Karlsruher Institut für Technologie

Bachelor Thesis

Massively Parallel 'Schizophrenic' Quicksort

Armin Wiebigke

Date: February 28, 2017

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Michael Axtmann

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Sorting algorithms for distributed memory systems are essential for quickly sorting large amounts of data. The de facto standard for communication in High Performance Computing (HPC) is the Message Passing Interface (MPI). MPI uses the concept of communicators to connect groups of PEs. Recursive algorithms may split a group of PEs into subgroups and then execute the algorithm on these subgroups recursively. A straightforward approach to implementing such an algorithm is to create a MPI subcommunicator for each subgroup. However, the time to create a MPI communicator is not negligible on a large group of PEs and may therefore drastically increase the running time of the algorithm.

In this thesis, we present a communication library based on MPI that supports communicator creation in constant time and without communication. The library supports both point-to-point communication and non-blocking collective operations. We also present the first efficient implementation of Schizophrenic Quicksort, a recursive sorting algorithm for distributed memory systems that is based on Quicksort. Schizophrenic Quicksort guarantees perfect data balance with the drawback that some PEs have to work on two groups simultaneously. The only previous implementation scales bad on large supercomputers. We integrate our communication library into the implementation of Schizophrenic Quicksort and thus eliminate the cost for the communicator creation almost completely. We also avoid problems caused by blocking collective operations. We also implement a better pivot selection and reduce the amount of communication in each level of recursion. Furthermore, we extend an implementation of Hypercube Quicksort with our communication library.

We perform an extensive experimental evaluation of our implementations. In our experiments, our library reduces the time to create a communicator by a factor of more than 10 000 compared to MPI. In contrast, the collective operations of MPI outperformed the collective operations of our library for most inputs. On large numbers of PEs, splitting a communicator and executing the operation Broadcast 50 times (with medium inputs) with our library is still faster than a single communicator split with MPI. Our experimental results show that we improve the performance of Schizophrenic Quicksort by a factor of up to 40 and the performance of Hypercube Quicksort by a factor of up to 15 if we use our library instead of MPI. We compare our implementation of Schizophrenic Quicksort with the implementation of Hypercube Quicksort. The results indicate that Schizophrenic Quicksort is not able to outperform Hypercube Quicksort. However, Schizophrenic Quicksort runs on any number of PEs while Hypercube Quicksort only runs on 2^k PEs.

Zusammenfassung

Sortieralgorithmen für verteilte Speichersysteme sind für die schnelle Sortierung großer Datenmengen unerlässlich. Der De-facto-Standard für die Kommunikation in High Performance Computing (HPC) ist das Message Passing Interface (MPI). MPI nutzt das Konzept der Kommunikatoren, um Gruppen von PEs zu verbinden. Rekursive Algorithmen können eine Gruppe von PEs in Untergruppen aufteilen und dann den Algorithmus auf diesen Untergruppen rekursiv ausführen. Ein einfacher Ansatz zur Implementierung eines solchen Algorithmus besteht darin, einen MPI-Subkommunikator für jede Untergruppe zu erstellen. Allerdings hat die Erstellung eines MPI-Kommunikators bei einer großen Gruppe von PEs eine nicht vernachlässigbare Laufzeit, so dass sich die Gesamtlaufzeit des Algorithmus drastisch erhöhen kann.

In dieser Arbeit präsentieren wir eine auf MPI basierende Kommunikationsbibliothek, die die Erstellung von Kommunikatoren in konstanter Zeit und ohne Kommunikation unterstützt. Die Bibliothek unterstützt sowohl Punkt-zu-Punkt-Kommunikation als auch nicht blockierende kollektive Operationen. Wir präsentieren auch die erste effiziente Implementierung von Schizophrenic Quicksort, einem rekursiven Sortieralgorithmus für verteilte Speichersysteme, der auf Quicksort basiert. Schizophrenic Quicksort garantiert eine perfekte Gleichverteilung der Daten, mit dem Nachteil, dass einige PEs gleichzeitig in zwei Gruppen arbeiten müssen. Die einzige vorherige Implementierung skaliert schlecht auf großen Supercomputern. Wir integrieren unsere Kommunikationsbibliothek in die Implementierung von Schizophrenic Quicksort und eliminieren so die Kosten für die Kommunikation des Kommunikators fast vollständig. Ebenfalls vermeiden wir damit Probleme, die durch blockierende kollektive Operationen verursacht werden. Wir implementieren auch eine bessere Pivot-Auswahl und reduzieren den Kommunikationsaufwand in jeder Rekursionsstufe. Darüber hinaus erweitern wir eine Implementierung von Hypercube Quicksort mit unserer Kommunikationsbibliothek.

