sums up to $\Theta(t'n/t)$ elements in total.

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

Proof of Lemma 3.5. From Lemma 3.1 follows that $T[l, r)$ covers position $(\underline{t}+1)n/t - 1$ but not position $\underline{t}n/t - 1$ of the input array. It also follows, that $T[l, r)$ covers position $\overline{t}n/t - 1$ but not position $(\overline{t}+1)n/t - 1$ of the input array. □

Lemma 3.6 (Relationship between PE and its sequential tasks)
PE i processes sequential tasks only containing elements from $A[in/t, (i+2)n/t - 1]$. This sums up to $\mathcal{O}(n/t)$ elements in total.

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

Proof of Lemma 3.6. We prove the following proposition: When a PE i starts processing sequential tasks, the tasks only contain elements from $A[in/t, (i+2)n/t - 1]$. From this proposition, Lemma 3.6 follows directly as PE i only processes sequential subtasks of these tasks.

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

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

$$\begin{aligned} i &= \min(\lfloor l_s t/n \rfloor, \overline{t} - 1) = \lfloor l_s t/n \rfloor \leq l_s t/n \\ \Longrightarrow l_s &\geq in/t \ . \end{aligned}$$

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

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

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

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

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

Assume $i \geq \overline{t} - 1$. In this case, i is essentially equal to $\overline{t} - 1$ as Lemma 3.2 tells us that a sequential subtask of a parallel task is assigned to a PE of the parallel task. From the calculation of PE i, we know that

$$\begin{aligned} i &= \min(\lfloor l_s t/n \rfloor, \overline{t} - 1) = \overline{t} - 1 \leq lt/n \\ \Longrightarrow l_s &\geq n/t(\overline{t} - 1) \ . \end{aligned}$$

The end r of the parent task can be bounded by

$$\begin{aligned} \bar{t} &= \lfloor rt/n \rfloor \geq rt/n - 1 \\ \Longrightarrow r &\leq (\bar{t} + 1)n/t \enspace . \end{aligned}$$

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

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

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

3.1.2.b) Dynamic Rescheduling

The task scheduler is extended to utilize computing resources of PEs that no longer have tasks. We implement a simplified version of *voluntary work sharing* proposed for parallel string sorting [BES17]. A global stack is used to transfer sequential tasks to idle PEs. PEs without sequential tasks increase a global atomic counter that tracks the number of idle PEs. PEs with sequential tasks check the counter after each partitioning step and move one task to the global stack if the counter is larger than zero. Then, an idle PE can consume the task from the global stack by decreasing the counter and processing the task. The algorithm terminates when the counter is equal to t which implies that no PE has a task left. We expect that we are able to amortize the additional cost in most cases or even reduce the work on the critical execution path: As long as no PE becomes idle, the counter remains valid in the PE's private cache and the PEs only access their local stacks. When a PE becomes idle, the local counter-copies of the other PEs are invalidated and the counter value is reloaded into their private cache. In most cases, we can amortize the counter reload by the previously processed task, as the task has typically more than $\Omega(kn_0)$ elements. When a PE adds an own task to the global stack, the task transfer is amortized by the workload reduction.

3.2 In-Place Parallel Super Scalar Radix Sort (IPS^{2}Ra)

We also use IPS^{4}o as an algorithmic framework to implement *In-place Parallel Super Scalar Radix Sort* (IPS^{2}Ra). For IPS^{2}Ra, we replaced the branchless decision tree of IPS^{4}o with a simple radix extractor function that accepts unsigned integer keys. For tasks with less than 2^{12} elements, we use SkaSort as a base case sorting algorithm. For small inputs ($n \leq 2^7$), SkaSort then falls back to quicksort that again uses insertion sort for $n \leq 2^5$. If we refer to IPS^{2}Ra in its sequential form, we use the term I1S^{2}Ra. IPS^{2}Ra only sorts data types with unsigned integer keys. The author of SkaSort [Ska16b; Ska16a] demonstrates that a radix sorter can be extended to sort inputs with floating-point keys and even compositions of primitive data types. We note that I1S^{2}Ra can be extended to sort these data types as well.

3.3 Analysis of IPS^{4}o

In this section, we analyze the additional memory requirement (Section 3.3.1), the I/O complexity (Section 3.3.2), and the local work (Section 3.3.3) of IPS^{4}o. For convenience, we use the term *local work* when we mean local work per PE. The analysis uses the asynchronous variant of the PEM model introduced in Section 1.3.3. Sections 3.3.2 and 3.3.3 assume the following constraints for IPS^{4}o (refer to Table 2.1 for a summary of the notation):

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

Assumption. Minimum number of elements per PE: $n/t \in \Omega(\max(M, bt))$.

Assumption. Restrict I/Os while sampling and buffers fit into private cache: $M \in \Omega(Bk \log k + bk)$ for some sufficiently constant of proportionality.

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

Assumption. Restrict maximum size of base cases: $n_0 \in \Omega(\log k) \cap \mathcal{O}(M/k)$.

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

3.3.1 Additional Memory Requirement

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

Theorem 3.7 (Memory usage of IPS^{4}o without local stack)

IPS^{4}o can be implemented with $\mathcal{O}(kb)$ additional memory per PE.

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

The classification phase stores elements only in the buffer blocks and the overflow buffer. As each PE reads its elements sequentially into its buffer blocks, there is always an empty block in the input array when a buffer block is flushed. When the size of the input array is not a multiple of the block size, a single overflow buffer may be required to store the overflow. The permutation phase only requires the swap buffers and the read and write pointers to move blocks into their target bucket. In the sampling phase, we do not need extra space as we swap the sample to the front of the input array. Nor do we need the local stacks (each of size $\mathcal{O}\big(k \log_k \frac{n}{tn_0}\big)$): Parallel tasks are processed immediately and we can use an implicit representation of the sequential tasks as described in Appendix A.2. □

Theorem 3.8 (Memory usage of IPS^{4}o with local stack)
With a local stack, IPS^{4}o can be implemented with $\mathcal{O}\big(k(b + t\log_k \frac{n}{n_0}\big)$ additional memory per PE.

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

3.3.2 Parallel Complexity and I/O Complexity

The PEM model measures the parallel complexity of an algorithm by counting the number of parallel I/O steps it performs. In particular, for external sorting algorithms, there are well-established lower bounds. For the PEM model this bound is $\Omega\big(\frac{n}{tB}\log_{M/B}\frac{n}{B}\big)$ [Arg+08]. We achieve this bound when the Assumptions 1–5 are fulfilled—essentially when the input is large enough. Since our main goal was to have both a reasonably simple algorithm and a reasonably simple analysis, the implied size bound is probably far from the best possible.[2]

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

Theorem 3.9 (I/O complexity of IPS^{4}o)
IPS^{4}o has an I/O complexity of $\mathcal{O}\big(\frac{n}{tB}\log_k\frac{n}{M}\big)$ memory block transfers with a probability of at least $1 - M/n$.

Before we prove Theorem 3.9, we prove that sequential partitioning steps exceeding the private cache are I/O-efficient (Lemma 3.10) and that parallel partitioning steps are I/O-efficient (Lemma 3.11).

Lemma 3.10 (I/Os of a sequential partitioning task)
A sequential partitioning task with $n' \in \Omega(M)$ elements transfers $\Theta(n'/B)$ memory blocks.

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

[2] Assumptions 1–5 imply $n = \Omega\big(t^3B\big)$

in the classification phase. We account $\mathcal{O}(1)$ I/Os for potential truncated memory blocks at the ends of the consecutive sequences. For k bucket boundaries, this sums up to $\mathcal{O}(k) \in \mathcal{O}(n'/B)$ as $n' \in \Omega(M) \in \Omega(Bk)$ (Assumptions 1 and 3). □

Lemma 3.11 (I/Os of a parallel task)

A parallel task with $\Theta(t'n/t)$ elements and t' PEs transfers $\mathcal{O}\left(\frac{n}{tB}\right)$ memory blocks per PE.

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

The same asymptotic cost occurs for moving blocks during the *permutation phase*. In each iteration of the main loop of the permutation phase, each PEs tries to acquire a block to be moved. If successful, it moves this block to its destination. We account $\mathcal{O}(b/B)$ I/O steps for a successful operation and $\mathcal{O}(t)$ I/O steps for an unsuccessful operation. Assumption 1 implies that $b/B = \Omega(t)$ so that both cases take care of the maximal possible contention for the fetch-and-add operations involved. Since overall there are only $\mathcal{O}(t'n/tb)$ blocks to be moved and since each PE will fail to acquire a block at most k times (once for each bucket), $\mathcal{O}(n/tb \cdot b/B + kt) = \mathcal{O}(n/tB + kt)$ I/O steps per PE suffice to finish the permutation phase. By assumptions 3, 2, and 1 we have $k = \mathcal{O}(M/b)$, $M = \mathcal{O}(n/t)$, and $b = \Omega(Bt)$, respectively. Thus, $kt = \mathcal{O}(Mt/b) = \mathcal{O}(n/b) = \mathcal{O}(n/tB)$ so that the number of I/O steps per PE simplifies to $\mathcal{O}(n/tB)$.

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

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

Proof of Theorem 3.9. In this proof, we can assume that IPS^{4}o performs $\mathcal{O}\left(\log_k \frac{n}{M}\right)$ recursion levels until the tasks have at most M elements. According to Theorem A.1, this assumption holds with a probability of at least $1 - M/n$. We do not consider the situation of many identical keys since the elements with these identical keys will not be processed at later recursion levels anymore.

From Theorem 3.7 we know that IPS^{4}o uses additional data structures that require $\mathcal{O}(kb)$ additional memory. In addition to the accesses to these data structures, task $T[l, r)$ only accesses $A[l\,..\,r-1]$. As $M \in \Omega(bk)$ (Assumption 3) we can keep the additional data structures in the

private cache. Thus, we only have to count the memory transfers of tasks from and to the input array.

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

Parallel tasks: IPS^{4}o processes the parallel tasks first. Parallel tasks transfer $\Theta\left(\frac{n}{tB}\right)$ memory blocks per PE (see Lemma 3.11). As a PE performs at most one parallel task on each recursion level (see Lemma 3.4), parallel tasks on the first $\mathcal{O}\left(\log_k\frac{n}{M}\right)$ recursion levels perform $\mathcal{O}\left(\frac{n}{tB}\log_k\frac{n}{M}\right)$ I/Os per PE. On subsequent recursion levels, no parallel tasks are executed: After $\mathcal{O}\left(\log_k\frac{n}{M}\right)$ recursion levels, the size of tasks is at most $\mathcal{O}(M)$. However, parallel tasks have $\Omega(M)$ elements. This follows from Lemma 3.5 and Assumption 2.

Large tasks (Sequential partitioning tasks not fitting into the main memory): A large task with n' elements takes $\Theta(n'/B)$ I/Os (see Lemma 3.10). A PE processes sequential tasks covering a continuous stripe of $\mathcal{O}(n/t)$ elements of the input array (see Lemma 3.6). Thus, the large tasks of a PE transfer $\mathcal{O}\left(\frac{n}{tB}\right)$ memory blocks on each recursion level. This sums up to $\mathcal{O}\left(\frac{n}{tB}\log_k\frac{n}{M}\right)$ I/Os per PE for the first $\mathcal{O}\left(\log_k\frac{n}{M}\right)$ recursion levels. After $\mathcal{O}\left(\log_k\frac{n}{M}\right)$ recursion levels, the size of tasks fits into the main memory, i.e., their size is $\mathcal{O}(M)$ for some sufficiently small constant of proportionality.

Middle tasks (Sequential tasks of size $\Omega(B)$ that fit into the main memory and that have a parallel or large task as parent task): In the first step of middle tasks, the classification phase, the PE reads the elements of the task from left to right. As the task fits into the private cache, the task does not perform additional I/Os after the classification phase. For a middle task of size n', we have $\mathcal{O}(\lfloor n'/B\rfloor)$ I/Os as $n'\in\Omega(B)$. Buckets of middle tasks are sequential subtasks that again fit into the main memory. Thus, each input element is only once part of a middle task and middle tasks cover disjoint parts of the input array. Additionally, we know from Lemma 3.6 that the sequential tasks of a PE contain $\mathcal{O}(n/t)$ elements. This implies that a PE transfers $\mathcal{O}\left(\frac{n}{tB}\right)$ memory blocks for middle tasks.

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

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

Comparing the I/O Volume of I1S^{4}o and S^{4}o. We analyze the constant factors of the I/O volume (i.e., data flow between cache and main memory) for the sequential algorithms I1S^{4}o (IPS^{4}o

	Subroutine	Types	Reps	Sum in n Bytes
S^4o	Copy back	r + w + wa	once	16 + 8
	Base Case	r + w	once	16
	Init Temp Array + Oracle	w	once	9
	Classification: Oracle	w + wa	per level	1 + 1
	Classification: Array	r	per level	8
	Redistribution: Oracle	r	per level	1
	Redistribution: Array	r + w + wa	per level	16+8
$I1S^4o$	Base Case	r + w	once	16
	Classification	r + w	per level	16
	Redistribution	r + w	per level	16

Table 3.1: I/O volume of read (r) and write (w) operations broken down into subroutines of $I1S^4o$ and S^4o. Additionally, potential write allocate operations (wa) are listed.

with $t = 1$) and S^4o. To simplify the discussion, we assume a single recursion level, $k = 256$, 8-byte elements, and an oracle with 1-byte entries for S^4o. Furthermore, we assume that the input does not fit into the private cache.

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

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

Additionally, S^4o writes an oracle sequence that indicates the bucket for each element in the classification phase and reads the oracle sequence in the distribution phase—$2n$ bytes per level. The algorithm also has to allocate the temporary arrays. For security reasons, that memory is zeroed by the operating system—$9n$ bytes.[3] If the number of levels is odd, S^4o has to copy the sorted result back to the input array—$16n$ bytes. For now, $I1S^4o$ (S^4o) has an I/O volume of $32n$ ($26n$) byte per level and $16n$ ($41n$) bytes once.

When S^4o writes to the temporary arrays or during copying back, cache misses happen when an element is written to a cache block that is currently not in memory. Depending on the cache replacement algorithm, a *write allocate* may be performed—the block is read from the memory to the cache even though none of the data in that block will ever be read. Detecting that the entire cache line will be overwritten is difficult as S^4o writes to the target buckets element by element. This amounts to an I/O volume of up to $9n$ bytes per level and $8n$ bytes once. $I1S^4o$ does not perform write allocates. The classification phase essentially sweeps a window of size $\Theta(bk)$ through the memory by reading elements from the right border of the window and

[3] In current versions of the Linux kernel this is done by a single PE and thus results in a huge scalability bottleneck in the parallel case.

writing elements to the left border. The permutation phase reads a block from the memory and replaces the "empty" memory bock with a cached block afterwards. Table 3.1 shows the I/O volume of the subroutines in detail. Overall, we get for I1S^{4}o (S^{4}o) a total I/O volume of $32n$ ($35n$) byte per level and $16n$ ($49n$) bytes once—S^{4}o with one level has a factor of 1.75 more I/O volume than I1S^{4}o. This is surprising since, at the first glance, the partitioning algorithm of I1S^{4}o writes the data twice, whereas S^{4}o does this only once. However, this is more than offset by "hidden" overheads of S^{4}o like memory management, memory page initialization, and write allocates.

Furthermore, S^{4}o may suffer more conflict misses than I1S^{4}o due to the mapping of data to cache lines. In the distribution phase, S^{4}o reads the input from left to right but writes element-wise to positions in the buckets which are not coordinated. For the same reasons, S^{4}o may suffer more TLB misses. I1S^{4}o, on the other hand, essentially writes elements to cached buffer blocks (classification) and swaps blocks of size b within the input array (block permutation). For an average case analysis on scanning multiple sequences, we refer to [MS03].

Much of this overhead can be reduced using measures that are non-portable (or hard to make portable). In particular, non-temporal writes eliminate the write allocates and also help to eliminate the conflict misses. One could also use a base case sorter that does the copying back as a side-effect when the number of recursion levels is odd. When sorting multiple times within an application, one can keep the temporary arrays without having to reallocate them. However, this may require a different interface to the sorter. Overall, depending on many implementation details, S^{4}o may require slightly or significantly more I/O volume.

3.3.3 Branch Mispredictions and Local Work

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

Our algorithm IPS^{4}o itself has virtually no branch mispredictions during element classification.[4] A branch misprediction only occurs when a bucket has to be flushed. This happens on average after b element classifications (after $b \log k$ element comparisons). The base case has about one misprediction per element for insertion sort. The permutation phase has a negligible amount of mispredictions since it basically moves whole blocks to their target position. The cleanup phase has mispredictions of the order of $\mathcal{O}(t+k)$ since the buckets are evenly distributed to the PEs and since each PE contributes one buffer block per bucket.

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

[4]Unless the comparator function causes branch mispredictions.

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

Theorem 3.12 (Local work of IPS^{4}o)

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

For the proof of Theorem 3.12, we need Lemmas 3.13–3.18. These lemmas use the term *small task* for tasks with at most kn_0 elements and the term *large task* for tasks with at least kn_0 elements.

Lemma 3.13 (Local work of a partitioning task)

A partitioning task with n' elements and t' PEs has a local work of $\mathcal{O}(n'/t' \log k)$ excluding the work for sorting the samples.

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

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

When the *cleanup phase* is executed sequentially, the elements of the local buffers are flushed into blocks that overlap into the next bucket. This may displace elements stored in these blocks. The displaced elements are written into empty parts of blocks. Thus, each element is moved at most once which sums up to $\mathcal{O}(n')$ work. For the cleanup phase of a parallel partitioning step, we conclude from the proof of IPS^{4}o's I/O complexity (see Theorem 3.9) that the local work is in $\mathcal{O}(n'/t')$: The proof of the I/O complexity shows that a parallel cleanup phase is bounded by $\mathcal{O}\left(\frac{n'}{t'B}\right)$ I/Os. Also, each element that is accessed in the cleanup phase is moved at most once and no additional work is performed. We account a local work of B for each memory block that a PE accesses. Thus, we can derive from $\mathcal{O}\left(\frac{n'}{t'B}\right)$ I/Os a local work of $\mathcal{O}(n'/t')$ for the parallel cleanup phase. □

Lemma 3.14 (Relationship between parallel tasks and small tasks)

At most one parallel task that is processed by a PE is a small task.

Proof. Assume that a PE processes at least two small parallel tasks p_1 and p_2. According to Lemma 3.4, a PE processes at most one of these tasks per recursion level. A parallel subtask of PE i is a subtask of a parallel task of PE i and represents a bucket of this task (see Lemma 3.3). Thus, p_1 and p_2 must be on different recursion levels, and p_1 processes a subset of elements

processed by p_2 or vice versa. However, this is a contradiction as buckets of small tasks are base cases. □

Lemma 3.15 (Local work of small partitioning tasks)

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

Proof. In this proof, we neglect the work for sorting the sample. Lemma 3.13 tells us that a partitioning task with n' elements and t' PEs has a local work of $\mathcal{O}(n'/t' \log k)$. Also, parallel tasks with t' PEs have $\Theta(t'n/t)$ elements (see Lemma 3.5). Thus, we have $\mathcal{O}(n/t \log k)$ local work for a small parallel task. Overall, we have $\mathcal{O}(n/t \log k)$ local work for small parallel tasks as each PE processes at most one of these tasks (see Lemma 3.14).

We now consider small sequential partitioning tasks. small sequential partitioning tasks that are processed by a single PE cover in total at most $\mathcal{O}(n/t)$ different elements (see Lemma 3.6). Each of these elements passes at most one of the small partitioning tasks since buckets of these tasks are base cases. Thus, a PE processes small partitioning tasks of size $\mathcal{O}(n/t)$ in total. As small partitioning tasks with n' elements require $\mathcal{O}(n' \log k)$ local work (see Lemma 3.13), the local work of all small partitioning tasks is $\mathcal{O}(n/t \log k)$. □

Lemma 3.16 (Local work of partitioning tasks)

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

Proof. Lemma 3.13 tells us that a partitioning task with n' elements and t' PEs has a local work of $\mathcal{O}(n'/t' \log k)$ excluding the work for sorting the samples. Parallel tasks with t' PEs have $\Theta(t'n/t)$ elements (see Lemma 3.5). Thus, we have $\mathcal{O}(n/t \log k)$ local work on a recursion level for parallel tasks. A PE processes sequential partitioning tasks covering a continuous stripe of $\mathcal{O}(n/t)$ elements of the input array (see Lemma 3.6). Thus, we also have $\mathcal{O}(n/t \log k)$ local work on a recursion level for sequential partitioning tasks. □

Lemma 3.17 (Local work of large partitioning tasks)

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

Proof. In this proof, we neglect the work for sorting the samples of a partitioning task. Large partitioning tasks create between $\sqrt{k}$ and k buckets (see Section 3.1.1.a)). According to Theorem A.1 in Appendix A.1, IPS^{4}o performs at most $\Theta\big(\log_{\sqrt{k}} \frac{n}{kn_0}\big) = \Theta\big(\log_k \frac{n}{kn_0}\big)$ recursion levels with a probability of at least $1 - kn_0/n$ until all partitioning tasks have less than kn_0 elements. However, this probability is not tight enough. Instead, we can perform up to $\mathcal{O}\big(\log_k n\big)$ recursion levels and still have $\mathcal{O}(n/t \log n)$ local work as each recursion level requires $\mathcal{O}(n/t \log k)$ local work (see 3.16). In this case, Theorem A.1 in Appendix A.1 tells us that all partitioning tasks have at most one element with a probability of $1 - n^{-1}$. This probability also holds if we stop partitioning buckets with less than kn_0 elements. □

Lemma 3.18 (Local work of base case tasks)
The local work of all base case tasks is in total $\mathcal{O}(n/t \log n)$ *with a probability of at least* $1 - n^{-1}$.

Proof. Sorting $\mathcal{O}(n)$ elements with the base case algorithm quicksort does not exceed $\mathcal{O}(\log n)$ recursion levels with probability $1 - n^{-1}$ [JáJ00]. Thus, an execution of quicksort with $\mathcal{O}(n/t)$ elements requires $\mathcal{O}(n/t \log n)$ local work with a probability of at least $1 - n^{-1}$ as it would also not exceed $\mathcal{O}(\log n)$ recursion levels with at least the same probability. Sorting all base cases of a PE is asymptotically at least as efficient as sorting $\mathcal{O}(n/t)$ elements at once: The base cases have in total at most $\mathcal{O}(n/t)$ elements (see Lemma 3.6) but the input is already prepartitioned. □

Lemma 3.19 (Local work for samples of small partitioning tasks)
The local work for sorting the samples of all small partitioning tasks is $\mathcal{O}(n/t \log n)$ *in total with a probability of at least* $1 - n^{-1}$

Proof. Small partitioning tasks with n' elements have $\mathcal{O}(n'/n_0)$ buckets and a sample of size $\mathcal{O}\big(n'/n_0 \log \frac{n'}{n_0}\big)$ (see Assumption 4 for the oversampling factor). The size of a sample is in particular bounded by $\mathcal{O}(n')$. For this, we use $n_0 \in \Omega(\log k)$ (Assumption 5) and $k \geq n'/n_0$ from which follows that $n_0 \in \Omega\big(\log \frac{n'}{n_0}\big)$ holds. Furthermore, small sequential partitioning tasks processed by a single PE cover in total at most $\mathcal{O}(n/t)$ different elements (see Lemma 3.6). Thus, the total size of all samples from small sequential partitioning tasks sorted by a PE is limited to $\mathcal{O}(n/t)$. We count one additional sample from a potential small parallel task (see Lemma 3.14). We can also limit its size to $\mathcal{O}(n/t)$ using Assumptions 2 and 5. We can prove that the work for sorting samples of a total size of $\mathcal{O}(n/t)$ is in $\mathcal{O}(n/t \log n)$ with a probability of at least $1 - n^{-1}$. We refer to the proof of Lemma 3.18 for details. □

Lemma 3.20 (Local work for samples of large partitioning tasks)
The local work for sorting the samples of all large partitioning tasks is $\mathcal{O}(n/t \log n)$ *in total with a probability of at least* $1 - n^{-1}$

Proof. A sequential large partitioning task with $\Omega(kn_0)$ elements has $\Omega(k \log k)$ elements (see Assumption 5) and a parallel large partitioning task has $\Omega(t'n/t)$ elements (see Lemma 3.5) with $n/t \in \Omega(k \log k)$ which is a result from Assumptions 2 and 3. Thus, a large partitioning task with t' PEs has $\Omega(t'k \log k)$ elements.

Each PE invokes at most $r = l \frac{n \log n}{tk \log^2 k}$ large partitioning tasks for a constant l – excluding the work for sorting the samples: On the one hand, Lemma 3.17 tells us that all partitioning tasks performed by a single PE sum up to $\mathcal{O}(n/t \log n)$ local work in total. On the other hand, this sum accounts $\mathcal{O}\big(k \log^2 k\big)$ local work or more for a large partitioning task since it accounts $\mathcal{O}(n'/t' \log k)$ local work for large partitioning tasks with n' elements and t' PEs (see Lemmas 3.13 and 3.16) and since a large partitioning task has $\Omega(t'k \log k)$ elements. Thus, when we execute large partitioning tasks, each PE performs at most r sample sorting routines – one for each task.

For the local work analysis, we consider a modified sample sorting algorithm. Instead of using quicksort, we use an adaption that restarts quicksort when it exceeds $\mathcal{O}\big(k \log^2 k\big)$ local work until the sample is finally sorted. The bounds for the local work that we obtain from

this variant also hold when IPS^{4}o executes quicksort until success instead of restarting the algorithm: A restart means to neglect the prepartitioned buckets which makes the problem unnecessarily difficult.

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

We observe that one execution of quicksort unsuccessfully sorts a sample with a probability of at most $p = k^{-3}\log_2^{-3} k$ as the size of the samples is bounded by $\mathcal{O}(k\log k)$. For this approximation, we use that sorting n elements with quicksort takes $\mathcal{O}(n\log n)$ work with high probability [JáJ00]. Each execution of quicksort is a Bernoulli trial as we have exactly two possible outcomes, "successful sorting in time" and "unsuccessful sorting in time", and the probability of failure is bounded by p each time. When we consider all quicksort invocations of IPS^{4}o, we need r successes. We define a binomial experiment that repeatedly invokes quicksort on the sample of the first large partitioning task until success and then continues with the second large partitioning task, until the sample of each partitioning step of a PE is sorted. Asymptotically, we can spend xr (re)starts of quicksort for any constant $x \geq 1$ such that the binomial experiment does not exceed $\mathcal{O}(n/t\log n)$ local work.

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

$$\begin{aligned}
I &= \mathrm{P}[X > (x-1)r] \\
&\leq \sum_{j>(x-1)r} \binom{xr}{j} p^j (1-p)^{xr-j} \leq \sum_{j>(x-1)r} \left(\frac{xre}{j}\right)^j p^j \\
&\leq \sum_{j>(x-1)r} \left(\frac{xre}{(x-1)r}\right)^j p^j = \sum_{j>(x-1)r} \left(\frac{pex}{x-1}\right)^j \\
&= \frac{\left(\frac{pex}{x-1}\right)^{(x-1)r+1}}{1-\frac{pex}{x-1}} \\
&\leq \left(\frac{pex}{x-1-pex}\right)\left(\frac{pex}{x-1}\right)^{\frac{(x-1)lk\log k\log n}{k\log^2 k}} \\
&\leq \left(\frac{pex}{x-1-pex}\right)(n)^{\frac{(x-1)l\log\left(\frac{pex}{x-1}\right)}{\log k}} \\
&\leq 2.13 n^{-\frac{1}{2}(x-1)} \leq 1/n
\end{aligned} \tag{3.1}$$

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

Proof of Theorem 3.12. According to Lemma 3.15, the small partitioning tasks excluding the work for sorting the samples require $\mathcal{O}(n/t \log k)$ local work. For large partitioning tasks, we have $\mathcal{O}(n/t \log n)$ local work with a probability of at least $1 - n^{-1}$ (see Lemma 3.17). The same holds for the base cases (see Lemma 3.18). Lemmas 3.19 and 3.20 bound the local work for sorting the samples of small and large partitioning tasks to $\mathcal{O}(n/t \log n)$, each with a probability of at least $1 - n^{-1}$. This sums up to a total local work of $\mathcal{O}(n/t \log n)$ with a probability of at least $1 - 4/n$. □

Chapter 4

Experiments and Conclusion

4

In this chapter, we present results from ten data distributions, generated for four different data types obtained on four different machines with one, two, and four processors and 21 different sorting algorithms. We extensively compare our in-place parallel sorting algorithms IPS^{4}o and IPS^{2}Ra as well as their sequential counterparts I1S^{4}o and I1S^{2}Ra to various competitors[1]:

- **Parallel in-place comparison-based sorting**
  - MCSTLbq (OpenMP): Two implementations (balanced and unbalanced) from the GCC STL library [SSP07] based on quicksort proposed by Tsigas and Zhang [TZ03].
  - TBB [Wen19] (TBB): Quicksort from the Intel® TBB library [Rei07].
- **Parallel non-in-place comparison-based sorting**
  - PBBS [Shu+12] (Cilk): $\sqrt{n}$-way samplesort [BGS10] implemented in the so-called problem based benchmark suite.
  - MCSTLmwm (OpenMP): Stable multiway mergesort from the GCC STL library.
  - PS^{4}o [Axt20a] (OpenMP or `std::thread`): Our parallel and stable implementation of S^{4}o from Section 4.1.
  - ASPaS [SyN18] (POSIX Threads): Stable mergesort that vectorizes the merge function with AVX2 [HWF15]. ASPaS only sorts `int`, `float`, and `double` inputs and uses the comparator function "<".
- **Parallel in-place radix sort**
  - RegionSort [Obe+19a] (Cilk): *Most Significant Digit* (MSD) radix sort [Obe+19b] that only sorts keys of unsigned integers. RegionSort skips the most significant bits that are zero for all keys.
  - IMSDradix [Pol14] (POSIX Threads): MSD radix sort [PR14] from Orestis and Ross with blockwise redistribution. The implementation, published by Orestis and Ross, however, requires 20 % of additional memory in addition to the input array and is very explorative.
- **Parallel non-in-place radix sort**
  - PBBR [Shu+12] (Cilk): A simple implementation of stable MSD radix sort from the so-called problem based benchmark suite.

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

  - RADULS2 [Gro17] (`std::thread`): MSD radix sort that uses non-temporal writes to avoid write allocate misses [KDD17]. RADULS2 requires 256-bit array alignment. Both keys and objects have to be aligned at 8-byte boundaries. We partially compiled the code with the flag `-O1` as recommended by the authors. The algorithm does not compile with the Clang compiler.

- **Sequential in-place comparison-based sorting**
  - BlockQ [Wei16a]: An implementation of BlockQuicksort [EW16] provided by the authors of the sorting algorithm publication.
  - BlockPDQ [Pet15]: Pattern-Defeating Quicksort that integrated the approach of BlockQuicksort in 2016. BlockPDQ has similar running times as BlockQuicksort using Lomuto's Partitioning [AH19], published in 2018.
  - DualPivot [Wei16b]: A C++ port of Yaroslavskiy's Dual-Pivot Quicksort [Yar10]. Yaroslavskiy's Dual-Pivot Quicksort is the default sorting routine for primitive data types in Oracle's Java runtime library since version 7.
  - std::sort: Introsort from the GCC STL library.
  - WikiSort [McF14]: An implementation of stable in-place mergesort [KK08].

- **Sequential non-in-place comparison-based sorting**
  - Timsort [GM14]: A C++ port of Timsort [Pet02]. Timsort is an implementation of stable mergesort that takes advantage of presorted sequences of the input. Timsort is part of Oracle's Java runtime library since version 7 to sort non-primitive data types.
  - S^4oS [Hüb16]: A recent implementation of non-in-place S^4o [SW04] optimized for modern hardware.
  - 1S^4o [Axt20a]: Our implementation of S^4o, which we describe in Section 4.1.

- **Sequential in-place radix sort**
  - SkaSort [Ska16b]: MSD radix sort [Ska16a] that accepts a key-extractor function returning primitive data types or pairs, tuples, vector, and arrays containing primitive data types. The latter ones are sorted lexicographically.

- **Sequential non-in-place radix sort**
  - IppRadix [Cor20]: Radix sort from the Intel® Integrated Performance Primitives library optimized with the AVX2 and AVX-512 instruction set.

We do not compare our algorithm to PARADIS as its source code is not publicly available. However, Omar et. al. [Obe+19b] compare RegionSort to the numbers reported in the publication of PARADIS and conclude that RegionSort is faster than PARADIS. Additionally, RegionSort and our algorithm have stronger theoretical guarantees (see Chapter 2).

Most radix sorters, i.e., IppRadix, IMSDradix, RADULS2, PBBR, and RegionSort, do not support all data types used in our experiments. Only the radix sorter SkaSort supports all data types as the types used here are either primitives or compositions of primitives, which are

Algorithm 3 Quartet comparison

```
function lessThan(l, r)
    if l.a ≠ r.a then
        return l.a < r.a
    else if l.b ≠ r.b then
        return l.b < r.b
    else return l.c < r.c
```

Algorithm 4 100B comparison

```
function lessThan(l, r)
    for i ← 0 to 9 do
        if l.k[i] ≠ r.k[i] then
            return l.k[i] < r.k[i]
    return False
```

sorted lexicographically. All algorithms are written in C++ and compiled with version 7.5.0 of the GNU compiler collection, using the optimization flags "`-march=native -O3`". We have not found a more recent compiler that supports Cilk threads, needed for RegionSort, PBBS, and PBBR.

Input Instances. We ran benchmarks with *double* (64-bit floating-points), *uint64* (64-bit unsigned integers), *uint32* (32-bit unsigned integers), and *Pair*, *Quartet*, and *100B* data types. Pair (Quartet) consists of one (three) 64-bit unsigned integers as key and one 64-bit unsigned integer of associated information. 100B consists of 10 bytes as key and 90 bytes of associated information. The keys of Quartet and 100B are compared lexicographically. Algorithms 3 and 4 show the lexicographical compare function that we used in our benchmarks. We want to point out that lexicographical comparisons can be implemented in different ways. We also tested `std::lexicographical_compare` for Quartet and `std::memcmp` for 100B. However, it turned out that these compare functions are (much) less efficient for all competitive algorithms. SkaSort is the only radix sorter that is able to sort keys lexicographically. For Quartet and 100B data types, we invoke SkaSort with a key-extractor function that returns the values of the key stored in a `std::tuple` object.

We ran benchmarks with ten different input distributions: *Uniform*, *Exponential*, and almost *AlmostSorted*, proposed by Shun et. al. [Shu+12]; *RootDup*, *TwoDup*, and *EightDup* from Edelkamp et. al. [EW16]; and *Zipf* (Zipf distributed input), *Sorted* (sorted Uniform input), *ReverseSorted*, and *Zero* (just zeros). The input distribution Exponential selects for each element a value $i \in [0..\lceil \log n \rceil]$ uniformly at random and uses the hash of another value selected uniformly at random from $[2^i..2^{i+1})$ as the key. The hash function has range $[0..2^w)$ for keys represented by w bits. RootDup sets $A[i] = i \mod \lfloor \sqrt{n} \rfloor$, TwoDup sets $A[i] = (i^2 + m/2) \mod m$, and EightDup sets $A[i] = (i^8 + m/2) \mod m$ with m being the minimum of n and the maximum finite value representable by the key type. The input distribution Zipf generates the integer number $k \in [1..10^6]$ with probability proportional to $1/k^{0.75}$. Figure 4.1 illustrates the nontrivial input distributions RootDup, Zipf, Exponential, TwoDup, EightDup, and AlmostSorted.

Machines. We ran our experiments on the following machines (all supporting 2 hardware threads per core):

- Machine *A1x16* with 32 GiB of memory and one AMD Ryzen 9 3950*X* 16-core CPU (512 KiB private L2-cache per core and 64 MiB shared L3-cache).

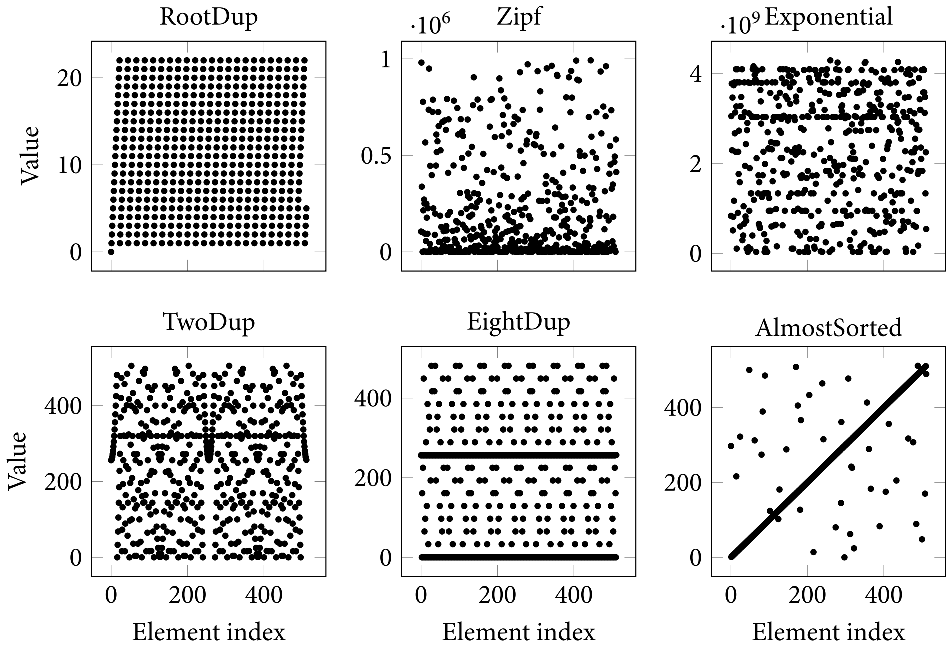


Figure 4.1: Examples of nontrivial input distributions for 512 uint32 values.

- Machine *A1x64* with 1024 GiB of memory and one AMD EPYC Rome 7702P 64-core CPU (512 KiB private L2-cache per core and 256 MiB shared L3-cache).
- Machine *I2x16* with 512 GiB of memory and two Intel Xeon E5-2683 v4 16-core CPUs (256 KiB private L2-cache per core and 40 MiB shared L3-cache per CPU).
- Machine *I4x20* with 768 GiB of memory and four Intel Xeon Gold 6138 20-core CPUs (1024 KiB private L2-cache per core and 27.5 MiB shared L3-cache per CPU).

Methodology. Each algorithm was executed on all machines with all input distributions and data types. The parallel (sequential) algorithms were executed for all input sizes with $n = 2^i$, $i \in \mathbb{N}^+$, until the input array exceeds 128 GiB (32 GiB). For $n < 2^{33}$ ($n < 2^{30}$), we perform each parallel (sequential) measurement 15 times and for $n \geq 2^{33}$ ($n \geq 2^{30}$), we perform each measurement twice. Unless stated otherwise, we report the average over all runs except the first one[2]. We note that non-in-place sorting algorithms which require an additional array of n elements will not be able to sort the largest inputs on *A1x16*, the machine with 32 GiB of memory. We also want to note that some algorithms did not support all data types because their interface rejects the key. In our figures, we emphasize algorithms that are “not general-purpose”

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

with red lines, i.e., because they assume integer keys (I1S^{2}Ra, IppRadix, RADULS2, RegionSort, and PBBR), make additional assumptions on the data type (RADULS2), or at least because they do not accept a comparator function (SkaSort). All non-in-place algorithms except RADULS2 return the sorted output in the input array. These algorithms copy the input back into the input array if the algorithm has executed an even number of recursion levels. Only RADULS2 returns the output in a second “temporary” array. To be fair, we copy the data back into the input array in parallel and include the time in the measurement.

We tested all parallel algorithms on Uniform input with and without hyper-threading. Hyper-threading did not slow down any algorithm compared to using half as many threads but the same number of cores. Thus, we give results of all algorithms with hyper-threading. Overall, we executed more than 500 000 combinations of different algorithms, input distributions, input sizes, data types, and machines. We now present an interesting selection of our measurements and discuss our results.

Overview. This chapter is divided as follows. Section 4.2 introduces and discusses the statistical measurement *average slowdown*, which we use to compare aggregated measurements. We present the results of I1S^{4}o and I1S^{2}Ra and their sequential competitors in Section 4.3. In Section 4.4, we discuss the influence of different memory allocation policies on the running time of parallel algorithms. Section 4.5 compares our parallel algorithm IPS^{4}o to its implementation presented in our conference version [Axt+17c]. We compare the results of IPS^{4}o and IPS^{2}Ra to their parallel competitors in Section 4.6. Section 4.7 evaluates the subroutines of IPS^{4}o, IPS^{2}Ra, and their sequential counterparts. Finally, we conclude our work on in-place (shared-memory) sorting algorithms in Section 7.6.

4.1 Implementation Details

IPS^{4}o has several parameters that can be used for tuning and adaptation. We performed our experiments using (up to) $k = 256$ buckets, an oversampling factor $\alpha = 0.2 \log n$, a base case size $n_0 = 16$ elements, and a block size of $b = \max(1, 2^{\lfloor 11 - \log_2 D \rfloor})$ elements, where D is the size of an element in bytes (i.e., b is about 2 KiB). Overall, the buffer blocks and the swap buffers sum up to less than 1.1 MB for each thread and take up the majority of the space required by IPS^{4}o. The space for local variables and k read and write pointers is negligible. Additionally, IPS^{4}o needs logarithmic space for the recursion stack and each thread needs space for a decision tree containing k elements. In the sequential case, we avoid the use of atomic operations on pointers and we use the recursion stack to store the tasks. On the last level, we perform the base case sorting immediately after the bucket has been completely filled in the cleanup phase, before processing the other buckets. This is more cache-friendly, as it eliminates the need for another sweep over the data. Furthermore, we use insertion sort as the base case sorting algorithm.

IPS^{4}o (I1S^{4}o) detects sorted inputs. If these inputs are detected, our algorithm reverses the input in the case that the input was sorted in decreasing order, and returns afterward. Note that such heuristics for detecting “easy” inputs are quite common [Obe+19b; Rei07].

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

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

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

We decided to implement our own version of the (non-in-place) algorithm S^{4}o from Sanders and Winkel [SW04]. This has two reasons. First, the initial implementation is only of explorative nature, e.g., the implementation does not handle duplicated keys, and, the implementation is highly tuned for outdated hardware architectures. Second, the reimplementation S^{4}oS [Hüb16] seemed to be unreasonably slow. We use IPS^{4}o as an algorithmic framework to implement PS^{4}o, our version of S^{4}o. For PS^{4}o, we had to implement the three main parts of S^{4}o: (1) the partitioning step, (2) the decision tree, and (3), the base case algorithm and additional parameters of the algorithm, e.g., for the maximum base case size, the number of buckets, and the oversampling factor. We replace the partitioning step of IPS^{4}o with the one described by Sanders and Winkel. For the element classification, we reuse the branchless decision tree of IPS^{4}o. We also reuse the base case algorithm and the parameters from IPS^{4}o, which seem to work very well for PS^{4}o. As we use IPS^{4}o as an algorithmic framework, we can execute PS^{4}o in parallel or with only one thread. If we refer to PS^{4}o in its sequential form, we use the term 1S^{4}o.

Our algorithms are rather slow for small inputs because of significant overhead for setting up distribution buffers and other data structures. To mitigate this effect, we also provide variants that can amortize this overhead over many sorting operations using *sorter objects*. The data structures are created when a sorter object is created. This object can then be used for many sorting calls with the same data type. For fairness reasons, we are *not* using these sorter objects in our experiments.

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

4.2 Statistical Evaluation

Many methods are available to compare algorithms. In our case, the cross product of machines, input distributions, input sizes, data types, and array types describes the possible inputs of our benchmark. In this work we consider the result of a benchmark input always averaged over all executions of the input, using the arithmetic mean. A common approach of presenting benchmark results is to fix all but two variables of the benchmark set and show a plot for these two variables, e.g., plot the running time of different algorithms over the input size in a graph for a specific input distribution, data type, and array type, executed on a specific machine.

Often, an interesting subset of all possible graphs is presented as the benchmark instances have too many parameters. However, in this case, a lot of data is not presented at all and general propositions require further interpretation and aggregation of the presented, and possibly incomplete, data. Besides running time graphs and speedup graphs, we use average slowdown factors (*average slowdowns*) and *performance profiles* to present our benchmark results.
Let $\mathcal{A}$ be a set of algorithms, let $\mathcal{I}$ be a set of inputs, let $S_A(\mathcal{I})$ be the inputs of $\mathcal{I}$ that algorithm A sorts successfully, and let $r(A, I)$ be the running time of algorithm A for input I. Furthermore, let $r(A, I, T)$ be the running time of an algorithm A for an input I with array type T. Note that A might not sort I successfully i.e., because its interface does not accept the data type or because A did not correctly sort the input. In this case, the running time of A is not defined.

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

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

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

$$s_{\mathcal{A},\mathcal{I}}(A) = \left(\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{A},I}(A) \right)^{\frac{1}{|S_A(\mathcal{I})|}}$$

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

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

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

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

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

$$s_{A,\mathcal{T},\mathcal{I}}(T) = \left(\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{T},A,I}(T) \right)^{\frac{1}{|S_A(\mathcal{I})|}}$$

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

Average slowdown factors are heavily used by Timo Bingmann [Bin18a] to compare parallel string sorting algorithms. We want to note that the average slowdown could also be defined as the arithmetic mean of the slowdown factors, instead of using the geometric mean. In this

case, the average slowdown would have a very strong meaning: The average slowdown of an algorithm A over a set of inputs is the expected average slowdown of A when an input of the benchmark set is picked at random to the fastest algorithm for this particular input. However, Timo Bingmann used in his work the geometric mean for the average slowdowns to “emphasize small relative differences of the fastest algorithms”. Additionally, the geometric mean is more robust against outliers and skewed measurements than the arithmetic mean [McG12, p. 229]. Furthermore, the arithmetic mean of ratios is “meaningless” in the general case. For example, Fleming and Wallance [FW86] state that the arithmetic mean is meaningless when different machine instances are compared relative to a baseline machine. In this case, ratios smaller than one and larger than one can occur. However, combining those numbers is “meaningless” as ratios larger than one depend linearly on the measurements but ratios smaller than one do not. Note that the slowdown factors in this work will never be smaller than one.

We use pairwise **performance profiles** [DM02] to compare the relative speed of two algorithms on a set of inputs: In a plot like Figure 4.3 on Page 62, the point (x, y) for an algorithm means that this algorithm sorts a fraction y of the inputs at most x times as slow as the other algorithm.

Our instance sets include all machines, input distributions, and input sizes with a few exceptions depending on the question at hand. Since we are mostly interested in the performance for large inputs, we exclude small inputs. We also exclude the easy instance distributions Zero, Sorted, and ReverseSorted because including them gives an undue advantage to algorithms that achieve large speedups by providing simple special case treatments.

4.3 Sequential Algorithms

In this section, we compare sequential algorithms for different machines, input distributions, input sizes, and data types. We begin with a comparison of the average slowdowns of I1S^{2}Ra, I1S^{4}o, and their competitors for ten input distributions executed with six different data types (see Section 4.3.1). This gives a first general view of the performance of our algorithms as the presented results are aggregated across all machines. Afterwards, we compare our algorithms to their competitors on different machines by scaling with input sizes for input distribution Uniform and data type uint64 (see Section 4.3.2). Finally, we discuss the performance profiles of the algorithms in Section 4.3.3.

4.3.1 Comparison of Average Slowdowns

Table 4.1 shows average slowdowns of sequential algorithms for different data types and input distributions aggregated over all machines and input sizes with at least 2^{18} bytes. The fact that the shown slowdowns are almost always larger than 1.0 indicates that no algorithm clearly dominates all others even for a particular input distribution. We also see that a sorting algorithm performs similarly well for inputs with “similar” input distributions. Thus, we divide the inputs into four groups: The largest group, *Skewed inputs*, contains inputs with duplicated keys and skewed key occurrences, i.e., Exponential, Zipf, RootDup, TwoDup, and EightDup. The second group, *Uniform inputs*, contains Uniform distributed inputs. For these inputs, each bit of the

Type	Distribution	**I1S^{4}o**	**BlockPDQ**	BlockQ	IS^{4}o	DualPivot	std::sort	Timsort	QMSort	WikiSort	**SkaSort**	IppRadix	**IPS^{2}Ra**
double	Sorted	1.05	1.70	25.24	**1.05**	12.90	20.49	1.09	62.42	2.81	21.83	62.61	
double	ReverseSorted	**1.04**	1.71	14.28	1.06	5.09	5.93	1.07	25.34	5.89	9.41	25.22	
double	Zero	**1.07**	1.77	21.20	1.10	1.20	14.98	1.08	2.72	3.58	16.36	24.23	
double	Exponential	**1.02**	1.13	1.28	1.27	2.30	2.57	4.23	4.04	4.20	1.29	1.38	
double	Zipf	**1.08**	1.25	1.42	1.37	2.66	2.87	4.63	4.21	4.72	1.17	1.28	
double	RootDup	**1.10**	1.50	1.83	1.65	1.44	2.30	1.32	6.01	3.12	1.90	2.69	
double	TwoDup	1.17	1.33	1.37	1.41	2.48	2.65	2.96	3.42	3.20	**1.07**	1.22	
double	EightDup	**1.01**	1.13	1.41	1.30	2.42	2.84	4.43	4.69	4.40	1.31	1.60	
double	AlmostSorted	2.33	1.15	2.21	2.99	1.68	1.80	**1.14**	6.76	2.57	2.39	4.53	
double	Uniform	1.08	1.21	1.22	1.28	2.35	2.43	3.59	2.98	3.58	**1.08**	1.29	
Total		**1.20**	1.24	1.50	1.54	2.15	2.47	2.81	4.42	3.61	1.40	1.77	
Rank		1	2	4	5	7	8	9	12	11	3	6	
uint64	Sorted	1.17	1.78	23.69	**1.02**	11.94	19.96	1.11	55.40	2.88	26.64	76.86	13.33
uint64	ReverseSorted	**1.03**	1.63	12.93	1.04	4.47	5.51	1.04	21.01	5.93	10.46	28.99	5.97
uint64	Zero	1.17	1.69	21.43	**1.06**	1.14	14.02	1.11	2.42	3.74	17.40	25.30	1.35
uint64	Exponential	1.06	1.22	1.37	1.37	2.28	2.64	4.52	3.82	4.51	1.21	1.74	**1.05**
uint64	Zipf	1.53	1.86	2.13	2.06	3.62	4.04	6.65	5.53	6.79	1.73	1.99	**1.01**
uint64	RootDup	1.25	1.73	2.19	2.07	1.60	2.60	1.70	6.34	3.91	2.08	2.88	**1.13**
uint64	TwoDup	1.73	2.07	2.11	2.17	3.56	3.88	4.54	4.65	4.93	1.58	2.66	**1.00**
uint64	EightDup	1.26	1.39	1.74	1.64	2.75	3.29	5.46	5.12	5.38	1.71	2.97	**1.02**
uint64	AlmostSorted	2.34	**1.11**	2.19	3.28	1.68	1.81	1.24	6.11	2.79	2.79	6.67	1.28
uint64	Uniform	1.35	1.60	1.60	1.71	2.85	3.02	4.62	3.49	4.63	1.20	2.19	**1.04**
Total		1.46	1.54	1.88	1.97	2.51	2.95	3.56	4.90	4.56	1.69	2.74	**1.07**
Rank		2	3	5	6	7	9	10	13	11	4	8	1
uint32	Sorted	2.44	3.89	57.73	2.42	28.63	53.53	**1.96**	139.13	6.34	46.41	44.93	29.91
uint32	ReverseSorted	1.40	2.06	17.70	1.47	6.09	8.37	**1.03**	29.37	5.57	10.08	20.92	7.28
uint32	Zero	2.30	3.71	59.44	2.28	2.28	37.19	**2.06**	6.19	8.98	24.29	14.03	3.05
uint32	Exponential	1.49	1.77	2.03	1.82	3.66	4.04	6.67	5.91	6.51	1.38	**1.08**	1.09
uint32	Zipf	1.82	2.33	2.75	2.37	4.97	5.46	8.68	7.55	8.81	1.41	1.27	**1.12**
uint32	RootDup	1.41	1.92	2.46	2.15	1.84	2.97	1.48	7.54	3.78	1.58	1.77	**1.18**
uint32	TwoDup	2.09	2.56	2.67	2.52	4.82	5.11	5.59	5.94	5.95	1.34	1.44	**1.09**
uint32	EightDup	1.40	1.68	2.09	1.76	3.67	4.19	6.47	6.23	6.45	1.35	1.77	**1.02**
uint32	AlmostSorted	3.07	1.45	2.79	4.24	2.15	2.58	**1.06**	8.24	2.97	2.66	5.45	1.51
uint32	Uniform	1.67	2.01	2.05	2.04	3.85	4.02	5.92	4.55	5.79	1.39	**1.08**	1.20
Total		1.78	1.93	2.39	2.32	3.37	3.93	4.09	6.47	5.45	1.54	1.67	**1.16**
Rank		4	5	7	6	8	9	10	12	11	2	3	1
Pair	Sorted	1.06	1.62	16.88	**1.04**	9.36	14.67	1.04	34.54	2.30	17.51		10.48
Pair	ReverseSorted	1.13	1.21	8.47	**1.08**	3.65	4.19	1.12	13.71	6.60	6.86		4.87
Pair	Zero	1.09	1.61	13.30	**1.03**	1.07	11.63	1.08	1.94	2.71	11.09		1.21
Pair	Exponential	1.10	1.92	1.20	1.36	1.84	2.12	3.87	3.11	4.14	1.16		**1.05**
Pair	Zipf	1.48	2.72	1.64	1.86	2.64	2.83	5.02	3.87	5.50	1.46		**1.01**
Pair	RootDup	1.27	1.44	1.78	1.84	1.42	2.16	1.83	4.70	4.05	1.69		**1.03**
Pair	TwoDup	1.63	2.81	1.69	1.92	2.71	2.84	3.62	3.45	4.35	1.41		**1.01**
Pair	EightDup	1.27	2.19	1.45	1.59	2.14	2.47	4.50	3.95	4.81	1.56		**1.00**
Pair	AlmostSorted	3.20	**1.01**	2.79	4.00	2.18	2.39	2.34	6.56	4.55	3.24		1.74
Pair	Uniform	1.37	2.46	1.45	1.66	2.40	2.45	3.89	2.88	4.26	1.17		**1.03**
Total		1.52	1.97	1.66	1.91	2.15	2.45	3.40	3.94	4.50	1.57		**1.10**
Rank		2	6	4	5	7	8	9	10	11	3		1
Quartet	Uniform	1.14	1.85	1.29	1.49	1.89	1.86	3.14	2.15	3.52	**1.02**		
Rank		2	5	3	4	7	6	9	8	10	1		
100B	Uniform	1.41	1.27	1.27	1.64	1.83	1.33	2.22	1.78	3.17	**1.06**		
Rank		5	2	3	6	8	4	9	7	10	1		

Table 4.1: Average slowdowns of sequential algorithms for different data types and input distributions. The slowdowns average over the machines and input sizes with at least 2^{18} bytes.

key has maximum entropy. Thus, we can expect that radix sort performs the best for these inputs. The third group, *Almost Sorted inputs*, are AlmostSorted distributed inputs. The last group, *(Reverse) Sorted inputs*, contains "easy inputs", i.e., Sorted, ReverseSorted, and Zero. For the average slowdowns separated by machine, we refer to Appendix A.3, Tables A.2–A.5.

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

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

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

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

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

However, I1S^{2}Ra could easily detect these instances by scanning the input array once. We want to note that I1S^{2}Ra already scans the input array to detect the significant bits of the input keys.

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

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

BlockQ shows similar performance as BlockPDQ for Uniform inputs. The reason is that BlockPDQ reimplemented the partitioning routine proposed BlockQ [EW16]. However, BlockQ does not take advantage of presorted sequences and BlockQ handles duplicated keys less efficiently. Thus, BlockQ is slower than BlockPDQ for (Reverse) Sorted inputs and Almost Sorted inputs.

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

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

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

We also compare I1S^{4}o to the mergesort algorithms Timsort, QMSort, and WikiSort. The in-place versions of mergesort, **QMSort** and **WikiSort**, are significantly slower than I1S^{4}o for all instances. **Timsort** is also much slower than I1S^{4}o for almost all input distributions—in most cases even more than a factor of three. Only for Almost Sorted inputs, Timsort is faster than I1S^{4}o and for (Reverse) Sorted inputs, Timsort has similar running times as I1S^{4}o.
We did not present the results of S^{4}oS, an implementation of Super Scalar Samplesort [SW04]. We made this decision as S^{4}oS is for all instances except 100B instances slower or significantly slower than 1S^{4}o, our implementation of Super Scalar Samplesort. For further details, we refer to Table A.6 in Appendix A.3 that shows average slowdowns of 1S^{4}o and S^{4}oS for different data types and input distributions. We did not the present results of the sequential version of ASPaS

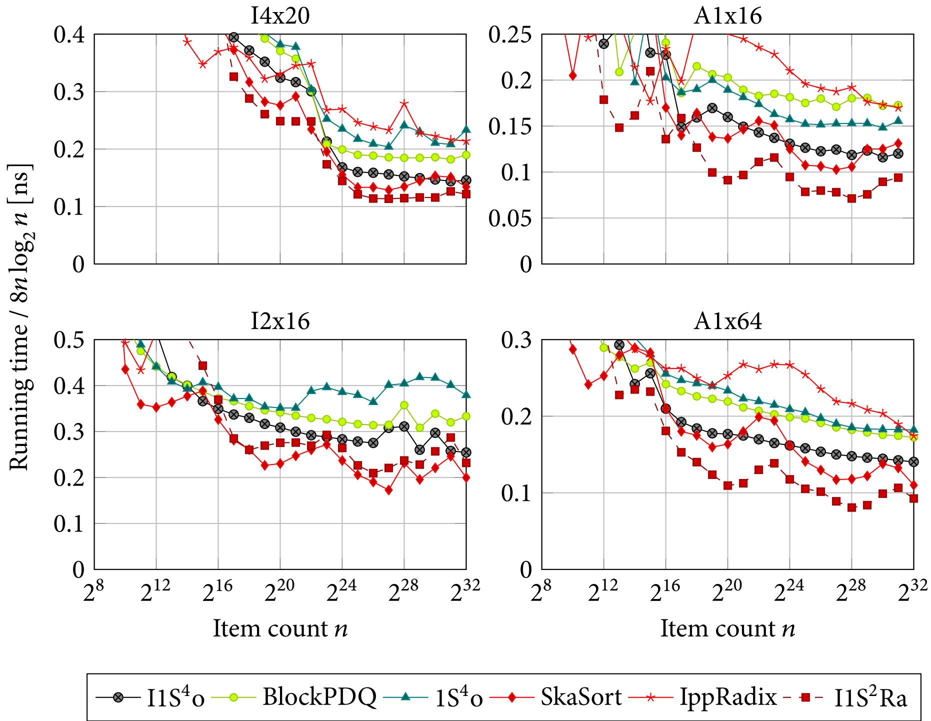


Figure 4.2: Running times of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, and WikiSort cannot be seen as their running times exceed the plot.

for three reasons. First, ASPaS performs worse than 1S^{4}o for all instances. Second, ASPaS only sorts inputs with the data type double. Finally, ASPaS returns unsorted output for inputs with at least 2^{31} elements.

4.3.2 Running Times for Uniform Input

In this section, we compare I1S^{2}Ra and I1S^{4}o to their closest sequential competitors for Uniform distributed uint64 inputs. Figure 4.2 depicts the running times of our algorithms I1S^{4}o and I1S^{2}Ra as well as their fastest competitors BlockPDQ and SkaSort separately for each machine. Additionally, we include measurements obtained from 1S^{4}o (which we used as a starting point to develop I1S^{4}o) and IppRadix (which is fast for uint32 data types with Uniform distribution). We refer to Table A.1 in Appendix A.3 for exact running time numbers. We decided to present results for uint64 inputs as our radix sorter does not support double inputs. We note that this decision is not a disadvantage for our fastest competitors as they show similar running times relative to our algorithms for both data types (see slowdowns in Table 4.1, Section 4.3.1).

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

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

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

We also compare our algorithm against **IppRadix**, which takes advantage of the *Advanced Vector Extensions* (AVX). All machines support the instruction extension `AVX2`. I4x20 additionally provides `AVX-512` instructions. We expected that IppRadix is competitive, at least on I4x20. However, IppRadix is significantly slower than I1S^{2}Ra on all machines. For example, I1S^{2}Ra outperforms IppRadix by a factor of 1.76 to 1.88 for the largest input size on A1x64, A1x16, and I4x20. On I2x16, I1S^{2}Ra is even a factor of 3.00 faster. We want to note that IppRadix is surprisingly fast for (mostly small) Uniform distributed inputs with data type uint32 (see Figure A.1 in Appendix A.3). Unfortunately, IppRadix fails to sort uint32 inputs with more than 2^{28} elements. In conclusion, it seems that `AVX` instructions only help for inputs whose data type size is very small, i.e., 32-bit unsigned integers in our case.

Comparison to I1S^{4}o. For most medium and large input sizes, **BlockPDQ** and **1S^{4}o** are significantly slower than I1S^{4}o. For example, on A1x16, I1S^{4}o is 1.29 times as fast as 1S^{4}o and 1.44 times as fast as BlockPDQ for the largest input size ($n = 2^{32}$). On the other machines, BlockPDQ is our closest competitor: I1S^{4}o is 1.23 to 1.44 (1.29 to 1.60) times as fast as BlockPDQ (1S^{4}o) for $n = 2^{32}$. According to the performance measurements, obtained with the Linux tool `perf`, there may be several reasons why I1S^{4}o outperforms BlockPDQ and 1S^{4}o. Consider the machine I4x20 and $n = 2^{38}$: BlockPDQ performs significantly more instructions (factor 1.30), more cache misses (factor 2.05), and more branch mispredictions (factor 1.69) compared to I1S^{4}o. Also, 1S^{4}o performs significantly more total cache misses (factor 1.79), more L3-store operations (factor 2.68), and more L3-store misses (factor 9.71). We note that the

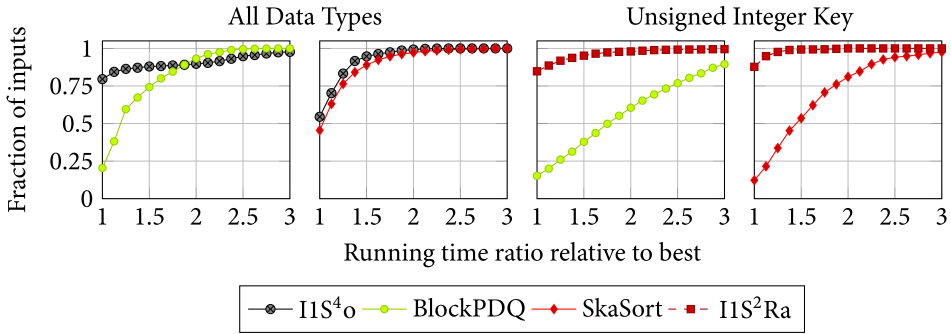


Figure 4.3: Pairwise performance profiles of our algorithms I1S^{4}o and I1S^{2}Ra to Block-PDQ and SkaSort for large inputs ($n \geq 2^{18}$) on all machines and all input distributions excluding the "easy" ones Sorted, ReverseSorted, and Zero. Profiles with the radix sorter I1S^{2}Ra only use inputs with with unsigned integer keys (uint32, uint64, and Pair data types).

comparison-based competitors DualPivot, std::sort, Timsort, QMSort, and WikiSort perform significantly more branch mispredictions than 1S^{4}o, BlockPDQ, and BlockQ. We think that this is the reason for their poor performance.

4.3.3 Comparison of Performance Profiles

In this section, we discuss the pairwise performance profiles[3] for inputs with at least 2^{18} bytes. Figure 4.3 compares our algorithms I1S^{4}o and I1S^{2}Ra to the fastest comparison-based competitor (BlockPDQ) and the fastest radix sort competitor (SkaSort).

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

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

Comparison to I1S^{4}o. The performance profile of I1S^{4}o is in most ranges significantly better than the profile of **BlockPDQ**. For example, I1S^{4}o sorts 79 % of the inputs faster than BlockPDQ.

[3] See Section 4.2 for an explanation of performance profiles.

Also, BlockPDQ sorts only 60 % of the inputs at least a factor of 1.25 faster than I1S^{4}o whereas I1S^{4}o sorts 86 % of the inputs at least 1.25 times as fast as BlockPDQ. We note that I1S^{4}o is significantly slower than BlockPDQ for some inputs. These inputs are AlmostSorted inputs. The performance profile of I1S^{4}o is slightly better than the profile of **SkaSort**. For example, I1S^{4}o sorts 54 % of the inputs faster than SkaSort. Also, I1S^{4}o (SkaSort) sorts 83 % (77 %) of the inputs at least a factor of 1.25 faster.

4.4 Influence of the Memory Allocation Policy

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

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

Benchmarks of sequential algorithms usually allocate and initialize the input array with a single thread with the default allocation policy (local allocation). The memory pages of the array are thus all allocated on a single NUMA node (*local arrays*). Local arrays are slow for many parallel algorithms because the NUMA node holding the array becomes a bottleneck. It is therefore recommended to use a different layout for parallel (sorting) algorithms. For example, the authors of RADULS2 recommend to use an array where the i-th stripe of the array is first touched by thread i (*striped array*). Another example is RegionSort for which the authors recommend to invoke the application with the interleaved allocation policy. We call arrays of those applications *interleaved arrays*. Orestis and Ross allocate for their benchmarks [PR14] on machines with m NUMA nodes m subarrays where subarray i is pinned to NUMA node i.

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

- For the *local array*, we allocate the array with the function `malloc` and a single thread initializes the array.

- For the *striped array*, we allocate the array with `malloc` and thread i initializes the i-th stripe of the input array.
- For the *interleaved array*, we activate the process-wide interleaved allocation policy using the Linux tool `numactl`.
- The *NUMA array* [Axt20b] uses a refined NUMA-aware array proposed by Lorenz Hübschle-Schneider[4]. The NUMA array pins the stripe i of the array to NUMA node i. This approach is similar to the array used by Orestis and Ross except that the NUMA array is a continuous array.

Table 4.2 shows the average slowdown of each array type for each algorithm on our machines. As expected, the machines with a single CPU, A1x16 and A1x64, do not benefit from NUMA allocations. On the NUMA machines, the local array performs significantly worse than the other arrays. Depending on the algorithm, the average slowdown of the local array is a factor of up to 1.49 larger than the average slowdown of the respectively best array on I2x16. On I4x20, the local array performs even worse: Depending on the algorithm, the average slowdown of the local array is a factor of 1.12 to 4.88 larger.

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

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

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

4.5 Evaluation of the Parallel Task Scheduler

The version of IPS^{4}o proposed in our conference article (IPS^{4}oNT) [Axt+17c] uses a very simple task scheduling. I.e., tasks with more than n/t elements are all executed with t threads (so-called parallel tasks) and sequential tasks are assigned to threads greedily in descending

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

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

Table 4.2: Average slowdowns of the local array (LA), the interleaved array with 4 KiB pages (IA), the striped array (SA), and the NUMA array (NA) for different parallel sorting algorithms on different machines. We only consider uint64 data types with at least $t \cdot 2^{21}$ bytes and input distribution Uniform.

order according to their size. The task scheduler of IPS^{4}o, described in Section 3.1.2, has three advantages. First, the number of threads processing a parallel task decreases as the size of the task decreases. This means that we can process small parallel subtasks more efficiently. Second, voluntary work sharing is used to balance the load of sequential tasks between threads. Finally, thread i predominantly accesses elements from $A[in/t \mathinner{..} (i+2)n/t - 1]$ in sequential tasks and in classification phases (see Lemmas 3.5 and 3.6). Thus, the access pattern of IPS^{4}o significantly reduces memory access to the nonlocal NUMA nodes when the striped array or the NUMA array is used.

Table 4.3 compares IPS^{4}o with IPS^{4}oNT. On machines with multiple NUMA nodes, i.e., I2x16 and I4x20, both algorithms are much slower when the local array is used. This is not surprising as the input is read in this case from a single NUMA node. On machine I4x20, I2x16, and A1x16, IPS^{4}o shows a slightly smaller average slowdown than IPS^{4}oNT for the same array type. It is hard to say whether this improvement is caused by the voluntary work sharing or by a better static scheduling. In any case, both algorithms do not execute parallel subtasks as $t \ll k$.

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

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

With the interleaved array, IPS^{4}o reports the fastest running times. For this array, the average slowdown ratio of IPS^{4}oNT to IPS^{4}o is 1.13. For the interleaved array, we expect that

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

Table 4.3: Average slowdown of IPS^{4}o and IPS^{4}oNT to the best of both algorithms for different array types and machines. The numbers in parentheses show the average running times of IPS^{4}o divided by $n/t \log_2 n$ in nanoseconds. We only consider uint64 data types with at least $t \cdot 2^{21}$ bytes and input distribution Uniform.

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

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

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

In conclusion, IPS^{4}o is (much) faster than IPS^{4}oNT for any array type tested here. IPS^{4}o shows the best performance for the interleaved array and the NUMA array, with the interleaved array performing slightly better. Both arrays allocate memory pages distributed among the NUMA nodes, and, compared to the striped array, pin the memory pages to NUMA nodes. For these arrays, the average slowdown ratio of IPS^{4}oNT to IPS^{4}o is between 1.13 and 2.54.

4.6 Parallel Algorithms

In this section, we compare parallel algorithms for different machines, input distributions, input sizes, and data types. We begin with a comparison of the average slowdowns of IPS^{4}o, IPS^{2}Ra, and their competitors for ten input distributions executed with six different data types (see Section 4.6.1). This gives a first general view of the performance of our algorithms as the presented results are aggregated across all machines. Afterwards, we compare the algorithms for input distribution Uniform with data type uint64 on different machines: We consider scaling with input sizes in Section 4.6.2 and scaling with the number of utilized cores in Section 4.6.3. Then, we discuss in Section 4.6.4 the running times for an interesting set of input distributions and data types, again by scaling the input size. In Section 4.6.5, we discuss the performance profiles of our algorithms and their most promising competitors. Finally, we separately compare IPS^{4}o to IMSDradix, which is only implemented in a very explorative manner and thus only works in some special cases (see Section 4.6.6).

4.6.1 Comparison of Average Slowdowns

Table 4.4 shows average slowdowns of parallel algorithms for different data types and input distributions aggregated over all machines and input sizes with at least $t \cdot 2^{21}$ bytes. For the average slowdowns separated by machine, we refer to Appendix A.3, Tables A.8–A.11. In this section, an *instance* describes the inputs of a specific data type and input distribution. We say that "algorithm A is faster than algorithm B (by a factor of C) for some instances" if the average slowdown of B is larger than the average slowdown of A (by a factor of C) for these instances.

Summary of the Results. Overall, the results show that IPS^{4}o is much faster than its competitors in most cases except for some "easy" instances and for some instances with uint32 data types. IPS^{4}o is faster than its **fastest comparison-based competitor PBBS**—in most cases by a factor of 1.5 or more. Except for some uint32 instances, IPS^{4}o is even significantly faster than its **fastest radix sort competitor RegionSort**. This indicates that parallel sorting algorithms are memory bound for most inputs, except for data types that only have a few bytes. In most cases, IPS^{4}o also outperforms our radix sorter IPS^{2}Ra. IPS^{2}Ra is faster for some instances with uint32 data types and, as expected, IPS^{2}Ra is faster for Uniform instances. IPS^{2}Ra has a better ranking than our fastest in-place radix sort competitor RegionSort. Thus, our approach of sorting data with parallel block permutations seems to perform better than the graph-based approach of RegionSort.

Comparison to IPS^{4}o. IPS^{4}o is the fastest algorithm for 30 out of 42 instances. IPS^{4}o is outperformed for 8 instances having "easy" input distributions, i.e, Sorted, ReverseSorted and Zero. For now on, we consider only these instances: TBB detects Sorted and Zero inputs as sorted and returns immediately. RegionSort detects that the elements of Zero inputs only have zero bits, and thus, also return immediately for Zero inputs. It is therefore not surprising that TBB and RegionSort sort easy inputs very fast. TBB (RegionSort) is for 4 (for 3) Zero instances better than IPS^{4}o. TBB is also better for 3 Sorted instances, i.e., with double, uint64, and Pair data types. Also, RegionSort is faster than IPS^{4}o for the ReverseSorted uint32 instance. Our algorithm also detects these instances but with a slightly larger overhead.

Type	Distribution	**IPS⁴o**	**PBBS**	PS⁴o	MCSTLmwm	MCSTLbq	TBB	**RegionSort**	PBBR	RADULS2	ASPaS	**IPS²Ra**
double	Sorted	1.42	10.96	2.02	15.47	13.36	**1.06**				42.23	
double	ReverseSorted	**1.06**	1.34	1.98	1.76	11.00	3.01				5.34	
double	Zero	1.54	12.83	1.80	14.55	166.67	**1.06**				41.78	
double	Exponential	**1.00**	1.82	1.97	2.60	3.20	10.77				4.97	
double	Zipf	**1.00**	1.96	2.12	2.79	3.55	11.56				5.33	
double	RootDup	**1.00**	1.54	2.22	2.52	3.88	5.54				6.28	
double	TwoDup	**1.00**	1.93	1.88	2.45	2.99	5.52				4.44	
double	EightDup	**1.00**	1.82	2.01	2.48	3.19	10.37				5.02	
double	AlmostSorted	**1.00**	1.73	2.40	5.12	2.18	3.54				6.37	
double	Uniform	**1.00**	2.00	1.85	2.53	2.99	9.16				4.39	
Total		**1.00**	1.82	2.06	2.83	3.10	7.46				5.21	
Rank		1	2	3	4	5	7				6	
uint64	Sorted	1.45	10.56	1.80	15.65	13.50	**1.09**	6.72	56.24	33.08		8.83
uint64	ReverseSorted	**1.17**	1.42	2.23	2.01	12.27	3.40	1.34	8.07	4.65		1.76
uint64	Zero	1.69	13.58	1.87	15.02	171.86	**1.13**	1.36	51.61	32.50		1.16
uint64	Exponential	**1.04**	1.74	2.10	2.62	3.41	10.38	1.79	1.58	2.58		1.20
uint64	Zipf	**1.00**	1.82	2.16	2.69	3.60	10.48	1.61	16.80	6.04		1.68
uint64	RootDup	**1.00**	1.47	2.24	2.52	3.84	5.78	1.59	9.89	7.00		1.54
uint64	TwoDup	**1.07**	1.91	2.04	2.54	3.20	5.83	1.30	10.00	3.89		1.34
uint64	EightDup	**1.02**	1.69	2.06	2.42	3.25	9.54	1.37	12.45	5.00		1.44
uint64	AlmostSorted	**1.11**	1.88	2.73	5.75	2.54	4.15	1.36	9.84	5.87		1.55
uint64	Uniform	1.13	2.10	2.14	2.80	3.32	9.57	1.59	1.41	1.49		**1.03**
Total		**1.05**	1.79	2.20	2.91	3.28	7.54	1.51	6.17	4.07		1.38
Rank		1	4	5	6	7	10	3	9	8		2
uint32	Sorted	**1.77**	10.03	2.77	11.64	14.68	1.91	5.28	7.86			4.98
uint32	ReverseSorted	1.51	1.84	2.46	2.03	11.96	5.17	1.22	1.44			**1.17**
uint32	Zero	1.59	15.94	1.95	19.35	286.17	**1.18**	1.50	73.11			1.20
uint32	Exponential	1.31	2.85	2.34	3.68	4.55	17.62	1.57	2.02			**1.02**
uint32	Zipf	**1.05**	2.54	2.06	3.22	4.05	15.68	1.33	6.39			1.41
uint32	RootDup	**1.09**	1.78	2.26	2.62	3.92	6.16	1.37	7.50			1.42
uint32	TwoDup	1.40	3.18	2.32	3.59	4.35	9.10	1.24	1.83			**1.02**
uint32	EightDup	1.23	2.84	2.26	3.41	4.24	16.24	1.33	1.84			**1.08**
uint32	AlmostSorted	1.38	2.08	2.63	5.66	3.22	4.54	1.32	1.62			**1.08**
uint32	Uniform	1.41	3.26	2.28	3.68	4.45	14.52	1.36	1.61			**1.03**
Total		1.26	2.59	2.30	3.60	4.09	10.75	1.36	2.49			**1.14**
Rank		2	6	4	7	8	9	3	5			1
Pair	Sorted	1.39	9.38	1.82	15.05	15.50	**1.03**	5.75	20.15	52.30		8.02
Pair	ReverseSorted	**1.09**	1.47	2.06	2.22	10.46	3.15	1.35	3.21	8.24		1.77
Pair	Zero	1.66	14.10	1.77	15.21	118.30	**1.08**	1.21	11.71	54.52		1.16
Pair	Exponential	1.12	1.77	2.22	2.76	3.09	6.92	1.92	**1.07**	9.52		1.39
Pair	Zipf	**1.00**	1.62	2.04	2.53	2.79	6.30	1.62	7.35	9.87		1.77
Pair	RootDup	**1.01**	1.58	2.08	2.81	3.84	4.88	1.58	4.35	11.76		1.52
Pair	TwoDup	**1.02**	1.67	2.02	2.44	2.96	4.10	1.43	4.88	7.54		1.48
Pair	EightDup	**1.02**	1.59	2.05	2.41	2.83	6.01	1.40	6.98	8.81		1.57
Pair	AlmostSorted	**1.05**	1.95	2.69	5.67	3.24	3.88	1.37	4.27	10.94		1.65
Pair	Uniform	1.08	1.81	2.12	2.62	2.93	6.15	1.67	1.20	5.36		**1.04**
Total		**1.04**	1.71	2.16	2.90	3.08	5.35	1.56	3.46	8.87		1.47
Rank		1	4	5	6	7	9	3	8	10		2
Quartet	Uniform	**1.01**	1.29	2.08	2.40	2.93	4.42					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.05**	1.14	2.14	2.35	3.18	3.55					
Rank		1	2	3	4	5	6					

Table 4.4: Average slowdowns of parallel algorithms for different data types and input distributions. The slowdowns average over the machines and input sizes with at least $t \cdot 2^{21}$ bytes.

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

IPS^{4}o is much faster than **PS^{4}o**. The only difference between these algorithms is that IPS^{4}o implements the partitioning routine in-place whereas PS^{4}o is non-in-place. We note that the algorithms share most of their code, even the decision tree is the same. The reason why PS^{4}o is slower than IPS^{4}o is that IPS^{4}o is more cache efficient than PS^{4}o: For example, PS^{4}o has about 46 % more L3-cache misses than IPS^{4}o for Uniform distributed uint64 inputs with 2^{27} elements whereas the number of instructions and the number of branch (misses) of PS^{4}o are similar to the ones of IPS^{4}o. The sequential results presented in Section 4.3 support this conjecture as the gap between the sequential versions is smaller than the gap between the parallel versions.

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

IPS^{2}Ra outperforms **IPS^{4}o** for instances with uint32 data types and some Uniform distributed instances. For uint32 instances, IPS^{2}Ra is faster than IPS^{4}o by a factor of 1.37. Interestingly, IPS^{2}Ra is not much faster than IPS^{4}o for Uniform instances with more than 32-bit elements. This indicates that the evaluation of the branchless decision tree is not a limiting factor for these data types in IPS^{4}o. For the remaining instances (data types with more than 32-bit elements and instances with noneasy distributions other than Uniform) IPS^{4}o is significantly faster than IPS^{2}Ra.
From now on, we do not present results for ASPaS, TBB, PS^{4}o, and MCSTLmwm. In regard to non-in-place comparison-based competitors, the algorithms ASPaS, PS^{4}o, and MCSTLmwm perform worse than PBBS. For non-in-place comparison-based competitors, the parallel quicksort algorithm TBB is for noneasy instances slower than the quicksort implementation MCSTLbq.

4.6.2 Running Times for Uniform Input

In this section, we compare IPS^{4}o and IPS^{2}Ra to their closest parallel competitors for Uniform distributed uint64 inputs. Figure 4.4 depicts the running times separately for each machine. We refer to Table A.7 in Appendix A.3 for exact running time numbers. The results of the algorithms obtained for double inputs are similar to the running times obtained for uint64 inputs. We decided to present results for uint64 inputs as our closest parallel competitors for

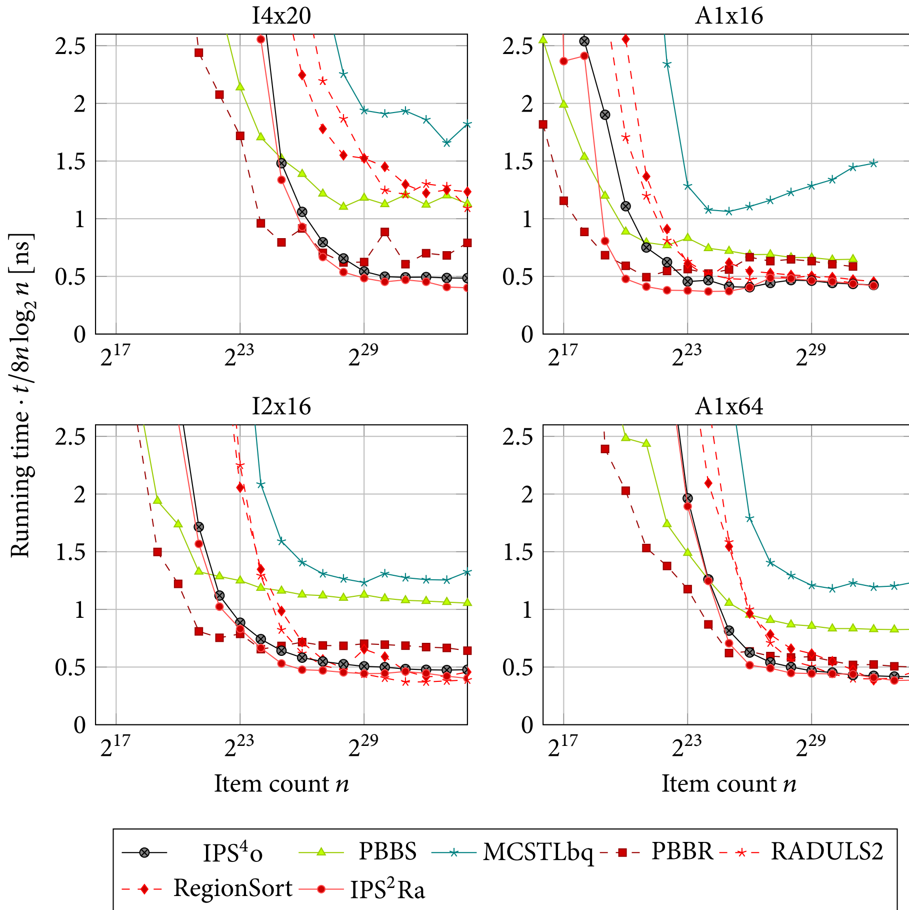


Figure 4.4: Running times of parallel algorithms sorting uint64 values with input distribution Uniform executed on different machines.

data types with "primitive" keys, i.e., RegionSort, PBBR, and RADULS2, do not support double inputs.

We outperform all comparison-based algorithms significantly for medium and large input sizes, e.g., by a factor of 1.49 to 2.45 for the largest inputs depending on the machine. For in-place competitors, the factor is even 2.55 to 3.71. For small inputs, the non-in-place competitors PBBS and PBBR are faster. Note however, that this advantage is reduced when the sorter object described in Section 4.1 is available. Also, the performance of PBBS and PBBR significantly decreases for larger inputs.

Exploiting integers slightly improves the performance of IPS²Ra compared to IPS⁴o. The radix sorters RADULS2 and RegionSort are only competitive for large input sizes. Still, they

are very inefficient even for these input sizes on I4x20, our largest machine. In particular, they are 2.72 respectively 3.08 times slower than IPS^{2}Ra for the largest input size on this machine.

We sort twice as much data as our non-in-place competitors (PBBS, RADULS2, and PBBR), which run out of memory for 2^{32} elements on A1x16. Also, the results in Table 4.4, Section 4.6.1, show that inputs with uint64 Uniform inputs are "best case" inputs for RADULS2. Other input distributions and data types are sorted by RADULS2 much less efficiently.

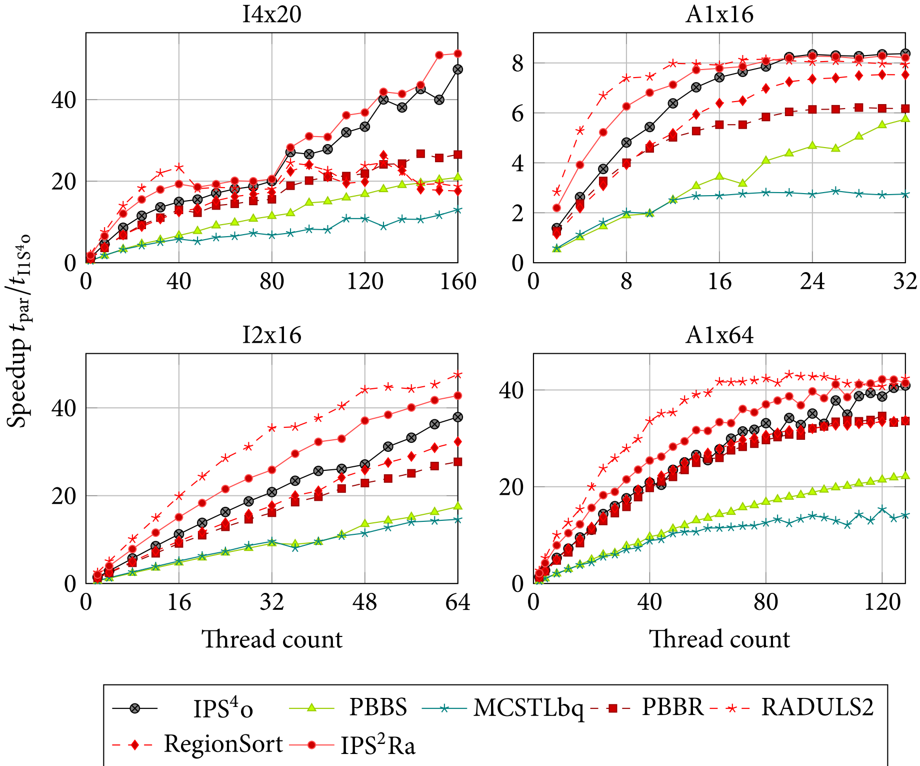


Figure 4.5: Speedup of parallel algorithms with different number of threads relative to our sequential implementation I1S^{4}o on different machines, sorting 2^{30} elements of uint64 values with input distribution Uniform.

4.6.3 Speedup Comparison and Strong Scaling

The goal of the speedup benchmark is to examine the performance of the parallel algorithms with increasing availability of cores. Benchmarks with $2i$ threads are executed on the first i cores, starting at the first NUMA node until it is completely used. Then we continue using the cores of the next NUMA node, and so on. Here, we mean by cores "physical cores" that run

two hardware threads on our machines and we use NUMA nodes as a synonym for CPUs.[5] Therefore, the benchmark always takes advantage of the "full capacity" of a core with hyperthreading. Preliminary experiments indicate that this gives around 10 % additional performance in most situations. No code is slowed down by hyperthreading. A plausible reason is that hyperthreading helps to do latency hiding during memory accesses or conditional branches.

Figure 4.5 depicts the speedup of parallel algorithms executed on different numbers of cores relative to our sequential implementation I1S^{4}o on our machines for Uniform inputs.[6] We first compare our algorithms to the non-in-place radix sorter **RADULS2**. This competitor is fast for Uniform inputs but it is slow for inputs with skewed key distributions and inputs with duplicated keys (see Table 4.4 in Section 4.6.1). On the machines with one CPU, A1x16 and A1x64, RADULS2 is faster when we use only a fraction of the available cores. When we further increase the available cores on these machines, the speedup of RADULS2 stagnates and our algorithms, IPS^{4}o, and IPS^{2}Ra, catch up until they have about the same speedup. RADULS2 also outperforms all algorithms on our machine with four CPUs, I4x20, when the algorithms use only one CPU. On the same machine, the performance of RADULS2 stagnates when we expand the algorithm to more than one CPU. When RADULS2 uses all CPUs, it is even a factor of 2.54 slower than our algorithm IPS^{4}o. We have seen the same performance characteristics when we executed IPS^{4}oNT on this machine. IPS^{4}o solved this problem of IPS^{4}oNT with a more sophisticated memory and task management. Thus, we conclude that the same problems also result in performance problems for RADULS2. Our algorithms IPS^{4}o and IPS^{2}Ra use the memory on this machine more efficiently and do not get memory-bound—the speedup of our algorithms increases on I4x20 linearly.

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

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

In conclusion, our in-place algorithms outperform their comparison-based competitors significantly on all machines independently of the number of assigned cores. For example, IPS^{4}o yields a speedup of 40.92 on the machine A1x64 whereas PBBS only obtains a speedup of 22.17. As expected, the fastest competitors for the (Uniform) input used in this experiment are radix sorters. The fastest radix sort competitor, non-in-place RADULS2, starts with a very large speedup when only a few cores are in use. For more cores, RADULS2 remains faster than

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

[6] The reference times of I1S^{4}o for 2^{30} elements of uint64 values with input distribution Uniform are 37.88, 29.91, 76.59, and 37.23 seconds on I4x20, A1x16, I2x16, and A1x64, respectively.

our algorithms on one machine (I2x16). On two machines (A1x64 and A1x16), the speedup of RADULS2 converges to the speedups of our algorithms. And, on our largest machine with four CPUs (I4x20), the memory management of RADULS2 seems to be not practical at all. On this machine, RADULS2 is even a factor of 2.54 slower than IPS^{4}o. The in-place radix sort competitor RegionSort is in all cases significantly slower than our algorithms. The speedup of IPS^{2}Ra is larger than the one of IPS^{4}o when they use only a few cores of the machine. However, the speedup levels out when the number of cores increases in most cases.

4.6.4 Input Distributions and Data Types

In this section, we compare our algorithms to our competitors for different input distributions and data types by scaling the input size. We show results of Uniform inputs for the data types double, Pair, and 100B. For a discussion of Uniform distributed uint64 data types, we refer to Section 4.6.2. For the remaining input distributions, we use the data type uint64 as a convenient example: In contrast to double, uint64 is supported by all algorithms in Figure 4.6. Additionally, we assume that uint64 is more interesting than uint32 in practice. For Figure 4.6, we decided to present results obtained on machine A1x64 as our competitors have the smallest absolute running time on this machine. For more details, we refer to Figures A.2–A.5 in Appendix A.3 that report the results separately for each machine.

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

For many inputs, our IPS^{4}o is faster than IPS^{2}Ra. For most inputs, IPS^{4}o (and to some extend IPS^{2}Ra) is much faster than RegionSort, our closest competitor. For example, IPS^{4}o is up to a factor of 1.61 faster for the largest inputs ($n = 2^{37}/D$) and up to a factor of 1.78 for inputs of medium size ($n = 2^{29}/D$). The results show that radix sorters are often slow for inputs with many duplicates or skewed key distributions (i.e., Zipf, Exponential, EightDup, RootDup). Yet, our algorithm seems to be the least affected by this. Our algorithms outperform their comparison-based competitors significantly for all input distributions and data types with $n \geq 2^{28}/D$. For example, IPS^{4}o outperforms PBBS by a factor of 1.25 to 2.20 for the largest inputs. Only for small inputs, where the algorithms are inefficient anyway, our algorithms are consistently outperformed by one algorithm (non-in-place PBBS). The remainder of this section compares our algorithms and their competitors in detail.
The non-in-place comparison-based **PBBS** is slower than IPS^{4}o for small inputs ($n \leq 2^{27}/D$). We note that all algorithms are inefficient for these small inputs. However, for inputs where the algorithms become efficient and for large inputs, IPS^{4}o significantly outperforms PBBS. For example, PBBS is a factor of 1.25 to 2.20 slower than IPS^{4}o for the largest input size. The difference between PBBS and IPS^{4}o is the smallest for 100B inputs. This input has very large elements, which are moved only twice by PBBS due to its $\sqrt{n}$-way partitioning strategy. We see this as an important turning point. While the previous state-of-the-art comparison-based algorithm worked non-in-place, it is now robustly outperformed by our in-place algorithms for inputs that are sorted efficiently by parallel comparison-based sorting algorithms.

The non-in-place radix sorter **PBBR** is tremendously slow for all inputs with skewed inputs and inputs with identical keys. In particular, its running times exceed the limits of Figure 4.6

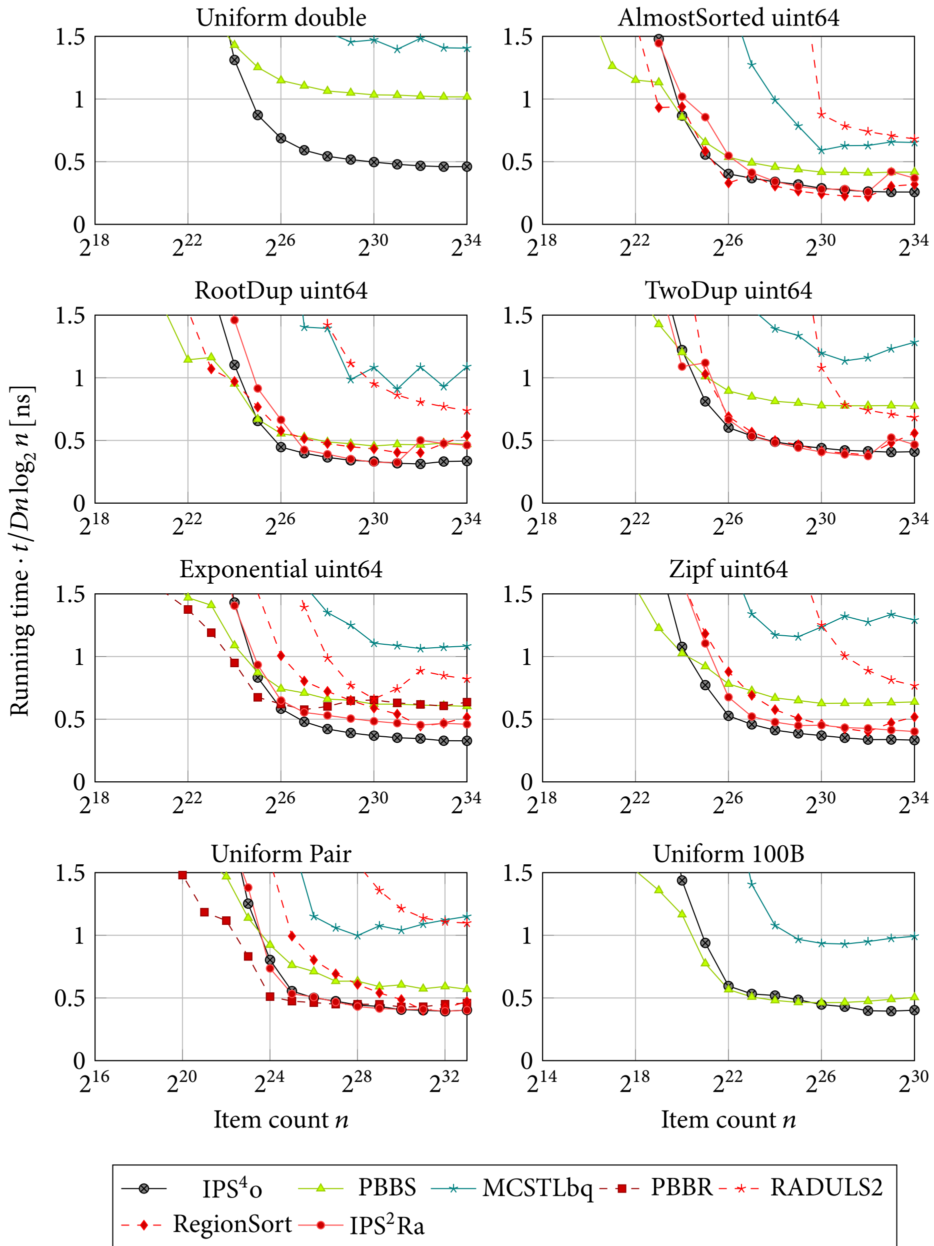


Figure 4.6: Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x64. The radix sorters PBBR, RADULS2, RegionSort, and IPS²Ra does not support the data types double and 100B.

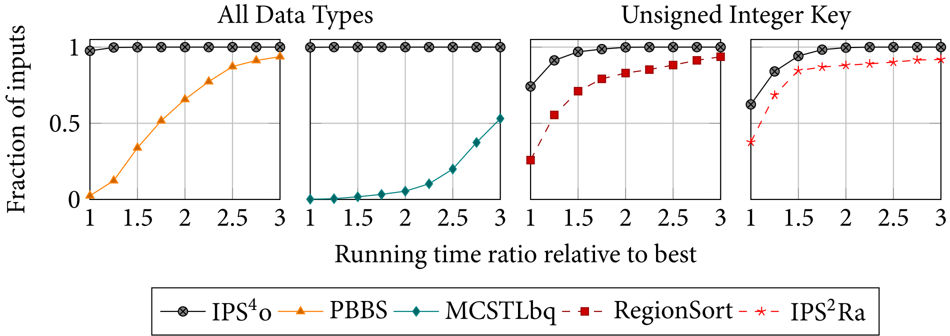


Figure 4.7: Pairwise performance profiles of IPS^{4}o to PBBS, MCSTLbq, RegionSort, and IPS^{2}Ra for input sizes with at least $t \cdot 2^{21}$ bytes on all machines and all input distributions excluding the "easy" ones Sorted, ReverseSorted, and Zero. Profiles with the radix sorters RegionSort and IPS^{2}Ra only use inputs with with unsigned integer keys (uint32, uint64, and Pair data types).

for AlmostSorted, RootDup, TwoDup, and Zipf inputs. Exceptions are Uniform inputs with Pair data type: For these inputs, PBBR is faster than our algorithms for small input sizes and performs similarly for medium and large inputs. However, this advantage disappears for other uniformly distributed inputs (see Table 4.4 in Section 4.6.1).

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

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

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

4.6.5 Comparison of Performance Profiles

In this section, we compare the pairwise performance profiles[7] of IPS^{4}o with the (non-)in-place comparison-based MCSTLbq (PBBS), and the radix sorter RegionSort as well as the pairwise performance profiles of IPS^{2}Ra and RegionSort for inputs with at least 2^{21} bytes.. The profiles are shown in Figure 4.7. We do not compare our algorithms to the radix sorters PBBR and RADULS2 as these are non-in-place and as their profiles are much worse than the profiles of the in-place radix sorter RegionSort. Overall, the performance of IPS^{4}o is much better than the performance of any other sorting algorithm. When we only consider radix sorters, the performance profile of IPS^{2}Ra is better than the one of RegionSort.

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

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

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

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

4.6.6 Comparison to IMSDradix

We compare our algorithm IPS^{4}o to the in-place radix sorter IMSDradix [PR14] separately as the available implementation works only in rather special circumstances—64 threads, $n > 2^{26}$, integer key-value pairs with values stored in a separate array. Also, that implementation is not in-place and requires a very specific input array: On a machine with m NUMA nodes, the input array must consist of m subarrays with a capacity of $1.2n/m$ each. The experiments in [Pol14] pin subarray i to NUMA node i. We nevertheless see the comparison as important since IMSDradix uses a similar basic approach to block permutation as IPS^{4}o.

Table 4.5 shows the average slowdowns of IPS^{4}o and IMSDradix for different input distributions executed on I2x16. We did not run IMSDradix on A1x64, I4x20, and A1x16 as these machines do not have exactly 64 hardware threads. The results show that IPS^{4}o is much faster than IMSDradix for all input distributions. For example, the average slowdown ratio of IMSDradix to IPS^{4}o is 7.10 for RootDup input on I2x16. Note that IMSDradix breaks for Zero input and its average slowdown are between 35.64 and 44.58 for some input distributions with duplicated keys (TwoDup, EightDup, and Zipf) and with a skewed key distribution (Zipf). For Sorted input, IMSDradix is also much slower because IPS^{4}o detects sorted inputs.

[7] See Section 4.2 for an explanation of performance profiles.

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

Table 4.5: Average slowdowns of IPS^{4}o and IMSDradix for Pair data types, different input distributions, and machine I2x16 with inputs containing at least $t \cdot 2^{21}$ bytes. IMSDradix breaks for Zero input.

4.6.6.a) Other Parallel Sorting Algorithms

We are aware that we compare our algorithms against a limited selection of sorting algorithms. A detailed comparison with all existing implementations is basically impossible. However, we had the aim to choose at least the most promising competitor from each sorting algorithm category, e.g., (non-)in-place, (non-)comparison-based, and so on. That said, it is sometimes hard to say which algorithms are the most promising competitors. For example, we recently became aware of the library *ParlayLib* [BAD20], which provides several parallel sorting algorithms. Preliminary experiments with Uniform distributed uint64 inputs on machine A1x16 show that these algorithms perform similar to PBBR and PBBS. For $n \leq 2^{21}$, the ParlayLib algorithms are, similar to PBBR and PBBS, faster than our algorithms. For larger inputs – which we focus on in this paper – the ParlayLib algorithms are significantly slower than our algorithms and again, perform similar to PBBR and PBBS. For example, IPS^{4}o outperforms these algorithms by a factor of at least 1.40 and 1.45 for 2^{25} respectively 2^{30} elements.

An interesting hybrid between radix sort and samplesort is LearnedSort [Kri+20; Kri20] that uses a sample to learn a piecewise linear approximation of the cumulative key distribution function. In principle, this combines a more robust adaptation to the input distribution than radix sort with potentially faster classification than samplesort. However the current implementation is outperformed by both relatives. In the technical report [Axt+20] we report detailed measurements on a version that had a performance bug when n is not a multiple of $n = 10^6$. We repeated a part of the measurements on a repaired implementation, which is, however, still significantly slower than either I1S^{4}o or the best radix sorters.

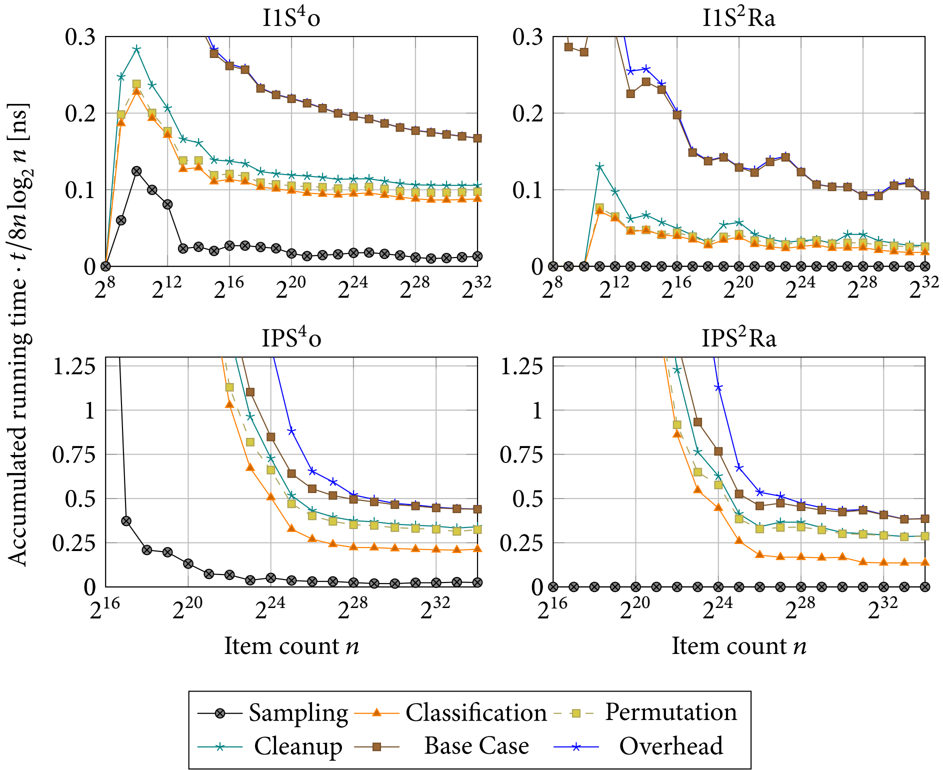


Figure 4.8: Accumulated running time (normalized by $t8n \log n$) of the phases of our sequential samplesort and radix sort algorithms (top left and top right) and their parallel counterparts (bottom left and bottom right) obtained on machine A1x64 for uint64 values with input distribution Uniform.

4.7 Phases of Our Algorithms

Figure 4.8 shows the running times of the sequential samplesort algorithm I1S^{4}o and the sequential radix sorter I1S^{2}Ra as well as their parallel counterparts IPS^{4}o and IPS^{2}Ra. The running times are split into the four phases of the partitioning step (sampling, classification, permutation, and cleanup), the time spent in the base case algorithm, and overhead for the remaining components of the algorithm such as initialization and scheduling. In the following discussion of the sequential and parallel execution times, we report numbers for the largest input size unless stated otherwise.

Sequential Algorithms

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

The time for **sampling** in the partitioning steps of I1S^{4}o is relatively small, i.e., 7.88 % of the total running time for $n = 2^{32}$. For I1S^{2}Ra, no sampling is performed.

The **classification phase** of I1S^{4}o takes up about half of the total running time. The **permutation phase** is about eight times as fast as its classification phase. As both phases transfer about the same data volume and as the classification phase performs a factor of $\Theta(\log k)$ more work ($\log k = 8$), we conclude that the classification phase is bounded by its work. It is interesting to note that this was very different in 2004. In the 2004 publication [SW04], data distribution dominated element classification. Since then, peak memory bandwidth of high-performance processors has increased much faster than internal speed of a single core. The higher investment in memory bandwidth was driven by the need to accommodate the memory traffic of multiple cores. Indeed, we will see below that for parallel execution, memory access once more becomes crucial. Classification and permutation phases of I1S^{2}Ra behave similarly as for I1S^{4}o. Since the work for classification is much lower for radix sort, the running time ratio between these two phases is smaller yet, with 2.43 still quite high ($n = 2^{32}$).

The **cleanup** takes less than five percent of the total running time of I1S^{4}o and less than two percent of I1S^{2}Ra. The sequential algorithms spend a significant part of the running time in the **base case**. The base case takes 36.71 % of the total running time for $n = 2^{32}$. For I1S^{2}Ra the base case even dominates the overall running time (70.29 % for $n = 2^{32}$) because it performs less work in the partitioning steps and because it uses larger base cases. The **overhead** caused by the data structure construction and task scheduling is negligible in the sequential case.

Parallel Algorithms

The partitioning steps of the parallel algorithms are significantly slower than the ones of the sequential algorithms. While the permutation phase of I1S^{4}o is almost negligible, it takes a significant fraction of the total running time of IPS^{4}o. For example, the total time across all threads needed for the permutation phase increases by a factor of 11.59 for IPS^{4}o and 19.88 for IPS^{2}Ra compared to their sequential counterparts. Since the permutation phase does little else than copying rather large blocks, the difference is mainly caused by memory bottlenecks.

Since memory access costs now dominate the running time, the performance advantage of radix sort over samplesort decreases when executed in parallel instead of sequentially. For other input distributions as well as other data types, the parallel radix sort is even slower than parallel samplesort (see Section 4.6.1). In other words, the price paid for *less work* in classification (for radix sort) is more data transfers due to *less accurate classification*. In the parallel setting, this tradeoff is harmful except for uniformly distributed keys and small element sizes.

When the input size is below $n = 2^{26}$, the overhead constitutes a major part of the execution time. Note that this overhead can be mostly eliminated using the sorter objects described in Section 4.1. For example, IPS^{4}o becomes 1.74 times faster for $n = 2^{24}$.

4.8 Conclusion

In-place Parallel Super Scalar Samplesort (IPS^{4}o) and In-place Parallel Super Scalar Radix Sort (IPS^{2}Ra) are among the fastest sorting algorithms both sequentially and on multi-core machines. The algorithms can also be used for data distribution and local sorting in distributed memory parallel algorithms (e.g., [Axt+15a]).

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

Our comparison-based algorithm parallel algorithm IPS^{4}o even mostly outperforms the integer sorting algorithms, despite having a logarithmic factor overhead with respect to executed instructions. Memory access efficiency of our algorithms is also the reason for the initially surprising observation that our in-place algorithms outperform algorithms that are allowed to use additional space.

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

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

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

We compare the algorithms for input arrays with various NUMA memory layouts. With our new locality-aware task scheduler, IPS^{4}o is robustly fast for all NUMA memory layouts.

Future Work

Several improvements of our algorithms can be considered in order to address the remaining cases where our algorithms are outperformed. For small inputs, not in-place variants of our algorithms with preallocated data structures, smaller values of the distribution factor k and smaller block sizes could be faster. Also, the base case sorter becomes more relevant. Here we could profit from several results on fast sorting for very small inputs [BMS20; Bra17; Cod+17].

As a further practical improvement, we could speed up the branchless decision tree with vector instructions. Preliminary results have shown improvements of up to a factor of 1.25 for I1S^{4}o with a decision tree using AVX-512 instructions. However, a general challenge remains how data-parallel instructions can be harnessed for sorting data with large keys and associated information and how to balance portability and efficiency.

With respect to the volume of accessed memory, which is a main distinguishing feature of our algorithms, further improvements are conceivable. One option is to reconsider the approach from most radix sort implementations and of the original super scalar samplesort [SW04] to first determine exact bucket sizes. This is particularly attractive for radix sorters since computing bucket indices is very fast. Then one could integrate the classification phase and the permutation phase of IPS^{4}o. To make this efficient, one should still work with blocks of elements moved between local buffer blocks and the input/output array. For samplesort, one would approximate bucket sizes using the sample and a cleanup would be required. Another difficulty may be a robust parallel implementation that avoids contention for all input distributions.