Wir führen eine umfangreiche experimentelle Auswertung unserer Implementierungen durch. In unseren Experimenten reduziert unsere Bibliothek die Zeit, um einen Kommunikator zu erstellen um einen Faktor von mehr als 10 000 im Vergleich zu MPI. Im Gegensatz dazu sind die kollektiven Operationen von MPI schneller als die kollektiven Operationen unserer Bibliothek für die meisten Eingabegrößen. Auf einer großen Anzahl von PEs dauert die Erstellung von Subkommunikatoren, gefolgt von einer fünfzigmaligen Ausführung der Operation Broadcast (mit mittlerer Eingabegröße) mit unserer Bibliothek immer noch kürzer als die Erstellung der entsprechenden Subkommunikatoren mit MPI. Unsere experimentellen Ergebnisse zeigen, dass wir die Performance von Schizophrenic Quicksort um einen Faktor von bis zu 40 verbessern, wenn wir unsere Bibliothek anstelle von MPI verwenden. Auf die gleiche Weise können wir die Performance von Hypercube Quicksort um einen Faktor von 15 steigern. Wir vergleichen unsere Implementierung von Schizophrenic Quicksort mit der Implementierung von Hypercube Quicksort. Die Ergebnisse zeigen, dass Schizophrenic Quicksort nicht in der Lage ist, Hypercube Quicksort zu übertreffen. Allerdings läuft Schizophrenic Quicksort auf einer beliebigen Anzahl von PEs, während Hypercube Quicksort nur auf 2^k PEs ausgeführt werden kann.

Acknowledgments

I would like to thank my supervisors M.Sc. Michael Axtmann and Prof. Dr. Peter Sanders for allowing me to work on this very interesting subject.
M.Sc. Michael Axtmann gave me valuable advice regarding theoretical questions as well as implementation techniques. Special thanks for proofreading my thesis and helping me to improve my writing skills. I also want to thank Tobias Heuer for providing me with the source code of his algorithm.

The authors 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 [19] 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).

The authors 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 (www.lrz.de).

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 27.02.2017

Contents

1 Introduction

Sorting is one of the most studied algorithmic problems and has many different applications. Sorting algorithms that run on a single processor are not well suited to sort very large amounts of data. Supercomputers make it possible to perform this task in a reasonable amount of time. The top performing supercomputers today have a distributed memory architecture. Such distributed memory systems require specialized algorithms that communicate the data between multiple PEs. There exist various sorting algorithms for supercomputers, e.g. algorithms based on Bitonic Sort [3, 18, 21], Samplesort [9, 27, 29] or Hypercube Quicksort [2, 32, 34].
Quicksort is a commonly used sequential sorting algorithm because of its optimal average complexity, simplicity and low overhead. When Quicksort is executed on an input, the data is partitioned based on a pivot element into small and large elements in each recursion. Then the small and large elements are both sorted recursively. Parallel Quicksort [10, 23, 30] is a simple approach to parallelize Quicksort. Firstly, all PEs collectively select a pivot element. Then each PE partitions its local data. The group of PEs is then split into two subgroups. The elements are redistributed between the PEs such that one subgroup contains all small elements and the other subgroup contains all large elements. Each subgroup sorts its element by executing Parallel Quicksort recursively. The number of small element is usually not equal to the number of large elements. The sizes of the subgroups are dynamically determined such that the data imbalance is minimal. Data imbalance means that some PEs have more local elements than other PEs. If a PE has more data, it executes the same task slower than other PEs. This mean that some PEs have finished the sorting while other PEs are still working. In this case the available resources are not fully utilized. Schizophrenic Quicksort is an adaptation of Parallel Quicksort that has been proposed [25, 30] but not intensively studied. Schizophrenic Quicksort retains perfectly balanced data, meaning all PEs have the same amount of data throughout the execution of the algorithm. The idea is that we redistribute the data such that each PE keeps the same number of elements. To achieve this, one PE has to have small and large elements. This PE is thus part of both subgroups and sorts on both subgroups simultaneously. A PE that works on two groups is called a *schizophrenic* PE. There exists a prototypical implementation of Schizophrenic Quicksort by Tobias Heuer [12] that does not scale well for large groups of PEs. The implementation uses the Message Passing Interface (MPI), which is the de facto standard for communication on supercomputers. MPI uses the concept of communicators to connect groups of PEs and provides collective operations that execute an operation on all PEs that are connected by a communicator. Using these collective operations can simplify the implementation of algorithms because it frees a developer from implementing and optimizing collective operations. To use the collective operations provided by MPI, we haves to create a MPI communicator for each subgroup. Creating communicators is expensive in comparison to performing a collective operation with a small input, at least for large groups of PEs. The prototypical implementation of Schizophrenic Quicksort creates two communicators on each level of recursion. For large groups of PEs, the communicator creation dominates the total running time for small and medium inputs. The prototypical implementation uses blocking collective operations of MPI. Blocking collective operations do not allow a schizophrenic PE to communicate on both groups simultaneously, thus the communication time is not optimal.