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

Our radix sorter IPS^{2}Ra is currently a prototype meant for demonstrating the usefulness of our scheduling and data movement strategies independently of a comparison-based sorter. It could be made more robust by adapting the function for extracting bucket indices to various input distributions (which can be approximated by analyzing a sample of the input). This could in particular entail various compromises between the full-fledged search tree of IPS^{4}o and the plain byte extraction of IPS^{2}Ra. For example, one could accelerate the search tree traversal of super scalar samplesort by precomputing a lookup table of starting nodes that are addressed by the most significant bits of the key. One could also consider the approach from the LearnedSort algorithm [Kri20] that addresses a large number of buckets using few linear functions. Perhaps, approximate distribution-learning approaches can be replaced by fast and accurate computational-geometry algorithms. Existing geometry algorithms [II86; DP73] might have to be adapted to use a cost function that optimizes the information gain from using a small number of piece-wise linear functions.

Adaptive sorting algorithms are an intriguing area of research in algorithms [EW92]. However, implementations [Pet02; MW18] currently cannot compete with the best nonadaptive algorithms except for some extreme cases. Hence, it would be interesting to engineer adaptive sorting algorithms to take the performance improvements of fast nonadaptive algorithms (such as ours) into account.

The measurements reported in this work were performed using somewhat non-portable implementations that use a 128-bit fetch-and-add instruction specific to x86 architectures (see also Section 4.1). Our portable variants currently use locks that incur noticeable overheads for inputs with only very few different keys. Different approaches can avoid locks without noticeable overhead but these would lead to more complicated source code.

Coming back to the original motivation for an alternative to quicksort variants in standard libraries, we see IPS^{4}o as an interesting candidate for sufficiently large inputs. Together with the variants discussed above, this might extend to the medium or even small inputs. A remaining issue is code complexity. When code size matters (e.g., as indicated by a compiler flag like `-Os`), one could use IPS^{4}o with fixed k and a larger base case size. Formal verification of the correctness of the implementation might help to increase trust in the remaining cases.

II

Engineering Distributed Sorting Algorithms

In this part of the thesis, we study distributed sorting algorithms. Chapter 5 introduces basic definitions and preliminaries, discusses problems and pitfalls of bulk data transfers, and gives an overview of existing distributed sorting algorithms. In Chapter 6, we propose new robust sorting algorithms—RFIS for very small input sizes (Section 6.1), RQuick for small input sizes (Sections 6.2 and 6.3), and RLM-sort as well as AMS-sort for large inputs (Sections 6.5 and 6.6). We then turn to an extensive experimental evaluation in Chapter 7.
We also invite the reader to Appendix B. Appendix B.1 addresses the problem of running time fluctuations caused by large message startup latencies. In Appendix B.2, we propose the communication library RangeBasedComm *(RBC) which provides scalable and efficient communication primitives on processor subsets—an essential feature for scalable implementations of recursive algorithms with sublinear running time.*

References and Attributions. Part II of the thesis is based on the conference articles Ref. [Axt+15a; AS17; AWS18]. The publication Ref. [Axt+15a] was jointly published with Peter Sanders, Timo Bingmann, and Christian Schulz. The majority of the publication has been written by Peter Sanders who also contributed the algorithmic ideas. The schematic illustrations of the algorithms were provided by Timo Bingmann. The author of this thesis provided the prototypical implementation of the algorithms and presented the experimental results. The publication Ref. [AS17] is joint work with Peter Sanders. Most parts of the publication were contributed and written by the author of this thesis. Peter Sanders also provided several notable parts of the publication. The implementations and the experimental evaluation presented in the publication are exclusive work of the author of this thesis. The publication Ref. [AWS18] is joint work with Armin Wiebigke and describes the RBC library. An initial version of this communication library has been implemented by Armin Wiebigke for his bachelor's thesis. Armin Wiebigke was supervised by the author of this thesis and publication Ref. [AWS18] has exclusively been written by the author of this thesis. The author extended the library by fast and asymptotically optimal communication primitives and incorporated the library into the algorithms presented here. All implementations and experiments proposed in Part II are exclusive work of the author of this thesis. Part II contains partial copies of the conference articles Ref. [Axt+15a; AS17].

Chapter 5

Overview of Distributed Sorting Algorithms

5

Chapter 5 introduces the reader to relevant definitions, preliminary information, and related work for distributed sorting. In Section 5.1, basic definitions and preliminaries are provided. We discuss problems and pitfalls of bulk data transfers in Section 5.2. Section 5.3 gives an overview of existing distributed sorting algorithms. In Section 5.4, we emphasize publications that highly influenced this thesis and we discuss work on sorting algorithms that we see orthogonal to the results presented in this thesis.

5.1 Definitions and Preliminaries

We denote t as the number of PEs available to us. The input of sorting algorithms are n elements with $\Theta(n/t)$ elements on each PE. The output must be globally sorted, i.e., each PE has elements with consecutive ranks and no element on PE i is larger than any elements on PE $i+1$. We also want $\mathcal{O}(n/t)$ output elements on each PE. Sometimes we more concretely consider perfectly balanced inputs and an output with at most $(1+\epsilon)n/t$ elements per PE for some small positive constant ϵ. We denote $1+\epsilon \geq 1$ as the imbalance factor of an algorithm. We call inputs *dense* when $n/t \geq 1$ and *sparse* when $n/t < 1$. In the case that the input size is smaller than the number of PEs, we allow $\mathcal{O}(1)$ local input and output elements. For simplicity, we assume that sparse input is stored on the first $\mathcal{O}(n)$ PEs. Table 5.1 gives an overview of the most important notation used in this chapter.

Table 5.1: Summary of notations

Symbol	Meaning
A	local input array
n	total number of elements
t	# of PEs
α	message startup overhead
β	time to communicate one element
k	distribution degree in recursive multiway sorting
r	# of levels in recursive multiway sorting

5.1.1 Communication Primitives

In high-performance computing, algorithms usually use the so-called *message passing interface* (MPI) [For12] for communication on distributed systems. In MPI, a communicator connects a group of PEs. An initial communicator contains all t PEs. MPI provides operations to create sub-communicators from parent communicators. MPI only allows communication between PEs of the same communicator. *Point-to-point message exchanges* are used for fine-grained communication. MPI offers various send and receive operations for message exchanges. E.g., *blocking operations* that return after the operation has been completed and *nonblocking operations* that return after the operation has been posted and require a test operation to determine completion. Point-to-point messages can be used to implement complex communication patterns. MPI defines important distributed operations that are frequently used by distributed algorithms. These so-called *collective operations* involve communication between all PEs of a communicator. In the following, we describe the collective operations used in this work.

Broadcast. One PE provides a vector of n elements and the collective operation stores the elements on each PE.

Gather. Each PE provides a vector of n/t elements and one PE returns the concatenation of these vectors.

All-Gather. Each PE provides a vector of n/t elements and the collective operation stores the concatenation of the elements on all PEs.

Reduce. Each PE provides a vector of n elements and one PE returns the partial reduction of the vectors.

All-Reduce. Each PE provides a vector of n elements and all PEs return the partial reduction of the vectors.

Prefix sum. Each PE provides a vector of n elements and PE i returns a vector of n elements containing the partial reduction of the vectors provided by PEs.

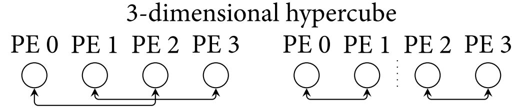
All-to-all. A data exchange in which each PE sends a distinct vector of n elements to every other PE.

In the single-ported message passing model, broadcast, (all-)gather, (all-)reduce, and prefix sum can be implemented to run in time $\mathcal{O}(\alpha \log t + \beta n)$ [Bat68; SST09] for vectors of size n on t PEs.

Many situations require collective operations with varying amounts of data. For example, an irregular gather operation receives different amounts of data from each PE. Another example is the irregular all-to-all operation that exchanges messages of different sizes between pairs of PEs. In MPI, $\Omega(t)$ time is a lower bound for these operations: For the irregular gather, the root process has to provide the input size of each PE. Still, irregular gather can be implemented in time $\mathcal{O}(\alpha \log t + \beta n)$ [Trä18]. For the irregular all-to-all, each PE must specify the number of elements the PE receives from each potential communication partner and sends to it. In Section 5.2, we discuss the cost of irregular all-to-all operations in different cost models. In practice, it also turns out that available sub-communicator creation routines are relatively slow. In particular, the routines of the most recent open-source implementations Open MPI 3.1 and

Algorithm 5 Hypercube algorithm design pattern

Input: a input vector, t number of PEs, i PE number
$d \leftarrow \log_2 t$
localComputation(a)
for $j \leftarrow d - 1$ **down to** 0 **do** ▷ or $[0 .. d)$
  $c \leftarrow i$ **xor** 2^j ▷ Calculate communication partner
  $m \leftarrow$ exchangeDataWith(c)
  localComputation(a, m)



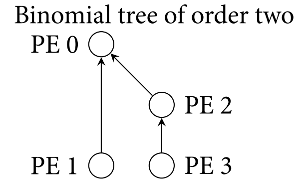
MPICH 3.4 take time $\Omega(t)$. Our experiments in Appendix B.2 indicate that this is also a lower bound for the Intel MPI library 2020.

Hypercube Algorithms. A hypercube network of dimension d consists of $t = 2^d$ PEs numbered $[0 .. t)$. Two nodes x and y are connected along dimension i if $x = y \oplus 2^i$. For this work, hypercubes are not primarily important as an actual network architecture. Rather, we extensively use communication in a conceptual hypercube as a design pattern for algorithms. To describe and understand hypercube algorithms, we need the concept of a subcube. A j-dimensional subcube consists of those PEs whose numbers have the same bits $[j .. d)$ in their binary representation. More specifically, the hypercube Algorithm 5 iterates through the dimensions of the hypercube. In iteration $j \in [0 .. d)$, each PE i is part of a $d - j$-dimensional hypercube that is split into two subcubes of dimension $d - j - 1$. Assume that PE i is the k-th PE of one of these subcubes. Then, PE i communicates in iteration j with the k-th PE of the other subcube. Depending on how this algorithm template is instantiated, one achieves a large spectrum of global effects [vdGei91; Kru92; HHL88]. We now give an overview of common instantiations—a full-length description is given by Sanders et al. [San+19, Chpt. 13]. For example, by repeatedly summing an initial local vector containing n elements, one gets an *all-reduce* in time $\mathcal{O}((\alpha + \beta n) \log t)$. Similarly, if we replace addition by concatenation, we perform an *all-gather* operation, which runs in time $\mathcal{O}(\beta t n + \alpha \log t)$. If the vectors are sorted sequences and we replace concatenation by merging, all PEs get the elements of all the local vectors in sorted order using time $\mathcal{O}(\beta t n + \alpha \log t)$. We call this operation *allgather-merge*.

We can also use a hypercube algorithm for routing data—each PE sends a distinct vector of size n to every other PE. In iteration j, data objects currently located on PE i and destined for PE t are moved if t and i differ in bit j (e.g., [Lei92]). This has the advantage that we need only $\mathcal{O}(\log t)$ startup overheads overall. This algorithm communicates $\mathcal{O}(tn \log t)$ data objects on the critical path. If every PE sends only a single object, it is known for random destination nodes that the running time remains $\mathcal{O}(\log t)$ [Lei92]. Unfortunately, for worst case inputs, even if every PE sends and receives only a single object, the routing leads to severe intermediate data imbalances. A worst case input is the *bit-reversal permutation* proposed by Thomson Leighton [Lei92, Sec. 3.4.2]. This permutation stores an element destined for PE j with bit representation $[b_{\log t} .. b_0]$ on PE i with bit representation $[b_0 .. b_{\log t}]$. Thomson Leighton shows that after $\frac{\log_k t}{2}$ iterations of the hypercube routing algorithm, $\lceil t^{1/2} \rceil$ PEs hold $\lfloor t^{1/2} \rfloor$ elements each.

Algorithm 6 Binomial-tree algorithm design pattern

Input: a input vector, t number of PEs, i PE number
$a \leftarrow$ LOCALCOMPUTATION(a)
$z \leftarrow$ TRAILINGZEROS(i) ▷ Number of 0-bits until first 1-bit occurs
$d \leftarrow \min(\lceil \log t \rceil,)$
for $j \leftarrow 0$ **up to** $d - 1$ **do**
  $c \leftarrow i + 2^j$ ▷ Calculate communication partner
  if $c \geq t$ **then break**
  $m' \leftarrow$ RECEIVEMESSAGEFROM(c)
  $a \leftarrow$ LOCALCOMPUTATION(a, m')
if $i > 0$ **then** SENDSOMEMESSAGETO$(i - 2^d)$
else return a



The required time to route the bit-reversal permutation is $\Theta\left(\alpha \log t + \beta t^{1/2}\right)$ and, generalized to n local input elements, we get $\Theta\left(\alpha \log t + \beta n t^{1/2}\right)$ time.

A k-way hypercube data exchange [Bok91] trades off $k \log_k t$ startup overheads for a bandwidth of $\mathcal{O}(\beta t n \log_k t)$. The idea behind this approach is to split the hypercube into k subcubes on each recursion level. Thus, for $k = 2^i$, one iteration of the k-way exchange performs i iterations of the ordinary hypercube algorithm. One consequence is that the k-way exchange can also lead to a load imbalance of $\Theta\left(t^{1/2}\right)$ for worst case inputs.

Binomial-Tree Algorithms. We construct a binomial tree T_k of order k as follows: Tree T_0 contains one node. Tree T_k contains a root node and k binomial subtrees of order $[0 .. k)$. We assign PEs to the tree nodes according to an in-order traversal that recurses on subtrees with lower order first. Algorithm 6 propagates information from each node to the root node. The algorithm has $\mathcal{O}(\log t)$ message startups since messages from subtrees with small order are received first and since the height of a tree T_k is k. Depending on how this algorithm template is instantiated, one achieves a large spectrum of global effects. For example, by repeatedly concatenating an initial local vector a, one gets the collective operation gather in time $\mathcal{O}(\beta t|a| + \alpha \log t)$. If the a's are sorted sequences and we replace concatenation by merging, the root PE gets the elements of all the local a's in sorted order using time $\mathcal{O}(\beta t|a| + \alpha \log t)$. We call this operation *gather-merge*.

5.1.2 Multiway Merging and Partitioning

Sequential multiway merging of k sequences with total length n can be implemented in time $\mathcal{O}(n \log k)$. An efficient practical implementation may use tournament trees [Knu73; San00; SSP07]. If k is small enough, this is even cache efficient, i.e., it incurs only $\mathcal{O}(n/B)$ cache faults where B is the cache block size. If k is too large, i.e., $k > M/B$ for cache size M, a multi-pass merging algorithm may be advantageous.

The dual operation for samplesort is partitioning the data according to $k-1$ splitters. This can be implemented with the same number of comparisons and similarly cache efficiently as k-way

merging. Multiway partitioning has the additional advantage that it can be implemented without causing branch mispredictions [SW04]. For more information on multiway partitioning, we refer the reader to Part I of this work.

5.1.3 Multisequence Selection

In its simplest form, given sorted sequences $d_1, \ldots, d_t$ (not necessarily of equal length) and a rank r, multisequence selection asks for finding an element x with rank r in the union of these sequences. If all elements are different, x also implicitly defines positions in the sequences such that there is a total number of r elements to the left of these positions.

There are several algorithms for multisequence selection, e.g. [Var+91; KK93; SSP07; SK10]. Here we use a particularly simple and intuitive method based on an adaptation of the well-known quick-select algorithm [Hoa61; SM08]. This algorithm may be folklore. See also [HS16]. The algorithm has also been on the publicly available slides of Sanders' lecture on parallel algorithms since 2008 [San08b]. The base case occurs if there is only a single element (and $r = 1$). Otherwise, a random element is selected as a pivot. This can be done in parallel by choosing the same pseudorandom number between 1 and $\sum_i |d_i|$ on all PEs. Using a collective prefix sum over the sizes of the sequences, this element can be located easily in time $\mathcal{O}(\alpha \log t)$. Where ordinary quickselect has to partition the input doing linear work, we can exploit the sortedness of the sequences to obtain the same information in time $\mathcal{O}(\log D)$ with $D := \max_i |d_i|$ by doing binary search in parallel on each PE. If items are evenly distributed, we have $D = \Theta\left(\frac{n}{t}\right)$, and thus only time $\mathcal{O}(\log \frac{n}{t})$ for the search that partitions all the sequences into two parts. Deciding whether we have to continue searching in the left or the right parts needs a collective reduce operation taking time $\mathcal{O}(\alpha \log t)$. The expected depth of the recursion is $\mathcal{O}(\log \sum_i |d_i|) = \mathcal{O}(\log n)$ as in ordinary quickselect. Thus, the overall expected running time is $\mathcal{O}\left((\alpha \log t + \log \frac{n}{t}) \log n\right)$.

In our application, we have to perform k simultaneous executions of multisequence selection on the same input sequences but on k different rank values. The involved collective communication operations will then get a vector of length k as input and their running time using an asymptotically optimal implementation is $\mathcal{O}(k\beta + \alpha \log t)$ [Bal+95a; SST09]. Hence, the overall expected running time of multisequence selection becomes

$$\mathcal{O}\left((\alpha \log t + k\beta + k \log \tfrac{n}{t}) \log n\right) \quad . \tag{5.1}$$

5.2 Delivering a Bulk of Data

In this work, we use the term *message assignment* to describe the problem of moving a bulk of data from sending PEs to receiving PEs. More specifically, the message assignment determines for each pair of PEs $\langle i, j \rangle$ the message that has to be delivered from PE i to PE j. In the most general version, the messages may be of arbitrary size and may even be empty. A *data exchange step* then executes the message assignment and routes the messages from the senders to the receivers.

We first consider a *regular data exchange*. The message assignment of a regular data exchange transfers the same number m of elements between each pair of PEs. The data exchange on

a hypercube takes time $(\alpha + \beta mt)\log t$. For large m delivering data directly is cheaper. E.g., the 1-factor algorithm from Sanders and Träff [ST02] exchanges in round $i \in [0 .. t)$ messages between the PE pairs $(u, (i - u) \bmod t)$, $u \in [0 .. t))$ summing up to time $\Theta(mt\beta + t\alpha)$.

Many bulk synchronous algorithms perform *irregular data exchanges*. We denote a data exchange as an irregular data exchange if it performs a message assignment that has arbitrary message sizes. The cost of an irregular data exchange depends on the message assignment and the algorithm that performs the actual data exchange. It is not clear how to implement these irregular data exchanges efficiently on a realistic parallel machine. In particular, the irregular data exchanges of the most recent open-source implementations Open MPI 3.1 and MPICH 3.4 send the data directly using up to t startups. For massively parallel machines, this is not scalable enough. A hypercube data exchange may only be efficient for small inputs as it moves the input $\log t$ times and as it may have huge intermediate load imbalances. Unfortunately, the BSP model only considers the bottleneck communication volume hg and "hides" the startup overheads in the term l that is independent of the actual message assignment. Thus, sending t messages of size h/t has the same cost as sending $t^{1/2}$ message of size $h/t^{1/2}$. The BSP* model [BDadH95] takes this into account by imposing a minimal message size. However, it also charges the cost for the maximal message size occurring in a data exchange for all its messages and thus, the BSP* model would still be too expensive for our sorting algorithms.

We therefore use our own generalization of the BSP model: We consider a black box data exchange function $\mathrm{Exch}(t, h, k)$ that tells us how long it takes to perform the data exchange step of a message assignment in a compact subnetwork of t PEs, with no PE receiving and sending more than h words, and k messages. Note that all three parameters of the function $\mathrm{Exch}(t, h, k)$ may be essential, as they model locality of communication (the subnetwork could be a rack of fully connected PEs), bottleneck communication volume (see [Bor13; SM13]), and startups respectively. Sometimes we also write $\widetilde{\mathrm{Exch}}(t, h, k)$ as a shorthand for $(1 + o(1))\mathrm{Exch}(t, h, k)$ in order to summarize a sum of $\mathrm{Exch}(\cdot)$ terms by the dominant one. We will also use that to absorb terms of the form $\mathcal{O}(\alpha \log t)$, $\mathcal{O}(\beta k)$, and $\mathrm{Exch}(t, k, k)$.

When comparing algorithms in different models, we can exploit that $\mathrm{Exch}(P, h, k) \leq \mathrm{BSP}(h)$ and $\mathrm{BSP}(h) \geq \mathrm{Exch}(t, h, t)$. In this work, we present a message assignment algorithm for multi-level sorting algorithms with $k \ll t$. In contrast to our model, the BSP model would treat these assignments the same. Compared to the single-ported message passing model we get $\mathrm{Exch}(P, h, k) \geq h\beta + k\alpha$ when data is delivered directly. There are reasons to believe that a data exchange algorithm comes close to this but we are not aware of actual matching upper bounds. There are offline scheduling algorithms that can deliver the data using time $h\beta$ when startup overheads are ignored (using edge coloring of bipartite multi-graphs). However, this chops messages into many blocks and also requires us to run a parallel edge-coloring algorithm.

5.3 Sorting Algorithms from Tiny to Huge Inputs

We now outline a spectrum of parallel sorting algorithms (old and new), analyze them, and use the result to compare their performance for different values of n and t. Table 5.2 summarizes the results, roughly going from algorithms for small inputs to ones for larger and larger inputs. We only give the cost terms with respect to latency (α) and communication volume

Table 5.2: Complexity of distributed sorting algorithms. Implicit $\mathcal{O}(\cdot)$. The symbol "*" indicates that the costs for splitter selection and sample selection of recursive multiway algorithms take more than $\mathcal{O}(\beta k \log t)$ time per recursion level. Unless stated otherwise, we assume deterministic or high probability running times.

Algorithm	Latency [α]	Comm. Vol. [β]	Remarks
(All)gather-merge-sort	$\log t$	n	see Sect. 5.3.1
Fast Work-Inefficient Sort [Axt+15a; AS17]	$\log t$	$n/t^{1/2}$	see Sect. 6.1
Bitonic sort [Joh84]	$\log^2 t$	$\frac{n}{t}\log^2 t$	deterministic
Minisort [SW11]	$\log^2 t$	$\log^2 t$	$n = t$
Hypercube quicksort			
– single sample element [Wag87]	$\log^2 t$	$\frac{n}{t}\log t$	best case; worst case: $\geq \beta n/t^{1/2}$; $t = 2^i$
– median of medians [LM92]	$\log^2 t$	$t + \frac{n}{t}\log t$	avg. case; worst case: $\geq \beta(t + n/t^{1/2})$; $2t = 2^i$
– random sample [SMB13]	$\log^2 t$	$t + \frac{n}{t}\log t$	avg. case; worst case: $\geq \beta(t + n/t^{1/2})$; $2t = 2^i$
Robust hypercube quicksort (see Sect. 6.3)	$\log^2 t$	$\frac{n}{t}\log t$	$t = 2^i$
Recursive BSP sorting [GV94; Goo99]	$t\log_k t$	$\frac{n}{t}\log_k t$	using cost function Exch(); arbitrary t
Recursive histogram sort [KK93]*	$k\log_k t$	$\frac{n}{t}\log_k t$	avg. case; worst case: αt; arbitrary t
HykSort [SMB13]*	$k\log_k t$	$t + \frac{n}{t}\log_k t$	avg. case; worst case: $\beta(t + n/t^{1/2})$; $t = 2^i$
Recursive HSS [HKS19]	$k\log_k t$	$t + \frac{n}{t}\log_k t$	avg. case; worst case: $\beta(t + n/t^{1/2})$; $t = 2^i$
RLM-sort [Axt+15a]* (see Sect. 6.5)	$k\log_k t$	$\frac{n}{t}\log_k t$	arbitrary t
AMS-sort [Axt+15a; AS17] (see Sect. 6.6)	$k\log_k t$	$\frac{n}{t}\log_k t$	arbitrary t
Samplesort [Ble+96]*	t	n/t	
Multiway mergesort [Var+91; SK10]*	t	n/t	

(β) per PE because the relevant local work of the algorithm is $\mathcal{O}\big(\frac{n}{t}\log n\big)$ or dominated by the communication volume. We now compare the algorithms with regard to running time and robustness.

5.3.1 (All)gather-Merge-Sort

Gathering-based sorting algorithms do not fulfill our requirement of sorted output (i.e., $\mathcal{O}(n/t)$ local output elements), but may be of general interest. We assume the algorithms described here to be folklore. When we sort the input locally and then perform a binomial-tree gather-merge (see Section 5.1.1), the root PE stores the global input in sorted order. We denote this operation *gather-merge-sort*. When we sort the input of a hypercube locally and then perform the collective operation allgather-merge (see Section 5.1.1), the global input is stored in sorted order on each PE. We denote this operation as *hypercube allgather-merge-sort*. We also get an *allgather-merge-sort* operation when we combine the binomial tree gather-merge-sort with a broadcast operation. In any case, the gathering-based sorting algorithms require time $\mathcal{O}\big(\alpha\log t + \beta n + \frac{n}{t}\log\frac{n}{t}\big)$ when $n \in \Omega(t)$. The running time also applies for sparse inputs when we use the binomial tree gather-merge. When $n < t$, the input is stored on the first n PEs and the root PE has received the input of PE $[1\,..\,2^i)$ after i recursion steps. Hypercube allgather-merge-sort requires a more complex placement of the input.

5.3.2 Fast Work-Inefficient Rank

A simple but fast ranking algorithm for small inputs is *Fast Work-Inefficient Ranking* (FIR). The algorithm has been on the publicly available slides of Sanders' lecture on parallel algorithms since 2008 [San08a] (also see [Ble+96]). In its simplest form, FIR may already have been considered as folklore before. It arranges n^2 PEs as a square matrix using PE indices from $[1\,..\,n]\times[1\,..\,n]$. Input element i is assumed to be present at PE (i,i) initially. First, the elements are broadcast along rows and columns. Then, PE (i,j) computes the result of comparing elements i and j (0 or 1). Summing these comparison results over row i yields the rank of element i. In total, the algorithm requires time $\mathcal{O}(\alpha\log t)$.

The lecture [San08a] also proposes a generalized version of FIR for $n \in \Omega(t)$. The PEs are arranged in an array of size $\mathcal{O}(t^{1/2})\times\mathcal{O}(t^{1/2})$ and each PE has $\mathcal{O}(n/t)$ elements as input. FIR first sorts the elements locally in time $\mathcal{O}\big(\frac{n}{t}\log\frac{n}{t}\big)$. Then each PE allgather-merges the elements of PEs in the same row as well as the elements of PEs in the same column. As each column and each row has $\mathcal{O}(n/t^{1/2})$ elements evenly distributed to the PEs, the allgather-merges take time $\mathcal{O}(\alpha\log t + \beta n/t^{1/2})$. Afterwards, every PE has the elements of all PEs in its row and column respectively, stored in a sorted way. Then the PEs rank each element received from their column in elements received from their row in time $\mathcal{O}(n/t^{1/2})$. Finally, FIR sums up the local ranks in each column with a collective all-reduce operation. The result is that each PE knows the global rank of all input elements in its row. The all-reduce operation with $\mathcal{O}(n/t^{1/2})$ ranks per column PE again takes $\mathcal{O}\big(\alpha\log t + \beta\frac{n}{t^{1/2}}\big)$ time. In total, the algorithm requires time $\mathcal{O}(\alpha\log t + \beta\frac{n}{t^{1/2}} + \frac{n}{t}\log\frac{n}{t})$ for $n \in \Omega(t)$. This bound does in general not apply for $n \in o(t)$—depending on the distribution of input elements to PEs, the allgather-merge operations can take more than $\mathcal{O}(\alpha\log t + \beta n/t^{1/2})$ time. We did not come up with a simple

and practical distribution pattern that guarantees the running time we devise for $n \in \Omega(t)$ also for the case that $n \in \mathrm{o}(t)$.

In theory, FIR is a very good algorithm for $n = t^{1/2}$ since it only takes logarithmic delay. However, since all other sorting algorithms need $\Omega(\log^2 t)$ startup overheads, the generalized version of FIR is even interesting for $n \in \mathcal{O}\big(\frac{\alpha}{\beta} t^{1/2} \log^2 t\big)$.

In Section 6.1 we refine FIR to run in time $\mathcal{O}(\alpha \log t + \beta \frac{n}{t^{1/2}} + \frac{n}{t} \log \frac{n}{t})$ for any input size. Also, our version calculates unique ranks $[0 .. n)$ without any additional element or lexicographic comparison—FIR computes the same rank for identical elements. We also show how to convert its output to a classical sorted permutation of the input with the same asymptotic guarantees.

5.3.3 Bitonic Sort

Bitonic sort [Bat68; Joh84] first sorts locally and then performs $\mathcal{O}\big(\log^2 t\big)$ pairwise exchange and merge operations. While the original bitonic sort only works when the number of PEs is a power of two, simple adaptions, e.g., [Lan06], allow an arbitrary number of PEs with the same asymptotic running time guarantees.

For small inputs, this running time is dominated by $\log^2 t$ startup overheads. This gives it a similar running time as the parallel quicksort algorithms to be discussed next. However, for $n = \omega(t\alpha/\beta)$ the term $\beta \frac{n}{t} \log^2 t$ dominates—all the data is exchanged $\log^2 n$ times. This makes it unattractive for large inputs compared to quicksort and other algorithms designed for these input sizes. Indeed, only for n superpolynomial in t ($n = \Omega(t^{\log t})$), bitonic sort eventually becomes efficient. But for such large inputs, algorithms like samplesort are much better since they exchange the data only once.

5.3.4 Parallel Quicksort

Since quicksort is one of the most popular sorting algorithms, it is not surprising that there are also many parallel variants. For distributed memory, a variant using the hypercube communication pattern is attractive since it is simple and can exploit locality in the network. A recursive subproblem is solved by a subcube of the hypercube. The splitter is broadcast to all PEs. Elements smaller than the pivot are moved to the 0-subcube, i.e., to the PEs whose least significant distinguishing bit is 0, and elements larger than the pivot are moved to the 1-subcube, i.e., to the PEs whose least significant distinguishing bit is 1. Since this hypercube quicksort obliviously splits the PEs in half, the imbalance accumulating over all levels of recursion is a crucial problem. For worst case inputs, the intermediate load increases to $n/t^{1/2}$—even in the case of perfect splitters. For a detailed discussion, we refer to Section 5.1.1. In its simplest original form by Wagar [Wag87], PE 0 uses its local median as a pivot. This is certainly not robust against skew and even for average case inputs it only works for rather large inputs. One can make this more robust—even in a deterministic sense—by using the global median of the local medians [LM92]. However, this introduces an additional βt term into the communication complexity and thus defeats the objective of having polylogarithmic execution time. The most recent implementation of hypercube quicksort that we have found has been proposed by Sundar et al. [SMB13]. They pick a random sample and use its median as a splitter. However, their approach is slow in practice even for random input: They pick a relatively large sample, the

sample is gathered in time $\Theta(\alpha + \beta t)$, and they use the operation `MPI_Comm_Split` to create sub-communicators whose current implementations need time $\Omega(\beta t)$. The implementation by Sundar et al. is also not robust with respect to duplicated keys. Therefore we propose a robust version of hypercube quicksort in Section 6.3. Robust hypercube quicksort turns worst case inputs into average case inputs and uses a median selection algorithm that is both fast and accurate.

Non-hypercube distributed memory quicksort is also an interesting option. It allows us to adapt the number of PEs working on a recursive subproblem to its problem size. However, it leads to more irregular communication patterns and has its own load balancing problems because we cannot divide PEs fractionally. For a more detailed discussion, we refer to our work on perfectly balanced quicksort [AWS18]. Siebert and Wolf [SW11] exploit the special case $n = t$ where this is no problem.

5.3.5 Multiway Sorting Algorithms

For large inputs, it is a disadvantage that quicksort exchanges the data at least $\log t$ times. We can improve that by partitioning the input with respect to $k - 1$ pivots at once. This decreases the number of times data is moved to $\mathcal{O}\big(\log_k t\big)$. However, the price we pay is latency $\Omega(\alpha k)$ for delivering data from k partitions to k different PEs. This gives us a lower bound of

$$\Omega\Big(\frac{n}{t}\log n\Big) + \beta\frac{n}{t}\log_k t + \alpha k \log_k t$$

for running generalized quicksort. We call these generalizations *multiway sorting algorithms*. However, getting an algorithm with a matching upper bound is not easy. We have to find pivots that partition the input in a balanced way and we have to execute the more complex data exchange problems efficiently.

The Single-Level Case. Many multiway algorithms have been devised for the special case $k = t$. We denote these algorithms as *single-level algorithms*. Single-level samplesort [Ble+96] is perhaps most well known because it is simple and effective. It achieves the above bound if a sample of size $S = \Omega(t \log n)$ is sorted using a parallel algorithm. Then, every S/k-th sample is used as a splitter. The algorithm achieves the desired bound for $n = \mathcal{O}\big(t^2/\log t\big)$. Solomonik and Kalé [SK10] describe single-level histogram sort which scales to 2^{15} PEs for large inputs. Single-level histogram sort uses an iterative sampling-based algorithm for finding high-quality approximate splitters. Histogramming is a compromise between the single shot algorithms used in samplesort and the exact (but slower) algorithms used for mergesort. Thus, histogram sort can be viewed as a hybrid between multiway mergesort and samplesort. It can also be seen as a generalization of multisequence selection. Single-level histogram sort overlaps data exchange and merging of the received data to minimize running time. Solomonik and Kalé do not provide performance bounds on histogramming for the approximate case. When we transfer the bounds of splitter selection with perfect histogramming [KK93]—no imbalance is allowed—to single-level histogram sort, it achieves the desired bound for $n = \mathcal{O}\big(t^2 \log t\big)$. Indeed, several multiway mergesort algorithms [Var+91; SSP07] that achieve perfect partitioning by finding optimal splitters have been proposed for the single-level case. This again requires $n = \mathcal{O}\big(t^2 \log t\big)$ for achieving the desired bound.

Allowing Multiple Levels. So-called *recursive multiway algorithms* sort the input recursively on r recursion levels while performing a k-way data exchange on each level. Recursive multiway algorithms pursue two different approaches for the data transfer. One approach uses a multiway hypercube data exchange that routes the elements deterministically to the target PEs. The other approach is more general and works for arbitrary t. On each recursion level, the approach calculates a message assignment depending on the local bucket sizes. A bulk data transfer then performs an irregular data exchange with point-to-point messages.

Kalé and Krishnan [KK93] propose recursive multiway histogram sort with perfect median selection and a **bulk data transfer** with direct message exchanges on each recursion level. This algorithm was implemented in the CHARM parallel programming system and does not run on today's supercomputers. Gerbessiotis and Valiant [GV94] develop recursive multiway samplesort. Goodrich [Goo99] gives communication efficient sorting algorithms based on recursive multiway merging. Both algorithms are developed for the BSP model and are not implemented in practice. However, these algorithms need a significant constant factor more communication per element than our algorithms. Moreover, bulk data transfers in the BSP model allow arbitrarily fine-grained communication at no additional cost. In particular, an implementation of the global data exchange primitive of BSP that delivers messages directly has a bottleneck of t message startups for every global message exchange. Also, see Section 6.4 for a discussion on why it is not trivial to adapt the BSP algorithms to a more realistic model of computation—it turns out that for worst case inputs, one PE receives data from t PEs. We have not found a recursive multiway sorting algorithm for arbitrary t that is not affected by this problem.

While the latter sorting algorithms work for arbitrary t, we now consider *(recursive) multiway hypercube sorting algorithms* which require that $t = k^r$ where r is the number of levels. Sundar et al. [SMB13] develop and implement the multiway hypercube algorithm *HykSort*, a generalization of hypercube quicksort. On each recursion level, HykSort selects k splitters with the approximate histogramming algorithm proposed for single-level sorting by Solomonik and Kalé [SK10] and redistributes the data to k subcubes. Similar to this single-level algorithm, HykSort overlaps data exchange and merging of the received data to minimize running time. The authors of HykSort also do not prove performance bounds on the approximate histogramming algorithm. However, when we transfer the bounds of splitter selection with perfect histogramming [KK93] to HykSort, it achieves the desired bound for $n = \mathcal{O}\left(t^{1+1/k} \log t\right)$ on uniformly distributed random keys. Unfortunately, HykSort and similar multiway hypercube sorting algorithms can lead to severe data imbalances in the worst case (see Section 5.1.1). The publication from Sundar et al. [SMB13] mentions measures for solving this problem, but does not analyze, implement, or evaluate them. Furthermore, HykSort uses the operation `MPI_Comm_Split` to create sub-communicators whose current implementations need time $\Omega(\beta t)$. HykSort is also not robust with respect to duplicated keys.

In 2019, Harsh et al. [HKS19] refined the approximate splitter selection algorithm of histogram sort and propose *histogram sort with sampling* (HSS). The authors propose two versions of HSS—single-level HSS and multiway hypercube HSS. When we consider the allowed imbalance of the sorted output as a constant, HSS has an isoefficiency function of $\mathcal{O}\left(t^{1+1/k} / \log t\right)$—the same isoefficiency as our recursive multiway samplesort algorithm AMS-sort proposed in 2015 (see Section 6.6 for a description of AMS-sort). HSS is the best

scaling single-level competitor we have seen so far and scales to 2^{14} PEs for relatively large inputs. Hypercube-based HSS uses HykSort as an algorithmic framework and replaces the splitter selection routine of HykSort by histogramming with sampling. Hypercube-based HSS, however, faces the same problems as HykSort.

Tie-Breaking. Many multiway sorting algorithms are not robust against the case of many equal elements [SK10; KK93; SMB13]. In regard to recursive multiway sorting, we have not found any algorithm that handles duplicates. A naive approach to generate unique keys appends unique identifiers to the elements [GV94; Goo99; Ble+96]. Then, lexicographic ordering makes the keys unique. However, this approach makes sorting considerably more expensive due to higher communication volume and more expensive comparisons. In Part I, we propose a low overhead measure that arranges elements equal to splitters into equality buckets.

Related Multiway Shared-Memory Approaches. Li and Sevcik [LS94] propose parallel sorting by overpartitioning for shared-memory multiprocessors. Their overpartitioning approach picks much more pivots than available PEs and the buckets are sorted in order of decreasing bucket sizes. However, their approach is not scalable enough to the largest machines since sample sorting is centralized and since a master-worker load balancer heuristically distributes buckets among all PEs. As a result, the output is not globally sorted and an additional data exchange step would be required to fulfill our output requirements. The overpartitioning algorithm from Section 6.6, which we propose for AMS-sort, is fully parallelized and optimally assigns consecutive ranges of buckets to consecutive PE-groups.

For more information on sequential and shared-memory sorting, we refer to Chapter 2. Some ideas of shared-memory algorithms, e.g., recursive multiway sorting [CR10; BGS10] for cache-oblivious shared memory models, may be interesting starting points for sorting on distributed machines. However, translating them to a distributed memory model seems nontrivial and is likely to result in worse bounds.

5.4 More Related Work

This thesis was highly influenced by two landmark papers. 25 years ago, Blelloch et al. [Ble+96] carefully studied a large number of algorithms, selected three of them (bitonic sort, radix sort, and samplesort), and compared them experimentally on up to 2^{15} PEs. Most other publications are much more focused on one or two (more or less) new algorithms sometimes missing the big picture. We felt that it was time again for a more wide view in particular because it became evident to us that a single algorithm is not enough on the largest machine and varying n/t. Helman et al. [HBJ98]—concentrating on samplesort—took robustness seriously and made systematic experiments with a wide variety on input distributions, which form the basis for our experiments. They also propose initial random shuffling to make samplesort robust against skewed input distributions (this was already proposed in [SH97]).

There has been a lot of work on making sorting fast on small systems exploiting SIMD, GPU, and shared memory parallelism. We view these results as largely orthogonal to ours. However, it should be noted that large performance gains due to exploiting low-level hardware properties are mostly observed when sorting unrealistically small objects like 32-bit numbers.

Also, since the communication inevitably becomes the bottleneck for very large machines, other optimizations become less and less important.

Chapter 6

Robust Scalable Distributed Sorting Algorithms

6

We propose four robust sorting algorithms for different input sizes which can be used to span the entire parameter space. The first algorithm sorts while data is routed to a single PE. This algorithm, gather-merge-sort, is already described in Section 5.3.1 since we assume the algorithm to be folklore. The second algorithm is a simple yet fast work-inefficient algorithm with logarithmic latency (RFIS). The third algorithm is an efficient variant of parallel quicksort with latency $\mathcal{O}(\log^2 t)$ (RQuick). The fourth algorithm is based on recursive multiway sample-sort and has latency $\mathcal{O}(rt^{1/r})$ when we allow data to be moved r times (AMS-sort). The output of the latter algorithm has an imbalance of $1 + \epsilon$. For the case of perfectly balanced output, we additionally propose recursive multiway mergesort (RLM-sort). For both multiway algorithms, we propose a new message assignment algorithm (DMA) that bounds the number of incoming and outgoing messages of the data exchanges to $\mathcal{O}(t^{1}/r)$.

6.1 Robust Fast Work-Inefficient Ranking and Sorting (RFIS)

We propose *Robust Fast Work-Inefficient Sorting* (RFIS) for sparse and very small local input sizes. We first perform our *Robust Fast Work-Inefficient Ranking* (RFIR) algorithm that computes the rank of each element in the global data. Then, optionally, a data exchange routine sends the elements to the appropriate PEs according to their ranks. RFIR refines the idea of FIR (see Section 5.3.2). In contrast to FIR, RFIR works for any input size and generates unique ranks—even in the case of duplicate elements. This is crucial as the data exchange routine only works without significant load imbalances if the ranks are unique.

Our generalization works for any number of elements n and a quadratic $a \times a$ grid of PEs. Initially, there are n elements uniformly distributed over the PEs. More precisely, each PE provides $\mathcal{O}(n/t)$ input elements if $n \geq t$. Otherwise, the elements are evenly distributed to the first n PEs. Figure 6.1 provides pseudocode. The algorithm performs six steps.

(i) On each PE-column, we perform a collective allgather-merge-sort operation. This operation takes time $\mathcal{O}\left(\alpha \log t + \beta \frac{n}{p^{1/2}} + \frac{n}{t} \log \frac{n}{t}\right)$, even for sparse inputs (recall Section 5.3.1). Now, each PE stores the elements of its column in sorted order.

(ii) PE (r, c) (the PE in row r and column c) exchanges its sorted column elements with PE (c, r). As a result, PE (r, c) stores two sequences—the sorted column elements of column r respectively c. We charge $\mathcal{O}(\alpha + \beta n/t^{1/2})$ time.

(iii) PE (r, c) computes the ranks of the sorted elements C_c from column c in the sorted elements C_r from column r. More precisely, for each element e in C_c, we count the

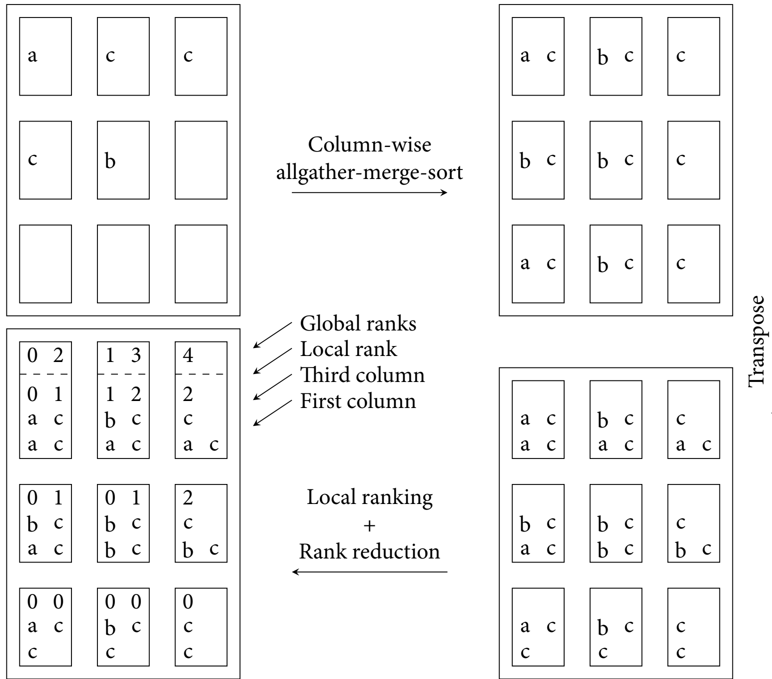


Figure 6.1: Robust fast work-inefficient ranking on a 3 × 3 PE-grid with five elements.

number of smaller elements in C_r. This can be done in time $\mathcal{O}(1 + n/t^{1/2})$ by merging these two sequences. For tie-breaking, we consider a case distinction with respect to r and c. If $c < r$, we consider an element $e_r \in C_r$ to be smaller than e if $e_r < e$. If $c > r$, we consider an element $e_r \in C_r$ to be smaller than e if $e_r \leq e$. If $c = r$, we set the rank of the i-th element to i. This schema applies tie-breaking by implicitly handling the m-th smallest element of column c, i.e., the element $C_c[m]$, as the triple $\langle C_c[m], c, m\rangle$.

(iv) Collectively reducing the "local ranks" of column c at PE $(0, c)$ with the operation "addition" yields the global rank of the element in C_c. We charge $\mathcal{O}(\alpha \log t + \beta n/t^{1/2})$ time.

Roughly spoken, RFIS performs an allgather-merge-sort operation to obtain sorted column input on each PE (i). The result of a transpose operation is that PE (r, c) stores the input of column r and c (ii). Step (iii) then ranks the elements of column c into the elements of column r using a simple tie-breaking approach and step (iv) calculates global ranks. In contrast, the initial version of FIR [AS17] (also see Section 5.3.2) performs an allgather-merge-sort operation on each PE-row and each PE-column. Compared to RFIS, the disadvantage of this approach is that these operations require complicated distributions of input elements to PEs in order to become efficient.

If desired, the global ranks can then be used for routing the elements in such a way that a globally sorted output is achieved. I.e., the element with rank i will be routed to PE $\lfloor it/n \rfloor$. We first broadcast the global ranks of C_c from PE $(0, c)$ to all PEs of column c. If $n \leq t$ we send the elements directly to the target PEs. This takes time $\mathcal{O}(\alpha)$ since each PE sends and receives a constant number of elements—each PE of column c must send only a constant fraction of C_c. If $n > t$, the data exchange problem is solved locally in each row. We now exploit the fact that each PE-row stores the complete ranked input. We thus can afford to discard all elements that are not mapped to the same row without losing any element. For routing the elements to their correct column, we can use the hypercube algorithm from Section 5.1.1 if the grid length a is a power of two. While routing the elements, we maintain their local order. The total number of elements in the row is n/a. After $\log a$ message exchange steps of the hypercube algorithm, each PE of the row will have n/a^2 elements. In the i-th iteration of this algorithm, $n/(a2^{i+1})$ elements have to be delivered to each of the two subcubes—the subcubes have size $a/2^{i+1}$. Even if all these elements are concentrated on the same PE, this requires at most $\alpha + \beta n/(t^{\frac{1}{2}} 2^i)$ time. Summing this over all iterations yields overall time $\mathcal{O}(\alpha \log t + \beta n/t^{1/2})$.

If the grid length is not a power of two, we can use the index algorithm by Bruck et al. [Bru+97] for the data exchange. The index algorithm works similar to the hypercube algorithm. The main difference is that the PE-group in the i-th iteration has up to $\lceil a/2^i \rceil$ (non-adjacent) PEs and $\lceil \log a \rceil$ iterations are performed. With the same argumentation as for hypercube quicksort, we get the same overall time $\mathcal{O}(\alpha \log t + \beta n/t^{1/2})$. Indeed, the asymptotic running time is the same for just ranking or ranking plus data exchange.

Either way, we get the overall execution time

$$\mathcal{O}\left(\alpha \log t + \beta \tfrac{n}{t^{1/2}} + \tfrac{n}{t} \log \tfrac{n}{t}\right) \ . \tag{6.1}$$

Note that for n polynomial in t this bound is $\mathcal{O}(\alpha \log t + \beta n/t^{1/2})$. This restriction is fulfilled for all reasonable applications of this sorting algorithm.

6.2 Building Blocks for Hypercube Quicksort

In this section, we describe building blocks that we need for robust hypercube quicksort in Section 6.3.

6.2.1 Bound for the Balls into Bins Problem

We consider the *balls into bins problem* [JK77]. This problem throws m balls independently and uniformly at random into b bins. The number of balls thrown into a bin is synonymous to the number of successes in a Bernoulli trial with m experiments and a success probability of $1/b$. Here, we modify the balls into bins problem by allowing to scale m and b.

Lemma 6.1 (Scaled balls into bins problem)

Let $t = 2^x \geq 2$ with x being a natural number and let $i \in [0 .. \log t)$. Furthermore, let $n/2^i$ balls be thrown into $t/2^i$ bins. Then, the probability that all bins have $\mathcal{O}(n/t + \log t)$ balls each is $\geq 1 - \min(n^{-c}, t^{-c})$ for any constant $c > 0$.

Proof. We first consider a fixed bin and $n \geq 13t \ln t$. Let X be the random variable determining the number of elements assigned to this bin. The number of elements in this bin is the result of a balls into bins problem with $m = n/2^i$ balls and $b = t/2^i$ bins. We bound the probability of having at least $(2+c)n/t$ balls in the bin to

$$\begin{aligned} \mathrm{P}[X \geq (2+c)n/t] &\leq e^{-\frac{(1+c)n}{3t}} = e^{-\frac{(1+c)n \ln n}{3t \ln n}} = n^{-\frac{(1+c)n}{3t \ln n}} \\ &\leq n^{-\frac{13(1+c)t \ln t}{3t \ln(13t \ln t)}} \leq n^{-c-1} \end{aligned} \tag{6.2}$$

for any $c > -1$. The first "≤" uses the Chernoff inequality from Section 1.4 with $\delta = 1 + c$, the second "=" uses $e^{\ln n} = n$, the second "≤" uses the assumption $n \geq 13t \ln t$, and the third "≤" uses $\frac{13t \ln t}{3t \ln(13t \ln t)} > 1$ for all $t \geq 2$.

For now, we have bounded the number of balls in a fixed bin. Applying this bound to all bins, no bin has $\omega(n/t)$ balls with probability $\geq 1 - n^{-c}$ since the probability that at least one of the $t/2^i$ bins gets assigned at least $(2+c)n/t$ balls can be bound by

$$t/2^i \cdot \mathrm{P}[X \geq (2+c)n/t] \leq t/2^i \cdot n^{-c-1} \leq n^{-c}$$

since we assume $n \geq t$.

We now consider the case $n < 13t \ln t$. When we pad the elements to a total of $n' = 13t \ln t$ elements, the maximum load of the bins is at most $(2+c)13 \ln t$ with probability $\geq 1 - n'^{-c}$ for $c > 0$. By removing the padding elements, this probability bound also holds for the initial n input elements. □

6.2.2 Randomized Shuffling on Hypercubes

It is a folklore observation (e.g., [SH97]) that data skew can be removed by shuffling the input data randomly. This is implemented for the samplesort algorithm of Helman et al. [HBJ98] by directly sending each element to a random destination. Note that this incurs an overhead of about $\alpha \min(t, n/t) + \beta n/t$. We propose a simple technique for small inputs where we need logarithmic startup latency. This can be achieved by routing the data according to a hypercube communication pattern—sending each element to a random side in each communication step. Here, we first communicate along dimension 0 and go up to dimension $\log t - 1$. Each communication step performs two operations. First, the PEs split their local elements into two random halves. Each element is assigned to one half with probability 1/2. Second, each PE sends one half to its communication partner, keeps one half, and receives one half from its communication partner.

Lemma 6.2 (Distribution of elements in randomized shuffling)
After each communication step, each element is stored on a random PE of its subcube.

Proof. We consider a random element e. Let e be input on PE j where $[b_0 \mathinner{..} b_{\log t})$ is the binary representation of j. In order to prove Lemma 6.2, we show the following proposition. After communication along dimension $i \in [0 \mathinner{..} \log t)$, element e is stored on a PE whose $i+1$ least significant bits are distributed uniformly at random and the $\log p - i - 1$ most significant

bits are $[b_{\log t-i-1} .. b_{\log t})$. We prove this proposition by induction over the communication steps. The base case is $i = 0$. In the first communication phase, the communication partner of j differs in the least significant bit. The base case is true since e is randomly routed to j or to its communication partner. We now consider communication along dimension $i+1$: By the induction hypothesis, we know that e is stored on a PE j whose i least significant bits are distributed uniformly at random and the $\log t - i$ most significant bits are $[b_{\log t-i} .. b_{\log t})$. Again, the binary representation of PE j and its communication partner only differ in bit $i+1$. Since e is randomly routed to j or its communication partner, the proposition also holds after communication along dimension $i+1$. □

Lemma 6.3 (Bound load of randomized shuffling)
After each communication step, each PE stores $\mathcal{O}(\max(n/t, \log t))$ elements with probability $\geq 1 - \min(n^{-c}, t^{-c})$ for any constant $c > 0$.

Proof. After communication step $i \in [0 .. t)$, each subcube has $\mathcal{O}(2^{i+1}n/t)$ elements in total. We consider a random subcube C. According to Lemma 6.2, the elements stored in C are randomly distributed to its PEs. Thus, the number of elements on the PEs of C can be described by the balls into bins problem with $\mathcal{O}(2^{i+1}n/t)$ balls and 2^{i+1} bins. From Lemma 6.1 in Section 6.2.1 follows that each PE of C stores at most $\mathcal{O}(n/t + \log t)$ elements with probability $\geq 1 - \min(n^{-c}, t^{-c})$. This probability bound also holds for all subcubes since the hypercube has only $2t-1$ subcubes. □

Lemma 6.4 (Running time of randomized shuffling on hypercubes)
The running time of our randomized shuffling algorithm with $\mathcal{O}(n/t)$ elements per PE on a hypercube of size t is

$$\mathcal{O}(\alpha \log t + \beta \max(n/t, \log t) \log t)$$

with probability $\geq 1 - \min(n^{-c}, t^{-c})$ for any constant $c > 0$.

Proof. From Lemma 6.3 follows each PE communicates $\mathcal{O}(\max(n/t, \log t) \log t)$ elements. We additionally account $\mathcal{O}(\alpha \log t)$ for communication startups. □

A naive approach labels each element with a random destination. Then, a hypercube algorithm redistributes the data. This approach increases the communication volume by a factor of two.

6.2.3 Approximate Median Selection with a Single Reduction

Siebert and Wolf [SW11] consider parallel quicksort for the case where $n = t$. They propose to select splitters using *ternary remedian*, an estimator to approximate the median of an input sequence initially introduced by Rousseeuw and Bassett [RB90]. Remedian uses a ternary tree whose leaves are the input elements. At each internal node, the median of the children elements is passed upward. Lemma 6.5 from Dean et al. [DJW14] shows that for randomly permuted inputs and a balanced tree this is a good approximation of the median.

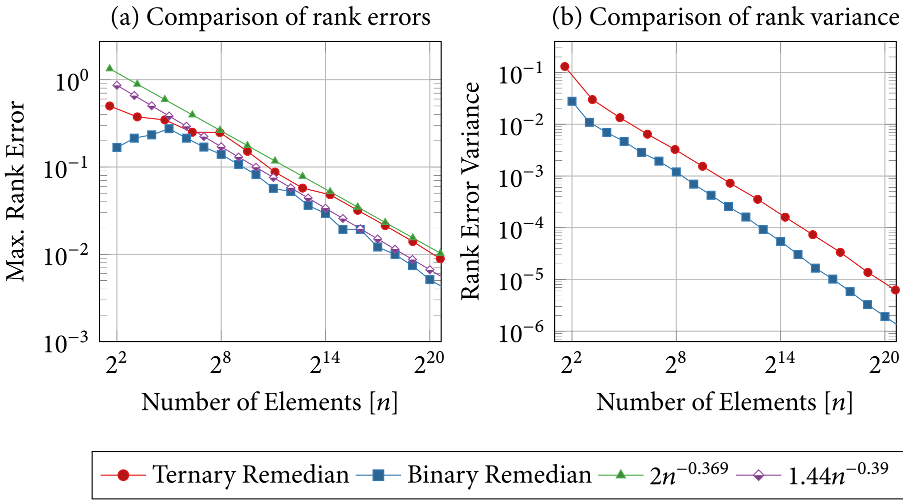


Figure 6.2: Comparison of ternary remedian and binary remedian with $k = 2$.

Lemma 6.5 (Ternary remedian)
Let S be a random sample from any probability distribution that supports efficient sampling (e.g., elements stored in an array). For each constant $c > 0$, there exists an n' for which the ternary remedian algorithm selects from $n > n'$ randomly shuffled sample elements S a splitter element of rank $[n/2(1 - 2n^{-0.369}) .. n/2(1 + 2n^{-0.369})]$ within the probability distribution with probability $\geq 1 - n^{-c}$ when n is a power of three.

But Siebert and Wolf [SW11] do not permute randomly. Even when t is a power of two, their method does not produce a completely balanced tree such that their implementation remains a heuristic.

We fix these restrictions by using a *binary remedian* and working with randomly permuted inputs: Consider a tuning parameter k (even). Assume that the local data is a sorted sequence $a[1..m]$ and that m is even. Each PE is a leaf of the binary tree and forwards $a[m/2 - k/2 + 1 .. m/2 + k/2]$ up the tree—a local approximation of the elements closest to the median. Undefined array entries to the left are treated like a very small key and undefined entries to the right as very large keys. If m is odd, we flip a coin whether $a[\lfloor m/2 \rfloor - k/2 + 1 .. \lfloor m/2 \rfloor + k/2]$ or $a[\lceil m/2 \rceil - k/2 + 1 .. \lceil m/2 \rceil + k/2]$ is forwarded. Internal nodes merge the received sequences and use the result as their sequence a analogously to the leaves. At the root, we flip a coin whether $a[k/2]$ or $a[k/2 + 1]$ is returned. Note that in most MPI implementations, this algorithm can be implemented by defining an appropriate reduction operator. The overall running time is $\mathcal{O}(\alpha \log t)$. For randomly permuted local inputs of size n/t, it is easy to show that our scheme yields a truthful estimator for the median, i.e., we get a result with expected rank $n/2$. We conjecture that similar quality bounds as from Dean et al. [DJW14] can be shown for our scheme even with $k = 2$, i.e., that we get rank $n/2(1 \pm cn^{-\gamma})$ with high probability for some constant c and γ.

Algorithm 7 Robust Hypercube Quicksort⁺

```
Input: A[1..n/t] data elements, t = 2^d number of PEs, PE number i
A ← randomly redistribute A                                              ▷ See Section 6.2.2
for j ← d − 1  down to 0 do                                              ▷ Iterate over cube dims
    s ← calculate splitter performing ⌈log t/(j+1)⌉ iterations of parallel ternary remedian ▷ See text
    if isEmpty(s) then return A                                          ▷ No elements in cube
    split A into L · R                                                   ▷ See text
    i' ← i ⊕ 2^j                                                         ▷ Communication partner
    if i' < i then
        send L to i' and receive R' from i'
        A ← concatenate R and R'
    else
        send R to i' and receive L' from i'
        A ← concatenate L and L'
Sort(A)
return A
```

We experimentally evaluate the median approximations of our binary remedian with $k = 2$ and the ternary remedian. We show that the binary remedian gives better median approximations than the ternary remedian and the rank can also be bounded by $n/2(1 \pm cn^{-\gamma})$ with better constants c and γ. Our benchmark runs both selection algorithms on uniformly distributed random integers in the range $[0, 2^{32} - 1]$ of up to 2^{20} elements. The input into the binary remedian is a power of two with nodes having two elements as input whereas the input into the ternary remedian is a power of three with nodes having three elements as input. We execute both algorithms 2 000 times for each input size. We measure the rank r for each output element and calculate the rank error

$$\left| \frac{r}{n-1} - \frac{1}{2} \right| .$$

Figure 6.2a shows the worst observed rank error. Compared to the ternary remedian, the binary remedian shows a smaller maximum rank error over 2 000 runs for all input sizes. Moreover, the rank error of the binary remedian is bounded by $1.44n^{-0.39}$ while the rank error of the ternary remedian is only limited by $2n^{-0.369}$. Figure 6.2b depicts the variance of the rank error for the binary remedian and the ternary remedian. For small input sizes, the variance of the binary tree median approximation is by a factor of two smaller. For large inputs, the difference in the variance even increases to a factor of three.

6.3 Robust Quicksort on Hypercubes

Previous implementations of hypercube-based quicksort (e.g., [Wag87; LM92; SMB13]) have three major drawbacks. First, they are not robust against duplicates as no tie-breaking is

performed. Second, the algorithms do not consider non-uniform input distributions. For hypercube-based routing algorithms, it is well known that skewed inputs exist where the intermediate imbalance factor increases to $\lfloor t^{1/2} \rfloor$ (see the discussion in Section 5.1.1). This bound even applies to unique input keys and true medians as a global splitter. Third, previous algorithms do not provide a fast high-quality splitter selection with median approximation guarantees. This is essential because, otherwise, quicksort gets impractical for large t as the load imbalance accumulates or the splitter selection dominates the running time for small inputs. Previous quicksort implementations use one of three different approaches to calculate the splitter. The first approach [Wag87] selects a random PE that just broadcasts its local median. The second approach [LM92] gathers the local median of each PE and calculates the median of medians. The third approach [SMB13] uses the median of a relatively large random sample.

In this section, we propose two versions of hypercube quicksort that overcome these problems. The first, *robust hypercube quicksort*$^+$ (RQuick$^+$), has a running time of $\mathcal{O}\left(\frac{n}{t}\log n + \beta\frac{n}{t}\log t + \alpha\log^2 t\right)$ with high probability. The second version, *robust hypercube quicksort* (RQuick), is a much simpler version for which we conjecture the same running time guarantee.

RQuick$^+$

Algorithm 7 provides pseudocode of RQuick$^+$. First, RQuick$^+$ randomly redistributes the input (see Section 6.2.2). The main loop goes through the dimensions $i \in [0, \ldots, \log t - 1]$ of the hypercube, highest dimension first. In the **splitter selection** step, we approximate the median of the data in each $i + 1$-dimensional subcube. The splitter selection performs the following six phases $\lceil \frac{\log t}{\log t - i} \rceil$ times and returns the best median approximation as the splitter candidate. (1) Let t' be the subcube size $t/2^i$ rounded up to the next power of three. Select t' samples with the parallel random sampling algorithm from Sanders et al. [San+18]. (2) Each sample element is sent to a random PE of the subcube with a direct message. (3) The elements are shuffled locally. (4) Enumerate the sample elements using a collective prefix sum. (5) Send the sample with rank j to PE $\lfloor j/3 \rfloor$ of the subcube with a direct message. (6) Use the shuffled samples as input to the ternary remedian algorithm by Rousseeuw and Bassett [RB90] (also see Section 6.2.3) to obtain a median approximation. In the **splitting step**, each PE partitions its local elements according to the splitter. In the last step, the **communication step**, PEs differing in bit j of their PE number exchange their data so that the PE with the 0-bit gets the elements smaller than s and the PE with the 1-bit gets elements larger or equal to s. For robustness against repeated keys, we use a low overhead greedy tie-breaking schema: Suppose that the 0-bit PE holds the sorted sequence a of the form $a = a_\ell \cdot s^u \cdot a_r$ and the 1-bit PE holds the sorted sequence b of the form $b = b_\ell \cdot s^v \cdot b_r$, i.e., the splitter s appears u respectively v times locally. We split a and b into four subsequences $a_\ell \cdot s^x$, $s^{u-x} \cdot a_r$, $b_\ell \cdot s^{v-y}$, and $s^y \cdot b_r$ with

$$x = \min(s^u, \max((|a| + |b|)/2 - a_\ell - b_\ell), 0)$$

and

$$y = \min(s^v, \max((|a| + |b|)/2 - a_r - b_r), 0))$$

and send $s^{u-x} \cdot a_r$ respectively $b_\ell \cdot s^{v-y}$ to the communication partner. This minimizes the load of the two PEs after the data exchange while simultaneously minimizing the communication volume between these PEs.

Theorem 6.6 (Running time of RQuick⁺)

RQuick$^+$ runs in time $\mathcal{O}\left(\frac{n}{t}\log n + \beta\frac{n}{t}\log t + \alpha\log^2 t\right)$ with probability $\geq 1 - t^{-c}$ for any constant $c > 0$ and inputs without duplicated keys.

The proof of Theorem 6.6 is structured as follows. **First**, we consider the imbalance introduced by the median approximation. This starts with Lemma 6.7 that limits the imbalance introduced by splitting the data of a subcube. Lemma 6.7 leads to Lemma 6.8 bounding the maximum load of subcubes. **Second**, we limit the load of PEs within subcubes. Here, the initial random shuffling is crucial. A result of the random shuffling is that the elements of each subcube are randomly distributed to its PEs, independent of the splitter quality. This is shown in Lemma 6.9. Lemma 6.10 uses this property and the bounded imbalance between subcubes (see Lemma 6.7) to limit the individual load of the PEs. **Finally**, we prove Theorem 6.6 with Lemma 6.10. In the case the splitter element occurs repeatedly, Lemma 6.9 also helps our simple local tie-breaking heuristic—each PE has a random sample of the global data of its subcube and, hence, our local balancing approximates a global balancing.

For the proof of Theorem 6.6, we oftentimes consider an algorithmic property that is valid with probability $\geq 1 - t^{-c}$ for any constant $c > 0$. In this case, we say that *the property is valid with probability* $1 - t^{-\Omega(c)}$. Note that we can use $\mathcal{O}(t)$ different properties that are each valid with probability $1 - t^{-\Omega(c)}$. All these properties are then again valid with probability $1 - t^{-\Omega(c)}$. For simplicity, we assume that $n \geq t$. For the proof, we assume that the input does not have any duplicated keys.

Lemma 6.7 (Quality of the splitter selection)

With probability $1 - t^{-\Omega(c)}$, there exists a constant $j \in \mathbb{N}$ for which no subcube of dimension $i \geq j$ of RQuick$^+$ introduces an imbalance factor of more than $1 + 2^{-0.369i}$.

Proof. First, we consider the splitter selection of an i-dimensional subcube. Let $j \in \mathbb{N}$ be an appropriate large constant and let $i \geq \log j$. We denote the execution of a single ternary remedian run as *failed* if the data of the subcube is split into two buckets with an imbalance factor larger than $1 + 2^{-\gamma i}$ for $\gamma = -0.369$. For the proof, we use 2^i rather than the actual sample size. This is an underestimate of the sample size and thus, the splitter quality. According to Lemma 6.5, the failure probability is $\geq 2^{-ci}$ for any constant $c > 0$ and a sufficiently large number of PEs. Let j be this number. Thus, the size 2^i of the current subcube is sufficiently large. Since we execute $\lceil \frac{\log t}{i} \rceil$ remedian repetitions, at least one remedian execution does not fail with probability

$$\leq 1 - \left(2^{-ci}\right)^{\lceil \frac{\log t}{i} \rceil} \leq 1 - \left(2^{-ci}\right)^{\frac{\log t}{i}} = 1 - \left(\left(2^i\right)^{\log t}\right)^{-\frac{c}{i}} = 1 - \left(t^i\right)^{-\frac{c}{i}} = 1 - t^{-c} \; .$$

Thus, the subcube does not introduce an imbalance factor of more than $1 + 2^{-\gamma i}$ with probability $1 - t^{-\Omega(c)}$.

With probability $1 - t^{-\Omega(c)}$, no subcube of dimension $i \geq j$ introduces an imbalance factor of more than $1 + 2^{-\gamma i}$ since the sorting algorithm has only $t - 1$ subcubes. □

Lemma 6.8 (Maximum load of subcubes in RQuick⁺)

With probability $1 - t^{-\Omega(c)}$*, there is no i-dimensional subcube of RQuick⁺ that has more than* $\mathcal{O}(2^i \max(\frac{n}{t}, \log t))$ *elements.*

Proof. After the initial shuffling, each PE stores $\mathcal{O}(\max(n/t, \log t))$ elements with probability $1 - t^{-\Omega(c)}$ (see Lemma 6.3 in Section 6.2.2). Thus, with the same probability, each i-dimensional subcube stores $\mathcal{O}\big(2^i \max(n/t, \log t)\big)$ elements. We now show that the additional load imbalance introduced by each step of the main loop can be limited by a constant factor.

According to Lemma 6.7, the last $\mathcal{O}(1)$ iterations of the main loop only introduce a constant imbalance. Since these iterations introduce a constant imbalance, we only have to consider the remaining iterations. With probability $1 - t^{-\Omega(c)}$, the imbalance factor introduced by an i-dimensional subcube is bounded by $1 + 2^{-\gamma i}$ with $\gamma = 0.369$ for the remaining iterations (also see Lemma 6.7). Now we can bound the imbalance factor I of the remaining iterations to

$$\begin{aligned} I &\leq \prod_{i<\log t} \left(1 + 2^{-\gamma i}\right) = e^{\ln \prod_{i<\log t}\left(1+2^{-\gamma i}\right)} \\ &= e^{\sum_{i<\log t} \ln\left(1+2^{-\gamma i}\right)} \leq e^{\sum_{i<\log t} 2^{-\gamma i}} \\ &= e^{\sum_{i<\log t} (2^{-\gamma})^i} \leq e^{\frac{1}{1-2^{-\gamma}}} = \mathcal{O}(1) \ . \end{aligned}$$

The first "≤" is used since we consider all iterations, the second "≤" uses the Taylor series development of ln, and the third "≤" uses a geometric sum. □

Lemma 6.9 (Placement of elements in a subcube of RQuick⁺)

Before step i of RQuick⁺'s main loop, the elements of a PE-subcube are randomly distributed to its PEs.

Proof. Consider an arbitrary input element e. Let e be stored on PE j of a subcube C with $t/2^i$ PEs before step i. Then, e had to be stored on a PE $j' \in [j2^i .. (j+1)2^i)$ when the main loop of RQuick⁺ started. Since the initial random shuffling had moved e to a random PE $j' \in [0..t)$ (see Lemma 6.2 in Section 6.2.2), PE j must also be a random PE of subcube C. □

Lemma 6.10 (Maximum load of PEs in RQuick⁺)

Each PE has $\mathcal{O}(\max(n, t\log t)/t)$ *elements after each step of the main loop of RQuick⁺ with probability* $1 - t^{-\Omega(c)}$.

Proof. Each subcube stores $\mathcal{O}(\max(\log t, n/t)/2^{i+1})$ elements with probability $1 - t^{-\Omega(c)}$ after step $i \in [0 .. \log t)$ of the main loop (see Lemma 6.8). The subcubes contain $t/2^{i+1}$ PEs each. Lemma 6.9 tells us that the elements of a subcube are randomly distributed among the PEs with probability $1 - t^{-\Omega(c)}$. Thus, the number of elements on the PEs of a subcube can be described by the balls into bins problem with $\mathcal{O}(\max(t\log t, n)/2^{i+1})$ balls and $t/2^{i+1}$ bins. From Lemma 6.1 in Section 6.2.1 follows that each PE of a subcube has at most $\mathcal{O}(\max(\log t, n/t))$ elements with

Algorithm 8 Robust Hypercube Quicksort

```
Input: A[1..n/t] data elements, t = 2^d number of PEs, PE number i
A ← randomly redistribute A                                   ▷ See Section 6.2.2
SORT(A)
for j ← d − 1 down to 0 do                                    ▷ Iterate over cube dimensions
    m ← median of A
    s ← calculate splitter with binary remedian using m      ▷ See Section 6.2.3
    if ISEMPTY(s) then return A                               ▷ No elements in cube
    split A into L · R according to splitter s                ▷ See text
    i' ← i ⊕ 2^j                                               ▷ Communication partner
    if i' < i then
        send L to i' and receive R' from i'
        A ← merge R with R'
    else
        send R to i' and receive L' from i'
        A ← merge L with L'
return A
```

probability $1 - t^{-\Omega(c)}$. This probability bound also holds for all subcubes since the hypercube has only $2t - 1$ subcubes. □

We finally come back to Theorem 6.6, i.e., the total running time

$$\mathcal{O}\Big(\frac{n}{t}\log n + \beta\frac{n}{t}\log t + \alpha\log^2 t\Big) \ .$$

Proof of Theorem 6.6. The first term covers the time for local sorting. The second and third term is an upper bound for the random shuffling (see Lemma 6.4 in Section 6.2.2). For this bound, we use that $\beta\log^2 t \in \mathcal{O}(\alpha\log^2 t)$. The second and third term is also an upper bound for the main loop of RQuick$^+$: In step i of the main loop, RQuick$^+$ executes $\lceil\frac{\log p}{\log p-i}\rceil$ rounds of splitter refinement. In each round, the sampling, the sample shuffling, and the collective prefix sum require time $\mathcal{O}(\alpha(\log p - i))$ each (see Lemma 6.1 for an upper bound for the sample shuffling). Additionally, Lemma 6.10 tells us that the local load of the main loop of RQuick$^+$ is $\mathcal{O}(\max(n, t\log t)/t)$. Thus, the data exchange and the partitioning runs in time $\mathcal{O}(\beta n/t\log t + \alpha\log^2 t)$. Again, we use that $\beta\log^2 t \in \mathcal{O}(\alpha\log^2 t)$. All bounds hold with probability $1 - t^{-\Omega(c)}$ since each bound holds with probability $1 - t^{-\Omega(c)}$. □

The startup overhead $\alpha\log p$ for shuffling is a lower order term compared to $\mathcal{O}(\log^2 p)$ for the main loop. Additionally, the cost of data transfer and local work for the shuffling phase is covered by the startup overhead of the main loop for $n \in \mathcal{O}(\alpha/\beta p\log p)$. For larger inputs, the cost of data transfer and local work for shuffling is still a logarithmic factor smaller compared to the cost of bitonic sort.

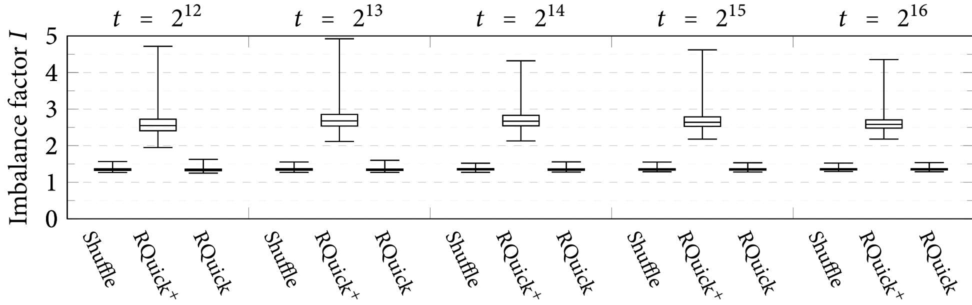


Figure 6.3: Distribution of the random shuffling (Shuffle), RQuick's and RQuick^{+}'s measured imbalance factors obtained from 2^{16} runs each.

RQuick

For RQuick, we use the binary remedian, which we conjecture to be more accurate than the ternary remedian algorithm used by RQuick^{+}. Additionally, we take advantage of the observation that in each iteration of the main loop of RQuick^{+}, the local data of the PEs is a random sample of their subcube's data (see Lemma 6.9). This is a consequence of the initial random shuffling. The idea now is to use the medians of the PEs' local data as input to the binary remedian since these medians are already very good approximations of the global median, especially compared to the samples used in RQuick^{+}. In return, RQuick only executes one round of median approximation. Overall, the splitter selection overhead for RQuick is significantly smaller than for RQuick^{+} since the splitter selection of RQuick boils down to propagating two elements from the leaf PEs of a binomial tree to its root and one element back to the leaves.

Algorithm 8 provides pseudocode for RQuick. First, RQuick randomly redistributes the input (see Section 6.2.2). Then the data is sorted locally. The main loop goes through the dimensions of the hypercube, highest dimension first. In each j-dimensional subcube, we use the binary remedian algorithm from Section 6.2.3 with $k = 2$ to calculate a splitter element s. Each PE provides its local median as input to the selection algorithm. In a communication step, PEs differing in bit j of their PE number exchange their data so that the PE with the 0-bit gets the elements smaller than s and the PE with the 1-bit gets elements larger or equal to s. These two sequences are then merged. For robustness against repeated keys, we use the low overhead greedy tie-breaking schema from RQuick^{+}.

Now, we evaluate the imbalance factor of the random shuffling, RQuick, and RQuick^{+} experimentally. The imbalance factor of a run is denoted as the maximum output imbalance of the PEs relative to the average load n/p. The imbalance of RQuick and RQuick^{+} is induced by the splitter selection quality—the better the median approximation quality, the smaller the imbalance between subcubes. After the last iteration of the main loop, we have one PE per

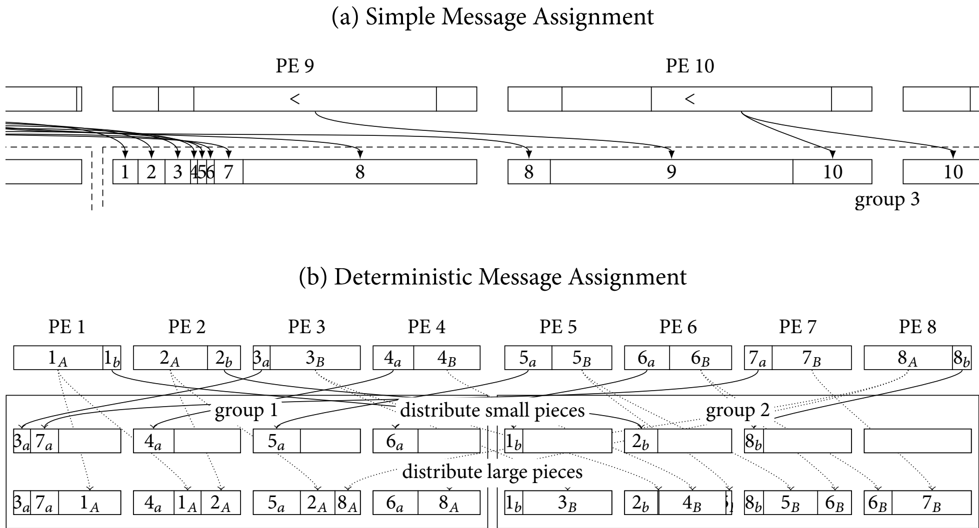


Figure 6.4: Illustration of the different message assignments.

subcube and the highest imbalance between subcubes. We see the intermediate imbalance factor of the initial random shuffling as the base line for the sorting algorithms. The box plot in Figure 6.3 depicts the distribution of the imbalance factors obtained by experiments with different numbers of PEs and $10t \log t$ elements. According to Figure 6.3, the imbalance factor distribution of initial random shuffling and RQuick are about the same. However, the imbalance of RQuick is always significantly smaller than the imbalance of RQuick$^+$, regardless of the measures minimum, maximum, median, first quartile, and third quartile. Thus, we conjecture that RQuick and RQuick$^+$ have the same asymptotic running time since the running time of RQuick$^+$ is already the lower bound for hypercube quicksort.

Conjecture 6.11 (Running time of RQuick)
RQuick runs in time $\mathcal{O}\left(\frac{n}{t}\log n + \beta\frac{n}{t}\log t + \alpha\log^2 t\right)$ with probability $\geq 1 - t^{-c}$ for any constant $c > 0$ and inputs without duplicated keys.

6.4 Delivering k Data Partitions to k PE-Groups

In the recursive multiway sorting algorithms considered here, we face the following data redistribution problem: Each PE has partitioned its n/t local elements into k pieces. The total size of pieces with number i is n/k. Pieces with number i have to be moved to PE-group i that consists of PEs $[it/k..(i+1)t/k)$. Each PE in a group should receive the same amount of data except for rounding issues. To simplify the discussion, we describe the algorithm for the case that every PE receives exactly n/t elements. We now want to compute a message assignment that minimizes the number of incoming and outgoing messages.

Algorithm 9 Computation of the Simple Message Assignment

Input: i current PE, t PEs, k PE-groups, $m = \{m_0, \dots, m_{k-1}\}$ number of elements for each group, a array of local elements
$t, l, s \in \mathbb{N}^k$
$t_j \leftarrow \sum_{x \in [0..t)} m_j$ @ PE x ▷ Global number of elements for group j (collective all-reduce)
$l_j \leftarrow \sum_{x \in [0..i)} m_j$ @ PE x ▷ Number of elements for group j on preceding PEs (collective excl. prefix sum)
for $w \in [0..k)$ **do** ▷ Calculate messages for PEs in group w
  $o \leftarrow \lceil t_w k/t \rceil$ ▷ Maximum number of elements received by PEs of group w
  if $m_w = 0$ **then**
    continue ▷ No elements for group w
  $b \leftarrow l_w / o$ ▷ First PE of group w that gets a message from PE i
  $e \leftarrow (l_w + m_w - 1)/o$ ▷ Last PE of group w that gets a message from PE i
  for $q \in [b..e]$ **do**
    Assign elements $a[\max(0, ot - l_w)..min((t+1)o - l_w, m_w))$ to PE $wt/k + t$

Simple Approach. We begin with a *simple message assignment* (SMA) [KK93; GV94; SW11] and then refine the assignment in order to handle bad cases. Algorithm 9 provides pseudocode. The basic idea is to compute a prefix sum over the piece sizes—this is a collective prefix sum with vector length k. As a result, each piece is labeled with a range of positions within the group it belongs to. Positions are numbers between 0 and $n/k - 1$. An element with number j in group i is sent to PE $it/k + \lfloor jt/n \rfloor$. This way, each PE sends and receives exactly n/t elements. Moreover, each piece is sent to one or two target PEs responsible for handling it in the recursion. Thereby, each PE *sends* at most $2k$ messages for the data exchange. Unfortunately, the number of *received messages*, although the same on average, may vary widely in the worst case. There are inputs where some PEs have to *receive* $\Theta(t)$ very short messages. This happens when many consecutively numbered PEs provide only very small pieces of data (see PE 9 in the top of Figure 6.4a). The SMA algorithm takes $\mathcal{O}(\alpha + \beta k)$ time and the data exchange step needs $\mathrm{Exch}(t, n/t, t)$ time in the worst case.

Reduce Message Transfers. We propose the *deterministic message assignment* (DMA), an assignment that limits the number of sent and received messages to $\mathcal{O}(k)$. Figure 6.4b illustrates the process. The basic idea is to distribute small and large pieces separately. We consider a piece to be small if it contains less than $n/(2tk)$ elements. Otherwise, the piece is large. Algorithm 11 provides pseudocode for the calculation of the message assignment.

The message assignment algorithm performs four steps. In step one, the description of the pieces is sent to their target group: For each group, we enumerate the small pieces respectively large pieces with two collective prefix sums. Overall, this takes $\mathcal{O}(\alpha \log t + \beta k)$ time. The result is a global ranking of the pieces. Then, PE i sends the description of its large respectively small piece with rank g destined for group j to PE $\lfloor g/k \rfloor$ respectively PE $(g \bmod k)$ of that group. This can be done in time $\mathrm{Exch}(t, \mathcal{O}(k), k)$. Also, each PE receives at most k small pieces. In the next two steps, each group independently produces an assignment of the received pieces. In step two, **each PE assigns the small pieces to itself** and subtracts the piece sizes from its

Algorithm 10 DMA compare function

```
function lessThan(l, r)
    if isEqualPieceType(l, r) then
        return l.i < r.i
    else if isResidualPiece(l) then
        return l.b ≤ r.b
    else
        return l.b < r.b
```

initial capacity n/t. This way, all small pieces are assigned without having to split them—each PE sends at most k small pieces and each PE receives at most k small pieces. Also, no receiving PE gets assigned more than half of its final load. In step three, the large pieces are assigned taking the residual capacity of the PEs into account. The assignment of large pieces to group j is collectively calculated by the PEs of that group. After step three, each PE has calculated $\mathcal{O}(k)$ assignments. Also, the pieces that have been initially delivered by a PE were assigned to $\mathcal{O}(k)$ PEs. In step four, the assignments are sent back to the initial PEs. This can be done in time $\mathrm{Exch}(t, \mathcal{O}(k), k)$. In the following, we describe step three, the assignment process of large pieces for a single group.

Conceptually, we enumerate the unassigned elements of the large pieces on the one hand and the unassigned element slots provided by the residual capacities on the other hand and then map element i to slot i.[1] To implement this, we compute a collective prefix sum of the residual capacities of the receiving PEs on the one hand and the sizes of the unassigned large pieces on the other hand. This yields two sorted sequences C and U respectively. On the one hand, each group PE i contributes to C one *residual piece* $(i, b, e) = c_i$ which indicates that this PE provides slots $[b \mathrel{..} e)$. On the other hand, the i-th *large piece* $(j, b, e) = u_i$ of U indicates that the source PE j of this piece will send elements with indices $[b \mathrel{..} e)$. The sequence C is then merged into sequence U. Our merge operation guarantees that the pieces from U remain on their PEs. Algorithm 10 depicts the comparator function. Roughly speaking, a residual piece c is smaller than a large piece u if and only if the first slot of c is smaller than or equal to the first element index of u. In the merged sequence, a subsequence of the form $\langle c_i, u_j, \ldots, u_{j+r}, c_{i+1}, z\rangle$ induces the following **element assignment**: The source PE of large piece $u \in [u_j \mathrel{..} u_{j+r}]$ has to send $|[u.b \mathrel{..} u.e) \cap [c_i.b \mathrel{..} c_i.e)|$ elements to group PE i. Piece $j + r$ may also wrap over to PE $i + 1$, and possibly to PE $i + 2$ if $z = c_{i+2}$. No further wrapping over of large piece u_{j+r} is possible since no piece can be larger than n/t and since every PE has a residual capacity of at least $n/(2t)$. Thus, **large pieces are assigned to a constant number of group PEs**. Similarly, **no PE gets assigned more than $\frac{n}{t} \cdot 2tk/n = 2k$ parts of large pieces** since large pieces have at most size n/t and since each PE has a residual capacity of at least $n/(2t)$.

Before we calculate the message assignment of large pieces as described above, residual piece c_i must be available on the PEs that store large pieces $[u_j \mathrel{..} u_{j+r-1}]$. Also, residual pieces c_i, c_{i+1}, and possibly z must be available on the PE that stores large piece u_{j+r}. We show that it

[1] A similar approach to data redistribution is described in [HS16]. However, here we can exploit special properties of the input to obtain a simpler solution that avoids segmented gather and scatter operations.

is sufficient to greedily send each residual piece to the preceding PE and the two succeeding PEs. This operation only requires time $\mathcal{O}(\alpha)$. In order to make the desired residual pieces available on the particular PEs, we take advantage that we merged sequence C into sequence U and that the pieces from U remained on their PEs. From this follows that residual pieces c_{i+1} and c_{i+2} may need to be sent back to the preceding PE since piece u_{j+r} is either stored on the PE holding these two residual pieces or on the preceding PE. Also, piece c_i may have to be forwarded to the next two PEs since pieces $[u_j \mathinner{..} u_{j+r-1}]$ can only be stored on the PE that holds piece c_i and on the two succeeding PEs. This is due to two reasons: On the one hand, the number of slots of piece c_i, and thus the total size of the pieces $[u_j \mathinner{..} u_{j+r-1}]$, is at most n/t. On the other hand, step one of the DMA algorithm guarantees that the left PEs of a PE-group store k large pieces each of size $\geq n/(2kt)$, followed by one PE that stores up to k pieces, followed by potential PEs without any large pieces.

The difficult part is to merge sequence C into sequence U efficiently. Here one can adapt and simplify the work efficient parallel merging algorithm for EREW PRAMs from Hagerup and Rüb [HR89]. Essentially, one first merges the t/k pieces of C with a compressed representation of U—each PE fuses its local large pieces of U into a single *super piece*. This merging operation can be done in time $\mathcal{O}(\alpha \log(t/k))$ using Batcher's merging network [Bat68] since each PE provides two pieces. Then each residual piece e of C has to be merged into $\leq k$ large pieces of U on one particular PE to obtain the merged sequence of C and U: For each piece e, we determine its particular PE by counting the number of super pieces to the left of e in the merged sequence. The counts can be obtained with a prefix sum in time $\mathcal{O}(\alpha \log(t/k))$. We then route the residual pieces to the particular PEs. Note that each PE will have to merge only $\mathcal{O}(k)$ residual pieces since it is impossible that the local large pieces (of total size $\leq kn/t$) fill more than $2k$ PEs (each providing residual capacity $> n/(2t)$). We also note that each PE sends at most two residual pieces. Thus, we can transfer the residual pieces to the appropriate PEs in time $\mathrm{Exch}(t, \mathcal{O}(k), \mathcal{O}(k))$. We now have everything at hand so that the PEs can merge the residual pieces into their large pieces. This can be done using local merging in time $\mathcal{O}(k)$ if we sort the received residual pieces with counting sort [Knu73]. In other words, the special properties of the considered sequences make it unnecessary to perform the contention resolution measures that make the algorithm from Hagerup and Rüb [HR89] somewhat complicated. Overall, we get the following deterministic result.

Theorem 6.12 (Data redistribution with the deterministic message assignment)
Data redistribution of $k \times t$ pieces to k groups can be implemented to run in time

$$\widetilde{\mathrm{Exch}}(t, \frac{n}{t}, \mathcal{O}(k)) \ .$$

Proof. We first consider the time to calculate the message assignment. Summing up the time we have accounted for the message assignment algorithm, we get time $\widetilde{\mathrm{Exch}}(t, \mathcal{O}(k), \mathcal{O}(k))$. Recall from Section 5.2 that $\widetilde{\mathrm{Exch}}(t, \cdot, k)$ absorbs terms of the form $\mathcal{O}(\alpha \log t + \beta k)$.

We now consider the time for the actual data exchange. Each PE sends and receives n/t elements since each PE provides n/t input elements and an initial capacity of n/t. The message assignment splits each large piece into at most three messages—small pieces are not split at all. This sums up to $\mathcal{O}(k)$ outgoing messages per PE. Each PE gets assigned at most k small

pieces of total size $\leq n/(2t)$ in step two of the message assignment. In step three, a maximum of $2k+1$ large pieces is assigned to each PE to cover its residual capacity—one piece may wrap over from the previous PE, followed by pieces of size $\geq n/(kt)$, and a final piece may wrap over to the next PE. Thus, the data exchange can be done in time $\mathrm{Exch}(t, \frac{n}{t}, \mathcal{O}(k))$.

Overall, the data redistribution takes time $\widetilde{\mathrm{Exch}}(t, \frac{n}{t}, \mathcal{O}(k))$ (recall from Section 5.2 that $\widetilde{\mathrm{Exch}}(t, \cdot, k)$ absorbs terms of the form $\widetilde{\mathrm{Exch}}(t, k, k)$). □

For applications with slightly imbalanced data like AMS-sort, some groups may receive up to $(1+\epsilon)n/t$ elements, some groups may receive much less than n/k elements, and k may not divide t evenly. We address this by adjusting residual capacities of all PEs to $(1+\epsilon)n/t$, ensuring that all PE-groups have a size of $\lfloor t/k \rfloor$ or $\lceil t/k \rceil$, and by carefully implementing corner cases. The data redistribution then takes time $\widetilde{\mathrm{Exch}}(t, (1+\epsilon)\frac{n}{t}, \mathcal{O}(k))$.

Algorithm 11 Deterministic Message Assignment

```
procedure DMA(n, t, k, m, i)
    Input: n total input size, t PEs, k PE-groups, m = [m_o .. m_k) number of elements for
            each group, i my PE index within my PE-group
    π ← enumerateSmallPieces(b)          ▷ Vectorized exclusive collective prefix sum
    Π ← enumerateLargePieces(b)                                          ▷ π, Π ∈ ℕ^k
    for j ∈ [0 .. t/k) do           ▷ Route pieces to PE-groups in time Exch(t, O(k), O(k))
        if m_j < n/(2tk) then Send small piece of size m_j to PE ⌊π_j/k⌋ of group j
        else Send large piece of size m_j to PE (Π_j mod k) of group j
    ⟨S, L⟩ ← receiveSmallAndLargePieces()         ▷ Pieces store source PE and size
    a ← calculateAssignmentToGroup(n, t, k, S, L, i)
    Send assignment a to requesters                          ▷ Exch(t, O(k), O(k))
    return receiveAssginment()
procedure calculateAssignmentToGroup(n, t, k, S, L, i)       ▷ Executed by PE-group
    a ← AssignPiecesToMyself(S)               ▷ Returns assignment of small pieces
    c ← n/t − ∑_{s∈S} size(s)                               ▷ Own residual capacity
    c' ← ∑_{j=0}^{i} c @ PE j   ▷ Residual capacity of preceding PEs (collective excl. prefix sum)
    C ← ⟨i, c', c' + c⟩                                            ▷ Residual piece
    u ← ∑_{l∈L} size(l)                             ▷ Total size of local large pieces
    μ ← ∑_{j=0}^{i−1} u @ PE j        ▷ Collective exclusive prefix sum over large piece sizes
    S ← ⟨i, μ, μ + ∑_{l∈L} size(l)⟩                              ▷ Create super piece
    for each l ∈ L do                  ▷ Create prefix representation of large pieces
        U ← U ∪ ⟨source(l), μ, μ + size(l)⟩
        μ ← μ + size(l)
    ⟨X, Y⟩ ← distributedMerge(C, S)         ▷ Merge sorted residual and super pieces
    γ ← countSuperPieces(X, Y)
    o ← ∑_{j=0}^{i−1} γ    ▷ Number of super pieces on preceding PEs (collective excl. prefix sum)
    if isResidualPiece(X) then           ▷ Bring residual pieces and large pieces together
        Send X to PE [o − 1 .. o + 2]
        if isResidualPiece(Y) then Send Y to PE [o .. o + 3]
    else
        if isResidualPiece(Y) then Send Y to PE [o − 1 .. o + 2]
    C ← receiveResidualPieces()                              ▷ Exch(t, O(k), O(k))
    return a ∪ createLargePieceAssignment(C, U)      ▷ Merge pieces and calculate
                                                       intersections
```

6.5 Generalizing Multiway Mergesort (RLM-sort)

In this section, we describe ***R**ecursive **L**ast Multiway **M**erge**sort*** (RLM-sort). RLM-sort is the first recursive multiway sorting algorithm that we propose in this thesis. Compared to our second recursive multiway algorithm AMS-sort, which we describe in the next section, RLM-sort simplifies several discussions since RLM-sort guarantees perfect load balancing. RLM-sort subdivides the PEs into "natural" groups of size t' on which it to recurses. Asymptotically, $k := t/t'$ around $t^{1/r}$ is a good choice if we want to execute r levels of recursion. However, we may also fix t' based on architectural properties. For example, in a cluster of many-core machines, we might choose t' as the number of cores in one node. Similarly, if the network has a natural hierarchy, we will adapt t' to that situation. For example, if PEs within a rack are more tightly connected than inter-rack connections, we may choose t' to be the number of PEs within a rack. Other networks, e.g., meshes or tori have less pronounced cutting points. However, it still makes sense to map groups to subnetworks with nice properties, e.g., nearly cubic subnetworks. For simplicity, we will assume that t is divisible by t', and that $k = \Theta\left(t^{1/r}\right)$.

There are several ways to define recursive multiway mergesort. We describe a method we call "recurse last" (see Figure 6.5) that needs to communicate the data only r times and avoids problems with many small messages. Every PE sorts locally first. Then each of these t sorted sequences is partitioned into k pieces in such a way that the sum of these piece sizes is n/k for each of these k resulting *parts*. In contrast to the single-level algorithm, we run only k multisequence selections in parallel and thus reduce the bottleneck due to multisequence selection by a factor of t'.

Now we have to move the data to the responsible groups. We refer to Section 6.4 which shows how this is possible using time $\widetilde{\mathrm{Exch}}(t, \frac{n}{t}, \mathcal{O}(t^{1/r}))$.

Afterwards, group i stores elements that are no larger than any element in group $i+1$ and it suffices to recurse within each group. However, we do not want to ignore the information already available—each PE does not store an entirely unsorted array but a number of sorted sequences. This information is easy to use though—we merge these sequences locally first and obtain locally sorted data, which can then be subjected to the next round of splitting. Algorithm 12 provides pseudocode of the RLM-sort algorithm.

Theorem 6.13 (Running time of RLM-sort with $\mathcal{O}(1)$ recursion levels)
RLM-sort with $r = \mathcal{O}(1)$ levels of recursion can be implemented to run in time

$$\mathcal{O}\Big(\big(\alpha \log t + t^{1/r}\,\beta + t^{1/r}\,\log \tfrac{n}{t} + \tfrac{n}{t}\big)\log n\Big) + \sum_{i=1}^{r} \widetilde{\mathrm{Exch}}\left(t^{i/r}, \tfrac{n}{t}, \mathcal{O}\left(t^{1/r}\right)\right) \quad . \tag{6.3}$$

Proof. (Outline) Local sorting takes time $\mathcal{O}\left(\frac{n}{t}\log n\right)$. For $r = \mathcal{O}(1)$ multiselections we get the bound $\mathcal{O}\left(\left(\alpha \log t + k\beta + k \log \frac{n}{t}\right)\log n\right)$ from Equation (5.1). Summing the latter two contributions, we get the first term of Equation (6.3).

In level i of the recursion we have k^i independent groups containing $t/k^i = t/t^{i/r} = t^{1-i/r}$ PEs each. An exchange within the group in level i costs $\widetilde{\mathrm{Exch}}(t^{1-i/r}, \frac{n}{t}, \mathcal{O}(t^{1/r}))$ time. Since all

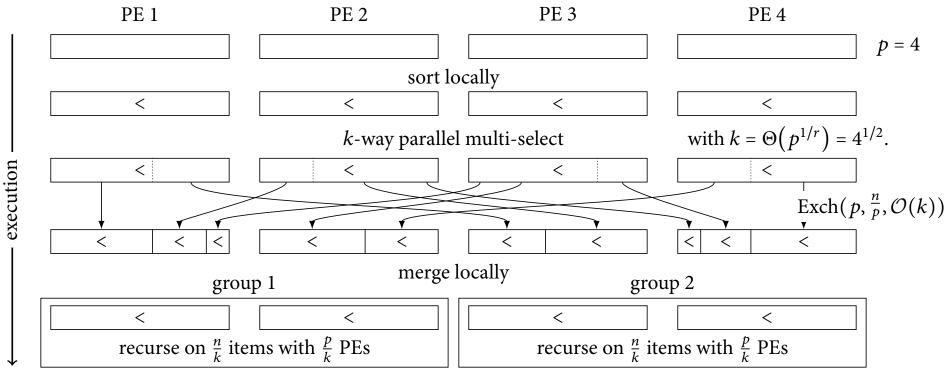


Figure 6.5: Algorithm schema of Recurse Last Parallel Multiway Mergesort.

independent exchanges are performed simultaneously, we only need to sum over the r recursive levels, which yields the second term of Equation (6.3). □

Equation (6.3) is a fairly complicated expression but using some reasonable assumptions we can simplify the equation. If all communications are equally expensive, the sum becomes $r\,\widetilde{\text{Exch}}(t, \frac{n}{t}, \mathcal{O}(t^{1/r}))$, i.e, we have r message exchanges involving all the data but we limit the number of startups to $\mathcal{O}(t^{1/r})$. On the other hand, on mesh or torus networks, the first (global) exchange will dominate the cost and we get $\widetilde{\text{Exch}}(t, \frac{n}{t}, \mathcal{O}(t^{1/r}))$ for the sum. If we also assume that data is delivered directly, $\Omega(t^{1/r})$ startups hidden in the $\widetilde{\text{Exch}}()$ term will dominate the $\mathcal{O}(\log^2 t)$ startups in the remaining algorithm. We can assume that n is bounded by a polynomial in t—otherwise, a traditional single-phase multiway mergesort would be a better algorithm. This implies that $\log n = \Theta(\log t)$. Furthermore if $n = \omega(t^{1+1/r} \log t)$ then $n/t = \omega(t^{\frac{1}{r}} \log t)$ and the term $\Omega(\beta \frac{n}{t})$ hidden in the data exchange term dominates the term $\mathcal{O}(\beta t^{\frac{1}{r}} \log n)$. Thus Equation (6.3) simplifies to $\mathcal{O}(\frac{n}{t} \log n)$ (essentially the time for internal sorting) plus the data exchange term.

If we also assume α and β to be constants and estimate $\widetilde{\text{Exch}}$-term as $\mathcal{O}(\frac{n}{t})$, we get execution time

$$\mathcal{O}\Big(t^{1/r} \log^2 t + \frac{n}{t} \log n\Big) \ .$$

From this, we can infer $\mathcal{O}(t^{1+1/r} \log t)$ as isoefficiency function.

6.6 Adaptive Multi-Level Samplesort (AMS-sort)

A good starting point is the recursive multiway samplesort algorithm by Gerbessiotis and Valiant [GV94]. However, they use centralized sorting of the sample and their data redistribution may lead to some PEs receiving $\Omega(t)$ messages (see also Section 6.4). We improve this algorithm in several ways to achieve a truly scalable algorithm. First, we sort the sample using

Algorithm 12 RLM-sort

```
Input: A[1..n/t] data elements, t PEs, k PE-groups
A' ← SORT(A)
return RLM-RECURSIVE(A', t, k)

function RLM-RECURSIVE(A, t, k)
    Input: A[1..n/t] data elements, t PEs, k PE-groups
    L ∈ ℕ^k ← temp. array of local bucket sizes induced by equidistant ranks
    G ∈ ℕ^k ← temp. array of global bucket sizes induced by equidistant ranks
    if t = 1 then
        return A
    for i ← [1..k) do vectorized
        L[i] ← MULTISEQUENCESELECTION(A, n/k · i) ▷ Select splitter by global rank n/k · i
        G[i] ← n/k · i
    M ← MESSAGEASSIGNMENT(L, G, k)   ▷ Assign local (sub)buckets to PEs of PE-groups
    M' ← DATAEXCHANGE(A, M)          ▷ Send messages M and receive messages M'
    A' ← MERGE(M')                   ▷ Merge received messages
    return RLM-RECURSIVE(A', t/k, k)
```

fast parallel sorting. Second, we use the advanced data exchange algorithms described in Section 6.4, and third, we give a scalable parallel adaptation of the idea of overpartitioning [LS94] in order to reduce the sample size needed for good load balance and a smaller memory footprint on the nodes. We state the lemmas and theorems in this section without proofs. For details, we refer to our initial publication of AMS-sort [Axt+15a].

But back to our version of recursive multiway samplesort (see Figure 6.6). As in RLM-sort, our intention is to split the PEs into k groups of size $t' = t/k$ each, such that each group processes elements with consecutive ranks. To achieve this, we **choose a random sample** of size abk where the *oversampling factor a* and the *overpartitioning factor b* are tuning parameters. The **sample is sorted** using a fast sorting algorithm. We assume the fast inefficient algorithm from Section 6.1. Its execution time is $\mathcal{O}\left(\frac{abk}{t}\log\frac{abk}{t} + \beta\frac{abk}{t^{1/2}} + \alpha\log t\right)$.

From the sorted sample, we **choose $bk - 1$ splitter** elements with equidistant rank. These splitters are broadcast to all PEs. This is possible in time $\mathcal{O}(\beta bk + \alpha\log t)$. Then every PE **partitions its local data** into bk *buckets* corresponding to these splitters. This takes time $\mathcal{O}\left(\frac{n}{t}\log(bk)\right)$.

Using a collective all-reduce, we then determine global bucket sizes in time $\mathcal{O}(\beta bk + \alpha\log t)$. These can be used to **assign buckets to PE-groups** in a load-balanced way: Given an upper bound L on the number of elements per PE-group, we can scan through the array of bucket sizes and skip to the next PE-group when the total load would exceed L. Using binary search on L, this finds an optimal value for L in time $\mathcal{O}(bk\log n)$ using a sequential algorithm.

Lemma 6.14 (Optimality of overpartitioning)

The above binary search scanning algorithm indeed finds the optimal L.

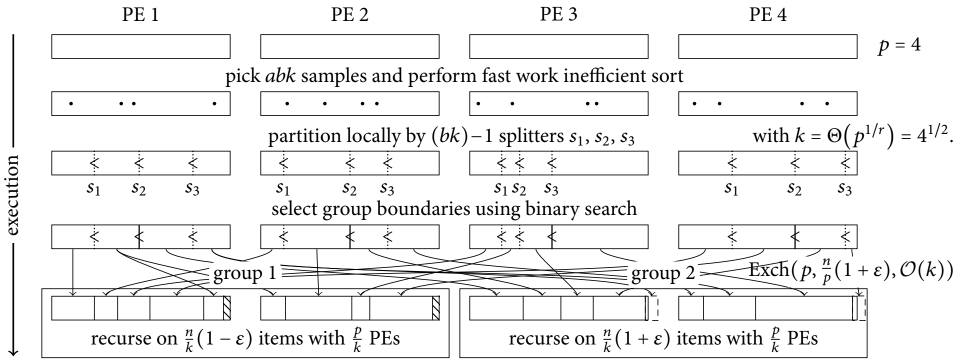


Figure 6.6: Algorithm schema of AMS-sort.

The binary scanning algorithm finds arbitrarily good splitters with high probability for appropriate a and b.

Lemma 6.15 (Imbalance bound of AMS-sort for one recursion level)
We can achieve $L = (1+\varepsilon)\frac{n}{k}$ *with high probability choosing appropriate* $b = \Omega(1/\varepsilon)$ *and* $ab = \Omega(\log k)$.

We improve the assignment of buckets to PE-groups to $\mathcal{O}(bk \log bk)$ and, using parallelization, even to $\mathcal{O}(bk + \alpha \log t)$. The first observation for improving the binary search algorithm is that a PE-group size can take only $\mathcal{O}\big((bk)^2\big)$ different values since it is defined by a range of buckets. We can modify the binary search in such a way that it operates not over all conceivable group sizes but only over those corresponding to ranges of buckets. When a scanning step succeeds, we can safely reduce the upper bound for the binary search to the largest PE-group actually used. On the other hand, when a scanning step fails, we can increase the lower bound: during the scan, whenever we finish a PE-group of size x because the next bucket of size y does not fit (i.e., $x + y > L$), we compute $z = x + y$. The minimum over all observed z-values is the new lower bound. This is safe since a value of the scanning bound L less than z will reproduce the same failed partition. This already yields an algorithm running in time $\mathcal{O}\big(bk \log((bk)^2)\big) = \mathcal{O}(bk \log(bk))$.

The second observation is that only values for L in the range $[\lceil n/r - 1\rceil .. (1 + \mathcal{O}(1/b))n/k]$ are relevant (see Lemma 6.15). Only $\mathcal{O}(bk)$ bucket ranges will have a total size in this range. To see this, consider any particular starting bucket for a bucket range. Searching from there to the right for range end points, we can skip all end buckets where the total size is below n/k. We can stop as soon as the total size leaves the relevant range. Since buckets have an average size of $\mathcal{O}(n/b)$, only a constant number of end points will be in the relevant range on the average. Overall, we get $\mathcal{O}(bk) \cdot \mathcal{O}(1) = \mathcal{O}(bk)$ relevant bucket ranges. Using this for initializing, the binary search speeds up the sequential algorithm by a factor of about two.

Using all t available PEs, we can do even better: in each iteration, we split the remaining range for L evenly into $t + 1$ subranges. Each PE tries one subrange end point for scanning and

Algorithm 13 AMS-sort

Input: $A[1..n/t]$ data elements, t PEs, k PE-groups, a oversampling factor, b overpartitioning factor
$l \in \mathbb{N}^{abk} \leftarrow$ temp. array of local bucket sizes induced by splitters
$g \in \mathbb{N}^{abk} \leftarrow$ temp. array of global bucket sizes induced by splitters
$L \in \mathbb{N}^{k} \leftarrow$ temp. array of local bucket sizes induced by grouped buckets
$G \in \mathbb{N}^{k} \leftarrow$ temp. array of global bucket sizes induced by grouped buckets
if $t = 1$ **then**
    $A' \leftarrow \textsc{sort}(A)$ **return** A'
$S \leftarrow \textsc{chooseRandomSample}(A, abk)$ ▷ $|S| = abk$
$R \leftarrow \textsc{FastWorkInefficientRanking}(S)$
$E \leftarrow \textsc{chooseEquidistant}(S, R, bk-1)$ ▷ $|E| = bk - 1$
$E' \leftarrow \textsc{allgatherMerge}(E)$
$l \leftarrow \textsc{localPartition}(A, E')$ ▷ Partition local data into bk buckets
$g \leftarrow \textsc{allreduce}(\mathrm{l})$ ▷ Determine global bucket sizes (collective all-reduce)
$(L, G) \leftarrow \textsc{fastBucketGrouping}(l, g)$
$M \leftarrow \textsc{messageAssignment}(L, G, k)$ ▷ Assign local (sub)buckets to PEs of PE-groups
$A' \leftarrow \textsc{dataExchange}(A, M)$ ▷ Exchange data according to messages M
return $\textsc{Ams-sort}(A', t/k, k, a, b)$ ▷ Recurse on PE-groups

uses the first observation to round up or down to an actually occurring size of a bucket range. Using a collective reduce operation we find the largest L-value $L_{\min}$ for a failed scan and the smallest L-value $L_{\max}$ for a successful scan. When $L_{\max} = L_{\min}$ we have found the optimal value for L. Otherwise, we continue with the range $[L_{\max} .. L_{\min}]$. Since the bucket range sizes in the feasible region are fairly uniformly distributed, the number of iterations will be $\log_{t+1} \mathcal{O}(bk)$. Since $t \geq k$, this is $\mathcal{O}(1)$ if b is polynomial in k. Indeed, one or two iterations are likely to succeed in all reasonable cases.

The data splitting defined by the bucket group is then the input for the **data exchange** algorithm described in Section 6.4. This takes time $\widetilde{\mathrm{Exch}}\,(t, (1+o(1))L, (2+o(1))k))$.

We **recurse on the PE-groups** similar to Section 6.6. Within the recursion it can be exploited that the elements are already partitioned into bk buckets.

We get the following overall execution time for one level:

Lemma 6.16 (Running time of single-level AMS-sort)
One level of AMS-sort works in time

$$\mathcal{O}\Big(\frac{n}{t}\log\frac{k}{\varepsilon} + \beta\frac{k}{\varepsilon}\Big) + \widetilde{\mathrm{Exch}}(t, (1+\varepsilon)\tfrac{n}{t}, \mathcal{O}(k)) \quad . \tag{6.4}$$

Compared to previous implementations of samplesort, including the one from Gerbessiotis and Valiant [GV94], AMS-sort improves the sample size from $\mathcal{O}\big(t\log t/\varepsilon^2\big)$ to $\mathcal{O}(t(\log k + 1/\varepsilon))$ and the number of startup overheads in the Exch-term from $\mathcal{O}(t)$ to $\mathcal{O}(k)$. In 2019, Harsh

et al. [HKS19] improves these results to a total of $\mathcal{O}\left(p \log \frac{\log p}{\epsilon}\right)$ samples in $\mathcal{O}\left(\log \frac{\log p}{\epsilon}\right)$ rounds of histogramming.

In the base case of AMS-sort, when the recursion reaches a single PE, the **local data is sorted sequentially**. Algorithm 13 provides pseudocode of AMS-sort.

Theorem 6.17 (Running time of AMS-sort with *r* recursion levels)
Adaptive multi-level samplesort (AMS-sort) with r levels of recursion and an imbalance factor $(1+\varepsilon)$ in the output can be implemented to run in time

$$\mathcal{O}\Big(\frac{n}{t}\log n + \beta\frac{r^2 t^{1/r}}{\varepsilon}\Big) + \sum_{i=1}^{r} \widehat{\mathrm{Exch}}\left(t^{\frac{i}{r}}, (1+\varepsilon)\tfrac{n}{t}, \mathcal{O}\big(t^{1/r}\big)\right)$$

if $r = \mathcal{O}(\log t / \log\log t)$ *and* $\frac{1}{\varepsilon} = \mathcal{O}\big(n^{1/r}\big)$.

Using a similar argument as for RLM-sort, for constant r and ε, we get an isoefficiency function of $t^{1+1/r}/\log t$ for $k = t^{1/r}$. This is a factor $\log^2 t$ better than for RLM-sort and an indication that AMS-sort might be the better algorithm—in particular, if some imbalance in the output is acceptable and if the inputs are rather small.

Another indicator for the good scalability of AMS-sort is that we can view it as a generalization of parallel quicksort that also works efficiently for very small inputs. For example, suppose $n = \mathcal{O}(t \log t)$ and $1/\varepsilon = \mathcal{O}(1)$. We run $r = \mathcal{O}(\log t)$ levels of AMS-sort with $k = \mathcal{O}(1)$ and $\varepsilon' = \mathcal{O}(r/\varepsilon)$. This yields running time $\mathcal{O}\big(\log^2 t \log\log t + \alpha \log^2 t\big)$. This does a factor $\mathcal{O}(\log\log t)$ more local work than an asymptotically optimal algorithm. However, this is likely to be irrelevant in practice since we expect that $\alpha \gg \log\log t$ for most cases. Also the factor $\log\log t$ would disappear in an implementation that exploits the information gained during bucket partitioning.

Robustness

In this work, we tested two tie-breaking approaches. The first approach turns inputs with duplicates into inputs without duplicates. Each element e is implicitly represented as a tuple (e, i, m) where i is the index of the source PE and m is the position in the local input of PE i. The partitioning step compares the local elements to the splitters lexicographically. An explicit representation of the tuples is only required for the splitters. Depending on the i- and m-values of elements and splitters, we break ties by either using the comparator function $<$ or by using the comparator function $\leq$. As all local elements share the same i-value and as elements that are close together have similar m-values we can apply the same comparator function to a large precalculated range of elements. Thus, the overhead is very small. However, this approach has two disadvantages. First, to become cache-efficient, we perform multiple partitioning passes if the number of partitions is large. This makes the tracking of the i- and m-values somewhat complicated. Second, this approach does not give us any flexibility in distributing duplicated elements to different PEs for the sake of low imbalance.

For AMS-sort, we propose a different approach. We partition the local elements into buckets and add additional equality buckets for elements corresponding to a splitter. To break ties, we

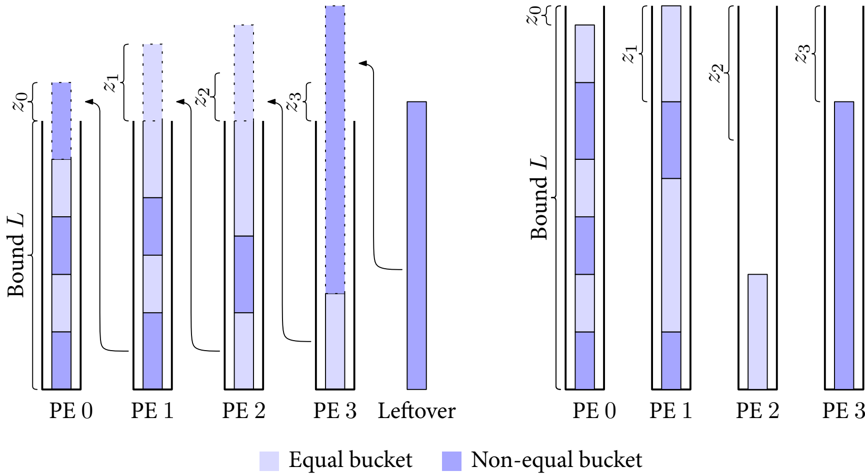


Figure 6.7: Scanning algorithm examples with bound L. L is a lower bound in the left example and an upper bound in the right example.

refine the algorithm that assigns buckets to PE-groups. To verify a bound L, we scan through the array of bucket sizes and skip to the next PE-group when the total load would exceed L. Equality buckets are allowed to overlap one or multiple PE-groups. Our binary search should not operate over all conceivable group sizes but only over those corresponding to ranges of buckets. Thus, we adjust L before we continue the binary search algorithm of the bucket assignment. If the scanning algorithm was not able to assign all buckets, we can safely increase L until the scanning algorithm moves a bucket to a prior PE-group. Otherwise, we decrease L until a bucket would have to be moved to a subsequent PE-group.

We first consider the case that the scanning algorithm failed. Let g be the residual capacity of PE-group i, let w be the capacity of the first bucket assigned to PE-group $i+1$, and let x be the number of PE-groups prior to PE-group i whose last bucket overlaps with its successor. Note that we can track x for all PE-groups in $\mathcal{O}(r)$ time when we process the PE-groups in ascending order. For PE-group i we compute $z_i = L + (w-g)/(x+1)$. The minimum over all observed z-values is the new lower bound. To understand that this is safe, we consider the z_j's from previous PE-groups $j \in [0..i)$. On the one hand, PE-group i needs $w-g$ additional residual capacity. On the other hand, an increase of L by $(w-g)/(x+1)$ increases the total capacity of PE-group i by $(w-g)/(x+1)$ and adds additional residual capacity of $x(w-g)/(x+1)$ to PE-groups $[i-x..i)$. This additional residual capacity must be pulled back by these PE-groups using their overlapping equality buckets to create $w-g$ additional residual capacity on PE-group i. Thus, z_i is a lower bound required to move the first bucket of PE-group $i+1$ to PE-group i. If $\min_{j\in[0..i)} x_j \leq z_i$, this lower bound is already undercut by previous PE-groups and we do not have to consider z_i at all. If $\min_{j\in[0..i)} x_j > z_i$, the value z_i is also an upper bound required to move the first bucket of PE-group $i+1$ to PE-group i since PE-groups $j \in [i-x..i)$ are actually able to pull back enough elements using their overlapping equality buckets.

When the scanning algorithm succeeds, we use a similar algorithm to update L. The idea is that we allow PE-groups to push parts of their equality buckets to subsequent PE-groups as long as the equality buckets do not wrap over to another PE-group. Figure 6.7 depicts two examples of the scanning algorithm. In one example, the bound L is a lower bound and we increase L by the minimum of the z-values (here z_0 respectively z_3). In the other example, the bound L is an upper bound and we decrease L by the minimum of the z-values (here z_0).

Compared to the first tie-breaking approach, we can exploit equality buckets to decrease the load imbalance and simplify the partitioning algorithm. For example, we can use the partitioning step from Section 3.1 to partition the local input and to create equality buckets efficiently. A disadvantage is that the assignment of buckets to PE-groups becomes more complicated.

Chapter 7

Experiments and Conclusion

7

In this chapter, we present extensive experimental results from 11 different sorting algorithms, ten data distributions, and two supercomputers with up to 262 144 cores. On the one hand, we compare our algorithms to their competitors for a wide spectrum of inputs. On the other hand, we consider each algorithm separately and push their robustness to the limits with publicly available data distributions but also with our own worst case input distributions. To better understand which algorithmic measures have an impact under which circumstances we examine different implementation stages of our algorithms.

Algorithms. We included the following implementations in our benchmark set:

- GatherMsort: Our simple binomial-tree gather-merge-sort from Section 5.3.1.
- RFIR: Our robust fast work-inefficient ranking from Section 6.1.
- RFIS: Our robust fast work-inefficient sorting from Section 6.1.
- RQuick: Our robust hypercube quicksort from Section 6.3.
- BitonicSort: Our bitonic sort for arbitrary values of t from Section 7.1.
- AMS-sort: Our recursive multiway samplesort algorithm from Section 6.6.
- Minisort: Our reimplementation of distributed quicksort from Siebert and Wolf [SW11] for the case $n = t$ (also see Section 7.1).
- RLMS-sort: Our MPI reimplementation of single-level histogram sort with sampling from Harsh et al. [HKS19] (again see Section 7.1).
- HykQuick: Hypercube quicksort from Sundar et al. [SMB13].
- HykSort: Multiway hypercube mergesort from Sundar et al. [SMB13].
- HSS: Multiway hypercube histogram sort with sampling from Harsh et al. [HKS19].

We do not present results of allgather-merge-sort from Sections 5.1.1 and 5.3.1 since allgather-merge-sort was slower than RFIS and GatherMsort for any input size we have tested.

Input Distributions. We ran benchmarks with ten input distributions of 64-bit floating-point elements. We report results for seven input distributions proposed by Helman et al. [HBJ98], i.e., *Uniform* (independent random values), *BucketSorted* (locally random, globally sorted), *DeterDupl* (only $\log t$ different keys), *Staggered* (hypercube-like routing with perfect splitters would have a load imbalance factor of 2), *g-Group* with $g \coloneqq t^{1/2}$, *Zero* (all elements are equal), and *RandDupl* (32 local buckets of random size, each filled with an arbitrary value from $[0\,..\,32)$). We do not present results for the input distribution *Gaussian* by Helman et al. [HBJ98] since all algorithms sort this distribution as fast as Uniform input. The input distribution

BucketSorted is meaningless for $n < t^2$ and the input distribution g-Group is meaningless for $n < t^{3/2}$. Additionally, we included the input distributions *AllToOne* and *Mirrored*. The first $n/t - 1$ elements of AllToOne on PE i are random elements from the range $[t + (t - i)(2^{32} - t)/t .. t + (t - i + 1)(2^{32} - t)/t]$ and the last element has the value $t - i$. At the first level of a naive implementation of recursive multiway samplesort, the last $\min(t, n/t)$ PEs would send a message to PE 0. The input of the distribution Mirrored on PE i are random numbers between $2^{31}(j)/t$ and $2^{31}(j + 1)/t$ where the integer bit representation of j is the reverse bit representation of i. Mirrored is a generalization of the bit-reversal permutation by Thomson Leighton [Lei92, Sec. 3.4.2] to arbitrary n. For naive hypercube-based sorting algorithms, this distribution has an intermediate imbalance of $\lfloor n/t^{1/2} \rfloor$ elements in the case of perfect splitters. We refer to Section 5.1.1 for more details.

Supercomputers. The results of our sorting experiments were obtained on two supercomputers. The first supercomputer, *SuperMUC-NG* [Lei18], is a distributed system with a two-level hierarchical network. The computation nodes of SuperMUC-NG are bundled into eight so-called *islands*, each equipped with 788 nodes. Each node has two Intel Skylake Xeon Platinum 8174 24-core CPUs with a standard frequency of 3.1 GHz and 96 GByte of memory. A fat tree network topology connects the nodes within an island for highly efficient communication using the Intel OmniPath Interconnect network technology. Computation nodes are connected to the fat tree by Intel OPA100 adapters. At the level of islands, the overall communication bandwidth is pruned by a factor of four. When we allocate a job on SuperMUC-NG, we always allocate full islands to minimize running time fluctuations caused by other jobs on the same island. The maximum number of islands available to us was four. However, we used only two instead of four islands since we measured large running time fluctuations for small inputs on four islands.

The second supercomputer, *JUQUEEN* [SD15], is an IBM BlueGene/Q based system of 56 so-called *midplanes*, each equipped with 512 compute nodes. Each compute node has one IBM PowerPC A2 16-core CPU with a nominal frequency of 1.6 GHz and 16 GByte of memory. A 5-D torus network with 40 GB/s and 2.5 μs worst case latency connects the nodes. Each midplane connects its compute nodes with a $4x4x4x4x2$ torus. The user can specify the number of midplanes per torus dimension. In our experiments, we configured the tori such that the size of the dimensions is as equal as possible. The maximum number of midplanes available to us was 32. The JUQUEEN has been switched off in 2018.

We also executed preliminary experiments on SuperMUC Phase 1 and Phase 2 [Lei15]. However, the running time of the algorithms fluctuated a lot on these machines for large t. In Appendix B.1, we discuss these fluctuations measured on the SuperMUC supercomputers in more detail.

Overview. This chapter is divided as follows. We give implementation details in Section 7.1 and describe our parameter tuning in Section 7.1.2. In Section 7.2 we define the methodology of our experiments. Section 7.3 presents results for our algorithms and our closest competitors HykSort and HSS for eight input distributions on SuperMUC-NG with 2^{16} cores and on JUQUEEN with 2^{18} cores. In Section 7.4, we compare our robust algorithms to their nonrobust versions and their competitors. In Section 7.5, we discuss the efficiency of RFIS, RQuick, and AMS-sort in terms of weak and strong scaling. Finally, we summarize our work on robust and scalable distributed sorting algorithms in Section 7.6.

7.1 Implementation Details

In our publication [AWS18] we propose the lightweight library *RangeBasedComm* (RBC) based on MPI (also see Appendix B.2). RBC provides routines to create RBC communicators of consecutive PE ranges, (non)blocking collective operations, and (non)blocking point-to-point communication. RBC provides the option to switch back to MPI communicators and collectives if desired. All algorithms that were implemented in this work use the RBC library. We also implemented RBC versions of our competitors HykQuick (hypercube quicksort from Sundar et al. [SMB13]), HykSort (multiway hypercube mergesort from Sundar et al. [SMB13]), and HSS (multiway hypercube histogram sort with sampling from Harsh et al. [HKS19]) by replacing the MPI routines with their counterparts of RBC.

Our binomial-tree gather-merge-sort implementation GatherMsort performs three steps. First, the PEs calculate the total input size of its subtree by sifting down the input sizes to the root node. Second, each PE allocates a temporary array that will store the data from its subtree. Finally, the actual binomial-tree gather-merge-sort algorithm gathers the input on the root node in sorted order.

Sundar et al. [SMB13] propose a distributed bitonic sort algorithm. However, it turned out that the bitonic sort implementation is not practical, i.e., it requires the same input size on each PE, it does not perform all routines in parallel when executed with multiple threads, and it is slowed down by unnecessary shared-memory code in the sequential case. Thus, we decided to compare our algorithms to BitonicSort, an own robust implementation of bitonic sort. BitonicSort allows arbitrary values of t [Lan06]. To minimize message startups, each pairwise data exchange of BitonicSort transfers all local elements and the PEs keep the left respectively the right half of the merged sequence. We also tested a version of BitonicSort with multisequence selection (see Section 5.1.3) to minimize the communication volume. However, this approach was significantly slower for small inputs where startup overheads matter and for large inputs, one would consider other algorithms anyway. In order to support an arbitrary number of local input elements, we determine the maximum local input size m of BitonicSort with a collective all-reduce. After each merge operation, we then split the result into one part containing the first $\lceil m/2 \rceil$ elements (if available) and one part containing the remaining elements.

The implementation of AMS-sort borrows the partitioning step from IPS^{4}o which already supports equality buckets. To become more cache-efficient, we limit the number of splitters in a partitioning step to $k = 256$ and perform multiple levels if required. For the data exchange step of AMS-sort, we extended the 1-factor data exchange algorithm [ST02] to send and receive messages using multiple connections simultaneously. For more details, we refer to the discussion on irregular data exchanges in Section 5.2. To guarantee a maximum imbalance factor of $1+\epsilon = 1.1$ we configure AMS-sort as follows: On each recursion level, r-level AMS-sort accepts an additional imbalance factor of $1 + \epsilon' = (1 + \epsilon)^{1/r}$. Thus, we use an overpartitioning factor $b = \frac{2}{\epsilon'}$ and an oversampling factor of $a = \max(\frac{2}{\epsilon'}, 13 \log_2 t)/b$. In our experiments, the imbalance of AMS-sort was always smaller than 0.1 (except for $n/t \leq 16$). If t is not a power of k, AMS-sort creates k PE-groups with $\lfloor t/k \rfloor$ respectively $\lceil t/k \rceil$ PEs. The merging step of our deterministic message assignment algorithm uses the merge operation of BitonicSort.

We provide our own implementation of Minisort, a distributed quicksort algorithm from Siebert and Wolf [SW11] for the case $n = t$, as the source code used in their experiments is not available anymore.

On the authors' recommendation, we use our own implementation (1LHSS) of single-level histogram sort with sampling from Harsh et al. [HKS19] as the original implementation uses the communication framework Charm++ instead of MPI. In each round of histogramming, 1LHSS adds each element to the sample with probability $5k/n$. We stop histogramming as soon as the imbalance factor is below 1.1. 1LHSS uses a fast tournament tree implementation [SSP07; Bin18b] for efficient multiway merging.

All algorithms use the sequential version of IPS^{4}o for local sorting. Our implementations, i.e., GatherMsort, RFIS, RFIR, RQuick, BitonicSort, AMS-sort, and 1LHSS, are in general capable of sorting all inputs (including the case in which some PEs do not have any local input elements) for arbitrary t. When t is not a power of two, RQuick routes the elements stored on the additional PEs to the embedded hypercube.

The algorithms are written in C++ and communicate with MPI. On SuperMUC-NG, we compiled with version 19.0 of the Intel compiler, using the full optimization flags *-xCORE-AVX512* and *-O3*. For inter-process communication, we use the Intel MPI library 2020. On JUQUEEN, the algorithms were compiled with version 4.8.1 of the GNU compiler collection, using the optimization flag `-O2`. For inter-process communication, we used the IBM MPI library `mpich2` version 1.5. The implementation of our algorithms AMS-sort, RFIS, RFIR, and RQuick as well as the reimplementations 1LHSS and BitonicSort can be found on the official website `https://github.com/MichaelAxtmann/KaDiS`. The implementation of GatherMsort is part of the RBC library published at `https://github.com/MichaelAxtmann/RBC`.

7.1.1 The *k*-way Data Exchange

We describe the implementation of the data exchange routine represented by the cost function $\mathrm{Exch}(a, b, c)$ used by AMS-sort and RLM-sort. We know that each PE receives at most b elements. Also, our data exchanges do not require a specific ordering of the incoming messages. Thus, we can receive messages in any order and we can preallocate sufficient storage at the receiving PEs. However, we need a technique that exchanges data without initially knowing the incoming messages at the receiving side—each PE only knows its outgoing messages. We use an implementation of the data exchange algorithm *NBX* [HSL10]. NBX posts all send operations in a bulk and receives incoming messages once available. It combines a nonblocking barrier with synchronous send operations to identify when all messages have been successfully received. NBX is designed for data exchanges with $\mathcal{O}(\log t)$ messages per PE and thus, can efficiently send the messages in a bulk. We use a refined version of NBX since our k-way sorting algorithms exchange $\mathcal{O}(k)$ messages. The value k is usually much larger than $\mathcal{O}(\log t)$, e.g., our recursive multiway algorithms with three levels use $k = t^{1/3}$ groups. Our implementation limits the number of incoming and outgoing messages to a maximum of m. We first create m slots for send respectively receive operations and post the first m send operations determined by the 1-factor algorithm. Then we continuously fill vacant send slots with the next send operation and vacant receive slots with a receive operation once an incoming message is ready for pickup.

Since version 2.1, the MPI library MPICH uses a similar technique to limit the number of simultaneous message transfers in irregular data exchanges.

7.1.2 Parameter Tuning

We performed extensive parameter tuning on JUQUEEN and SuperMUC-NG. Each algorithm has a range of input sizes where it is designed to be the fastest. We optimized each algorithm to run as fast as possible in this input range. All recursive multiway algorithms in the benchmark portfolio are attractive for relatively large inputs (i.e., $n/t \geq 2^{12}$ on SuperMUC-NG and $n/t \geq 2^{14}$ JUQUEEN) as they come with large overheads for small inputs. We tested all recursive multiway algorithms with these input sizes and Uniform input distribution. AMS-sort, HykSort, and HSS were executed with $k = 16, 32, 64, 128$. For each algorithm, we select the k for which they performed best on average for large inputs: On SuperMUC-NG, the algorithms sorted large inputs the fastest with $k = 64$. On JUQUEEN, AMS-sort and HykSort performed best with $k = 64$ respectively $k = 32$. The HSS algorithm had not yet been published at that time we ran the experiments on JUQUEEN. AMS-sort does not provide shared-memory parallelism. HSS and HykSort were significantly slower on SuperMUC-NG with more than one thread. On JUQUEEN, HykSort was slightly faster with eight instead of one thread. For the sake of consistency, we present results for all algorithms with one thread per process. More levels speed up the recursive multiway algorithms for smaller inputs (up to 50%) but slow them down for large inputs. Fewer levels slightly speed up the algorithms for very large inputs.

7.2 Methodology

In our experiments, we always execute one MPI process per core. For the sake of consistency, we continue to use the term PE, which from now on is also a synonym for MPI process. Unlike stated otherwise, we use 65 536 cores on SuperMUC-NG respectively 262 144 cores on JUQUEEN and the input distribution Uniform.

Before each measurement, we invoke a global barrier to synchronize the PEs. We then measure the local wall times and use the maximum wall time as the running time of the run. On JUQUEEN we repeat each measurement five times ($n \leq 2^{17}$) respectively two times ($n \geq 2^{18}$) and average over runs. We do not show error bars since the ratio between the maximum execution time and the average execution time is consistently less than a few percent. On SuperMUC-NG we noticed fluctuations in running time while warmup as well as while executing sparse data exchanges with non-deterministic communication partners. We study these issues in Appendix B.1. As a consequence, we decided to sort each input repeatedly until the running time exceeds two seconds, but at least twice, and average over runs. This experimental setup compensates for fluctuations in the running time of quickly sorted (small) inputs and limits the time of expensive (large) inputs. Overall, we spent more than twenty million core-hours on JUQUEEN and SuperMUC-NG.

We benchmark algorithms in separate jobs since the running time of some algorithms influenced the running time of another algorithm executed in the same job. Our competitors HykSort and HSS suffer from deadlocks on some small inputs and some large input distributions

with duplicates. As a consequence, we sort small inputs first and we submit multiple jobs for each of these algorithms—the jobs separate input distributions and inputs with $n/t \leq 2^7$ and $n/t > 2^7$.

The algorithms were executed for dense inputs with $n/t = 2^i$, $i \in [0..22]$ and sparse inputs with sparsity-factors 3^i, $i \in [1..5]$. Sparse inputs with a *sparsity-factor* t/n mean that only the first n PEs hold an input element. Dense inputs are evenly distributed among the PEs.

Unless stated otherwise, we executed the algorithms as follows: On SuperMUC-NG and JUQUEEN, our implementations GatherMsort, RFIR, RFIS, RQuick, BitonicSort, AMS-sort, and Minisort used RBC for communication. On SuperMUC-NG, we benchmarked our competitors HykQuick, HykSort, and HSS with RBC as well. On this supercomputer, our competitors were much slower with pure MPI—even when we excluded the time for communicator splitting. We refer the reader to Appendix B.2 for a detailed performance comparison of HykQuick, HykSort, and HSS with pure MPI and with RBC on SuperMUC-NG. On JUQUEEN, we benchmarked our competitors HykQuick, HykSort, and HSS with pure MPI since the collective communication operations provided by MPI were extremely fast on this supercomputer. When we execute our competitors with MPI collectives, they also require the MPI sub-communicators. Unfortunately, the MPI communicator creation was also slow on this supercomputer. Thus, we decided to exclude the time for communicator splitting when we report results of our competitors obtained on JUQUEEN. We argue that the communicators could be stored and reused in a sorting object over many runs with arbitrary inputs. Thus, we view the overhead for MPI communicator splitting as a precomputation.

7.3 Input Size Analysis and Algorithm Comparison

Motivation. In this section, we present the results of our sorting algorithms GatherMsort, BitonicSort, RFIS, RQuick, AMS-sort, and competitors HSS respectively HykSort. We exclude the competitor Minisort from the discussion in this section as Minisort is not competitive for any input distribution (see Section 7.4.2). We expect to need multiple algorithms to sort the entire parameter space of input sizes as fast as possible. The reason is that the algorithms trade off startup overheads and communication volume differently. Our asymptotic analysis suggests that algorithms with small overhead sort small inputs the fastest whereas for larger inputs, algorithms with larger overheads benefit from less communication volume. Besides the algorithm comparison for inputs with a sparsity-factor of 243 to 2^{22} elements per PE, we examine their robustness for eight input distributions—the input distributions from Helman et al. [HBJ98] and the distribution Mirrored.

Experimental Results. Figures 7.1 and 7.3 show the running times obtained on the supercomputers SuperMUC-NG and JUQUEEN. Figures 7.2 and 7.4 give running time ratios of each algorithm compared to the fastest algorithm. We do not present the results of our competitor HykSort obtained on SuperMUC-NG as HSS (HykSort with a refined splitter selection algorithm) turned out to be faster in all situations. On JUQUEEN, we have to compare our results to HykSort as HSS had not yet been published when JUQUEEN was decomposed.

GatherMsort sorts very sparse inputs ($n/t \leq 3^{-3}$) up to 1.8 times as fast as the other sorting algorithms on JUQUEEN. On SuperMUC-NG, the performance is slightly better as we use

fewer PEs on this supercomputer. Remember that GatherMsort does not fulfill the balance constraint of sorted output.

For all input distributions, RFIS is the fastest sorting algorithm when $3^{-3} < n/t \leq 4$ on JUQUEEN. For example, RFIS sorts Uniform sparse inputs up to 3.6 times as fast as its competitors and a single element per PE more than twice as fast as RQuick and BitonicSort. On SuperMUC-NG, RFIS is the fastest sorting algorithm on a rather narrow range of input sizes.

On JUQUEEN, RQuick sorts small ($n/t = 2^3$ to 2^{14}) Uniform inputs up to 3.4 times as fast as any other algorithm (more than 8 times if we exclude our algorithm AMS-sort). RQuick sorts these small inputs up to 3.3 times as fast as on SuperMUC-NG. In general, the running times of RQuick for Uniform inputs do not differ much from the running times for other input distributions. There is no need for BitonicSort: When the running time of RQuick is bound by its message startups, BitonicSort has a similar performance. However, for these input sizes, one would prefer RFIS in most cases. For larger input sizes, RQuick is in most cases at least twice as fast as BitonicSort. Thus, even when t is not a power of two, we expect RQuick to outperform BitonicSort.

AMS-sort is more efficient than the multiway hypercube algorithms HSS and HykSort for small inputs. Also, HSS and HykSort crashed on input sizes of distributions DeterDupl, RandDupl, and Zero. Apart from that, AMS-sort, HykSort, and HSS are the fastest algorithms for large inputs (more than 2^{15} elements per PE on JUQUEEN and more than 2^{13} elements per PE on SuperMUC-NG). For example, RQuick is for the largest input between 2.08 (Uniform) and 12.35 (Zero) times slower than RQuick. We now compare AMS-sort to HSS on SuperMUC-NG: AMS-sort is faster than HSS for all large input instances. For Uniform, BucketSorted, and g-Group inputs, HSS is almost as fast as AMS-sort. Also, AMS-sort sorts the skewed input Staggered up to 2.69 times as fast as HSS (more than 8 times if we consider Mirrored, which is hard for nonrobust hypercube algorithms). Finally, we compare AMS-sort to HykSort on JUQUEEN: AMS-sort is always faster for 2^{15} to 2^{17} elements per PE. For $n/t > 2^{17}$, the situation becomes more distribution-dependent. For uniform inputs, HykSort is up to 1.10 times as fast as AMS-sort. However, HykSort sorts Staggered inputs up to 1.70 times slower than AMS-sort. This slowdown is caused by the skewed input distribution. Mirrored input is even harder for HykSort. I.e., HykSort sorts Mirrored input 9.24 times slower ($n/t = 2^{16}$) than AMS-sort and crashed for $n/t \geq 2^{16}$. We want to note that we use a refined version of HSS that uses our RBC library for communication. For more information, we refer to Appendix B.2 which illustrates that the refinements speed up the original version by up to a factor of 101.

Conclusion. GatherMsort is the fastest algorithm for very sparse inputs. For sparse and very small inputs one would prefer RFIS. One would choose RQuick for medium-sized inputs. And, AMS-sort seems to be a good compromise between robustness and performance for large inputs. Note that AMS-sort sorts inputs on arbitrary supercomputer sizes but HykSort and HSS require that $t = k^r$ where r is the number of levels. HykSort and HSS are competitive to AMS-sort for some input distributions but sort other distributions slowly or even crash. Due to the robustness of our sorting algorithms, their running time is not sensitive to the input distribution. As a result, one could once execute a performance test for Uniform inputs on a supercomputer instance and then choose the appropriate algorithm only depending on the input size.

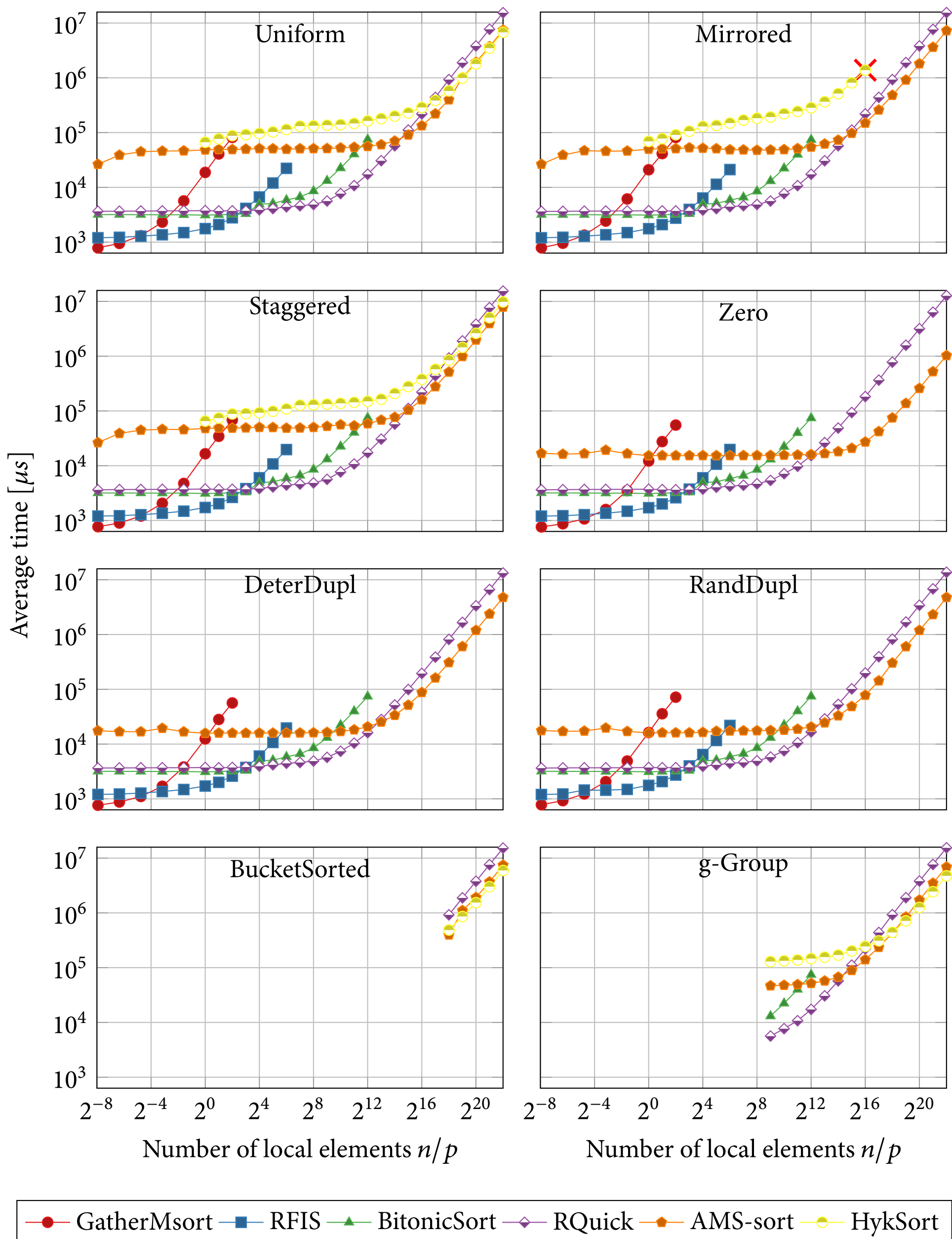


Figure 7.1: Running times of distributed sorting algorithms on JUQUEEN. The red cross indicates a break due to memory overflow.

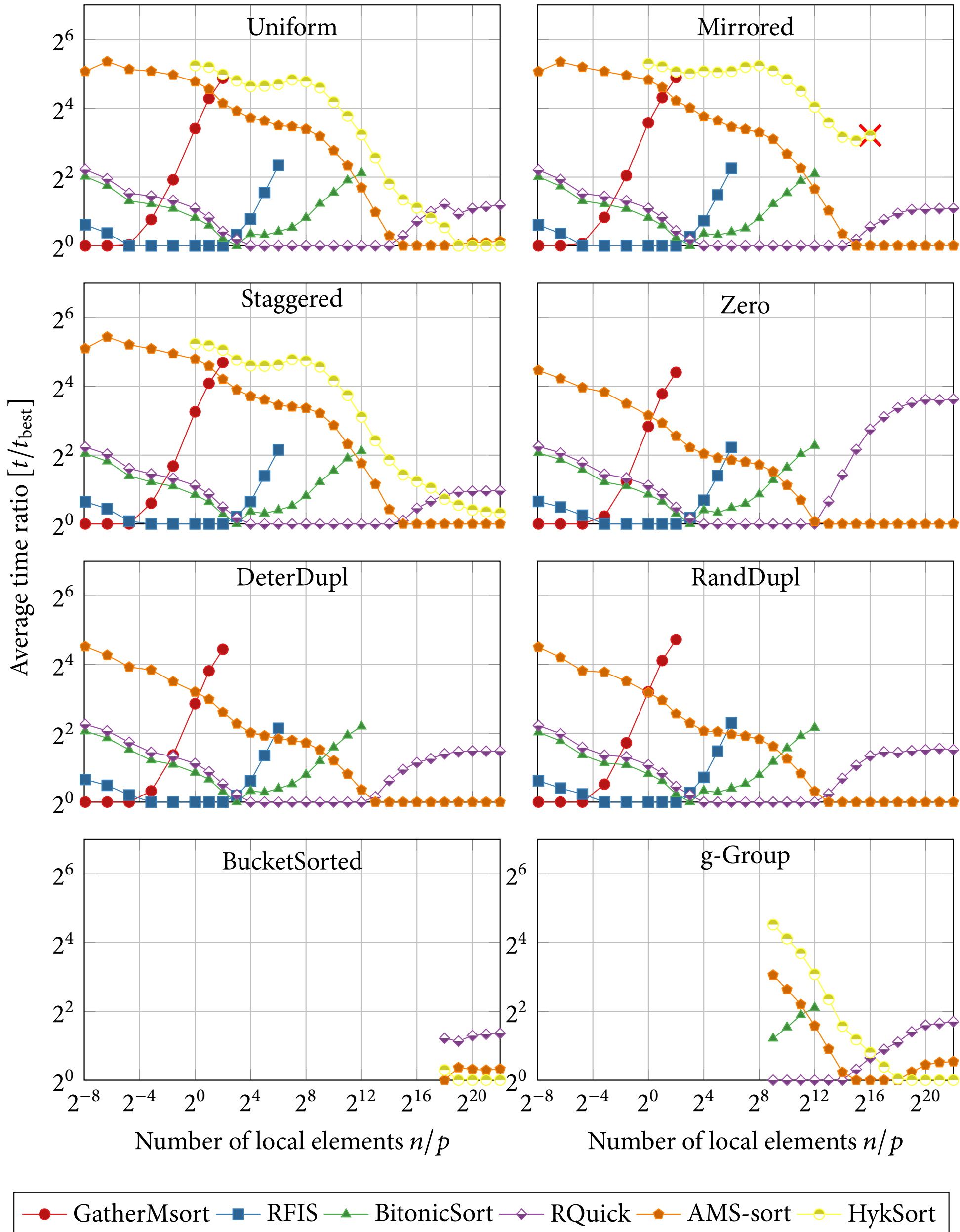


Figure 7.2: Running ratios of each algorithm to the fastest algorithm on JUQUEEN. The red cross indicates a break due to memory overflow.

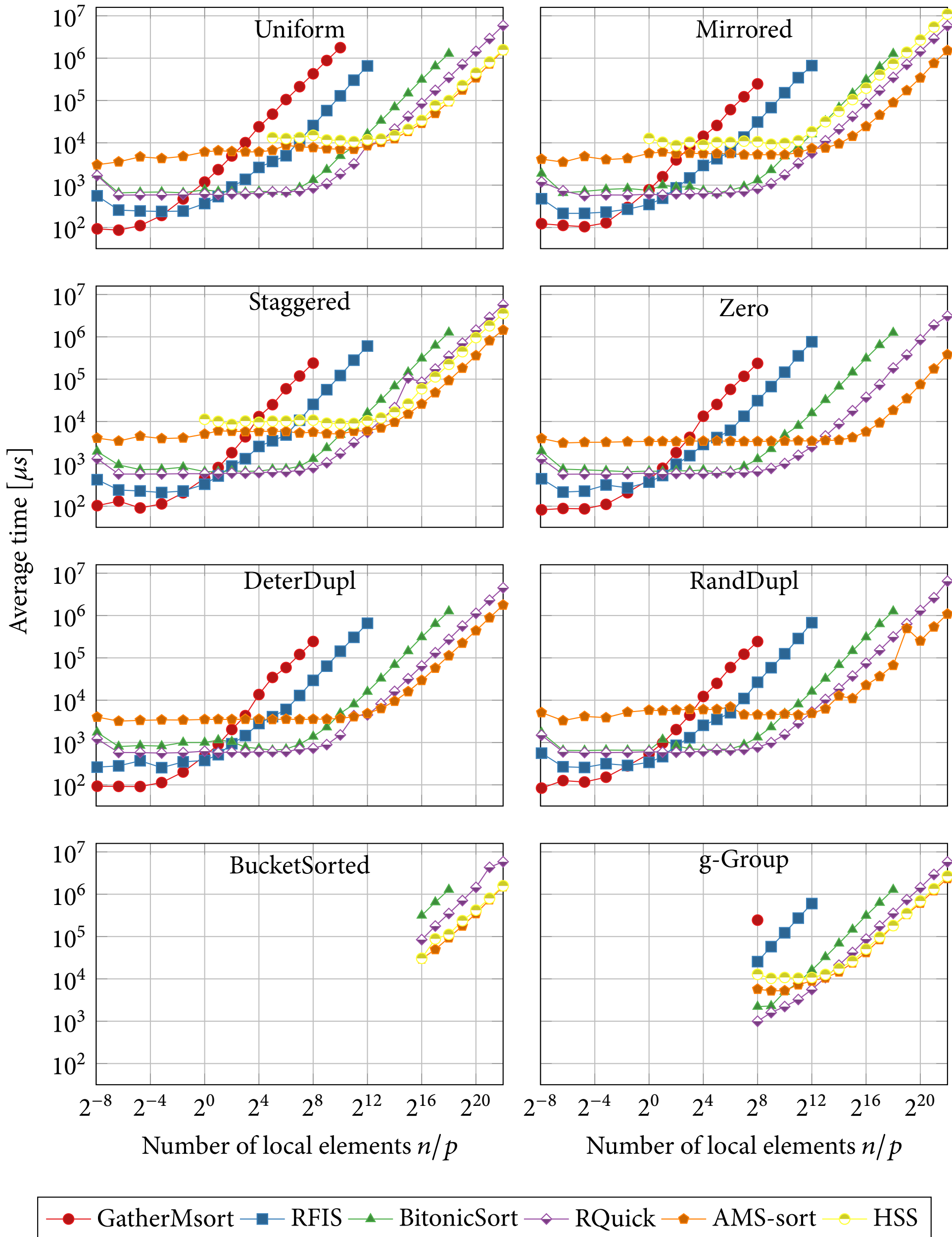


Figure 7.3: Running times of distributed sorting algorithms on SuperMUC-NG.

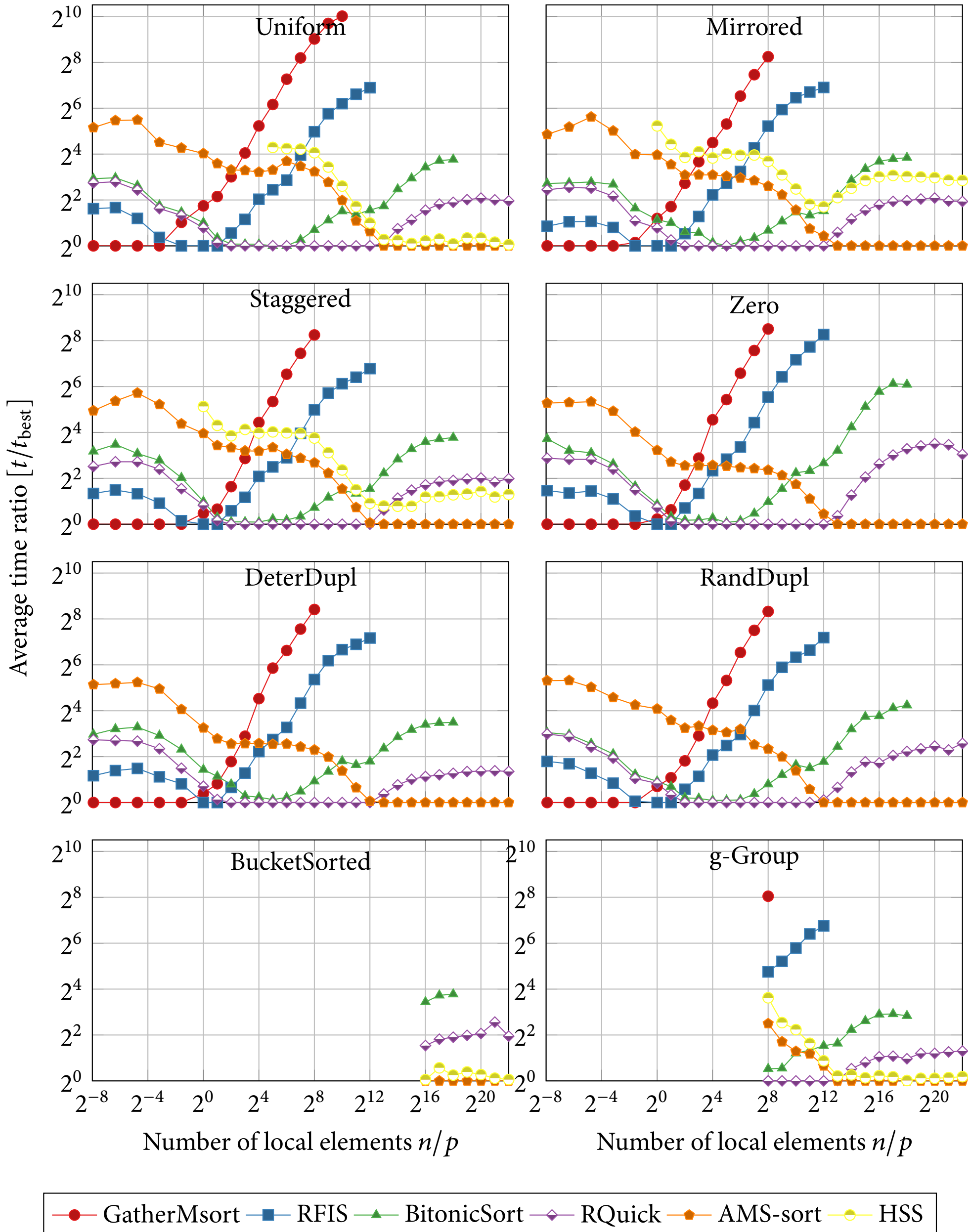


Figure 7.4: Running time ratios of each algorithm to the fastest algorithm on SuperMUC-NG.

7.4 Robustness Analysis

We now have a closer look at the impact of various algorithmic measures to improve the robustness of our algorithms and discuss the robustness of our competitors. We consider three major issues with respect to the robustness of a massively parallel sorting algorithm: (1) Its *scalability*, i.e., its running time as a function of n and t, (2) how the algorithm behaves with respect to *skewed input distributions*, and (3) how it handles *repeatedly occurring keys*. We show results for a selection of input distributions: Uniform for randomly distributed input, DeterDupl for inputs with duplicates, and Staggered as well as Mirrored input for skewed respectively very skewed input. For the robustness experiments of AMS-sort, we also show results for the skewed input distribution AllToOne, which is a hard input for some recursive multiway algorithms.

We executed the robustness experiments on SuperMUC-NG and on JUQUEEN. However, we present results obtained on SuperMUC-NG in most cases. The reasons are manifold: First, the results on SuperMUC-NG were very similar to the results on JUQUEEN. Second, the nonrobust versions of the algorithms have an advantage on SuperMUC-NG as this supercomputer is equipped with more local memory and the number of PEs is lower. Third, SuperMUC-NG is currently listed at position 15 of the TOP500 list (November 2020) whereas JUQUEEN has already been decomposed. Finally, algorithms that initially seemed to be robust on JUQUEEN were not robust on SuperMUC-NG. An example is the routing of the ranked elements to their final PEs in RFIS. For $n/t < \log t$ it should be more efficient to route an element with rank r to PE rn/t with a single point-to-point message. However, on SuperMUC-NG these sparse data exchanges caused running time fluctuations and we achieved much more reliable results when we use a hypercube data routing algorithm. For a more detailed experimental analysis, we refer to Appendix B.1.

Overview. In Section 7.4.1, we examine the robustness of RFIS for different input distributions and compare its performance to the performance of different gather-based sorting approaches. Section 7.4.2 extensively evaluates the robustness of RQuick compared to the nonrobust version of RQuick for different input distributions. In this section, we also compare RQuick to quicksort for the special case $n/t = 1$ (Minisort from Siebert and Wolf [SW11]) and to HykQuick, a hypercube quicksort implementation from Sundar et al. [SMB13]. In Section 7.4.3, we discuss the robustness of our competitors HykSort and HSS. Finally, we extensively compare AMS-sort to nonrobust AMS-sort in Section 7.4.4. In this section, we also compare the performance of AMS-sort to the performance of the single-level sorting algorithms 1LAMS-sort and 1LHSS.

7.4.1 Robustness of Logarithmic Latency Algorithms

Motivation. In this section, we examine the robustness of algorithms with logarithmic message startups in three parts. First, we compare the running time of GatherMsort to much simpler gathering-based sorting algorithms and test the robustness of these simple algorithms in terms of scalability and efficiency. Second, we compare the cost of the ranking algorithm RFIR to the cost of converting the ranked elements to sorted output (RFIS). Finally, we study the robustness of RFIS to skewed inputs and inputs with duplicates.

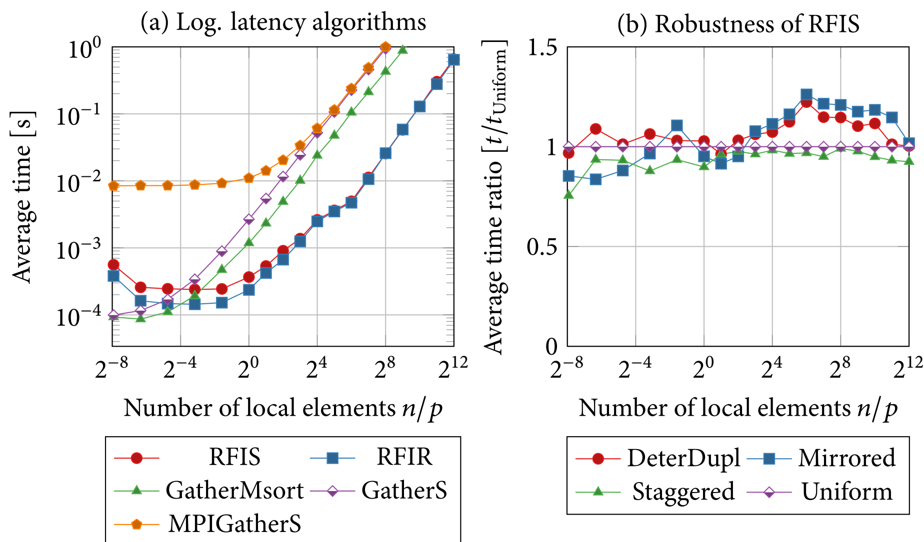


Figure 7.5: Running time comparison of gather-merge algorithms, RFIS, and RFIR for Uniform input (a) and running time comparison of RFIS for different input distributions (b) obtained on the SuperMUC-NG.

Sorting by Gathering. We compare our GatherMsort algorithm to *GatherS*, a simplified version of GatherMsort that sorts the whole input on the root PE, and *MPIGatherS*, a collective gather with MPI collectives (`MPI_Allgather` for the message size exchange and `MPI_Gatherv` for the actual element gathering) followed by local sorting of the result. Figure 7.5a depicts the running times for Uniform input. MPIGatherS is up to two orders of magnitude slower than GatherMsort and GatherS for small inputs. The reason is that `MPI_Gatherv` requires the local input sizes on each PE to handle arbitrary input sizes on each PE. These sizes are distributed with an `MPI_Allgather` operation that has a communication volume linear to t. In contrast, GatherMsort and GatherS only need $\log t$ message startups since they propagate the sizes with a binomial-tree operation to the parent PEs. For all inputs, GatherMsort is faster than GatherS, e.g., more than a factor of two for $n/t > 1$. The reason is that GatherMsort sorts the input locally and merges the elements while routing in parallel.

Overall, we have seen very poor scalability of the gathering-based sorting algorithms. In particular, the running time of MPIGatherS increased rapidly when we sorted sparse inputs on more than a hundred PEs. The fact that GatherMsort outperforms all algorithms for the smallest input comes with the price that it needs $\mathcal{O}(n)$ local memory. Therefore it was not surprising that gathering-based sorting ran out of memory for $n/t > 2^8$ on JUQUEEN.

From Ranking to Sorting. Figure 7.5a shows the running time of RFIS and RFIR. For very small inputs, RFIS is up to a factor of two slower than RFIR as the redistribution of the elements requires $\Theta(\log t)$ startup overheads. For larger inputs, these overheads are dominated by the

time for the actual data transfer and the local work. It turned out that the reduce operation, which calculates the final ranks, dominates the running time for large inputs. Thus, RFIS and RFIR have similar running times for these inputs.

Fast Work-Inefficient Sorting. Figure 7.5b compares the running time of RFIS for different input distributions. The time for routing the elements to their final PEs is limited by $\mathcal{O}(\alpha \log t + \beta n/t^{1/2})$. While input of some input distributions come close to this upper bound, other distributions need much less element transfers. For example, uniformly distributed inputs only need $\mathcal{O}(\alpha \log t + \beta n/t)$ with high probability. Thus, RFIS sorts large Uniform inputs up to 1.26 times as fast as DeterDupl and Mirrored inputs in our experiments. We see some running time fluctuations for sparse inputs (factor of ≤ 1.25). These seem to be random overheads caused by the machine or the MPI library as they occurred randomly over multiple runs. On JUQUEEN, we did not see these fluctuations.

7.4.2 Robustness of RQuick

Motivation. In this section, we study the robustness of our algorithm RQuick by comparing its running time and imbalance to other quicksort algorithms. First, we compare RQuick to a variant of RQuick without redistribution and tie-breaking (*NRQuick*) to illustrate the quality of these robustness measures. Then, we compare RQuick to HykQuick from Sundar et al. [SMB13], a simple implementation of hypercube quicksort that selects splitters by gathering and locally sorting a random sample. We expect that the approximate median selection of RQuick selects high-quality splitters while still being much faster than the splitter selection of HykQuick—even for random input. We want to note that we compare RQuick to an already refined version of HykQuick that uses our RBC library for communication. For more information, we refer to Appendix B.2 which illustrates that the refinements speed up the original version by up to a factor of six. Finally, we compare our algorithm to Minisort, which sorts inputs with one element on each PE. Minisort sorts inputs for arbitrary t by adjusting the PE-groups for the recursive calls depending on the splitter rank in the global input. However, we assume this approach to be slower than RQuick for two reasons: First, the number of recursion levels of Minisort will be larger. Second, the non-deterministic PE-groups cause communication between alternating communication partners. Preliminary experiments on several supercomputers have shown that those non-deterministic communication patterns are significantly slower when the main bottleneck is caused by message startups. For a detailed discussion, we refer to Appendix B.1.

Imbalance of RQuick and HykQuick. Figure 7.6a illustrates the maximum imbalance of RQuick for different input distributions and sizes. Inputs without duplicates have a very small imbalance, e.g. the maximum output size is $1.07n/t$ for $n/t = 2^{12}$ for Mirrored, Staggered, and Uniform. However, we also see considerable imbalances for almost all DeterDupl distributed inputs. An exception are large inputs for which the imbalance converges to the imbalance of the other input distributions. We want to note that we see the distribution DeterDupl as a worst case distribution for the tie-breaking algorithm of RQuick: Assuming a perfect splitter selection algorithm, a perfect splitter e would split the elements of a subcube into two parts of equal size on each recursion level—one part with elements smaller than e and one part with elements equal to e. The tie-breaking algorithm would then either perfectly balance the

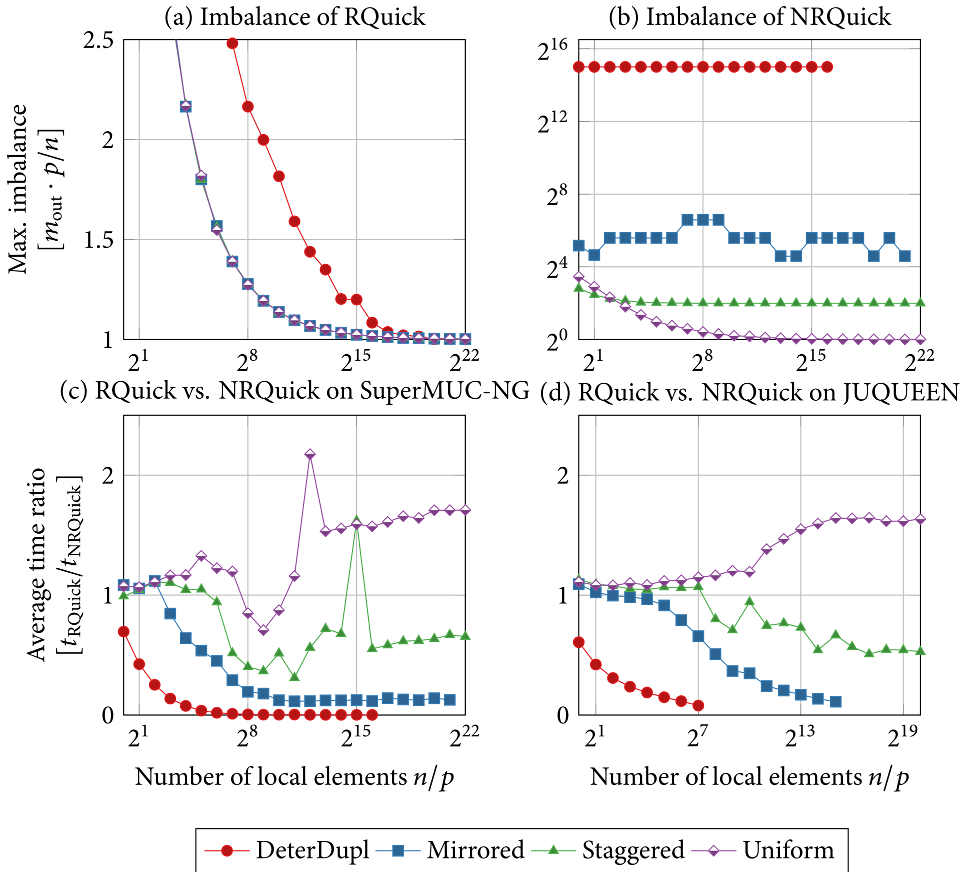


Figure 7.6: Average running time ratios of RQuick to NRQuick on SuperMUC-NG (a) and on JUQUEEN (b). Maximum imbalance ratio of RQuick (c) and NRQuick (d).

elements between two communication partners (in case the majority of the elements are equal to e) or the algorithm would move more than half of the elements to the communication partner with the smaller index (in case the minority of the elements are equal to e).

Looking at Figure 7.6b, we see that the imbalance of NRQuick (RQuick without tie-breaking and redistribution) deteriorates tremendously. Inputs with duplicates (DeterDupl) and very skewed inputs (Mirrored) come with huge imbalances and run out of memory for large inputs. Staggered sorts all input sizes but has an imbalance of around four. Uniformly distributed inputs (i.e., Uniform inputs) were the only inputs for which the imbalance of NRQuick was as low as the imbalance of RQuick.

Running Time Comparison of RQuick and HykQuick. Figure 7.6c-d depicts the running time ratio of RQuick over NRQuick on SuperMUC-NG and on JUQUEEN. The price of

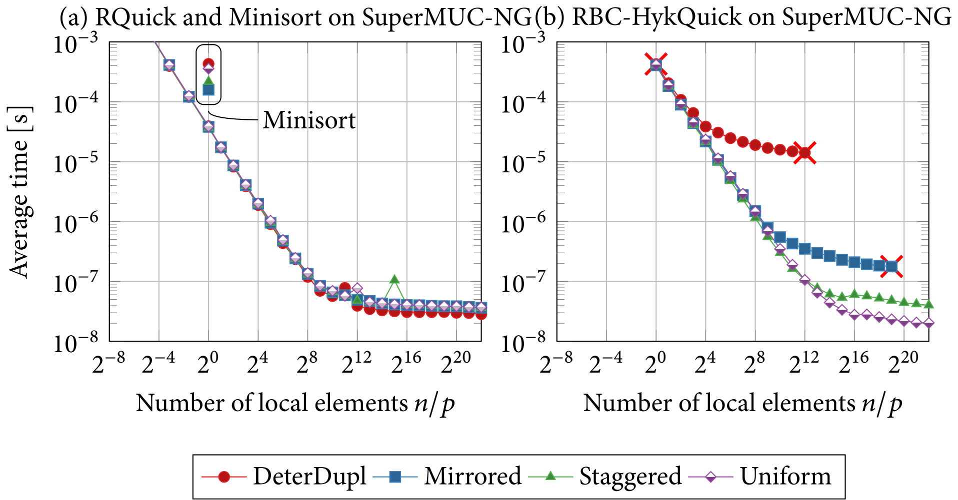


Figure 7.7: Running times of RQuick and Minisort (a) as well as RBC-HykQuick (b) for different input distributions on SuperMUC-NG. The red crosses indicate breaks due to memory overflow.

robustness for simple input distributions such as Uniform is an additional data redistribution. For large Uniform inputs, this slows down RQuick by a factor of up to 1.71. On the other hand, RQuick sorts skewed input distributions such as Staggered and Mirrored robustly. Thus, the redistribution makes RQuick robust against skewed inputs. Compared to NRQuick, it decreases the total running time by a factor of up to 9 ($n/t = 2^{15}$) on JUQUEEN. Figure 7.6c does not provide running times for skewed input distributions for $n/t \geq 2^{22}$ (Staggered input) respectively $n/t \geq 2^{17}$ (Mirrored input) as NRQuick runs out of memory for these input sizes on SuperMUC-NG. On JUQUEEN, NRQuick already starts to run out of memory for skewed inputs with 2^8 elements per PE.

Running Time of RQuick for Different Distributions. Figure 7.7a depicts the running time of RQuick for different input distributions. We see that the running time of RQuick is very robust, meaning that RQuick does not sort any input distribution slower than random inputs (i.e., Uniform inputs) with the exception of a few outliers. For small inputs ($n \leq 2^{12}$), the running time is about the same for all distributions. In the following, we consider large inputs, i.e., $n > 2^{12}$, for which the situation is more diverse. The sorting times of large skewed inputs (Staggered and Mirrored) are very similar and close to the running times of large Uniform inputs. Exceptions are large inputs with many duplicates, i.e., DeterDupl distributed inputs: RQuick sorts large DeterDupl inputs faster than the other input distributions, e.g., up to 1.31 times as fast as Uniform. At the first glance, this is surprising since RQuick has somewhat large imbalances when sorting DeterDupl inputs (see Figure 7.6c). However, the reasons for short sorting times of DeterDupl inputs are that the local sorting algorithm sorts these inputs faster (due to equality buckets) and that RQuick performs less element transfers (due to tie-breaking).

Comparison to Minisort and HykQuick. RQuick is up to one order of magnitude faster than Minisort. One reason is that Minisort trades off more recursion levels for zero imbalance. We also noticed that the splitter quality of Minisort is worse than the splitter quality of RQuick, especially for the skewed inputs Mirrored and Staggered. However, this does not fully explain the large performance gap between RQuick and Minisort: Even though Minisort selects worse splitters with distributions Mirrored and Staggered than with distributions DeterDupl and Uniform, the running times with the former distributions were shorter than with the latter distributions. To explain this observation, one needs to take a look at the number of different communication partners separately for each input distribution. Since the distributions Mirrored and Staggered are skewed, the path an element takes to its final PE is more deterministic. And indeed, the average number of different communication partners for DeterDupl and Uniform inputs over many runs were much larger than for Mirrored and Staggered inputs. This confirms our observation in Appendix B.1 that alternating communication partners increase the message startup overhead.

Figure 7.7b shows the running times of our competitor HykQuick for different input distributions. Compared to RQuick (see Figure 7.7a), HykQuick is much slower and much less robust than RQuick: (1) HykQuick deadlocks for sparse inputs. (2) For small inputs, HykQuick is more than one order of magnitude slower than our algorithm. We found that the splitter selection of HykQuick needs a tremendously large amount of time. (3) HykQuick runs out of memory while sorting large Mirrored inputs as its hypercube routing algorithm gathers too many elements on "hot" PEs. This is a known issue of naive hypercube routing algorithms (see Section 5.1.1). (4) HykQuick runs out of memory for relatively large DeterDupl inputs as the algorithm does not sort inputs with duplicates robustly. Only the largest Uniform inputs are sorted faster by HykQuick in comparison to our algorithm. The reason is the redistribution that our algorithm uses to make skewed inputs easy. However, one would use multiway sorting algorithms for these large inputs anyway.

7.4.3 Robustness of Multiway Hypercube-Based Sorting

Motivation. In Section 7.3, we have already noticed that our competitors HykSort and HSS are not robust in several aspects. This is not surprising since hypercube-based routing is known to have severe intermediate load imbalances for worst case inputs (see the discussion in Section 5.1.1). Here, we study the robustness of HykSort and HSS based on the results obtained on JUQUEEN and SuperMUC-NG in more detail (see Figures 7.1–7.4 of Section 7.3).

Experimental Results. Both algorithms are not robust for three reasons. The first reason is that HykSort and HSS only work when $t = k^r$ where r is the number of levels. The second reason is that the implementations are rather prototypical. Both algorithms crash for Zero, DeterDupl, and RandDupl inputs as they do not break ties. Additionally, they do not support sparse inputs even though we have already fixed some bugs that caused them to crash for small inputs. The third reason is that the algorithms do not sort skewed inputs robustly. On the first recursion level, they route the input of the distribution Staggered to $t/2$ PEs. The load imbalance is even worse for Mirrored input: As the multiway hypercube algorithms perform $\log k$ levels of hypercube data routing in one step, after $r = \lfloor \frac{\log_k t}{2} \rfloor$ recursions of HykSort, t/k^r

PEs hold nk^r/t elements each. As a result, HSS sorts Staggered (Mirrored) input 2.25 (6.95) times slower than non-skewed Uniform inputs on SuperMUC-NG ($n/t = 2^{22}$). On JUQUEEN, HykSort even runs out of memory while sorting Mirrored input with more than 2^{16} elements per PE. The algorithms do not break for this input distribution on SuperMUC-NG, as we have more local memory and fewer PEs available than on JUQUEEN.

To run HykSort and HSS for any number of t, it would be possible to send the data from the additional PEs to the embedded hypercube in a preprocessing step. To become robust against skewed inputs, a preprocessing step could redistribute the input. We expect both approaches as not practical since they would increase the local work and the communication volume significantly. This would only have little effect on their running time for relatively small input sizes. However, one would prefer RQuick for these inputs anyway. These robustness measures, however, turned out to work quite well for RQuick: For small inputs, the element redistribution did not increase the performance of RQuick much since it only adds $\log t$ additional message startups and RQuick still remains about as fast as BitonicSort. For larger inputs, i.e., for which RQuick is bandwidth- and compute-bound, RQuick with element redistribution is still twice as fast as its closest competitor BitonicSort.

7.4.4 Robustness of AMS-sort

Motivation. In the first part of this section, we show that our recursive multiway algorithm AMS-sort is much more efficient than single-level sorting algorithms for the input sizes we consider in this work. The second part studies the robustness measures of AMS-sort in detail. In Section 7.3 we have already seen that AMS-sort sorts all kinds of input distributions robustly and very efficiently whereas its closest competitors HSS and HykSort are less efficient for small inputs and much less robust when the input becomes larger. However, without the robustness measures of AMS-sort, namely our message assignment algorithm DMA and our tie-breaking approach, the situation would be very different. To illustrate the advantages of our robustness measures we show that they come with low overhead for "easy" inputs but reduce the running time of AMS-sort tremendously for worst case inputs.

Comparison of Message Assignment Algorithms. Figure 7.8a-b shows the running time ratio of AMS-sort over SMA-AMS, a version of AMS-sort that uses the simple message assignment algorithm proposed by Kalé and Krishnan [KK93] (see Section 5.2) instead of our deterministic message assignment algorithm. We present results for different input distributions obtained on SuperMUC-NG and JUQUEEN. DeterDupl, Mirrored, Staggered, and Uniform are "easy" inputs for the message assignment algorithms DMA and SMA, i.e., regardless of the message assignment algorithm that AMS-sort uses, the number of messages that are sent and received by each individual PE is very small. Our experiments show that the overhead of DMA over SMA for these inputs is very small. An exception are inputs with $n/t \approx k$ for which DMA has an overhead of up to a factor of two. However, for such small inputs RQuick would be used preferably in any case. For the input distribution AllToOne, SMA-AMS sends $\min(n/t, t)$ messages to PE 0 on the first recursion level. In this case, AMS-sort actually can take advantage of DMA as it reduces the startups in the data exchange step to $\mathcal{O}(k)$. This speeds up AMS-sort over SMA-AMS by a factor of up to 1240 on SuperMUC-NG. On JUQUEEN, the factor is 5.21

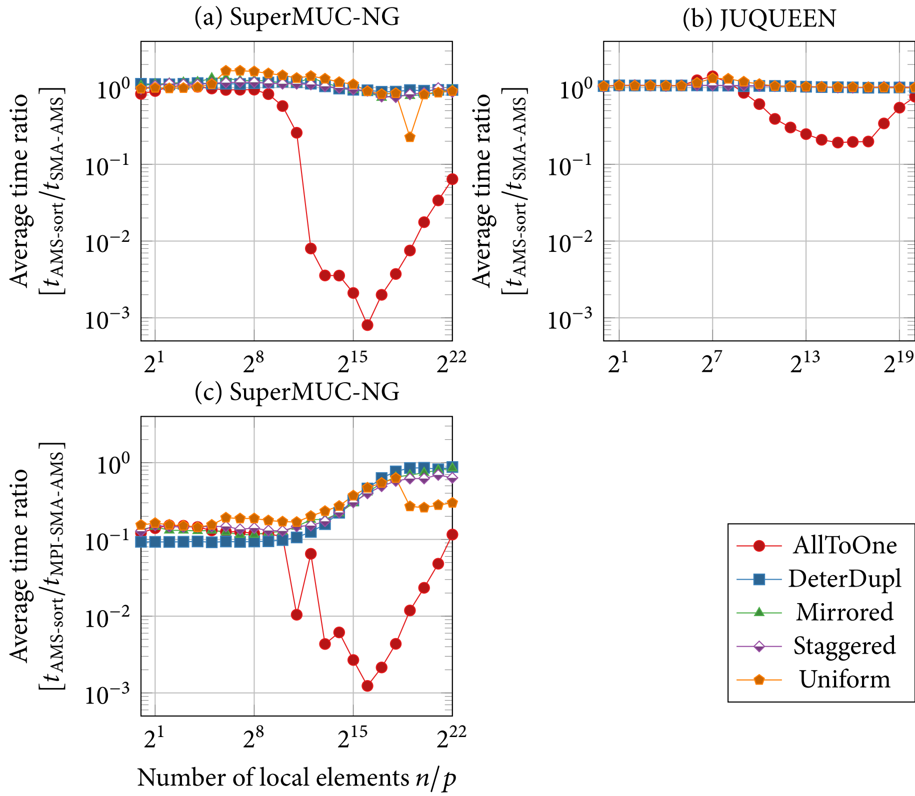


Figure 7.8: Running time ratios of different algorithms with and without DMA. The results were obtained on the SuperMUC-NG using 2^{16} cores and on the JUQUEEN using 2^{17} cores.

as JUQUEEN has very small message startup costs. The positive effect begins for $n/t \geq 256$ elements per PE and increases rapidly. The effect decreases when the time for the message exchange dominates the startup time. One might assume that SMA-AMS sorts AllToOne inputs slowly as we use our own implementation for the actual data exchange step. Thus, we conclude the benchmark by presenting results of MPI-SMA-AMS, a version of SMA-AMS that uses collective operations provided by MPI, e.g., for communicator creation and for the data exchange step (`MPI_Alltoall` for exchanging the message sizes and `MPI_Alltoallv` for the data exchange). Figure 7.8c depicts the running times of MPI-SMA-AMS relative to the ones of AMS-sort obtained on SuperMUC-NG. MPI-SMA-AMS sorts AllToOne inputs two orders of magnitude slower than AMS-sort. Furthermore, MPI-SMA-AMS is more than one order of magnitude slower for inputs of small and medium size—regardless of the input distribution.

In preliminary experiments, we have also tested AMS-sort with a straightforward implementation of the data exchange step, i.e., the data exchange step posted all sent and receive

operations at once and counted on the MPI library to execute the operations as efficiently as possible. This approach turned out to be relatively fast when AMS-sort uses our message assignment algorithm DMA. However, when we disable DMA and use the simple message assignment algorithm (SMA) instead, the MPI library on JUQUEEN crashed while sorting AllToOne inputs as the PEs posted to many messages simultaneously.

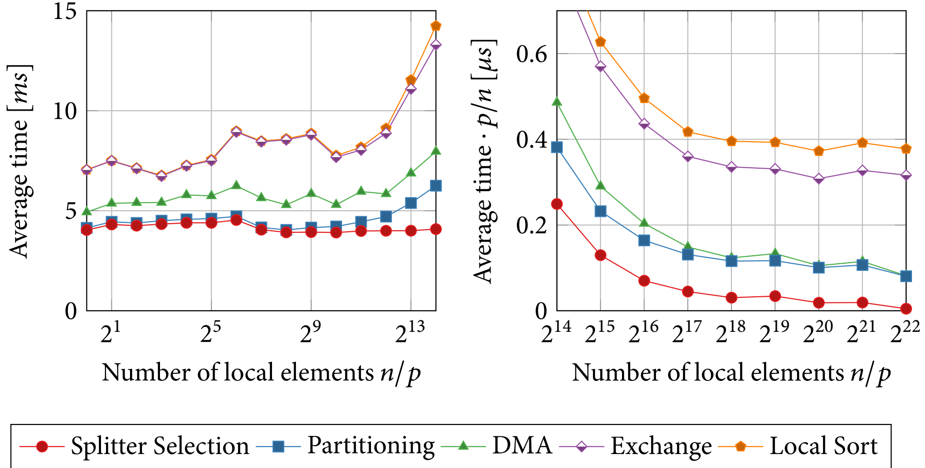


Figure 7.9: Accumulated running time of the phases of AMS-sort on SuperMUC-NG.

Subroutines of AMS-sort. We divide each level of AMS-sort into five distinct phases: splitter selection, partitioning, deterministic message assignment (DMA), data exchange, and local sorting. To measure the time of each phase, we place a barrier before each phase. Timings for these phases are accumulated over all recursion levels. The splitter selection phase includes the sample selection and sorting as well as the overpartitioning algorithm.

Figure 7.9 shows the running times of the phases for Uniform inputs on SuperMUC-NG. The extra cost for the message assignment DMA is relatively small. For example, DMA takes less than 19 % of the total running time and less than 13 % when we look at the range where AMS-sort is faster than RQuick ($n/t > 2^{13}$). For a comparison of AMS-sort and RQuick, we refer to Section 7.3. For small inputs ($n/t \leq 2^{12}$), the splitter selection takes more than 50 % of the total running times. This appears to be large at a first glance. However, even when we exclusively consider the cost of the data exchange step, AMS-sort would not be faster than RQuick for small inputs. For inputs larger than $n/t \geq 2^{17}$, the running time increases almost linear and the phases partitioning, data exchange, and local sorting dominate the total running time with 85 % and more.

Tie-Breaking. Figure 7.10 shows the running time ratio of AMS-sort and AMS-sort without tie-breaking during local data partitioning (*NTB-AMS*) on SuperMUC-NG. When sorting small inputs without unique keys (e.g., AllToOne, Mirrored, Staggered, and Uniform), the extra work to calculate splitters with tie-breaks slightly slows down AMS-sort compared to

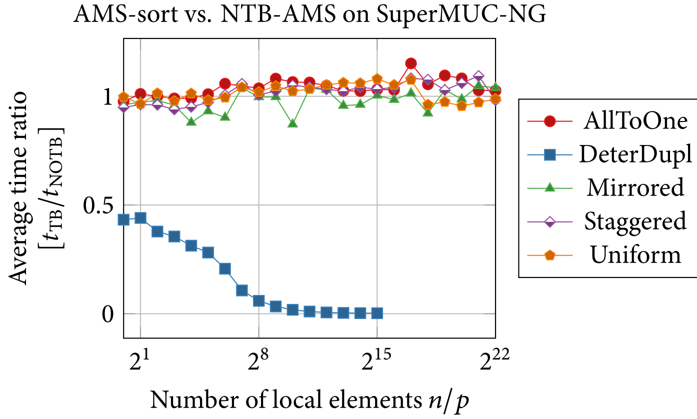


Figure 7.10: Ratio of AMS-sort and NTB-AMS on the SuperMUC-NG.

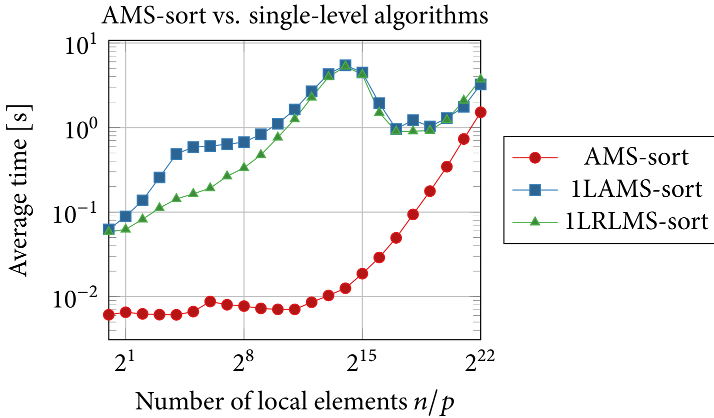


Figure 7.11: Running times of single-level algorithms and AMS-sort on SuperMUC-NG.

NTB-AMS (factor of 1.10). NTB-AMS sorts inputs of the distribution DeterDupl much slower than AMS-sort as this distribution contains many duplicates. The nonrobust version runs out of memory for $n \geq 2^{16}$ when we sort DeterDupl inputs because too many keys were gathered on "hot" PEs.

Comparison to Single-Level Algorithms. We sum up the robustness analysis of (recursive multiway) AMS-sort by comparing its performance to the performance of single-level sorting algorithms. The fastest single-level algorithms we have found for the input sizes considered in this work are our reimplementation of single-level histogram sort with sampling 1LHSS (see Section 7.1) and our multiway AMS-sort algorithm executed with $k = t$ (1LAMS-sort).

1LAMS-sort and 1LHSS use pure MPI for communication. In particular, they exchange the message sizes with the MPI collective `MPI_Alltoall` and exchange the data with the MPI collective `MPI_Alltoallv`. To achieve the best performance of 1LHSS and 1LAMS-sort, we also tested both algorithms with RBC collectives in a preliminary experiment. The versions with RBC collectives perform the data exchange with our data exchange routine described in Section 5.2. 1LAMS-sort and 1LHSS with pure MPI were faster for $n/t > 2^8$ (and much faster for $n/t > 2^{15}$) since our own data exchange routine has not been optimized for a dense t-way data exchange. For smaller inputs, 1LAMS-sort and 1LHSS were up to twice as fast when we use RBC.

Figure 7.11 depicts the running times of AMS-sort, 1LAMS-sort, and 1LHSS obtained on SuperMUC-NG for Uniform inputs. AMS-sort is between one and two orders of magnitude faster than the single-level algorithms for inputs of small and medium size ($n/t \leq 218$). For the largest input ($n/t = 2^{22}$), AMS-sort is still faster by a factor of 2.13.

Both, the data exchange as well as the splitter selection contribute to the poor performance of 1LAMS-sort and 1LHSS. It turned out that the splitter selection contributes to the performance very much in particular for $n/t \geq 2^{10}$. For larger inputs the t-way data exchange dominates the running time. Looking at Figure 7.11, we see that 1LAMS-sort and 1LHSS have a peak around $n/t = 2^{14}$. For these input sizes, the data exchange operation `MPI_Alltoallv` is particularly slow. We assume that `MPI_Alltoallv` switches to a different implementation for larger inputs.

In preliminary experiments, we also tested the multiway hypercube implementation of histogram sort with sampling provided by Harsh et al. [HKS19] with $k = t$. However, our implementation 1LHSS turned out to be faster as the data exchange routine `MPI_Alltoallv` performed better than the k-way hypercube exchange with $k = t$.

7.5 Efficiency in Scaling Experiments

The experimental setting of the efficiency test is as follows: We benchmarked RFIS, RQuick, and AMS-sort on 2^i, $i \in [10..16]$ PEs for different input sizes with Uniform inputs obtained on SuperMUC-NG. For each algorithm, we have chosen input sizes for which the particular algorithm is faster than its competitors. Table 7.1 shows the efficiency of these algorithms compared to our sequential sorting algorithm I1S^{4}o.

RFIS and RQuick. Even though RFIS respectively RQuick sort small inputs ($n/t \leq 2^{11}$) the fastest, they are still very inefficient. RQuick, for example, has an efficiency of less than 0.1 on all machine sizes. For 2^9 elements per PE, its efficiency is only 0.043 on 2^{16} PEs. The efficiency of RFIS is even smaller, e.g., less than 0.0002 on 2^{16} PEs for $n/t \leq 2$. However, RFIS sorts these input sizes still 12 times as fast as the sequential sorting algorithm I1S^{4}o. This is still a pessimistic comparison we stored the input of I1S^{4}o on a single PE.

AMS-sort. When we assume that the local input size is constant, the efficiency of AMS-sort increases by a factor of 2.2 to 2.6 when we use 2^{10} instead of 2^{16} PEs (strong scaling). When we assume that the number of PEs is constant, i.e., 2^{10} PEs respectively 2^{16} PEs, the efficiency of AMS-sort also increases by a factor of 2.2 to 2.6 when we use 2^{22} instead of 2^{16} elements on each PE (scaling the local input size). For $n/t \geq 2^{18}$, the efficiency changes only slightly. This input size seems to be a turning point. When the number of local elements goes below 2^{18}, the

Algorithm	n/p	p 2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
AMS-sort	2^{22}	0.462	0.437	* 0.375	* 0.320	* 0.306	* 0.275	* 0.207
	2^{20}	0.435	0.425	0.379	0.318	* 0.293	* 0.248	* 0.190
	2^{18}	0.425	0.403	0.357	0.283	0.272	0.247	* 0.197
	2^{16}	0.347	0.278	0.271	0.206	0.194	0.172	0.150
	2^{14}	0.217	0.179	0.156	0.128	0.121	0.096	0.083
RQuick	2^{11}	0.099	0.086	0.071	0.062	0.060	0.051	0.043
	2^{9}	0.070	0.050	0.031	0.029	0.027	0.019	0.018
	2^{7}	0.026	0.023	0.020	0.017	0.008	0.009	0.008
	2^{5}	0.008	0.006	0.005	0.004	0.004	0.002	0.002
RFIS	2^{1}	0.00115	0.00089	0.00045	0.00034	0.00035	0.00026	0.00018
	2^{0}	0.00057	0.00047	0.00047	0.00019	0.00018	0.00018	0.00011

Table 7.1: Efficiency of sorting algorithms compared to I1S^{4}o on the SuperMUC-NG. A single node was not able to store inputs with $n \geq 2^{34}$ elements. We linearly scaled the running time of I1S^{4}o for these inputs and labeled the efficiency with a "*".

efficiency decreases rapidly—independent of the machine size. Note that we had to extrapolate the running time of I1S^{4}o for $n \geq 2^{34}$ as a single node would not store these large inputs.

7.6 Conclusion

We have shown how practical parallel sorting algorithms like recursive multiway mergesort and samplesort can be generalized so that they scale on massively parallel machines without incurring a large additional amount of communication volume. Our implementation of AMS-sort shows very competitive performance that is probably the best by orders of magnitude for large t and moderate n. For large n it can compete with the best single-level algorithms. Indeed, AMS-sort is the only scalable recursive multiway sorting algorithm that works for arbitrary t. For smaller inputs, we devised RQuick and RFIS, two work-inefficient, yet practical and fast algorithms with (poly)logarithmic latency.

So far, most "practical" research on massively parallel sorting algorithms had focused on average case inputs. Our work closed the gap between these algorithms and theoretical publications on achieving asymptotically fast and efficient parallel sorting algorithms, usually involving large constant factors, by turning worst case inputs into average cases. The result is that we obtain robustness of massively parallel sorting with respect to the input size by using three algorithms: RFIS for sparse inputs and very small inputs, RQuick for small inputs, and AMS-sort for moderate and large inputs. Robustness with respect to skew can be achieved by careful randomization (RQuick) and clever message assignments (AMS-sort). Our competitors

HSS and HykSort and nonroubst implementations of our algorithms crash on sorting hard input distributions with skewed input due to tremendously large load imbalances (HykSort, HSS, and nonrobust RQuick) or become extremely slow due to immense message startup overheads (nonrobust AMS-sort, also see algorithms such as [KK93]). Robustness with respect to duplicated keys can be achieved at reasonable overhead using careful implicit implementations of the brute-force tie-breaking idea (RFIS), equality buckets (AMS-sort), and data shuffling in combination with greedy tie-breaking (RQuick). All these algorithms would crash on sorting hard input distributions with duplicated keys.

Our communication library RBC closed the gap between theory and practice a step further by providing asymptotically optimal implementations for range-based communicator splitting and collective communication operations. If we had relied on MPI functionality, scalable implementations of our algorithms would have been impossible. Moreover, a fair comparison to our competitors would have been impossible, too. Only after applying RBC to our competitors, their running times matched with the running times one would expect from the theoretical analysis. In regard to our competitors, RBC speeds up their running time by multiple orders of magnitude for inputs of small and medium size.

Outlook. Still further improvements are possible. For large inputs, more overlapping of communication and computation, as well as a specialized shared-memory implementation for node-local partitioning, seems useful for AMS-sort. We could combine RFIS, RQuick, and AMS-sort to a single robust algorithm. A simple solution could perform binary searches on several machine sizes to determine the range of input sizes that each algorithm sorts the fastest (and interpolate the result).

Another approach could combine the advantages of gather-merge-sort, RQuick, and recursive multiway samplesort by adaptively selecting the appropriate sorting routine for each level of recursion. As long as the startup overheads for the splitter selection dominate the remaining running time of RQuick, a gather-merge-sort could ship all data to few PEs and replaces the first levels of RQuick. The last levels of RQuick could again be replaced by a multiway samplesort routine or by a second gather-merge-sort, which collects all data of each subcube to its first PE.

On the SuperMUC supercomputers, we measured large fluctuations in the message startup latency on very large machine instances (SuperMUC-NG) and on some smaller instances (SuperMUC Phase 1–2). On the one hand, one could try to develop sorting algorithms being more robust to these fluctuations. On the other hand, our experiments indicated that the fluctuations depend on the specific MPI implementation and the supercomputer at hand. In our opinion, a more universal solution would therefore be more robust MPI implementations. If we go a step further, one would want fault-tolerant sorting algorithms with low overhead.

Appendix

Appendix A

Sequential and Shared-Memory Sorting A

In Appendix A.1, we limit the recursion depth of IPS^{4}o. Appendix A.2 describes a technique how to store the recursion stack of IPS^{4}o implicitly—a requirement to make the space consumption of IPS^{4}o independent of n. In Appendix A.3, we report additional measurements of sequential and shared-memory sorting algorithms.

A.1 Limit the Number of Recursions

We prove the following theorem:

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

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

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

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

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

The probability $\Pr[X_{ir} = 1] = 1.5\frac{n}{k} \cdot \frac{1}{n} = \frac{1.5}{k}$ is independent of the sample s_i as the samples are selected with replacement. Thus, the expected value of the number of samples selected from R_r is $X_r = \sum_{i=1}^{\alpha k} X_{ir}$ is $E\left[X_r\right] = 1.5/k \cdot \alpha k = 1.5\alpha$. We use the Chernoff bound to limit the probability of less than α samples in R_r to $\Pr[X_r < \alpha] = \Pr[X_r < (1-1/3)E[X_r]] < e^{-1/2(1/3)^2 E[X_r]} = e^{-1/12\alpha}$. When R_j as well as $R_{j-1.5n/k}$ both provide at least α samples, R_j as

well as $R_{j-1.5n/k}$ provide a splitter and e_j is in a bucket containing at most $3n/k$ elements. The probability is $\Pr[X_j \geq S \wedge X_{j-1.5n/k} \geq S] = 1 - \Pr[X_j < S \vee X_{j-1.5n/k} < S] \geq 1 - \Pr[X_j < S] - \Pr[X_{j-1.5n/k} < S] > 1 - 2e^{-1/12\alpha} = 1 - 2k^{-1/12c}$. □

Lemma A.3 (Limit recursion depth of an arbitrary element)

Let c be a constant, let $\alpha = c\log k$ be the oversampling ratio of IPS⁴o ($c \geq 36 - 2.38/\log(0.34\cdot k)$), and let IPS⁴o execute $2\log_{k/3}\frac{n}{M}$ recursion levels. Then, an arbitrary but fixed input element of IPS⁴o passes at least $\log_{k/3}\log\frac{n}{M}$ successful recursion levels with a probability of at least $1-(n/M)^{-2}$.

Proof. We execute IPS⁴o $2\log_{k/3}\frac{n}{M}$ recursion levels and bound the probability that an arbitrary but fixed input element passes at least $\log_{k/3}\log\frac{n}{M}$ successful recursion levels. This experiment is a Bernoulli trial as we have exactly two possible outcomes, "successful recursion step" and "non-successful recursion step", and the probability of success is the same on each level. Let denote the random variable X as the number of non-successful recursion steps after $2\log_{k/3}\frac{n}{M}$ recursion levels, t the probability of a non-successful recursion step, and let $c \geq 36 - 2.38/\log(0.34\cdot k)$. The probability I

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

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

Proof of Theorem A.1. We first assume that $M \geq k^2 n_0$ holds. In this case, we select $kc\log k$ samples. Let $l = \log_{k/3}\frac{n}{M}$ and let e be an arbitrary but fixed input element of IPS⁴o after $2l$ recursion levels. Lemma A.3 tells us that e has passed at least l successful recursion steps with a probability of at least $1-(n/M)^{-2}$ when IPS⁴o has performed $2\log_{k/3}\frac{n}{M}$ recursion levels. Element e is, in this case, in a bucket containing more than $n(3/k)^l = M$ elements as each successful recursion step shrinks the bucket by a factor of at least $3/k$. Let $E = [e_1 .. e_n]$ be

the input elements of IPS^{4}o in sorted order and let $Q = \{e_{iM} | 1 \leq i < n/M \wedge i \in \mathbb{N}\}$ every n/M-th element. We now examine buckets containing elements in Q after $2l$ recursion levels. The probability that any element Q is in a bucket containing more than M elements is less than $n/M \cdot (n/M)^{-2} = (n/M)^{-1}$—this follows from the former insight and Boole's inequality. In other words, the probability that all elements in Q are in buckets containing less than M elements is larger than $1 - M/n$. As this holds for all elements in Q, every n/M-th element in the sorted output, the probability that all elements after $2l$ recursion level are in buckets containing less than M elements is larger than $1 - M/n$. □

A.2 From In-Place to Strictly In-Place

We now explain how the recursion stack of IPS^{4}o can be implicitly represented in a rather simple way by adapting the strictly in-place approach of quicksort [Dur86]. The technique which we describe—making the space consumption of IPS^{4}o independent of n—requires that (sequential and parallel) partitioning steps mark the beginning of each subtask by storing the largest element of a subtask in its first position. We now exploit that the sequential tasks of a PE are stored next to each other when the PE has finished its last parallel task:

Lemma A.4 (Sequential tasks of a PE are adjacent)

When PE i starts its sequential phase, the sequential tasks assigned to PE i cover a consecutive subarray $A[l \mathinner{..} r-1]$ *of the input array, i.e., there is no gap between the tasks in the input array.*

For reasons of better readability, we appended the proof of Lemma A.4 to the end of this section. The proof of this lemma also implicitly describes the technique to track the values of l and r until the last parallel task had been processed.

In the sequential phase, the PEs have to sort their elements $A[l \mathinner{..} r-1]$, which are already partitioned into sequential tasks. The boundaries of the sequential tasks are implicitly represented by the largest element at the beginning of each task. Algorithm 14 emulates recursion on the sequential tasks of $A[l \mathinner{..} r-1]$ in constant space.

Algorithm 14 Sequential task execution of the subarray $A[b, e-1]$ without a local stack

```
Input: A[0..n−1] an array of n input elements, b begin of the subarray, e end of the subarray
n ← r − l                                                ▷ total size of sequential tasks
i ← l                                                    ▷ first element of current task
j ← searchNextLargest(A[i], A, i + 1, n)                 ▷ first element of next task
while i < n do
    if onlyEqualElements(A, i, j) then i ← j             ▷ skip equal tasks
    else if j − i < n_0 then smallSort(A, i, j);  i ← j  ▷ base case
    else partition(A, i, j)                              ▷ partition first unsorted task
    j ← searchNextLargest(A[i], A, i + 1, n)             ▷ find beginning of next task
```

Starting a search at the leftmost task, the first element larger than the first element of a task defines the first element of the next task. Note that the time required for the search is

only logarithmic to the task size when using an exponential/binary search. We assume that the corresponding function *searchNextLargest* returns $n + 1$ if no larger elements exist—this happens for the last task. The function *onlyEqualElements* checks whether a task only contains identical elements. We have to skip these "equal" tasks to avoid an infinite loop. The following pseudocode uses this approach to emulate recursion in constant space on the sequential tasks.

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

Proof of Lemma A.4. Let PE i process a parallel task $T[l,r)$ with PE group $[\underline{t}..\overline{t})$ in one of the following states:

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

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

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

Assume for now that these propositions hold—we will prove them later. We use the proposition to show that Lemma A.4 holds when a PE i starts processing its sequential tasks: Let $T[l,r)$ be the last parallel task of PE i, executed with the PE group $[\underline{t}..\overline{t})$. No matter in which state $T[l,r)$ has been executed, the sequential tasks of PE i cover a consecutive subarray of the input array at the beginning of its sequential phase.

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

interrupted by a parallel task. Also, the subtasks are added in sorted order from left to right.

We will now prove the propositions by induction.

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

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

- *PE i has executed $T[l, r)$ in the state Middle and PE i is executing $T[l_s, r_s)$ in the state Middle.* From the induction hypothesis, we know that PE i did not have sequential tasks when $T[l, r)$ was executed. We have to show that PE i did not get sequential subtask of $T[l, r)$, after $T[l, r)$ has been executed. A sequential subtask $T[a, b)$ would have been added to PE i, if $i = \min(at/n, t - 1)$. We show that no $T[a, b)$ with this property exists. As PE i is not the rightmost PE of $T[l, r)$, we have $i < \overline{t} - 1$. This means that a sequential subtask $T[a, b)$ is only assigned to PE i if $i = \lfloor at/n \rfloor$ holds, i.e., $a \in [in/t, (i + 1)n/t)$ is required. However, there is no sequential subtask of $T[l, r)$ that begins in the i-th stripe of the input array: As PE i is not the leftmost PE of $T[l_s, r_s)$, the parallel subtask $T[l_s, r_s)$ contains the subarray $A[in/t, (i + 1)n/t - 1]$ completely (see Lemma 3.5). Thus, a second (sequential) subtask $T[a, b)$ with $in/t \leq a < (i + 1)n/t$ cannot exist.

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

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

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

A.3 More Measurements

		2^{8}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}	2^{32}
I4x20	I1S^{4}o	2.02	0.84	0.75	**0.55**	**0.45**	**0.37**	**0.32**	**0.30**	**0.17**	**0.16**	**0.15**	**0.15**	**0.15**
	BlockPDQ	**1.09**	**0.75**	0.69	0.61	0.49	0.41	0.37	**0.30**	0.20	0.19	0.18	0.19	0.19
	1S^{4}o	1.42	1.00	**0.62**	0.57	0.47	0.42	0.38	**0.30**	0.24	0.21	0.24	0.21	0.23
	std::sort	1.28	1.20	1.04	1.21	0.92	0.82	0.76	0.45	0.42	0.41	0.41	0.41	0.41
	I1S^{2}Ra	**0.85**	0.80	0.50	0.44	0.46	**0.29**	**0.25**	0.25	**0.14**	**0.11**	**0.11**	**0.12**	**0.12**
	SkaSort	0.86	**0.62**	0.52	0.66	0.43	0.32	0.28	**0.23**	0.16	0.13	0.13	0.15	0.13
	IppRadix	2.25	0.67	**0.49**	**0.39**	**0.37**	0.36	0.33	0.35	0.27	0.24	0.28	0.22	0.21
A1x16	I1S^{4}o	0.68	**0.37**	**0.24**	0.34	0.23	**0.16**	**0.16**	**0.14**	**0.13**w	**0.12**	**0.12**	**0.12**	
	BlockPDQ	**0.43**	0.29	0.36	0.25	0.24	0.22	0.20	0.18	0.18	0.18	0.18	0.17	
	1S^{4}o	0.87	0.42	0.35	**0.20**	**0.20**	0.19	0.19	0.17	0.16	0.15	0.15	0.15	
	std::sort	0.60	0.36	0.59	0.35	0.34	0.38	0.37	0.34	0.33	0.33	0.34	0.34	
	I1S^{2}Ra	0.65	0.43	**0.18**	**0.16**	**0.14**	**0.13**	**0.09**	**0.11**	**0.09**	**0.08**	**0.07**	**0.09**	
	SkaSort	**0.46**	**0.21**	0.32	0.37	0.17	0.16	0.14	0.16	0.12	0.11	0.11	0.13	
	IppRadix	0.74	0.30	0.25	0.21	0.23	0.27	0.25	0.24	0.21	0.19	0.19	0.17	
I2x16	I1S^{4}o	1.12	0.62	0.52	0.40	**0.35**	**0.33**	**0.31**	**0.29**	**0.28**	**0.28**	**0.31**	**0.30**	**0.25**
	BlockPDQ	**0.67**	**0.53**	**0.44**	0.40	0.37	0.36	0.34	0.33	0.32	0.31	0.36	0.34	0.33
	1S^{4}o	1.10	0.57	**0.44**	**0.39**	0.40	0.37	0.35	0.39	0.39	0.36	0.40	0.42	0.38
	std::sort	0.81	0.73	0.70	0.70	0.68	0.67	0.67	0.66	0.66	0.66	0.71	0.66	0.66
	I1S^{2}Ra	0.70	0.60	0.98	0.50	0.37	**0.26**	0.28	0.27	0.26	0.21	0.24	0.26	0.23
	SkaSort	**0.65**	**0.44**	**0.35**	**0.38**	**0.33**	**0.26**	**0.23**	**0.26**	**0.24**	**0.19**	**0.23**	**0.22**	**0.20**
	IppRadix	1.31	0.49	0.52	0.60	0.65	0.57	0.54	0.73	0.73	0.70	0.72	0.76	0.70
A1x64	I1S^{4}o	0.80	0.66	0.35	**0.24**	**0.21**	**0.18**	**0.18**	**0.17**	**0.16**	**0.15**	**0.15**	**0.14**	**0.14**
	BlockPDQ	**0.44**	**0.34**	**0.29**	0.26	0.24	0.23	0.22	0.21	0.20	0.19	0.18	0.18	0.17
	1S^{4}o	0.81	0.59	0.38	0.30	0.26	0.24	0.23	0.22	0.21	0.20	0.19	0.18	0.18
	std::sort	0.52	0.53	0.47	0.54	0.48	0.45	0.45	0.44	0.44	0.44	0.43	0.43	0.44
	I1S^{2}Ra	0.52	0.55	0.32	**0.23**	**0.18**	**0.14**	**0.11**	**0.13**	**0.12**	**0.10**	**0.08**	**0.10**	**0.09**
	SkaSort	**0.41**	**0.29**	**0.25**	0.29	0.21	0.17	0.16	0.20	0.16	0.13	0.12	0.14	0.11
	IppRadix	1.08	0.47	0.32	0.29	0.26	0.25	0.25	0.26	0.27	0.24	0.22	0.20	0.17

Table A.1: Running times in nanoseconds of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. The running times are scaled by $1/8n\log_2 n$. The fastest comparison-based and non-comparison-based running time is highlighted.

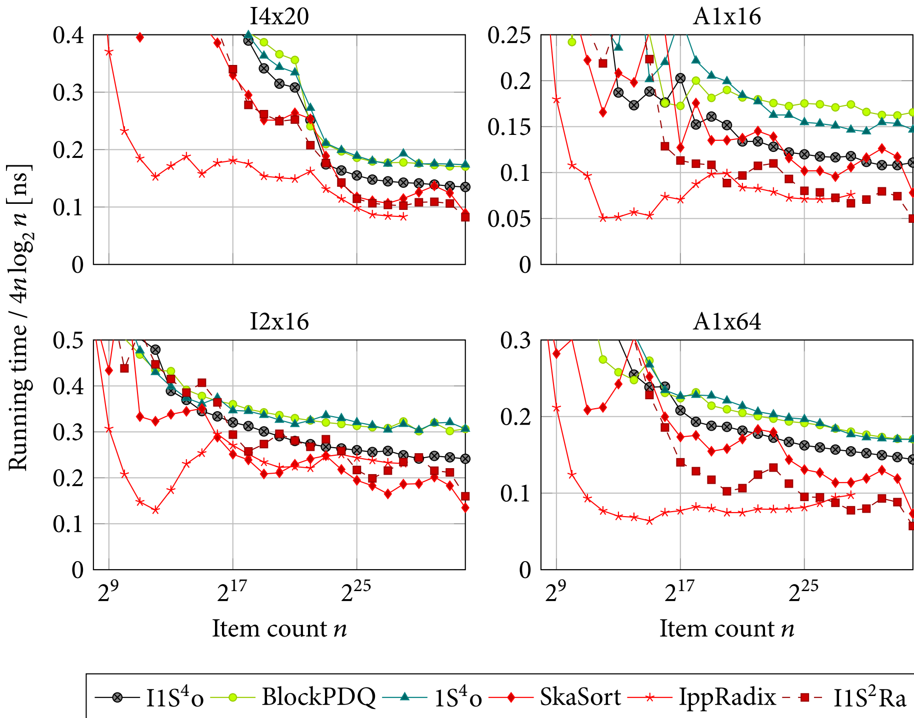


Figure A.1: Running times of sequential algorithms of uint32 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, and WikiSort cannot be seen as their running times exceed the plot.

Type	Distribution	I1S^{4}o	BlockPDQ	BlockQ	IS^{4}o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	IPS^{2}Ra
double	Sorted	1.11	1.77	20.31	1.04	11.54	17.14	**1.03**	51.78	2.90	16.90	49.53	
double	ReverseSorted	**1.04**	1.87	15.20	1.08	5.57	6.40	1.07	25.97	6.08	9.18	24.82	
double	Zero	1.14	1.93	17.48	1.11	1.29	13.56	**1.02**	2.93	3.72	13.09	18.38	
double	Exponential	**1.05**	1.10	1.24	1.28	2.31	2.56	4.15	3.79	3.98	1.27	1.12	
double	Zipf	1.19	1.32	1.44	1.45	2.86	3.07	4.77	4.23	4.84	1.26	**1.14**	
double	RootDup	**1.10**	1.39	1.73	1.62	1.51	2.50	1.50	5.43	2.81	1.70	2.43	
double	TwoDup	1.21	1.33	1.40	1.42	2.50	2.73	2.93	3.27	3.12	**1.11**	1.14	
double	EightDup	**1.02**	1.08	1.36	1.27	2.46	2.78	4.28	4.29	4.07	1.21	1.35	
double	AlmostSorted	2.22	**1.05**	1.87	2.79	1.57	1.62	1.25	5.85	2.26	2.05	4.22	
double	Uniform	1.13	1.24	1.27	1.32	2.47	2.57	3.62	2.94	3.56	1.14	**1.07**	
Total		1.23	**1.21**	1.46	1.53	2.19	2.51	2.89	4.15	3.43	1.36	1.55	
Rank		2	1	4	5	7	8	9	12	11	3	6	
uint64	Sorted	1.10	1.77	17.70	1.06	8.93	16.60	**1.06**	42.12	2.82	17.88	61.80	11.03
uint64	ReverseSorted	**1.02**	1.73	14.04	1.08	4.92	6.25	1.03	22.35	6.16	10.28	35.04	6.71
uint64	Zero	1.10	1.50	15.49	1.04	1.08	12.05	**1.04**	2.35	3.34	12.65	16.71	1.29
uint64	Exponential	1.09	1.22	1.35	1.40	2.65	2.84	4.69	3.74	4.62	1.23	1.37	**1.05**
uint64	Zipf	1.45	1.71	1.92	1.93	3.54	3.82	6.08	5.02	6.23	1.60	1.50	**1.04**
uint64	RootDup	**1.06**	1.44	1.77	1.70	1.43	2.55	1.64	5.16	3.24	1.70	2.28	1.08
uint64	TwoDup	1.55	1.84	1.89	2.00	3.34	3.57	4.02	4.07	4.29	1.37	2.32	**1.00**
uint64	EightDup	1.20	1.32	1.56	1.58	2.77	3.15	5.07	4.76	5.03	1.49	2.68	**1.02**
uint64	AlmostSorted	2.13	**1.06**	1.85	3.03	1.52	1.71	1.35	5.36	2.41	2.37	6.14	1.23
uint64	Uniform	1.28	1.47	1.51	1.63	2.84	2.97	4.24	3.18	4.32	1.17	1.57	**1.05**
Total		1.36	1.42	1.68	1.84	2.45	2.86	3.42	4.40	4.14	1.52	2.24	**1.06**
Rank		2	3	5	6	8	9	10	12	11	4	7	1
uint32	Sorted	2.84	4.29	49.59	2.87	24.36	60.07	**1.94**	121.52	6.43	35.09	48.04	27.84
uint32	ReverseSorted	1.55	2.27	20.24	1.46	6.38	11.16	**1.01**	31.74	5.86	9.79	29.71	8.44
uint32	Zero	2.56	3.97	48.98	2.53	2.26	33.81	**1.94**	6.54	9.05	20.41	12.16	3.12
uint32	Exponential	1.54	1.85	2.07	1.89	4.37	4.57	7.00	5.93	6.71	1.47	**1.03**	1.18
uint32	Zipf	1.89	2.31	2.65	2.40	5.27	5.67	8.57	7.40	8.91	1.33	1.20	**1.18**
uint32	RootDup	1.19	1.55	1.97	1.85	1.63	2.76	1.44	5.98	3.15	1.23	1.52	**1.11**
uint32	TwoDup	1.93	2.46	2.50	2.46	5.05	5.07	5.07	5.20	5.47	1.22	1.46	**1.10**
uint32	EightDup	1.34	1.64	1.99	1.77	4.17	4.56	6.50	5.74	6.43	1.22	1.83	**1.01**
uint32	AlmostSorted	2.65	1.25	2.21	3.50	1.83	2.74	**1.14**	6.79	2.45	2.08	4.92	1.33
uint32	Uniform	1.75	2.05	2.06	2.04	4.10	4.23	5.89	4.55	5.91	1.41	**1.00**	1.32
Total		1.70	1.83	2.19	2.21	3.46	4.09	4.09	5.88	5.15	1.40	1.58	**1.17**
Rank		4	5	6	7	8	10	9	12	11	2	3	1
Pair	Sorted	1.12	1.57	13.51	1.04	7.57	12.35	**1.02**	28.08	2.31	13.08		8.61
Pair	ReverseSorted	1.11	1.41	9.31	**1.01**	3.78	4.63	1.05	14.28	7.20	6.49		5.00
Pair	Zero	1.16	1.65	10.91	1.05	1.08	10.21	**1.03**	1.97	2.74	9.02		1.22
Pair	Exponential	1.15	2.05	1.29	1.38	2.11	2.45	4.26	3.19	4.18	1.25		**1.05**
Pair	Zipf	1.45	2.75	1.67	1.82	2.69	2.84	4.82	3.73	5.27	1.48		**1.02**
Pair	RootDup	1.20	1.46	1.68	1.71	1.44	2.30	1.86	4.39	3.78	1.60		**1.03**
Pair	TwoDup	1.74	3.04	1.83	2.01	2.90	3.10	3.74	3.56	4.41	1.47		**1.00**
Pair	EightDup	1.30	2.39	1.53	1.65	2.28	2.65	4.51	3.97	4.93	1.46		**1.01**
Pair	AlmostSorted	2.73	**1.02**	2.29	3.40	1.86	2.06	2.29	5.47	3.91	2.58		1.48
Pair	Uniform	1.41	2.54	1.47	1.71	2.46	2.48	3.82	2.88	4.22	1.24		**1.00**
Total		1.50	2.06	1.66	1.88	2.20	2.53	3.43	3.81	4.36	1.54		**1.07**
Rank		2	6	4	5	7	8	9	10	11	3		1
Quartet	Uniform	1.06	1.91	1.26	1.39	1.92	1.78	3.08	2.01	3.22	**1.04**		
Rank		2	6	3	4	7	5	9	8	10	1		
100B	Uniform	1.21	1.16	1.13	1.51	1.52	1.21	2.02	1.55	2.65	**1.09**		
Rank		4	3	2	6	7	5	9	8	10	1		

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

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

Table A.3: Average slowdowns of sequential algorithms for different data types and input distributions on A1x16. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	I1S^{4}o	BlockPDQ	BlockQ	1S^{4}o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	IPS^{2}Ra
double	Sorted	1.07	1.83	31.60	**1.01**	14.48	28.79	1.02	92.63	3.93	27.46	94.22	
double	ReverseSorted	**1.03**	1.70	19.10	1.06	5.72	7.75	1.14	36.85	7.45	12.07	37.44	
double	Zero	1.09	1.79	26.88	**1.01**	1.15	19.12	1.05	3.02	4.87	21.95	28.47	
double	Exponential	**1.00**	1.11	1.23	1.29	2.34	2.69	4.44	4.42	4.19	1.19	1.65	
double	Zipf	**1.08**	1.23	1.38	1.48	2.79	3.11	5.01	4.70	4.79	1.08	1.39	
double	RootDup	**1.04**	1.23	1.52	1.51	1.23	2.12	1.34	6.01	2.57	1.59	2.39	
double	TwoDup	1.20	1.35	1.38	1.52	2.60	2.85	3.19	3.79	3.18	**1.02**	1.45	
double	EightDup	**1.00**	1.06	1.31	1.33	2.43	2.90	4.64	5.02	4.35	1.16	1.61	
double	AlmostSorted	2.29	**1.01**	2.07	2.76	1.60	1.87	1.30	7.43	2.22	2.26	5.61	
double	Uniform	1.05	1.20	1.20	1.34	2.43	2.54	3.82	3.26	3.63	**1.01**	1.78	
Total		1.18	**1.16**	1.42	1.55	2.13	2.55	3.00	4.79	3.45	1.28	2.00	
Rank		2	1	4	5	7	8	10	12	11	3	6	
uint64	Sorted	1.11	1.83	28.42	**1.02**	13.30	26.46	1.05	84.18	3.33	34.08	116.92	16.69
uint64	ReverseSorted	1.06	1.73	17.76	**1.02**	5.28	7.00	1.04	32.52	7.72	13.81	43.72	7.59
uint64	Zero	1.10	1.68	25.90	1.03	1.11	17.94	**1.01**	2.73	4.42	22.45	28.05	1.47
uint64	Exponential	**1.02**	1.15	1.24	1.38	2.23	2.59	4.42	4.09	4.14	1.11	2.06	1.10
uint64	Zipf	1.28	1.56	1.71	1.85	3.20	3.61	5.76	5.23	5.72	1.27	1.75	**1.01**
uint64	RootDup	**1.06**	1.22	1.52	1.63	1.16	1.95	1.39	5.35	2.73	1.43	2.26	1.35
uint64	TwoDup	1.49	1.67	1.65	1.90	2.99	3.29	3.76	4.19	3.85	1.21	2.26	**1.00**
uint64	EightDup	1.16	1.20	1.47	1.58	2.51	3.00	4.95	5.04	4.74	1.36	2.54	**1.02**
uint64	AlmostSorted	2.37	**1.02**	1.91	3.08	1.60	1.76	1.41	6.81	2.39	2.69	7.30	1.21
uint64	Uniform	1.25	1.41	1.38	1.61	2.61	2.80	4.09	3.43	4.01	**1.00**	2.80	1.12
Total		1.32	1.30	1.54	1.80	2.21	2.64	3.25	4.77	3.79	1.37	2.66	**1.11**
Rank		3	2	5	6	7	8	10	13	11	4	9	1
uint32	Sorted	2.82	4.45	83.97	2.32	35.39	95.00	**2.00**	234.86	8.93	61.73	92.69	42.36
uint32	ReverseSorted	1.47	2.38	26.97	1.51	7.54	12.80	**1.00**	49.47	7.45	14.09	51.86	10.10
uint32	Zero	2.51	4.09	80.92	**1.99**	2.49	54.23	2.11	7.59	12.12	34.89	17.80	4.03
uint32	Exponential	1.29	1.53	1.67	1.62	3.20	3.56	5.92	5.77	5.61	1.13	1.11	**1.05**
uint32	Zipf	1.67	2.10	2.40	2.18	4.76	5.14	8.11	7.72	7.94	**1.09**	1.34	1.22
uint32	RootDup	1.35	1.43	1.85	1.70	1.41	2.42	1.26	6.66	2.76	**1.09**	1.67	1.61
uint32	TwoDup	1.86	2.16	2.22	2.15	4.12	4.40	4.86	5.67	4.89	**1.04**	1.78	1.18
uint32	EightDup	1.28	1.46	1.77	1.59	3.29	3.76	6.00	6.28	5.70	**1.04**	1.91	1.08
uint32	AlmostSorted	2.77	1.22	2.43	3.56	1.89	2.76	**1.09**	8.40	2.23	2.30	7.36	1.30
uint32	Uniform	1.35	1.62	1.56	1.63	3.20	3.25	4.79	4.01	4.53	**1.01**	1.12	1.20
Total		1.59	1.61	1.96	1.98	2.91	3.51	3.68	6.22	4.45	**1.19**	1.84	1.22
Rank		3	4	6	7	8	9	10	13	11	1	5	2
Pair	Sorted	1.08	1.72	19.73	1.02	9.96	19.02	**1.01**	51.62	2.65	20.73		11.81
Pair	ReverseSorted	1.07	1.18	10.15	1.08	3.48	4.63	**1.07**	17.97	7.55	7.46		5.07
Pair	Zero	1.12	1.64	17.10	**1.01**	1.05	16.22	1.12	2.13	3.05	13.49		1.15
Pair	Exponential	**1.04**	1.94	1.18	1.52	1.82	2.07	3.87	3.35	4.04	1.10		1.07
Pair	Zipf	1.39	2.68	1.53	2.06	2.53	2.72	4.83	4.09	5.25	1.25		**1.00**
Pair	RootDup	**1.05**	1.15	1.39	1.66	1.09	1.76	1.65	4.31	3.24	1.23		1.12
Pair	TwoDup	1.48	2.67	1.56	2.00	2.47	2.66	3.35	3.55	3.95	1.21		**1.02**
Pair	EightDup	1.24	2.20	1.40	1.78	2.08	2.47	4.52	4.35	4.85	1.37		**1.00**
Pair	AlmostSorted	3.40	**1.00**	2.62	4.12	2.17	2.50	2.80	7.94	4.55	3.17		1.66
Pair	Uniform	1.29	2.39	1.35	1.76	2.24	2.34	3.75	2.98	4.00	**1.04**		1.08
Total		1.43	1.88	1.53	2.01	2.00	2.34	3.37	4.16	4.22	1.37		**1.12**
Rank		3	5	4	7	6	8	9	10	11	2		1
Quartet	Uniform	1.22	2.00	1.31	1.82	2.00	2.02	3.37	2.39	3.71	**1.02**		
Rank		2	5	3	4	6	7	9	8	10	1		
100B	Uniform	1.51	1.30	1.29	1.88	1.84	1.38	2.39	1.98	3.40	**1.04**		
Rank		5	3	2	7	6	4	9	8	10	1		

Table A.4: Average slowdowns of sequential algorithms for different data types and input distributions on I2x16. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	I1S^{4}o	BlockPDQ	BlockQ	IS^{4}o	DualPivot	std::sort	Timsort	QMSort	WikiSort	SkaSort	IppRadix	IPS^{2}Ra
double	Sorted	**1.03**	1.64	29.22	1.11	16.69	22.54	1.24	70.63	2.51	27.84	66.47	
double	ReverseSorted	1.01	1.56	11.86	**1.00**	4.75	4.98	1.04	21.65	4.68	8.75	19.44	
double	Zero	**1.01**	1.64	21.64	1.17	1.11	16.84	1.19	2.62	3.03	18.42	32.61	
double	Exponential	**1.03**	1.10	1.20	1.24	2.62	2.92	4.78	4.32	4.79	1.36	1.32	
double	Zipf	**1.02**	1.11	1.27	1.26	2.95	3.12	4.92	4.28	5.06	1.08	1.21	
double	RootDup	1.14	1.64	1.96	1.86	1.79	2.69	**1.14**	7.04	3.71	2.32	3.24	
double	TwoDup	1.18	1.26	1.31	1.37	2.80	2.95	3.25	3.48	3.54	1.09	**1.09**	
double	EightDup	**1.02**	1.06	1.29	1.25	2.72	3.13	4.86	4.93	4.82	1.38	1.52	
double	AlmostSorted	3.03	1.22	3.02	4.47	2.43	2.33	**1.00**	9.01	3.56	3.52	4.81	
double	Uniform	1.09	1.15	1.15	1.25	2.64	2.68	3.96	3.07	3.95	1.10	**1.04**	
Total		1.25	**1.21**	1.51	1.61	2.54	2.82	2.89	4.83	4.16	1.53	1.71	
Rank		2	1	3	5	7	8	9	11	10	4	6	
uint64	Sorted	1.33	1.92	29.77	**1.00**	16.75	24.57	1.03	59.38	2.81	37.63	79.94	16.30
uint64	ReverseSorted	1.01	1.54	10.29	**1.01**	4.11	4.67	1.01	15.95	4.66	9.84	20.49	5.05
uint64	Zero	1.37	2.06	24.96	1.10	1.18	16.98	**1.01**	2.57	3.81	22.53	39.42	1.39
uint64	Exponential	**1.04**	1.18	1.33	1.34	2.51	2.89	5.09	3.73	5.07	1.26	1.62	1.04
uint64	Zipf	1.73	2.06	2.39	2.33	4.80	5.32	9.03	6.36	9.19	2.15	2.51	**1.00**
uint64	RootDup	1.59	2.32	2.90	2.77	2.42	3.61	1.80	7.98	5.65	3.15	4.26	**1.00**
uint64	TwoDup	2.04	2.47	2.55	2.62	4.94	5.36	6.36	5.47	6.95	1.99	3.41	**1.00**
uint64	EightDup	1.37	1.52	1.91	1.77	3.43	4.06	6.89	5.57	6.84	2.13	3.39	**1.00**
uint64	AlmostSorted	2.93	1.19	2.94	4.64	2.39	2.41	**1.03**	7.08	3.83	4.03	7.29	1.66
uint64	Uniform	1.43	1.73	1.73	1.82	3.59	3.78	5.86	3.75	5.84	1.34	2.02	**1.00**
Total		1.65	1.71	2.17	2.30	3.30	3.78	4.17	5.51	6.00	2.12	3.13	**1.08**
Rank		2	3	5	6	8	9	10	11	12	4	7	1
uint32	Sorted	2.48	4.51	67.12	2.80	37.74	54.21	**1.93**	139.72	6.14	64.10	28.28	35.51
uint32	ReverseSorted	1.42	1.88	13.03	1.55	5.38	6.34	**1.06**	19.68	4.27	8.92	7.12	5.83
uint32	Zero	2.19	3.95	60.33	2.38	2.21	41.48	**1.96**	6.01	7.85	27.01	17.69	2.97
uint32	Exponential	1.60	1.84	2.13	1.97	4.12	4.76	7.67	5.90	7.78	1.51	**1.00**	1.09
uint32	Zipf	2.04	2.51	3.01	2.67	6.12	6.86	10.77	7.84	11.07	1.66	1.25	**1.06**
uint32	RootDup	1.67	2.48	3.21	2.83	2.56	4.07	1.50	9.10	5.14	2.41	2.08	**1.00**
uint32	TwoDup	2.65	3.13	3.32	3.13	6.46	7.09	7.72	6.89	8.31	1.63	**1.09**	1.10
uint32	EightDup	1.53	1.81	2.31	1.93	4.31	5.07	7.80	6.48	7.89	1.55	1.30	**1.00**
uint32	AlmostSorted	5.23	2.10	4.95	8.25	3.93	3.96	**1.00**	12.63	5.42	5.34	5.24	2.70
uint32	Uniform	2.21	2.53	2.60	2.57	5.38	5.77	8.32	5.38	8.39	1.75	**1.02**	1.29
Total		2.21	2.31	2.97	2.94	4.51	5.24	4.84	7.49	7.49	2.03	1.53	**1.24**
Rank		4	5	7	6	8	10	9	11	12	3	2	1
Pair	Sorted	1.03	1.65	20.71	1.04	12.67	18.52	**1.03**	35.04	2.41	23.54		12.43
Pair	ReverseSorted	1.08	1.18	6.89	1.07	3.77	3.90	**1.05**	10.76	5.49	6.82		4.58
Pair	Zero	**1.02**	1.65	13.38	1.04	1.06	12.81	1.03	2.03	2.78	12.80		1.24
Pair	Exponential	1.06	2.00	1.09	1.20	1.96	2.18	4.16	2.72	4.40	1.13		**1.04**
Pair	Zipf	1.53	3.18	1.63	1.74	3.06	3.31	5.96	3.77	6.50	1.58		**1.00**
Pair	RootDup	1.62	1.90	2.06	2.17	1.86	2.85	1.96	5.37	5.22	2.22		**1.00**
Pair	TwoDup	1.67	3.23	1.69	1.86	3.10	3.34	4.26	3.32	4.99	1.55		**1.00**
Pair	EightDup	1.24	2.37	1.34	1.44	2.31	2.70	4.96	3.64	5.16	1.72		**1.00**
Pair	AlmostSorted	3.51	**1.00**	3.26	4.62	2.70	2.91	1.96	6.48	5.18	4.17		2.03
Pair	Uniform	1.41	2.82	1.40	1.58	2.71	2.81	4.62	2.75	4.90	1.24		**1.00**
Total		1.60	2.21	1.68	1.90	2.48	2.85	3.69	3.83	5.16	1.77		**1.11**
Rank		2	6	3	5	7	8	9	10	11	4		1
Quartet	Uniform	1.13	1.82	1.24	1.28	1.96	1.84	3.04	1.95	3.67	**1.01**		
Rank		2	5	3	4	8	6	9	7	10	1		
100B	Uniform	1.53	1.38	1.40	1.68	2.10	1.42	2.25	1.79	3.50	**1.01**		
Rank		5	2	3	6	8	4	9	7	10	1		

Table A.5: Average slowdowns of sequential algorithms for different data types and input distributions on A1x64. The slowdowns average over input sizes with at least 2^{18} bytes.

Type	Distribution	1S⁴o	S⁴oS
double	Sorted	**1.00**	35.77
double	ReverseSorted	**1.00**	16.15
double	Zero	**1.00**	17.03
double	Exponential	**1.00**	1.41
double	Zipf	**1.00**	1.34
double	RootDup	**1.02**	1.39
double	TwoDup	**1.00**	1.23
double	EightDup	**1.00**	1.52
double	AlmostSorted	**1.02**	1.13
double	Uniform	**1.00**	1.23
Total		**1.01**	1.32
Rank		1	2
uint64	Sorted	**1.00**	37.39
uint64	ReverseSorted	**1.00**	15.50
uint64	Zero	**1.00**	16.41
uint64	Exponential	**1.00**	1.41
uint64	Zipf	**1.00**	1.31
uint64	RootDup	**1.02**	1.33
uint64	TwoDup	**1.00**	1.24
uint64	EightDup	**1.00**	1.48
uint64	AlmostSorted	**1.04**	1.11
uint64	Uniform	**1.00**	1.24
Total		**1.01**	1.30
Rank		1	2
uint32	Sorted	**1.00**	39.40
uint32	ReverseSorted	**1.01**	15.16
uint32	Zero	**1.00**	20.16
uint32	Exponential	**1.00**	1.49
uint32	Zipf	**1.00**	1.38
uint32	RootDup	**1.01**	1.45
uint32	TwoDup	**1.00**	1.25
uint32	EightDup	**1.00**	1.56
uint32	AlmostSorted	**1.04**	1.14
uint32	Uniform	**1.00**	1.25
Total		**1.01**	1.35
Rank		1	2
Pair	Sorted	**1.00**	24.42
Pair	ReverseSorted	**1.00**	10.47
Pair	Zero	**1.00**	12.16
Pair	Exponential	**1.00**	1.32
Pair	Zipf	**1.01**	1.20
Pair	RootDup	**1.02**	1.24
Pair	TwoDup	**1.00**	1.19
Pair	EightDup	**1.00**	1.37
Pair	AlmostSorted	**1.04**	1.07
Pair	Uniform	**1.01**	1.16
Total		**1.01**	1.22
Rank		1	2
Quartet	Uniform	**1.01**	1.12
Rank		1	2
100B	Uniform	1.09	**1.04**
Rank		2	1

Table A.6: Average slowdowns of 1S⁴o and S⁴oS for different data types and input distributions. The slowdowns average over the machines and input sizes with at least 2^{18} bytes.

		2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}	2^{32}	2^{34}
I4x20	IPS^{4}o	272.84	51.73	8.20	3.00	**1.06**	**0.66**	**0.50**	**0.49**	**0.49**
	PBBS	**10.15**	**4.62**	**2.86**	**1.70**	1.39	1.10	1.13	1.12	1.13
	MCSTLbq	229	352.11	63.94	15.79	4.58	2.25	1.91	1.86	1.82
	IPS^{2}Ra	237.91	37.12	10.54	2.56	0.93	**0.54**	**0.45**	**0.45**	**0.40**
	PBBR	**13.96**	**4.03**	**2.08**	**0.96**	**0.92**	0.62	0.89	0.70	0.79
	RADULS2	721.12	146.75	37.14	9.29	3.09	1.87	1.25	1.30	1.09
	RegionSort	314.64	65.06	15.72	4.22	2.25	1.55	1.45	1.22	1.23
A1x16	IPS^{4}o	14.39	**1.11**	**0.62**	**0.47**	**0.41**	**0.47**	**0.44**	**0.42**	
	PBBS	**1.99**	1.20	0.79	0.83	0.72	0.69	0.66	0.65	
	MCSTLbq	57.04	13.62	2.34	1.08	1.11	1.23	1.34	1.48	
	IPS^{2}Ra	2.41	**0.48**	**0.38**	**0.37**	**0.41**	**0.48**	**0.46**	**0.42**	
	PBBR	**1.16**	0.68	0.49	0.56	0.56	0.63	0.63	0.59	
	RADULS2	9.13	2.87	1.20	0.63	0.48	0.50	0.51	0.44	
	RegionSort	9.88	2.56	0.91	0.49	0.55	0.51	0.49	0.45	
I2x16	IPS^{4}o	21.33	2.89	**1.12**	**0.74**	**0.58**	**0.53**	**0.50**	**0.48**	**0.48**
	PBBS	**2.93**	**1.74**	1.29	1.19	1.13	1.10	1.10	1.07	1.05
	MCSTLbq	149.59	36.86	6.07	2.09	1.41	1.27	1.31	1.26	1.33
	IPS^{2}Ra	10.25	2.63	1.02	0.67	**0.48**	**0.45**	0.45	0.45	0.40
	PBBR	**2.63**	**1.22**	**0.75**	**0.66**	0.72	0.68	0.69	0.67	0.64
	RADULS2	61.36	14.46	4.15	1.29	0.62	0.47	**0.40**	**0.37**	**0.39**
	RegionSort	57.67	13.92	3.58	1.35	0.72	0.49	0.59	0.42	0.46
A1x64	IPS^{4}o	26.17	11.88	3.34	**1.26**	**0.63**	**0.50**	**0.45**	**0.42**	**0.42**
	PBBS	**4.72**	**2.48**	**1.74**	**1.26**	0.95	0.87	0.83	0.83	0.83
	MCSTLbq	408.96	95.10	27.22	5.30	1.79	1.29	1.18	1.19	1.24
	IPS^{2}Ra	45.22	11.72	3.20	1.25	**0.52**	**0.45**	0.44	0.41	**0.39**
	PBBR	5.80	**2.03**	**1.38**	**0.87**	0.64	0.58	0.55	0.52	0.50
	RADULS2	154.04	36.82	9.82	2.79	1.00	0.56	**0.44**	0.40	0.40
	RegionSort	103.12	26.75	6.62	2.09	0.96	0.66	0.55	**0.39**	0.46

Table A.7: Running times in nanoseconds of parallel algorithms sorting uint64 values with input distribution Uniform executed on different machines. The running times are scaled by $t/8n\log_2 n$. The fastest comparison-based and non-comparison-based running time is highlighted.

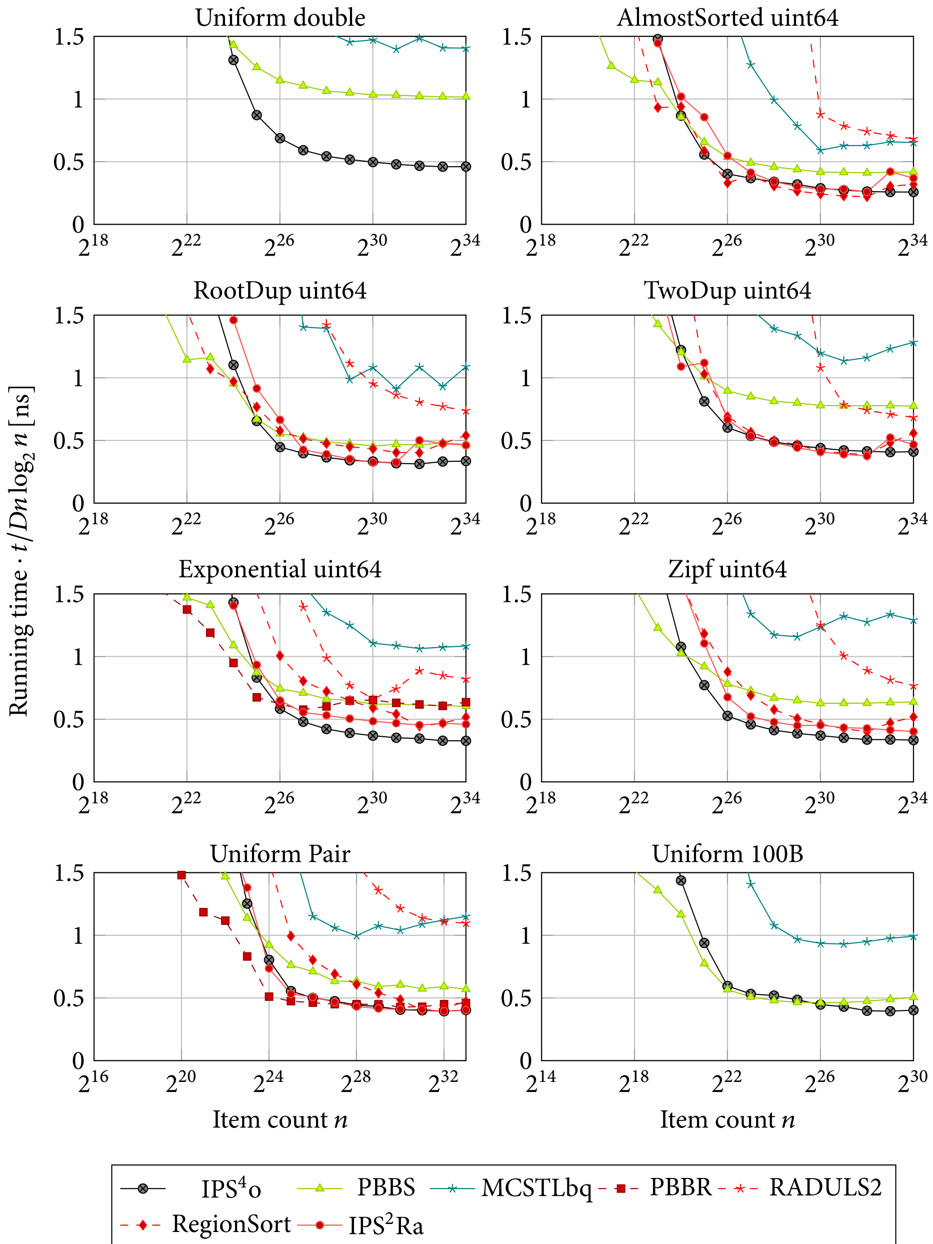


Figure A.2: Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x64. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

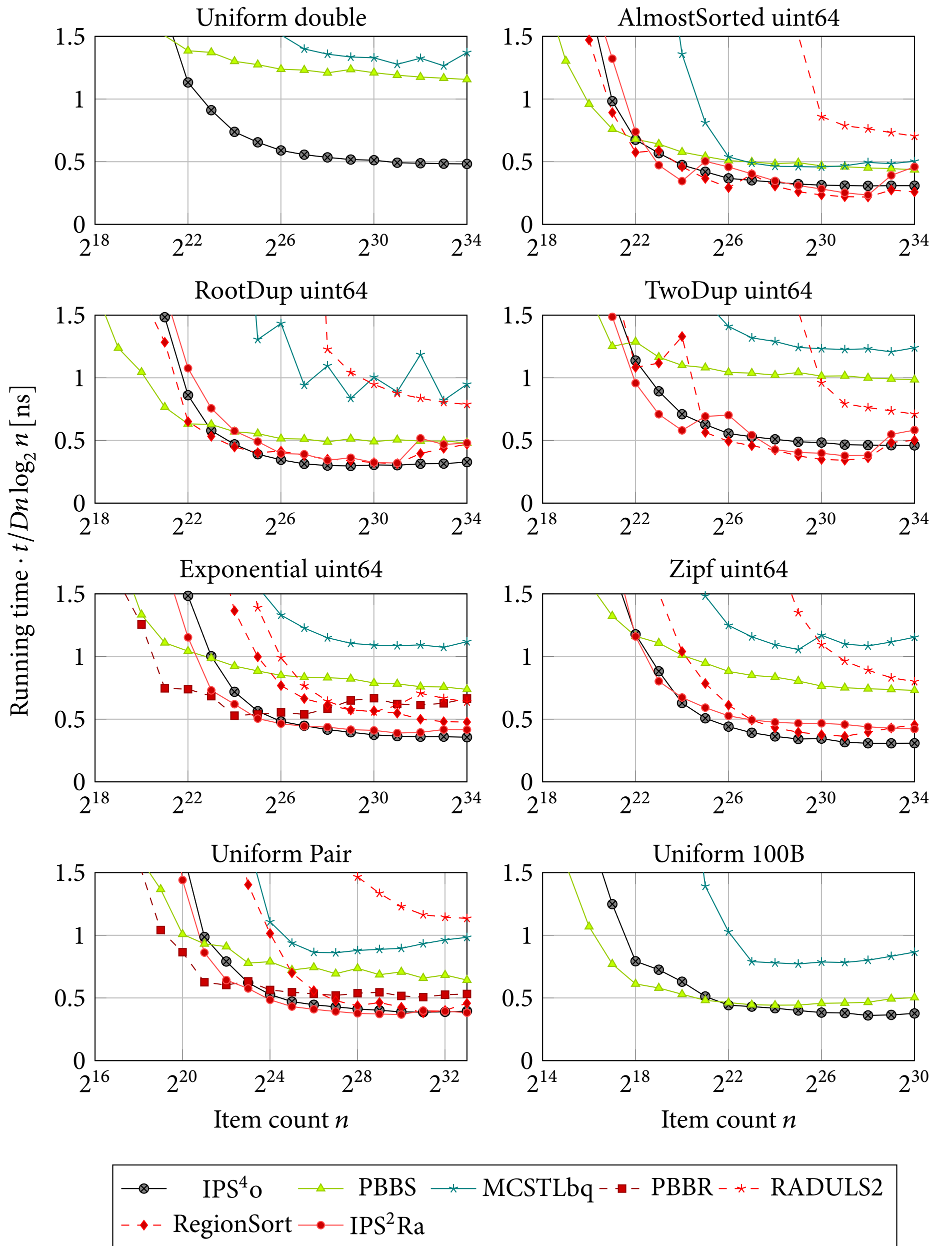


Figure A.3: Running times of parallel algorithms on different input distributions and data types of size D executed on machine I2x16. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

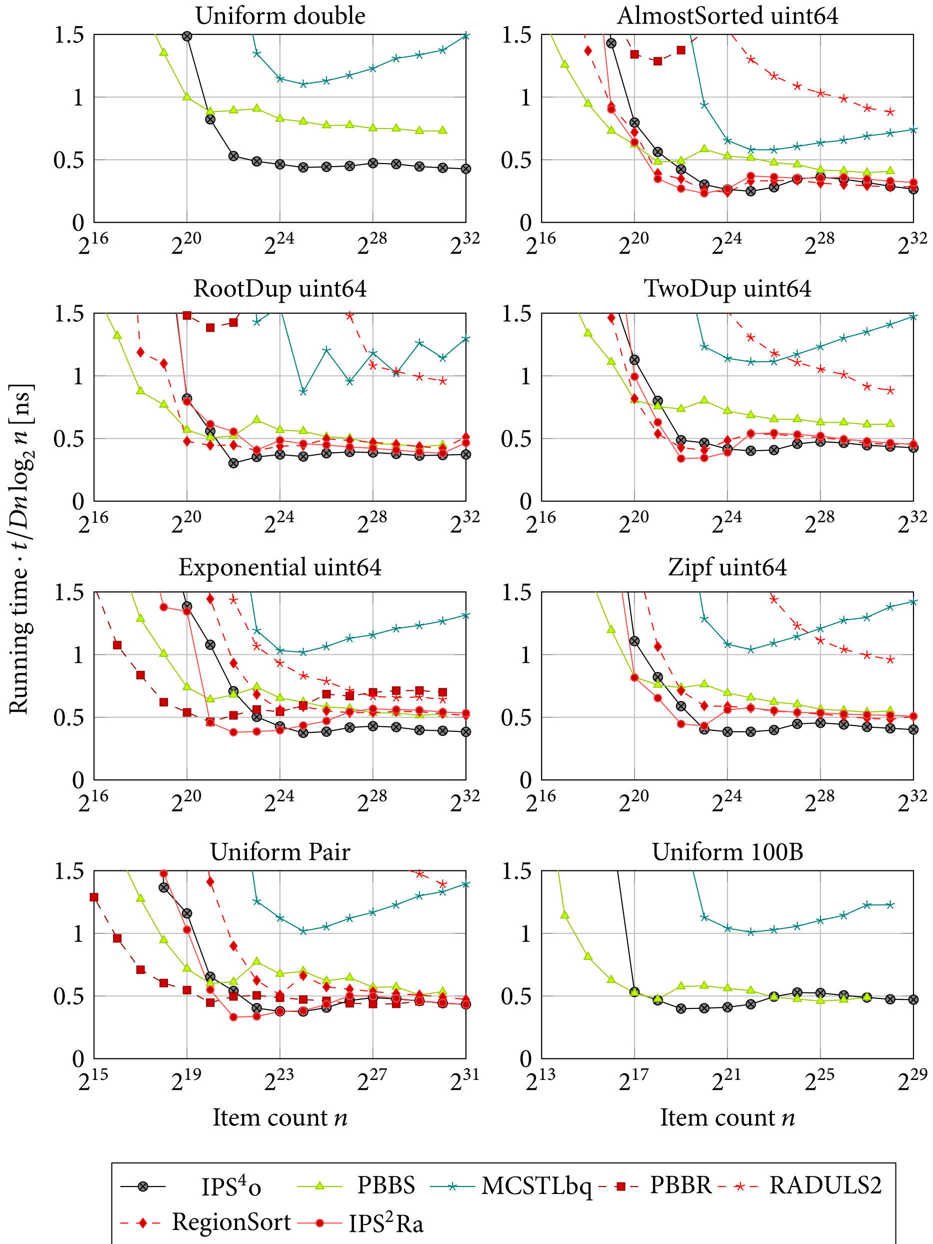


Figure A.4: Running times of parallel algorithms on different input distributions and data types of size D executed on machine A1x16. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

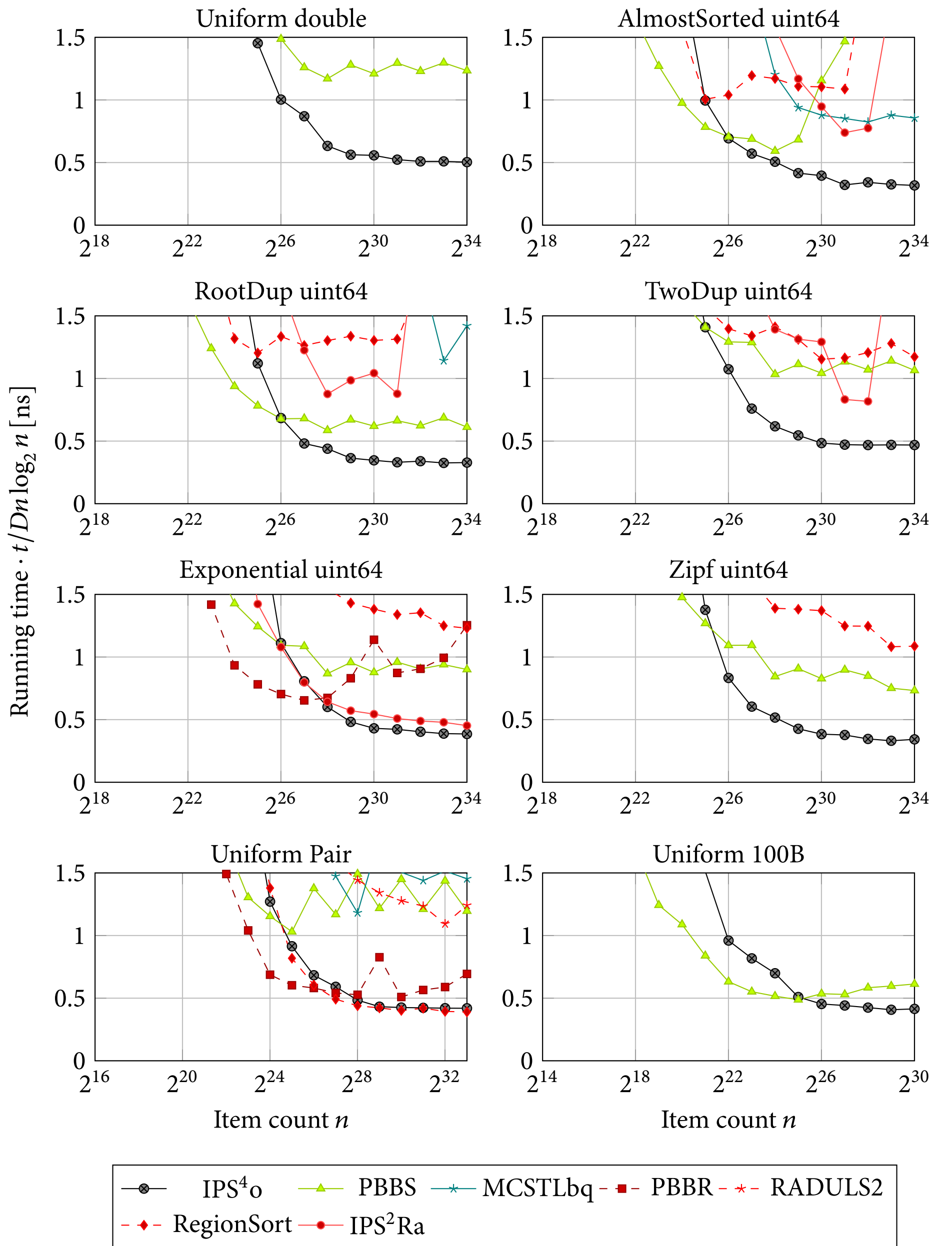


Figure A.5: Running times of parallel algorithms on different input distributions and data types of size D executed on machine I4x20. The radix sorters PBBR, RADULS2, RegionSort, and IPS^{2}Ra does not support the data types double and 100B.

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

Table A.8: Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine A1x64. The slowdowns average input sizes with at least $t \cdot 2^{21}$ bytes.

Type	Distribution	IPS⁴o	PBBS	PS⁴o	MCSTLmwm	MCSTLbq	TBB	RegionSort	PBBR	RADULS2	ASPaS	IPS²Ra
double	Sorted	2.47	18.27	2.81	22.46	19.43	**1.02**				60.33	
double	ReverseSorted	**1.05**	1.20	1.78	1.40	8.79	2.24				4.29	
double	Zero	2.17	17.91	2.30	26.99	292.91	**1.03**				60.63	
double	Exponential	**1.00**	2.06	1.92	2.76	2.79	11.65				4.17	
double	Zipf	**1.00**	2.36	2.08	3.16	3.33	13.48				4.65	
double	RootDup	**1.00**	1.63	2.29	2.62	3.53	5.71				5.71	
double	TwoDup	**1.00**	2.15	1.79	2.60	2.64	5.37				3.52	
double	EightDup	**1.00**	2.10	1.98	2.67	2.93	11.29				4.29	
double	AlmostSorted	**1.00**	1.47	2.07	4.21	1.57	2.72				5.05	
double	Uniform	**1.00**	2.23	1.78	2.70	2.65	9.35				3.43	
Total		**1.00**	1.97	1.98	2.92	2.71	7.54				4.34	
Rank		1	2	3	5	4	7				6	
uint64	Sorted	2.32	16.98	2.91	21.77	18.67	**1.01**	8.19	93.89	50.76		12.20
uint64	ReverseSorted	1.34	1.43	2.28	1.75	11.17	2.83	**1.00**	8.30	4.24		1.47
uint64	Zero	1.62	18.42	2.30	26.88	291.98	**1.07**	1.09	85.76	52.94		1.09
uint64	Exponential	**1.04**	1.98	2.06	2.79	3.01	11.02	1.58	1.45	1.96		1.08
uint64	Zipf	**1.00**	2.17	2.09	2.99	3.24	12.17	1.32	18.30	5.64		1.30
uint64	RootDup	**1.00**	1.55	2.27	2.53	3.50	5.64	1.17	9.66	6.40		1.26
uint64	TwoDup	1.19	2.36	2.10	2.91	3.12	6.18	**1.09**	11.02	3.39		1.15
uint64	EightDup	**1.05**	2.02	2.11	2.66	2.98	10.68	1.14	14.02	4.45		1.15
uint64	AlmostSorted	1.23	1.73	2.62	5.24	1.99	3.42	**1.04**	9.95	5.18		1.26
uint64	Uniform	1.21	2.50	2.15	3.09	3.11	10.31	1.36	1.54	1.15		**1.06**
Total		**1.10**	2.02	2.19	3.08	2.95	7.80	1.23	6.43	3.49		1.18
Rank		1	4	5	7	6	10	3	9	8		2
uint32	Sorted	3.33	14.36	3.55	14.59	18.15	**1.96**	6.28	8.67			6.47
uint32	ReverseSorted	1.94	2.12	2.80	2.07	12.90	5.32	**1.02**	1.28			1.14
uint32	Zero	1.97	19.35	1.99	32.52	473.11	**1.06**	1.08	105.42			1.09
uint32	Exponential	1.46	3.38	2.43	4.20	4.33	19.28	1.32	1.93			**1.00**
uint32	Zipf	1.10	2.99	2.03	3.73	3.90	17.67	**1.05**	5.83			1.10
uint32	RootDup	1.21	1.94	2.41	2.96	3.48	6.35	**1.01**	7.00			1.36
uint32	TwoDup	1.64	3.79	2.48	4.08	4.38	9.68	**1.04**	2.12			1.06
uint32	EightDup	1.38	3.48	2.43	3.95	4.29	18.54	1.10	1.84			**1.09**
uint32	AlmostSorted	1.72	2.25	2.76	6.11	2.87	4.77	1.17	1.34			**1.01**
uint32	Uniform	1.53	3.63	2.23	3.95	4.09	14.73	1.09	1.73			**1.08**
Total		1.42	2.98	2.39	4.06	3.87	11.54	1.11	2.43			**1.09**
Rank		3	6	4	8	7	9	2	5			1
Pair	Sorted	2.17	14.22	2.93	19.64	17.52	**1.03**	7.26	34.12	95.13		11.46
Pair	ReverseSorted	1.14	1.31	2.05	1.74	9.42	2.76	**1.03**	3.11	8.53		1.53
Pair	Zero	1.95	20.27	2.40	25.29	197.76	**1.03**	1.06	19.66	97.93		1.06
Pair	Exponential	**1.06**	1.49	2.02	2.36	2.45	6.71	1.52	1.07	8.42		1.20
Pair	Zipf	**1.00**	1.58	1.93	2.38	2.55	6.87	1.35	7.99	9.45		1.35
Pair	RootDup	**1.01**	1.34	2.05	2.31	3.06	4.89	1.16	4.30	10.97		1.13
Pair	TwoDup	**1.05**	1.65	1.99	2.30	2.48	4.08	1.15	4.92	6.91		1.17
Pair	EightDup	**1.02**	1.50	2.04	2.22	2.48	6.44	1.16	7.57	8.34		1.21
Pair	AlmostSorted	1.06	1.67	2.60	4.49	2.12	3.25	**1.03**	3.97	11.00		1.33
Pair	Uniform	1.06	1.73	2.01	2.41	2.39	6.19	1.36	1.32	4.74		**1.01**
Total		**1.04**	1.56	2.08	2.56	2.49	5.31	1.24	3.55	8.26		1.20
Rank		1	4	5	7	6	9	3	8	10		2
Quartet	Uniform	**1.00**	1.26	2.01	2.20	2.28	4.50					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.01**	1.16	1.94	1.93	2.16	3.33					
Rank		1	2	4	3	5	6					

Table A.9: Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine I2x16. The slowdowns average input sizes with at least $t \cdot 2^{21}$ bytes.

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

Table A.10: Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine A1x16. The slowdowns average input sizes with at least $t \cdot 2^{21}$ bytes.

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

Table A.11: Average slowdowns of parallel algorithms for different data types and input distributions obtained on machine I4x20. The slowdowns average input sizes with at least $t \cdot 2^{21}$ bytes.

Appendix B

B On the Performance of MPI Libraries

In high-performance computing, algorithms usually use an MPI library for communication. The target of MPI libraries is to simplify the development of algorithms by providing efficient and scalable implementations of frequently used operations for communication. However, it turned out that point-to-point message exchanges, collective operations, and operations to create new communicators are much less efficient in many situations than one would expect: Depending on the MPI library, the startup overhead of data exchanges with alternating communication partners is tremendously large, collective operations are slow, and the time to create MPI communicators makes it impossible to implement algorithms with polylogarithmic latency. In Appendix B.1, we study the problem of performing fast data exchanges with point-to-point messages on different supercomputers with different MPI implementations. On one supercomputer, we did not see any running time fluctuations. On two supercomputers, data exchanges with alternating communication partners are not efficient at all. And on one supercomputer, the time for data exchanges fluctuates only slightly. In Appendix B.2, we study the performance of MPI collective operations and MPI communicators. We also present RBC, a communication library based on MPI that creates range-based communicators in constant time without communication. These RBC communicators provide very scalable and efficient implementations of collective operations. For an illustration, we apply RBC to three sorting algorithms from Harsh et al. [HKS19] and Sundar et al. [SMB13]. RBC improves the performance of these algorithms by multiple orders of magnitude.

B.1 Startup Overheads of Communication Patterns

In this section, we discuss the reliability of four supercomputers and their MPI libraries in terms of startup overhead of message exchanges. On each supercomputer, we have executed a simple communication pattern with point-to-point messages in different situations to challenge the network and the MPI implementation. This is an overview of what we have found out: The result differed widely between the supercomputers: On *JUQUEEN*, the communication cost was very reliable—in most cases we had cost $\alpha + n\beta$ for about the same constants α and β. On *SuperMUC-NG*, *SuperMUC Phase 1*, and *SuperMUC Phase 2*, the first communication between two PEs is tremendously large. The supercomputers SuperMUC Phase 1 and SuperMUC Phase 2 are only fast when we use fixed communication partners. OnSuperMUC-NG, the time of data exchanges between fixed communication partners is slightly smaller than the time of exchanges between alternating partners. Since the results obtained on SuperMUC-NG and on JUQUEEN are much more reliable, we assume that the bad performance of SuperMUC

Phase 1 and 2 cannot be generalized. Thus, we decided to use SuperMUC-NG and JUQUEEN for our sorting benchmarks.

The experimental setting of the startup overhead test was as follows: Our benchmark measured the running time of a single PE performing ping-pong message transfers with 256 different PEs. We present results of benchmark runs obtained after two different warmup types. The *local warmup* performs exactly the same ping-pong massage transfers with the same PEs as the actual benchmark. The *global warmup* executes a point-to-point message transfer between all PEs beforehand. These two benchmarks provide minimal working examples to understand the impact of startup overheads of two different data exchanges. The benchmark after a local warmup represents data exchanges with fixed communication partners executed over and over again, e.g., hypercube data exchanges. The benchmark after a global warmup represents a situation in which we execute data exchanges with alternating communication partners over and over again. We also present results obtained after a *cold start*, i.e., the running time of the benchmark directly executed after system startup.

JUQUEEN. The communication cost on JUQUEEN matched very well with the single-ported message passing model. The communication bandwidth as well as the startup overhead for inter-node communication was almost the same between any pair of nodes. As expected, the intra-node communication was somewhat faster. Additionally, the time for local computations was very predictable—probably as the nodes ran with a constant frequency and as a 17-th core processed OS-related tasks. Overall, the fluctuation in running time of our sorting algorithms was very small on JUQUEEN and only a few runs were necessary to obtain reliable results. On JUQUEEN, we used the IBM MPI library `mpich2` version 1.5.

SuperMUC Phase 1 and 2. We present results obtained on SuperMUC Phase 2 [Lei15], which is the predecessor of SuperMUC-NG. We do not present results obtained on SuperMUC Phase 1 as the results were very similar. For the measurements we used the IBM MPI library version 1.4 and the Intel MPI library 2018. We now discuss the startup overheads on SuperMUC Phase 2 in detail.

Figure B.1 shows the results obtained with IBM MPI. By default, IBM MPI 1.4 uses the bulk transfer protocol for messages with at least 2^{16} bytes. As this protocol has a big impact on the performance of IBM MPI, we ran the benchmark once with bulk transfer (default) and once without. In both cases, we got reliable results after a local warmup (fixed communication partners) but not after global warmups (alternating communication partners). We first describe the results of the default configuration. When we executed the benchmark after a cold start, the startup overhead of messages that use bulk transfer internally was about a factor of 37 larger than the overhead of shorter messages. Even a global warmup did not prohibit these large startup overheads for long messages (factor of 18.05). Even worse, the startup overhead of short messages increased by more than one order of magnitude when we used a global warmup instead. Only a local warmup (fixed communication partners) avoids these large running times. When we disabled the bulk transfer protocol, the overhead for long messages disappeared. However, the startup overhead after a global warmup was still one order of magnitude larger than the overhead after a local warmup.

The results obtained with Intel MPI 2018 were even worse (see Figure B.2). Similar to IBM MPI, Intel MPI supported data exchanges with alternating communication partners (global

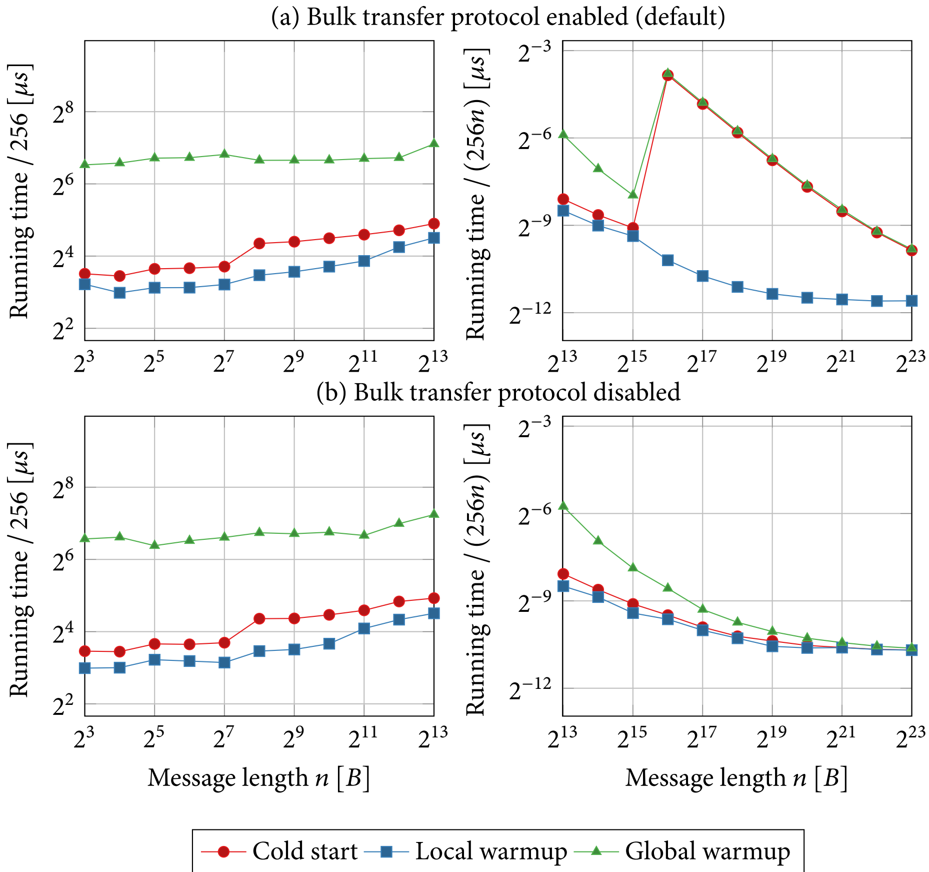


Figure B.1: Ping-pong benchmark with 256 partners on SuperMUC Phase 2 with IBM MPI.

warmup) very poorly. The startup latency after a global warmup was 32.65 times larger than the latency after a local warmup for long messages. Even worse, after a cold start, exchanges of short messages took almost two orders of magnitude longer than after a local warmup.

The results obtained on SuperMUC Phase 2 were obtained on a job instance with 7 168 PEs. For small numbers of t, we were able to avoid these fluctuations by disabling so-called "dynamic connections". Unfortunately, this did not help for large t, probably since the total number of open connections is limited.

SuperMUC-NG. Figure B.3 depicts the results obtained on SuperMUC-NG with the Intel MPI library 2020. Communicating the longest messages took about the same time for all benchmarks. We conclude that the choice of the communication partners is not a crucial factor when PEs send a large bulk of data. The results are different for short messages: After a global

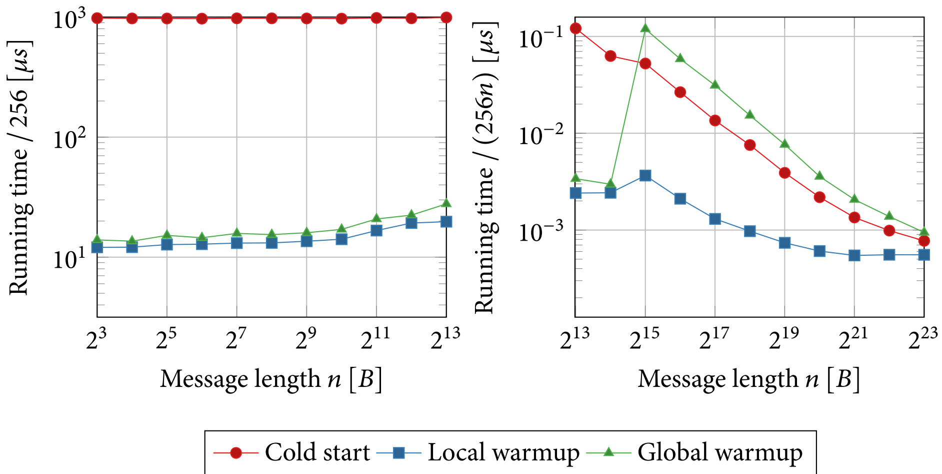


Figure B.2: Ping-pong benchmark with 256 partners on SuperMUC Phase 2 with Intel MPI.

warmup, the startup overhead was "only" twice as large as the overhead after a local warmup. However, the startup overhead after a cold start was more than a magnitude larger than the overhead after a local warmup. Thus, the first execution of a data exchange will have increased startup overheads. If subsequent data exchanges involve new communicating partners, the startup overheads will also be large for these connections. That may be very problematic if the data exchanges of an algorithm use few but always different connections.

B.2 Faster Sorting with MPI—Communicators and Collectives

The size of supercomputers rapidly increased to petascale machines with millions of cores. The de facto standard for communication in High Performance Computing (HPC) is the Message Passing Interface (MPI). Many applications need a flexible management of PE groups, e.g., to adjust the scope of parallelism for load balancing, to achieve parallelism on multiple levels, to divide tasks into fine-grained subproblems, or to recursively sort data [Din+11; Bal+09; SMB13; HKS19]. MPI uses the concept of communicators to enable multiple levels of parallelism by connecting groups of PEs. MPI provides (collective) communication operations between PEs of a communicator to ensure scalability, portability, and comfortable programming with a high-level interface [HLR07; Bal+95b]. The group context of a communicator guarantees that collective communication and point-to-point communication within one communicator as well as over different communicators does not interfere. In this work, we do not need a separation of communication between communicators—in fact, all MPI sorting algorithms we have found only communicate between PEs of one group at the same time.

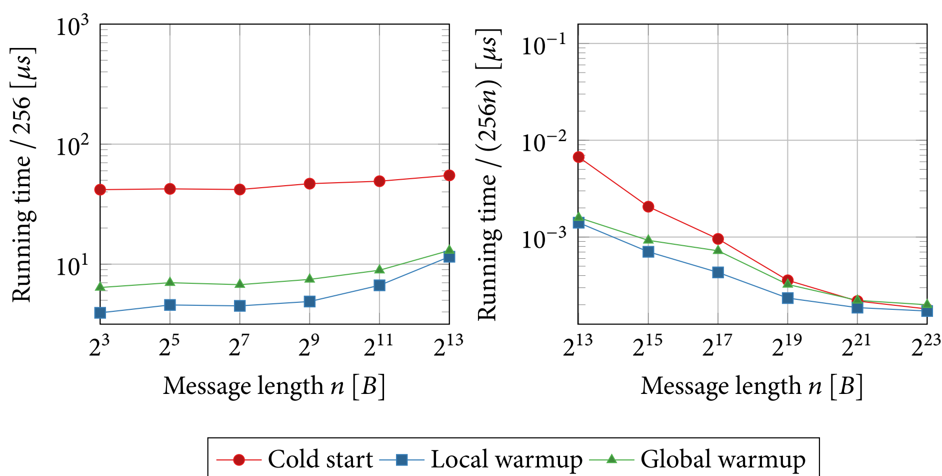


Figure B.3: Ping-pong benchmark with 256 partners on SuperMUC-NG with Intel MPI.

We did not find an MPI library that provides fast communicator creation routines in sublinear time. Thus, collective communicator creation in MPI becomes a bottleneck for sublinear algorithms. E.g., the most recent open-source implementations Open MPI 3.1 and MPICH 3.4 create an array of PE IDs when the user creates a communicator. In this case, the construction time of a communicator is linear to the number of group members. Mohamad Chaarawi and Edgar Gabriel [CG08] integrated sparse data storages into Open MPI. Their implementation reduces the footprint of an existing communicator. But the PE group is still stored explicitly during the communicator construction. Unfortunately, MPI does not provide a method to invoke collective operations on a subset of PEs without creating a new communicator. Before performing the collective operation, the user must create a communicator of the subset of PEs with a blocking communicator creation routine.

Besides inefficient communicator creation routines, it turned out that some supercomputers used in this work provide poorly implemented collective operations. However, when we want to use the full functionality of the message passing interface to implement efficient algorithms, e.g., sub-communicators or collective operations, we need fast MPI libraries. Additionally, scalable and efficient MPI libraries are crucial for a fair comparison of algorithms. Otherwise, poorly implemented MPI operations would penalize algorithms that use these operations more frequently. Thus, we decided to implement a library that provides the functionality required for scalable and efficient sorting algorithms.

We present the lightweight library *RangeBasedComm* (RBC) based on MPI. RBC creates new communicators, containing a consecutive PE range of a parent communicator, in constant time without communication. Our range-based communicators provide (non)blocking collective operations and (non)blocking point-to-point communication. As RBC can not access the context ID of a message, the library does not fully support the nonblocking model of the MPI-

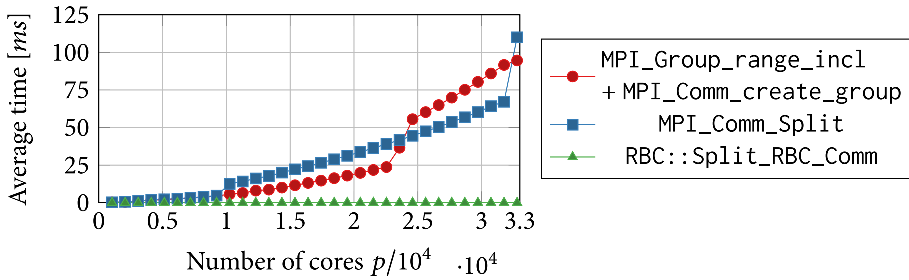


Figure B.4: Running time of communicator splitting into two communicators of equal size on SuperMUC-NG with different number of cores.

standard. Even though we restricted the semantics of communication, the library is applicable to all sorting algorithms used in this work. For example, we replaced the MPI primitives used by our competitors RQuick, HykSort, and HSS with RBC by substituting the prefix "`MPI_`" with the prefix "`RBC::`". It was also easy to replace the routines that create new MPI communicators because all sorting algorithms that we have found create communicators of consecutive PEs. Our RBC communicators only need the index of the first and the last PE of the new PE range. Our experiments show that our library reduces the time to create a new communicator by multiple orders of magnitude whereas the performance of collective operations increases in some cases by more than one order of magnitude. RBC improves the running times of RQuick, HykSort, and HSS proposed by Sundar et al. [SMB13] and Harsh et al. [HKS19] by multiple orders of magnitude.

The RBC library has been proposed with focus on perfectly balanced quicksort [AWS18] in cooperation with our bachelor student Armin Wiebigke. The RBC library is available at `https://github.com/MichaelAxtmann/RBC/`. For a detailed description of RBC as well as perfectly balanced quicksort, we refer to the conference publication [AWS18] and to the bachelor's thesis of Armin Wiebigke [Wie17a]. Since we have published the first version of RBC, we extended the collective operations of RBC with implementations of the two-tree algorithms for full bandwidth broadcast, (all-)reduction, and (exclusive) prefix sum proposed by Sanders et al. [SST09]. We consider the implementation of these algorithms as a separate contribution of this work since the source code of the two-tree algorithms has never been published as open source.

Experimental Results. We still owe you an experimental evaluation of the RBC library. We compare the performance of RBC to the Intel MPI library 2020 on SuperMUC-NG. First, we present benchmark results of communicator creation routines and collective operations. In a second step, we evaluate the performance of RBC on sorting algorithms as a whole.

MPI offers two methods to create sub-communicators of consecutive PE ranges. The first method requires each PE to provide the index of its new group and its rank within that group (`MPI_Comm_Split`). For the second method, the PEs of the new communicator have to be provided by every single PE. Fortunately, MPI provides an efficient interface to enumerate consecutive PE ranges—`MPI_Group_range_incl` only requires the index of the first and last

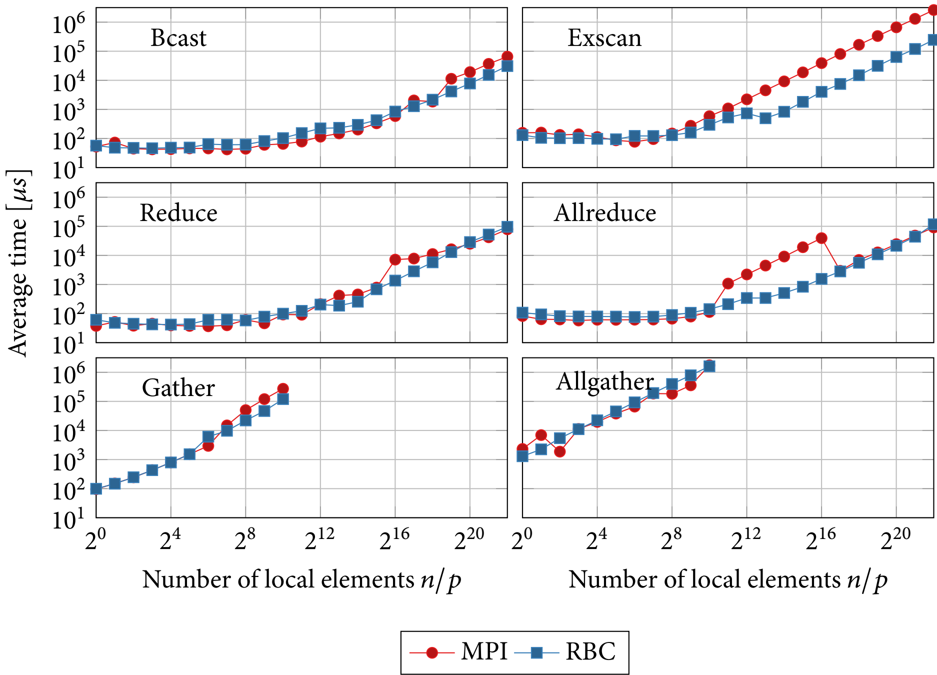


Figure B.5: Running time of collective operations with n output elements on SuperMUC-NG with 32 784 cores.

PE of the range. Figure B.4 depicts the time to split a communicator into two ranges of equal size with MPI respectively RBC. This splitting technique is, for example, frequently used by hypercube quicksort algorithms. We see that the time to split MPI communicators increases piece-wise linearly with the communicator's size. Thus, it depends on the communicator which MPI splitting method works the best. Overall, communicator splitting with MPI is very inefficient. For example, splitting a communicator containing 2^{15} PEs takes between 94.62 and 110.00 milliseconds. On the contrary, communicator splitting with RBC is multiple orders of magnitude faster.

Figure B.5 depicts the running times of the MPI collectives broadcast (`MPI_Bcast`), (all-)reduction (`MPI_Reduce` resp. `MPI_Allreduce`), exclusive prefix sum (`MPI_Exscan`), and (all-)gather (`MPI_Gather` resp. `MPI_Allgather`) as well as their counterparts provided by the RBC library. The exclusive prefix sum and the all-reduction is the most expensive MPI collective. RBC performs these collectives more than one order of magnitude faster. Note that the prefix sum and the reduction are important subroutines of sorting algorithms, e.g., to sum up local bucket sizes or to compute the rank of an element within a bucket across PE boundaries. The other collectives are sometimes slightly faster either with RBC or with MPI.

We now study the performance impact of RBC on our competitors HykSort, HSS, and HykQuick. RBC improves the performance of HykSort, HSS, and HykQuick significantly.

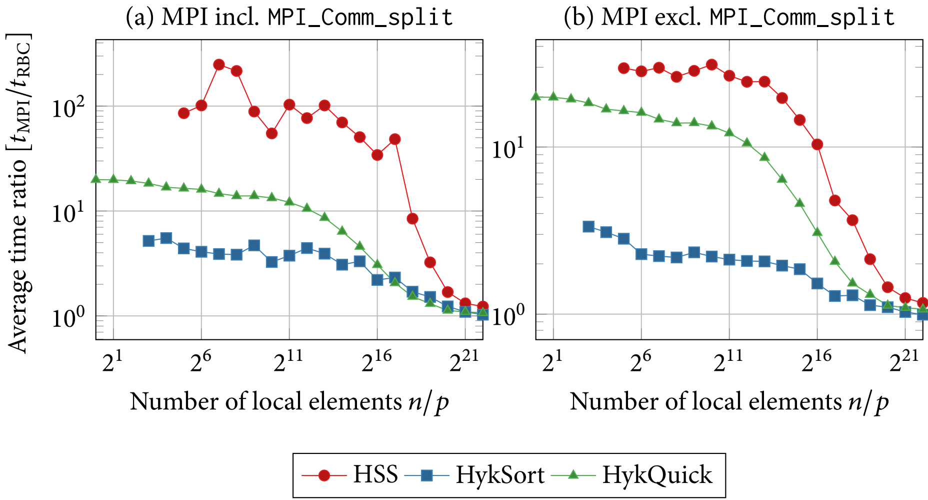


Figure B.6: Running time ratios of algorithms with MPI collectives over algorithms with RBC collectives. The left figure includes the running times of the MPI collective `MPI_Comm_split` for sub-communicator creation. The right figure excludes the running times of the MPI collective `MPI_Comm_split`.

Figure B.6a shows the speed of HykSort, HSS, and HykQuick with RBC operations over the original implementations that use pure MPI. On SuperMUC-NG, HSS is our closest competitor for $n/t \geq 2^{12}$. For these input sizes, RBC speeds up HSS up to a factor of 101. When we exclude the time to construct MPI communicators, RBC still speeds up HykSort, HSS, and HykQuick significantly (see Figure B.6b). For example, RBC decreases the running time of HSS by a factor of 25 for $n/t = 2^{12}$.

Our sorting algorithms RQuick, RFIS, and AMS-sort also benefit from the RBC library enormously. When we configure RBC to use MPI operations, RFIS is slowed down by a factor 39.75 for $n/t = 1/243$, RQuick by a factor of 39.51 for $n/t = 2^6$, and AMS-sort by a factor of 2.47 for $n/t = 2^{17}$. For these input sizes, the algorithms ran the fastest compared to the other algorithms.

List of Algorithms

2 Overview of Sequential and Shared-Memory Sorting Algorithms
1 Element classification of the first $u\lfloor n/u \rfloor$ elements . . . 20

3 Robust Scalable In-Place Sorting Algorithms
2 Task Scheduler . . . 35

4 Experiments and Conclusion
3 Quartet comparison . . . 51
4 100B comparison . . . 51

5 Overview of Distributed Sorting Algorithms
5 Hypercube algorithm design pattern . . . 89
6 Binomial-tree algorithm design pattern . . . 90

6 Robust Scalable Distributed Sorting Algorithms
7 Robust Hypercube Quicksort^{+} . . . 107
8 Robust Hypercube Quicksort . . . 111
9 Computation of the Simple Message Assignment . . . 114
10 DMA compare function . . . 115
11 Deterministic Message Assignment . . . 118
12 RLM-sort . . . 121
13 AMS-sort . . . 123

A Sequential and Shared-Memory Sorting
14 Sequential task execution of the subarray $A[b, e-1]$ without a local stack . . . 157

List of Figures

1 Introduction and Overview

1.1 Running time of ping-pong benchmarks between two fixed PEs. 7

2 Overview of Sequential and Shared-Memory Sorting Algorithms

2.1 Branchless decision tree. 19

3 Robust Scalable In-Place Sorting Algorithms

3.1 Overview of a parallel k-way partitioning step. 26
3.2 Classification of IPS^{4}o. 27
3.3 Input array after parallel classification. 27
3.4 Invariant during block permutation. 28
3.5 Block permutation examples. 29
(a) Swapping a block into its correct position. 29
(b) Moving a block into an empty position, followed by refilling the swap buffer. 29
3.6 An example of a permutation phase. 30
3.7 An example of the steps performed during cleanup. 31
3.8 Example schedule of a task execution in IPS^{4}o with 8 PEs. 34

4 Experiments and Conclusion

4.1 Examples of nontrivial input distributions for 512 uint32 values. 52
4.2 Running times of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. 60
4.3 Pairwise performance profiles of sequential algorithms. 62
4.4 Running times of parallel algorithms. 70
4.5 Speedup of parallel algorithms. 71
4.6 Running times of parallel algorithms on machine A1x64. 74
4.7 Pairwise performance profiles of parallel algorithms. 75
4.8 Accumulated running time of IPS^{4}o and IPS^{2}Ra. 78

6 Robust Scalable Distributed Sorting Algorithms

6.1 Robust fast work-inefficient ranking on a 3×3 PE-grid with five elements. . . 102
6.2 Comparison of ternary remedian and binary remedian with $k = 2$. 106
6.3 Distribution of the random shuffling (Shuffle), RQuick's and RQuick$^+$'s measured imbalance factors obtained from 2^{16} runs each. 112

6.4 Illustration of the different message assignments. 113
6.5 Algorithm schema of Recurse Last Parallel Multiway Mergesort. 120
6.6 Algorithm schema of AMS-sort. 122
6.7 Scanning algorithm for tie-breaking in AMS-sort. 125

7 Experiments and Conclusion
7.1 Running times of distributed sorting algorithms on JUQUEEN. 134
7.2 Running time ratios of each algorithm to the fastest algorithm on JUQUEEN. 135
7.3 Running times of distributed sorting algorithms on SuperMUC-NG. 136
7.4 Running time ratios of each algorithm to the fastest algorithm on SuperMUC-NG. 137
7.5 Running time comparison of gather-merge algorithms, RFIS, and RFIR. . . . 139
7.6 Average running time and maximum imbalance ratios of RQuick to NRQuick. 141
7.7 Running times of RQuick, Minisort, and RBC-HykQuick. 142
7.8 Running time ratios with and without DMA. 145
7.9 Accumulated running time of the phases of AMS-sort on SuperMUC-NG. . 146
7.10 Ratio of AMS-sort and NTB-AMS on the SuperMUC-NG. 147
7.11 Running times of single-level algorithms and AMS-sort. 147

A Sequential and Shared-Memory Sorting
A.1 Running times of sequential algorithms of uint32 values with input distribution Uniform executed on different machines. 162
A.2 Running times of parallel algorithms on machine A1x64. 169
A.3 Running times of parallel algorithms on machine I2x16. 170
A.4 Running times of parallel algorithms on machine A1x16. 171
A.5 Running times of parallel algorithms on machine I4x20. 172

B On the Performance of MPI Libraries
B.1 Ping-pong benchmark with 256 partners on SuperMUC Phase 2 with IBM MPI. 179
B.2 Ping-pong benchmark with 256 partners on SuperMUC Phase 2 with Intel MPI. 180
B.3 Ping-pong benchmark with 256 partners on SuperMUC-NG with Intel MPI. . 181
B.4 Running time of communicator splitting. 182
B.5 Running time of collective operations 183
B.6 Running time ratios of algorithms with MPI collectives over algorithms with RBC collectives. 184

List of Tables

2 Overview of Sequential and Shared-Memory Sorting Algorithms
2.1 Summary of notations 18

3 Robust Scalable In-Place Sorting Algorithms
3.1 I/O volume of read and write operations broken down into subroutines. . . . 42

4 Experiments and Conclusion
4.1 Average slowdowns of sequential algorithms. 57
4.2 Average slowdowns for different arrays. 65
4.3 Average slowdown of IPS^{4}o and IPS^{4}oNT. 66
4.4 Average slowdowns of parallel algorithms. 68
4.5 Average slowdowns of IPS^{4}o and IMSDradix. 77

5 Overview of Distributed Sorting Algorithms
5.1 Summary of notations 87
5.2 Complexity of distributed sorting algorithms. 93

7 Experiments and Conclusion
7.1 Efficiency of distributed sorting algorithms. 149

A Sequential and Shared-Memory Sorting
A.1 Running times of sequential algorithms of uint64 values with input distribution Uniform from different machines. 161
A.2 Average slowdowns of sequential algorithms on I4x20. 163
A.3 Average slowdowns of sequential algorithms on A1x16. 164
A.4 Average slowdowns of sequential algorithms on I2x16. 165
A.5 Average slowdowns of sequential algorithms on A1x64. 166
A.6 Average slowdowns of 1S^{4}o and S^{4}oS. 167
A.7 Running times of parallel algorithms of uint64 values with input distribution Uniform from different machines. 168
A.8 Average slowdowns of parallel algorithms on machine A1x64. 173
A.9 Average slowdowns of parallel algorithms on machine I2x16. 174
A.10 Average slowdowns of parallel algorithms on machine A1x16. 175
A.11 Average slowdowns of parallel algorithms on machine I4x20. 176

List of Theorems

3 Robust Scalable In-Place Sorting Algorithms

Lemma	3.1	Relationship between array position and PE index	32
Lemma	3.2	Relationship between PE of sequential task and parallel parent task	32
Lemma	3.3	Parallel subtasks are processed by subgroup of PEs	33
Lemma	3.4	One parallel task per recursion level and PE	33
Lemma	3.5	Relationship between subarray and PE-group	35
Lemma	3.6	Relationship between PE and its sequential tasks	36
Theorem	3.7	Memory usage of IPS^{4}o without local stack	38
Theorem	3.8	Memory usage of IPS^{4}o with local stack	39
Theorem	3.9	I/O complexity of IPS^{4}o	39
Lemma	3.10	I/Os of a sequential partitioning task	39
Lemma	3.11	I/Os of a parallel task	40
Theorem	3.12	Local work of IPS^{4}o	44
Lemma	3.13	Local work of a partitioning task	44
Lemma	3.14	Relationship between parallel tasks and small tasks	44
Lemma	3.15	Local work of small partitioning tasks	45
Lemma	3.16	Local work of partitioning tasks	45
Lemma	3.17	Local work of large partitioning tasks	45
Lemma	3.18	Local work of base case tasks	46
Lemma	3.19	Local work for samples of small partitioning tasks	46
Lemma	3.20	Local work for samples of large partitioning tasks	46

6 Robust Scalable Distributed Sorting Algorithms

Lemma	6.1	Scaled balls into bins problem	103
Lemma	6.2	Distribution of elements in randomized shuffling	104
Lemma	6.3	Bound load of randomized shuffling	105
Lemma	6.4	Running time of randomized shuffling on hypercubes	105
Lemma	6.5	Ternary remedian	106
Theorem	6.6	Running time of RQuick$^+$	109
Lemma	6.7	Quality of the splitter selection	109
Lemma	6.8	Maximum load of subcubes in RQuick$^+$	110
Lemma	6.9	Placement of elements in a subcube of RQuick$^+$	110
Lemma	6.10	Maximum load of PEs in RQuick$^+$	110
Theorem	6.12	Data redistribution with the deterministic message assignment	116

Theorem 6.13 Running time of RLM-sort with $\mathcal{O}(1)$ recursion levels 119
Lemma 6.14 Optimality of overpartitioning 121
Lemma 6.15 Imbalance bound of AMS-sort for one recursion level 122
Lemma 6.16 Running time of single-level AMS-sort 123
Theorem 6.17 Running time of AMS-sort with r recursion levels 124

A Sequential and Shared-Memory Sorting
Theorem A.1 Recursion depth of IPS^{4}o 155
Lemma A.2 Probability of a successful recursion step 155
Lemma A.3 Limit recursion depth of an arbitrary element 156
Lemma A.4 Sequential tasks of a PE are adjacent 157

Publications and Supervised Theses

In Conference Proceedings

Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. "Practical Massively Parallel Sorting". In: *27th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015, pages 13–23. DOI: 10.1145/2755573.2755595

Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *2016 IEEE International Conference on Big Data*. 2016, pages 172–183. DOI: 10.1109/bigdata.2016.7840603

Michael Axtmann and Peter Sanders. "Robust Massively Parallel Sorting". In: *29th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2017, pages 83–97. DOI: 10.1137/1.9781611974768.7

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)". In: *25th Annual European Symposium on Algorithms (ESA)*. 2017, 9:1–9:14. DOI: 10.4230/LIPIcs.ESA.2017.9

Michael Axtmann, Armin Wiebigke, and Peter Sanders. "Lightweight MPI Communicators with Applications to Perfectly Balanced Quicksort". In: *32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pages 254–265. DOI: 10.1109/IPDPS.2018.00035

Technical Reports

Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. "Practical Massively Parallel Sorting". In: *Computing Research Repository (CoRR)* (Aug. 2015). arXiv: 1410.6754

Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *Computing Research Repository (CoRR)* (Aug. 2016). arXiv: 1608.05634

Michael Axtmann and Peter Sanders. "Robust Massively Parallel Sorting". In: *Computing Research Repository (CoRR)* (June 2016). arXiv: 1606.08766

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-place Parallel Super Scalar Samplesort (IPS^{4}o)". In: *Computing Research Repository (CoRR)* (May 2017). arXiv: 1705.02257

Michael Axtmann, Armin Wiebigke, and Peter Sanders. "Lightweight MPI Communicators with Applications to Perfectly Balanced Quicksort". In: *Computing Research Repository (CoRR)* (Oct. 2017). arXiv: 1710.08027

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. *Engineering In-place (Shared-memory) Sorting Algorithms*. Computing Research Repository (CoRR). Sept. 2020. arXiv: 2009.13569

Theses

Michael Axtmann. "Echtzeitfähige Implementierung, Visualisierung und Analyse von STaC". Bachelor's Thesis. Karlsruhe Institute of Technology, Feb. 2012

Michael Axtmann. "Structural Variant Detection in Genome Sequencing". Master's Thesis. Karlsruhe Institute of Technology, May 2014

Supervised Theses

Armin Wiebigke. "Massively Parallel Schizophrenic Quicksort". Bachelor Thesis. Karlsruhe Institute of Technology, Feb. 2017

Bibliography

[AH19] Martin Aumüller and Nikolaj Hass. “Simple and Fast BlockQuicksort using Lomuto’s Partitioning Scheme”. In: *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019, pages 15–26. DOI: 10.1137/1.9781611975499.2. [see page 50]

[AMD15] AMD Inc. *AMD Athlon™ 64 X2 Dual-Core Processor Product Data Sheet*. 2015. URL: https://www.amd.com/system/files/TechDocs/33425.pdf, archived at https://web.archive.org/web/20190315070511/https://www.amd.com/system/files/TechDocs/33425.pdf on Mar. 15, 2019. [see page 1]

[AMD20] AMD Inc1. *AMD Ryzen™ Threadripper™ PRO 3995WX*. 2020. URL: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-pro-3995wx, archived at https://web.archive.org/web/20201128100115/https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-pro-3995wx on Nov. 28, 2020. [see page 1]

[App20] Apple Inc. *Apple iPad Air with A14 Bionic*. 2020. URL: https://www.apple.com/newsroom/2020/09/apple-unveils-all-new-ipad-air-with-a14-bionic-apples-most-advanced-chip/, archived at https://web.archive.org/web/20201003014200/https://www.apple.com/newsroom/2020/09/apple-unveils-all-new-ipad-air-with-a14-bionic-apples-most-advanced-chip/ on Oct. 3, 2020. [see page 1]

[Arg+08] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. “Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors”. In: *20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2008, pages 197–206. DOI: 10.1145/1378533.1378573. [see pages 4, 39]

[AS16] Michael Axtmann and Peter Sanders. “Robust Massively Parallel Sorting”. In: *Computing Research Repository (CoRR)* (June 2016). arXiv: 1606.08766. [see page 193]

[AS17] Michael Axtmann and Peter Sanders. “Robust Massively Parallel Sorting”. In: *29th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2017, pages 83–97. DOI: 10.1137/1.9781611974768.7. [see pages 10–12, 85, 93, 102, 193]

[AWS17] Michael Axtmann, Armin Wiebigke, and Peter Sanders. “Lightweight MPI Communicators with Applications to Perfectly Balanced Quicksort”. In: *Computing Research Repository (CoRR)* (Oct. 2017). arXiv: 1710.08027. [see page 194]

[AWS18] Michael Axtmann, Armin Wiebigke, and Peter Sanders. "Lightweight MPI Communicators with Applications to Perfectly Balanced Quicksort". In: *32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pages 254–265. DOI: 10.1109/IPDPS.2018.00035. [see pages 10, 11, 85, 96, 129, 182, 193]

[Axt+15a] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. "Practical Massively Parallel Sorting". In: *27th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2015, pages 13–23. DOI: 10.1145/2755573.2755595. [see pages 10, 80, 85, 93, 121, 193]

[Axt+15b] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. "Practical Massively Parallel Sorting". In: *Computing Research Repository (CoRR)* (Aug. 2015). arXiv: 1410.6754. [see page 193]

[Axt+17a] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-place Parallel Super Scalar Samplesort (IPS^{4}o)". In: *Computing Research Repository (CoRR)* (May 2017). arXiv: 1705.02257. [see page 193]

[Axt+17b] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)". In: *25th Annual European Symposium on Algorithms (ESA)*. 2017, 9:1–9:14. DOI: 10.4230/LIPIcs.ESA.2017.9. [see page 193]

[Axt+17c] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)". In: *25th Annual European Symposium on Algorithms (ESA)*. 2017, 9:1–9:14. DOI: 10.4230/LIPIcs.ESA.2017.9. [see pages 9, 15, 22, 53, 64]

[Axt+20] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. *Engineering In-place (Shared-memory) Sorting Algorithms*. Computing Research Repository (CoRR). Sept. 2020. arXiv: 2009.13569. [see pages 9, 15, 77, 194]

[Axt12] Michael Axtmann. "Echtzeitfähige Implementierung, Visualisierung und Analyse von STaC". Bachelor's Thesis. Karlsruhe Institute of Technology, Feb. 2012. [see page 194]

[Axt14] Michael Axtmann. "Structural Variant Detection in Genome Sequencing". Master's Thesis. Karlsruhe Institute of Technology, May 2014. [see page 194]

[Axt20a] Michael Axtmann. *(Parallel) Super Scalar Sample Sort*. https://github.com/ips4o/ps4o. Accessed: 2020-09-01. 2020. [see pages 49, 50]

[Axt20b] Michael Axtmann. *NUMA Array*. https://github.com/ips4o/NumaArray. Accessed: 2020-09-01. 2020. [see page 64]

[Bad13] Michael Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*. Volume 9. Texts in Computational Science and Engineering. Springer, 2013. DOI: 10.1007/978-3-642-31046-1. [see page 3]

[BAD20] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. "ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines". In: *32nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2020, pages 507–509. DOI: 10.1145/3350755.3400254. [see page 77]

[Bal+09] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jesper Larsson Träff. "MPI on a Million Processors". In: *16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*. 2009, pages 20–30. DOI: 10.1007/978-3-642-03770-2_9. [see page 180]

[Bal+95a] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers". In: *IEEE Transactions on Parallel and Distributed Systems* 6.2 (1995), pages 154–164. DOI: 10.1109/71.342126. [see page 91]

[Bal+95b] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers". In: *IEEE Transactions on Parallel and Distributed Systems* 6.2 (1995), pages 154–164: IEEE. DOI: 10.1109/71.342126. [see page 180]

[Bat68] Kenneth E. Batcher. "Sorting Networks and Their Applications". In: *American Federation of Information Processing Societies (AFIPS) Conference*. 1968, pages 307–314. DOI: 10.1145/1468075.1468121. [see pages 12, 88, 95, 116]

[BDadH95] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. "Truly Efficient Parallel Algorithms: c-Optimal Multisearch for an Extension of the BSP Model". In: *3rd Annual European Symposium on Algorithms (ESA)*. 1995, pages 17–30. DOI: 10.1007/3-540-60313-1_131. [see page 92]

[BES17] Timo Bingmann, Andreas Eberle, and Peter Sanders. "Engineering Parallel String Sorting". In: *Algorithmica* 77.1 (2017), pages 235–286: Springer. DOI: 10.1007/s00453-015-0071-1. [see pages 2, 18, 21, 37]

[BFV07] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. "Engineering a cache-oblivious sorting algorithm". In: *ACM Journal of Experimental Algorithmics* 12 (2007), 2.2:1–2.2:23. DOI: 10.1145/1227161.1227164. [see page 2]

[BGS10] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. "Low depth cache-oblivious algorithms". In: *22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2010, pages 189–199. DOI: 10.1145/1810479.1810519. [see pages 2, 21, 49, 98]

[Bin+16a] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *2016 IEEE International Conference on Big Data*. 2016, pages 172–183. DOI: 10.1109/bigdata.2016.7840603. [see page 193]

[Bin+16b] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *Computing Research Repository (CoRR)* (Aug. 2016). arXiv: 1608.05634. [see page 193]

[Bin18a] Timo Bingmann. "Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools". PhD thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, 2018. 374 pages. DOI: 10.5445/IR/1000085031. URL: https://arxiv.org/abs/1808.00963. [see page 55]

[Bin18b] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. 2018. URL: https://panthema.net/tlx, archived at https://web.archive.org/web/20191103202232/https://panthema.net/tlx on Nov. 3, 2019. [see page 130]

[BK86] Huang Bing-Chao and Donald E Knuth. "A one-way, stackless quicksort algorithm". In: *BIT Numerical Mathematics* 26.1 (1986), pages 127–130: Springer . [see page 20]

[Ble+96] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. "A Comparison of Sorting Algorithms for the Connection Machine CM-2". In: *3rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1996, pages 273–297. DOI: 10.1145/113379.113380. [see pages 2, 3, 21, 93, 94, 96, 98]

[BMS20] Timo Bingmann, Jasper Marianczuk, and Peter Sanders. *Engineering Faster Sorters for Small Sets of Items*. Computing Research Repository (CoRR). Aug. 2020. arXiv: 2002.05599. [see page 80]

[Bok91] Shahid H. Bokhari. "Multiphase Complete Exchange on a Circuit Switched Hypercube". In: *International Conference on Parallel Processing*. 1991, pages 525–529. [see page 90]

[Bor13] Shekhar Borkar. "Exascale Computing - A Fact or a Fiction?" In: *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2013, page 3. Keynote presentation. DOI: 10.1109/IPDPS.2013.121. [see page 92]

[Bra17] Berenger Bramas. "A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake". In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 8.10 (2017), pages 337–344. DOI: 10.14569/ijacsa.2017.081044. [see page 80]

[Bru+97] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. "Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 8.11 (1997), pages 1143–1156. DOI: 10.1109/71.642949. [see page 103]

[CG08] Mohamad Chaarawi and Edgar Gabriel. "Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators". In: *8th International Conference on Computational Science (ICCS)*. 2008, pages 297–306. DOI: `10.1007/978-3-540-69384-0_35`. [see page 181]

[Cho+15] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. "PARADIS: An Efficient Parallel Algorithm for Inplace Radix Sort". In: *International Conference on Very Large Data Bases (VLDB)* 8.12 (2015), pages 1518–1529. DOI: `10.14778/2824032.2824050`. [see page 22]

[Cod+17] Michael Codish, Luís Cruz-Filipe, Markus Nebel, and Peter Schneider-Kamp. "Optimizing sorting algorithms by using sorting networks". In: *Formal Aspects of Computing* 29.3 (2017), pages 559–579. DOI: `10.1007/s00165-016-0401-3`. [see page 80]

[Col88] Richard Cole. "Parallel Merge Sort". In: *SIAM Journal on Computing* 17.4 (1988), pages 770–785. DOI: `10.1137/0217049`. [see page 3]

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. [see page 1]

[Cor20] Intel Corporation. *Intel® Integrated Performance Primitives*. `https://software.intel.com/en-us/ipp-dev-reference`. Version 2020 Initial Release. 2020. [see pages 2, 50]

[CR10] Richard Cole and Vijaya Ramachandran. "Resource Oblivious Sorting on Multicores". In: *37th International Colloquium on Automata, Languages, and Programming (ICALP)*. 2010, pages 226–237. DOI: `10.1007/978-3-642-14165-2_20`. [see page 98]

[CR72] Stephen A. Cook and Robert A. Reckhow. "Time-Bounded Random Access Machines". In: *4th Annual ACM Symposium on Theory of Computing (STOC)*. 1972, pages 73–80. DOI: `10.1145/800152.804898`. [see page 4]

[Den03] John M. Dennis. "Partitioning with Space-Filling Curves on the Cubed-Sphere". In: *17th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2003, page 269. DOI: `10.1109/IPDPS.2003.1213486`. [see page 3]

[Din+11] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. "Noncollective Communicator Creation in MPI". In: *18th European MPI Users' Group Meeting (EuroMPI)*. 2011, pages 282–291. DOI: `10.1007/978-3-642-24449-0_32`. [see page 180]

[DJW14] Brian C. Dean, Rommel Jalasutram, and Chad G. Waters. "Lightweight Approximate Selection". In: *27th Annual European Symposium on Algorithms (ESA)*. 2014, pages 309–320. DOI: `10.1007/978-3-662-44777-2_26`. [see pages 105, 106]

[DM02] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical Programming* 91.2 (2002), pages 201–213: Springer. DOI: `10.1007/s101070100263`. [see page 56]

[DP73] David H Douglas and Thomas K Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2 (1973), pages 112–122: University of Toronto Press . [see page 81]

[Dur86] Branislav Durian. “Quicksort Without a Stack”. In: *Mathematical Foundations of Computer Science*. 1986, pages 283–289. DOI: 10.1007/BFb0016252. [see pages 20, 157]

[EKS12] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. “Branch Mispredictions Don’t Affect Mergesort”. In: *11th International Symposium on Experimental Algorithms (SEA)*. 2012, pages 160–171. DOI: 10.1007/978-3-642-30850-5_15. [see page 22]

[EW16] Stefan Edelkamp and Armin Weiß. “BlockQuicksort: Avoiding Branch Mispredictions in Quicksort”. In: *24th Annual European Symposium on Algorithms (ESA)*. 2016, 38:1–38:16. DOI: 10.4230/LIPIcs.ESA.2016.38. [see pages 2, 21, 50, 51, 59]

[EW19] Stefan Edelkamp and Armin Weiß. “Worst-Case Efficient Sorting with QuickMergesort”. In: *21st Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019, pages 1–14. DOI: 10.1137/1.9781611975499.1. [see page 22]

[EW92] Vladimir Estivill-Castro and Derick Wood. “A Survey of Adaptive Sorting Algorithms”. In: *ACM Computing Surveys* 24.4 (1992), pages 441–476. DOI: 10.1145/146370.146381. [see page 81]

[FG05] Gianni Franceschini and Viliam Geffert. “An in-place sorting with $\mathcal{O}(n \log n)$ comparisons and $O(n)$ moves”. In: *Journal of the ACM* 52.4 (2005), pages 515–537. DOI: 10.1145/1082036.1082037. [see page 22]

[FM70] W. Donald Frazer and A. C. McKellar. “Samplesort: A Sampling Approach to Minimal Storage Tree Sorting”. In: *Journal of the ACM* 17.3 (1970), pages 496–507. DOI: 10.1145/321592.321600. [see page 21]

[For12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.0. mpi-forum.org, 2012, pages 1–852. [see page 88]

[FP92] Rhys S. Francis and L. J. H. Pannan. “A parallel partition for enhanced parallel QuickSort”. In: *Parallel Computing* 18.5 (1992), pages 543–550. DOI: 10.1016/0167-8191(92)90089-P. [see pages 2, 20]

[Fra04] Gianni Franceschini. “Proximity Mergesort: optimal in-place sorting in the cache-oblivious model”. In: *15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2004, pages 291–299. [see pages 2, 22]

[Fri56] Edward H. Friend. “Sorting on Electronic Computer Systems”. In: *Journal of the ACM* 3.3 (1956), pages 134–168. DOI: 10.1145/320831.320833. [see page 10]

[FW78] Steven Fortune and James Wyllie. “Parallelism in Random Access Machines”. In: *4th Annual ACM Symposium on Theory of Computing (STOC)*. 1978, pages 114–118. DOI: 10.1145/800133.804339. [see page 4]

[FW86] Philip J. Fleming and John J. Wallace. "How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results". In: *Communications of the ACM* 29.3 (1986), pages 218–221. DOI: 10.1145/5666.5673. [see page 56]

[GG10] Viliam Geffert and Jozef Gajdos. "Multiway in-place merging". In: *Theoretical Computer Science* 411.16-18 (2010), pages 1793–1808: Elsevier. DOI: 10.1016/j.tcs.2010.01.034. [see page 22]

[Gib89] Phillip B. Gibbons. "A More practical PRAM Model". In: *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, SPAA '89, Santa Fe, New Mexico, USA, June 18-21, 1989*. 1989, pages 158–168. DOI: 10.1145/72935.72953. [see page 6]

[GL91] Xiaojun Guan and Michael A. Langston. "Time-Space Optimal Parallel Merging and Sorting". In: *IEEE Transactions on Computers* 40.5 (1991), pages 596–602: IEEE. DOI: 10.1109/12.88483. [see page 22]

[GM14] Fuji Goro and Morwenn. *TimSort*. https://github.com/timsort/cpp-TimSort.git. Accessed: 2020-09-01. 2014. [see page 50]

[Goo99] Michael T. Goodrich. "Communication-Efficient Parallel Sorting". In: *SIAM Journal on Computing* 29.2 (1999), pages 416–432. DOI: https://doi.org/fdrws9. [see pages 3, 93, 97, 98]

[GOS21] Yan Gu, Omar Obeya, and Julian Shun. *Parallel In-Place Algorithms: Theory and Practice*. Computing Research Repository (CoRR). Jan. 2021. arXiv: 2103.01216. [see pages 17, 22]

[Gro17] REFRESH Bioinformatics Group. *RADULS2*. https://github.com/refresh-bio/RADULS. Accessed: 2020-09-01. 2017. [see page 50]

[GV94] Alexandros V. Gerbessiotis and Leslie G. Valiant. "Direct Bulk-Synchronous Parallel Algorithms". In: *Journal of Parallel and Distributed Computing* 22.2 (1994), pages 251–267: Elsevier. DOI: 10.1006/jpdc.1994.1085. [see pages 3, 93, 97, 98, 114, 120, 123]

[HBJ98] David R. Helman, David A. Bader, and Joseph JáJá. "A Randomized Parallel Sorting Algorithm with an Experimental Study". In: *JPDC* 52.1 (1998), pages 1–23. DOI: 10.1006/jpdc.1998.1462. [see pages 98, 104, 127, 132]

[HHL88] Sandra Mitchell Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman. "A survey of gossiping and broadcasting in communication networks". In: *Networks* 18.4 (1988), pages 319–349. DOI: 10.1002/net.3230180406. [see page 89]

[HKS19] Vipul Harsh, Laxmikant V. Kalé, and Edgar Solomonik. "Histogram Sort with Sampling". In: *The 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2019, pages 201–212. DOI: 10.1145/3323165.3323184. [see pages 3, 11, 12, 93, 97, 124, 127, 129, 130, 148, 177, 180, 182]

[HL92] Bing-Chao Huang and Michael A. Langston. "Fast Stable Merging and Sorting in Constant Extra Space". In: *The Computer Journal* 35.6 (1992), pages 643–650: IEEE. DOI: 10.1093/comjnl/35.6.643. [see page 22]

[HLR07] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. “Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2007, page 52. DOI: 10.1145/1362622.1362692. [see page 180]

[HNR90] Philip Heidelberger, Alan Norton, and John T. Robinson. “Parallel Quicksort Using Fetch-and-Add”. In: *IEEE Transactions on Computers* 39.1 (1990), pages 133–138: IEEE. DOI: 10.1109/12.46289. [see pages 2, 20]

[Hoa61] C. A. R. Hoare. “Algorithm 65: Find”. In: *Communications of the ACM* 4.7 (1961), pages 321–322. DOI: 10.1145/366622.366647. [see page 91]

[Hoa62] C. A. R. Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (1962), pages 10–15: IEEE. DOI: 10.1093/comjnl/5.1.10. [see pages 2, 20]

[HR89] Torben Hagerup and Christine Rüb. “Optimal Merging and Sorting on the EREW PRAM”. In: *Information Processing Letters* 33.4 (1989), pages 181–185. DOI: 10.1016/0020-0190(89)90138-5. [see page 116]

[HS16] Lorenz Hübschle-Schneider and Peter Sanders. “Communication Efficient Algorithms for Top-k Selection Problems”. In: *30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pages 659–668. DOI: 10.1109/IPDPS.2016.45. [see pages 91, 115]

[HSL10] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. “Scalable communication protocols for dynamic sparse data exchange”. In: *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2010, pages 159–168. DOI: 10.1145/1693453.1693476. [see page 130]

[Hüb16] Lorenz Hübschle-Schneider. *Super Scalar Sample Sort*. https://github.com/lorenzhs/ssssort. Accessed: 2020-09-01. 2016. [see pages 50, 54]

[HWF15] Kaixi Hou, Hao Wang, and Wu-chun Feng. “ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors”. In: *29th International Conference on Supercomputing (ICS)*. 2015, pages 383–392. DOI: 10.1145/2751205.2751247. [see pages 2, 49]

[II86] Hiroshi Imai and Masao Iri. “An optimal algorithm for approximating a piecewise linear function”. In: *Journal of Information Processing* 9.3 (1986), pages 159–162 . [see page 81]

[JáJ00] Joseph JáJá. “A Perspective on Quicksort”. In: *Computing in Science and Engineering* 2.1 (2000), pages 43–49: IEEE. DOI: 10.1109/5992.814657. [see pages 46, 47]

[JK77] Norman L. Johnson and Samuel Kotz. *Urn Models and Their Application: An Approach to Modern Discrete Probability Theory*. Approach to Modern Discrete Probability Theory. Wiley, 1977. ISBN: 9780471446309. [see page 103]

[JM14] Tomasz Jurkiewicz and Kurt Mehlhorn. “On a Model of Virtual Address Translation”. In: *ACM Journal of Experimental Algorithmics* 19.1 (2014), 1.9:1–1.9:28: ACM. DOI: 10.1145/2656337. [see page 2]

[Joh84] S Lennart Johnsson. "Combining Parallel and Sequential Sorting on a Boolean N-Cube". In: *ICPP*. 1984, pages 444–448. [see pages 12, 93, 95]

[KDD17] Marek Kokot, Sebastian Deorowicz, and Maciej Dlugosz. "Even Faster Sorting of (Not Only) Integers". In: *5th International Conference on Man-Machine Interactions (ICMMI)*. 2017, pages 481–491. DOI: 10.1007/978-3-319-67792-7_47. [see pages 2, 50]

[KK08] Pok-Son Kim and Arne Kutzner. "Ratio Based Stable In-Place Merging". In: *Theory and Applications of Models of Computation (TAMC)*. 2008, pages 246–257. DOI: 10.1007/978-3-540-79228-4_22. [see pages 22, 50]

[KK93] Laxmikant V. Kalé and Sanjeev Krishnan. "A Comparison Based Parallel Sorting Algorithm". In: *International Conference on Parallel Processing*. 1993, pages 196–200. DOI: 10.1109/ICPP.1993.17. [see pages 3, 12, 91, 93, 96–98, 114, 144, 150]

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 0-201-03803-X. [see pages 2, 90, 116]

[KPT96] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. "Practical In-Place Mergesort". In: *Nordic Journal of Computing* 3.1 (1996), pages 27–40: ACM . [see page 22]

[Kri+20] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. "The Case for a Learned Sorting Algorithm". In: *ACM International Conference on Management of Data (SIGMOD)*. 2020, pages 1001–1016. DOI: 10.1145/3318464.3389752. [see page 77]

[Kri20] Ani Kristo. *LearnedSort*. https://github.com/learnedsystems/LearnedSort. Accessed: 2021-02-11. 2020. [see pages 77, 81]

[Kru92] David W. Krumme. "Fast Gossiping for the Hypercube". In: *SIAM Journal on Computing* 21.2 (1992), pages 365–380. DOI: 10.1137/0221026. [see page 89]

[KS06a] Kanela Kaligosi and Peter Sanders. "How Branch Mispredictions Affect Quicksort". In: *14th Annual European Symposium on Algorithms (ESA)*. 2006, pages 780–791. URL: https://doi.org/10.1007/11841036%5C_69. [see pages 2, 21]

[KS06b] Kanela Kaligosi and Peter Sanders. "How Branch Mispredictions Affect Quicksort". In: *14th Annual European Symposium on Algorithms (ESA)*. Springer. Sept. 2006, pages 780–791. DOI: 10.1007/11841036_69. [see page 2]

[Kum+94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley Longman, Amsterdam, 1994, page 597. ISBN: 0805331700. [see pages 1, 9]

[Kus+14] Shrinu Kushagra, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro. "Multi-Pivot Quicksort: Theory and Experiments". In: *16th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2014, pages 47–60. DOI: 10.1137/1.9781611973198.6. [see page 20]

[Lan06] Hans-Werner Lang. *Bitonic sorting network for n not a power of* 2. 2006. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm, archived at https://web.archive.org/web/20180910044034/http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm on Sept. 10, 2018. [see pages 95, 129]

[LBH07] Justin Luitjens, Martin Berzins, and Tom Henderson. "Parallel space-filling curve generation through sorting". In: *Concurrency and Computation: Practice and Experience* 19.10 (2007), pages 1387–1402: John Wiley & Sons, Ltd. DOI: 10.1002/cpe.1179. [see page 3]

[Lei15] Leibniz-Rechenentrum. *SuperMUC-NG*. 2015. URL: https://www.lrz.de/services/compute/supermuc/systemdescription/, archived at https://web.archive.org/web/20181110053315/https://www.lrz.de/services/compute/supermuc/systemdescription/ on Nov. 10, 2018. [see pages 128, 178]

[Lei18] Leibniz-Rechenentrum. *SuperMUC-NG*. 2018. URL: https://doku.lrz.de/display/PUBLIC/SuperMUC-NG, archived at https://web.archive.org/web/20201111112101/https://doku.lrz.de/display/PUBLIC/SuperMUC-NG on Nov. 11, 2020. [see pages 6, 128]

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, 1992, pages 389–783. ISBN: 978-1-4832-0772-8. DOI: 10.1016/C2013-0-08299-0. [see pages 89, 128]

[LM92] Youran Lan and Magdi A. Mohamed. "Parallel Quicksort in hypercubes". In: *1992 Symposium on Applied Computing (SAC)*. 1992, pages 740–746. DOI: 10.1145/130069.130085. [see pages 3, 93, 95, 107, 108]

[LS94] Hui Li and Kenneth C. Sevcik. "Parallel Sorting by Over Partitioning". In: *The 6th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1994, pages 46–56. DOI: 10.1145/181014.192329. [see pages 98, 121]

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. "Engineering Radix Sort". In: *Computing Systems* 6.1 (1993), pages 5–27 . [see page 21]

[McF14] Mike McFadden. *WikiSort*. https://github.com/BonzaiThePenguin/WikiSort. Accessed: 2020-09-01. 2014. [see page 50]

[McG12] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012. ISBN: 978-0-521-17301-8. [see page 56]

[MG89] Charles U. Martel and Dan Gusfield. "A Fast Parallel Quicksort Algorithm". In: *Information Processing Letters* 30.2 (1989), pages 97–102. DOI: 10.1016/0020-0190(89)90116-6. [see pages 2, 20]

[MS03] Kurt Mehlhorn and Peter Sanders. "Scanning Multiple Sequences Via Cache Memory". In: *Algorithmica* 35.1 (2003), pages 75–93: Springer. DOI: 10.1007/s00453-002-0993-2. [see page 43]

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. ISBN: 978-0-521-83540-4. DOI: 10.1017/CBO9780511813603. [see page 8]

[Mus97] David R. Musser. "Introspective Sorting and Selection Algorithms". In: *Software: Practice and Experience* 27.8 (1997), pages 983–993: John Wiley & Sons, Ltd . [see page 20]

[MW18] J. Ian Munro and Sebastian Wild. "Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs". In: *26th Annual European Symposium on Algorithms (ESA)*. 2018, 63:1–63:16. DOI: 10.4230/LIPIcs.ESA.2018.63. [see page 81]

[Obe+19a] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. *RegionSort*. https://github.com/omarobeya/parallel-inplace-radixsort. Accessed: 2020-09-01. 2019. [see page 49]

[Obe+19b] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. "Theoretically-Efficient and Practical Parallel In-Place Radix Sorting". In: *The 31st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM. 2019, pages 213–224. [see pages 2, 22, 49, 50, 53]

[Pet02] Tim Peters. *Timsort*. http://svn.python.org/projects/python/trunk/Objects/listsort.txt. Accessed: 2020-03-31. 2002, archived at https://web.archive.org/web/20070301000000/ on Mar. 1, 2007. [see pages 50, 81]

[Pet15] Orson Peters. *Pattern-defeating quicksort*. https://github.com/orlp/pdqsort. Accessed: 2020-09-01. 2015. [see page 50]

[Pol14] Orestis Polychroniou. *In-place MSB*. http://www.cs.columbia.edu/~orestis/publications.html. Accessed: 2020-09-01. 2014. [see pages 49, 76]

[PR14] Orestis Polychroniou and Kenneth A. Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort". In: *ACM International Conference on Management of Data (SIGMOD)*. 2014, pages 755–766. DOI: 10.1145/2588555.2610522. [see pages 2, 21, 49, 63, 76]

[RB90] Peter J Rousseeuw and Gilbert W Bassett Jr. "The remedian: A Robust Averaging Method for Large Data Sets". In: *Journal of the American Statistical Association* 85.409 (1990), pages 97–104: Taylor & Francis Group. DOI: 10.2307/2289530. [see pages 105, 108]

[Rei07] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN: 978-0-596-51480-8. [see pages 2, 49, 53]

[San+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online". In: *ACM Transactions on Mathematical Software (TOMS)* 44.3 (Apr. 2018), 29:1–29:14: ACM. DOI: 10.1145/3157734. [see page 108]

[San+19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0. [see pages 1, 4, 6, 89]

[San00] Peter Sanders. "Fast Priority Queues for Cached Memory". In: *ACM Journal of Experimental Algorithmics* 5 (2000), page 7. DOI: 10.1145/351827.384249. [see page 90]

[San08a] Peter Sanders. *Course on Parallel Algorithms, Lecture slides, 100–105*. 2008. URL: https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf, archived at https://web.archive.org/web/20190819145046/https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf on Aug. 19, 2019. [see page 94]

[San08b] Peter Sanders. *Course on Parallel Algorithms, Lecture slides, 126–128*. 2008. URL: https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf, archived at https://web.archive.org/web/20190819145046/https://algo2.iti.kit.edu/sanders/courses/paralg08/vorlesung1.pdf on Aug. 19, 2019. [see page 91]

[SD15] Michael Stephan and Jutta Docter. "Jülich Supercomputing Centre. JUQUEEN: IBM Blue Gene/Q Supercomputer System at the Jülich Supercomputing Centre". In: *Journal of Large-Scale Research Facilities* (2015). DOI: 10.17815/jlsrf-1-18. [see pages xi, 128]

[SH97] Peter Sanders and Thomas Hansch. "Efficient Massively Parallel Quicksort". In: *4th Solving Irregularly Structured Problems in Parallel (IRREGULAR)*. 1997, pages 13–24. DOI: 10.1007/3-540-63138-0_2. [see pages 98, 104]

[Shu+12] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. "Brief announcement: the problem based benchmark suite". In: *24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2012, pages 68–70. DOI: 10.1145/2312005.2312018. [see pages 2, 49, 51]

[SK10] Edgar Solomonik and Laxmikant V. Kalé. "Highly Scalable Parallel Sorting". In: *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2010, pages 1–12. DOI: 10.1109/IPDPS.2010.5470406. [see pages 3, 91, 93, 96–98]

[Ska16a] Malte Skarupke. *I Wrote a Faster Sorting Algorithm*. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/. Accessed: 2020-03-31. 2016. [see pages 2, 21, 37, 50]

[Ska16b] Malte Skarupke. *Ska Sort*. https://github.com/skarupke/ska_sort. Accessed: 2020-09-01. 2016. [see pages 37, 50]

[SM08] Peter Sanders and Kurt Mehlhorn. *Algorithms and Data Structures - The Basic Toolbox*. Springer, 2008. ISBN: 978-3-540-77977-3. DOI: 10.1007/978-3-540-77978-0. [see page 91]

[SM13] Peterand Sebastian Schlag Sanders and Ingo Müller. "Communication Efficient Algorithms for Fundamental Big Data Problems". In: *2013 IEEE International Conference on Big Data*. 2013, pages 15–23. DOI: 10.1109/BigData.2013.6691549. [see pages xi, 92]

[SMB13] Hari Sundar, Dhairya Malhotra, and George Biros. "HykSort: a New Variant of Hypercube Quicksort on Distributed Memory Architectures". In: *27th International Conference on Supercomputing (ICS)*. 2013, pages 293–302. DOI: 10.1145/2464996.2465442. [see pages 3, 11, 12, 93, 95, 97, 98, 107, 108, 127, 129, 138, 140, 177, 180, 182]

[SS63] John C. Shepherdson and Howard E. Sturgis. "Computability of Recursive Functions". In: *Journal of the ACM* 10.2 (1963), pages 217–255. DOI: 10.1145/321160.321170. [see page 4]

[SSP07] Johannes Singler, Peter Sanders, and Felix Putze. "MCSTL: The Multi-core Standard Template Library". In: *13th International Euro-Par Conference*. 2007, pages 682–694. DOI: 10.1007/978-3-540-74466-5_72. [see pages 2, 3, 23, 49, 90, 91, 96, 130]

[SST09] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. "Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan". In: *Parallel Computing* 35.12 (2009), pages 581–594: Elsevier. DOI: 10.1016/j.parco.2009.09.001. [see pages 88, 91, 182]

[ST02] Peter Sanders and Jesper Larsson Träff. "The Hierarchical Factor Algorithm for All-to-All Communication". In: *8th International Euro-Par Conference*. 2002, pages 799–804. DOI: 10.1007/3-540-45706-2_112. [see pages 92, 129]

[SW04] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort". In: *12th Annual European Symposium on Algorithms (ESA)*. 2004, pages 784–796. DOI: 10.1007/978-3-540-30140-0_69. [see pages 2, 18, 21, 50, 54, 59, 79, 81, 91]

[SW11] Christian Siebert and Felix Wolf. "Parallel Sorting with Minimal Data". In: *18th European MPI Users' Group Meeting (EuroMPI)*. 2011, pages 170–177. DOI: 10.1007/978-3-642-24449-0_20. [see pages 3, 11, 93, 96, 105, 106, 114, 127, 130, 138]

[SyN18] Virginia Tech SyNeRGy Lab. *ASPaS*. https://github.com/vtsynergy/aspas_sort. Accessed: 2020-09-01. 2018. [see page 49]

[top05] top500.org. *TOP 10 Sites for November 2005*. 2005. URL: https://www.top500.org/lists/top500/2005/11/, archived at https://web.archive.org/web/20210103032640/https://www.top500.org/lists/top500/2005/11/ on Jan. 3, 2021. [see page 1]

[top20] top500.org. *TOP 10 Sites for November 2020*. 2020. URL: https://www.top500.org/lists/top500/2020/11/, archived at https://web.archive.org/web/20210103025922/https://www.top500.org/lists/top500/2020/11/ on Jan. 3, 2021. [see page 1]

[Trä18] Jesper Larsson Träff. "Practical, distributed, low overhead algorithms for irregular gather and scatter collectives". In: *Parallel Computing* 75 (2018), pages 100–117. DOI: 10.1016/j.parco.2018.04.003. [see page 88]

[TZ03] Philippas Tsigas and Yi Zhang. "A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000". In: *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP)*. 2003, page 372. DOI: 10.1109/EMPDP.2003.1183613. [see pages 2, 20, 49]

[Val90] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (1990), pages 103–111. DOI: 10.1145/79173.79181. [see page 4]

[Var+91] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. "Merging Multiple Lists on Hierarchical-Memory Multiprocessors". In: *Journal of Parallel and Distributed Computing* 12.2 (1991), pages 171–177: Elsevier. DOI: 10.1016/0743-7315(91)90022-2. [see pages 3, 91, 93, 96]

[vdGei91] Robert A van de Geijn. "Efficient global combine operations". In: *16th Distributed Memory Computing Conference*. IEEE Computer Society. 1991, pages 291–292. DOI: 10.1109/DMCC.1991.633147. [see page 89]

[vNeu45] John von Neumann. *The First Draft Report on the EDVAC (Electronic Discrete Variable Automatic Calculator)*. Technical report. Contract no. W-670-ORD-4926 between the United States Army Ordinance Department and the University of Pennsylvania. Moore School of Electrical Engineering, University of Pennsylvania, June 1945. [see page 4]

[Wag87] Bruce Wagar. "Hyperquicksort: A fast sorting algorithm for hypercubes". In: *2nd Conference on Hypercube Multiprocessors*. 1987, pages 292–299. [see pages 3, 93, 95, 107, 108]

[Weg87] Lutz M. Wegner. "A Generalized, One-Way, Stackless Quicksort". In: *BIT Numerical Mathematics* 27.1 (1987), pages 44–48: Springer. DOI: 10.1007/BF01937353. [see page 20]

[Wei16a] Armin Weiss. *BlockQuicksort*. www.github.com/weissan/BlockQuicksort. Accessed: 2020-09-01. 2016. [see page 50]

[Wei16b] Armin Weiss. *Yaroslavskiy's Dual-Pivot Quicksort*. https://github.com/weissan/BlockQuicksort/blob/master/Yaroslavskiy.h++. Accessed: 2020-09-01. 2016. [see page 50]

[Wen19] Jokob Wenzel. *Intel Threading Building Blocks with CMake build system*. https://github.com/wjakob/tbb. TBB 2019 Update 6. 2019. [see page 49]

[Wie17a] Armin Wiebigke. „Massively Parallel Schizophrenic Quicksort". Bachelor's Thesis. Department of Computer Science, Karlsruhe Institute of Technology, Germany, Feb. 2017. [see page 182]

[Wie17b] Armin Wiebigke. "Massively Parallel Schizophrenic Quicksort". Bachelor Thesis. Karlsruhe Institute of Technology, Feb. 2017. [see page 194]

[WS11] Jan Wassenberg and Peter Sanders. "Engineering a Multi-core Radix Sort". In: *International Euro-Par Conference*. 2011, pages 160–169. DOI: 10.1007/978-3-642-23397-5_16. [see page 2]

[Yar10] Vladimir Yaroslavskiy. *Question on sorting*. http://mail.openjdk.java.net/pipermail/core-libs-dev/2010-July/004649.html. Accessed: 2020-09-01. 2010. [see pages 2, 20, 50]