1.1 Contributions of this Thesis

In this thesis we present a communication library based on MPI that provides an interface very close to MPI. The main feature of the library is that communicators can be created in constant time. The library supports point-to-point communication and multiple non-blocking collective operations. Existing implementations that use MPI can integrate our library with very few modifications of the code. We present an efficient implementation of Schizophrenic Quicksort. The focus of the implementation is a good scalability for large numbers of PEs. The implementation is based on an existing implementation by Tobias Heuer [12]. We use our communication library to reduce the cost of the communicator creation. Furthermore, we implement an effective pivot selection and minimize the amount of communication in each level of recursion of Schizophrenic Quicksort. We integrate our library into an implementation of Robust Hypercube Quicksort [2] and show that this increases the performance by a factor of up to 15. We perform an experimental evaluation of all implementations on two different supercomputers using up to 262 144 cores. For the implementations of Schizophrenic Quicksort and Hypercube Quicksort, we run experiments with nine different input instances with an input size of up to 2^{20} elements on each PE.

1.2 Structure of this Thesis

In Chapter 2 we introduce basic terms and problem definitions. Chapter 3 gives an overview of related research about sorting algorithms designed for distributed systems and the implementation of collective operations. In Chapter 4 we explain the concept of collective operations and communicators in MPI. We also give scenarios in which commonly used implementations of MPI are unpractical. In Chapter 5 we explain the interface and the implementation of our communication library. In Chapter 6 we present our implementation of Schizophrenic Quicksort examines the problems of SchizoQS and presents our improvements for Schizophrenic Quicksort. Chapter 7 gives notable implementation details on the communication library and on the implementation of Schizophrenic Quicksort. In Chapter 8 we present an extensive experimental evaluation of the communication library and our implementation of Schizophrenic Quicksort.

2 Preliminaries

We study the problem of sorting n elements distributed over p processing elements (PEs) numbered $0..p{-}1$. For simplicity of exposure, we assume that $n = kp$ with $k \in \mathbb{N}$. The elements are evenly distributed, meaning each PE has the same number of elements $\frac{n}{p}$ as input. The output requirement is that the PEs store a permutation of the input elements such that the elements on each PE are sorted and that no element on PE i is larger than any elements on PE $i + 1$. We call multiple PEs that solve a problem collectively a *group*. Figure 2.1 shows a correct output schematically. When we split a group into two subgroups, we call the group with the lower indexes the left group and the other group the right group.

Memory Architecture and Communication

Modern supercomputers consist of a multitude of nodes. Each node has its own main memory and processor. The processor of a node usually contains more than one core. Each core is seen as an independent PE. To collectively solve a problem using multiple PEs, data is exchanged through a network via message passing. Stonebreaker [31] describes three architectures for multiprocessor high transaction rate systems: *Shared memory* (one central memory for all PEs), *shared disk* (each PE has its own memory but one disk is shared among all PEs) and *shared nothing* (each PE has its own memory and the PEs are connected by a network). By this classification, we design our algorithms for shared nothing. The *Message Passing Interface* (MPI) is the de facto standard for communication on supercomputers. The MPI standard defines syntax and semantics of a communication library and defines an interface for the operations of the library. There are several efficient and well-tested implementations of MPI for multiple programming languages, including Fortran and C. The unified interface allows the writing of portable code that can be compiled on different supercomputers. An important concept of MPI are communicators. A MPI communicator connects a (possibly empty) group of PEs. Each communicator assigns each contained PE a unique identifier (*rank*). MPI provides point-to-point communication and collective communication. In a point-to-point communication, a message containing data is transmitted between a pair of twos PEs. Such a pair consist of one PE that sends the message and one PE that receives the massage. A collective operation executes a distributed operation on multiple PEs. The communication involved in the execution of this operation can be described by a communication pattern. A communication patter defines which pairs of PEs

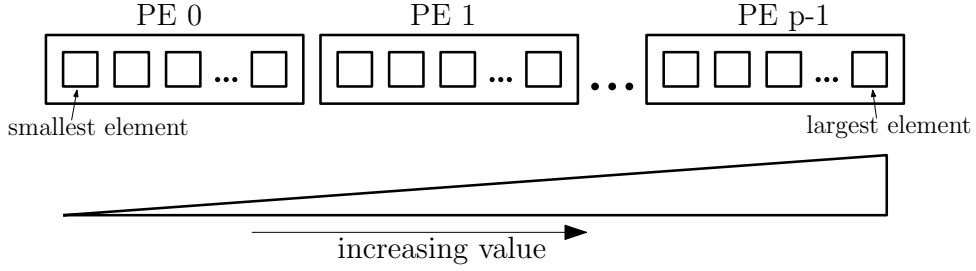

Figure 2.1: The data distribution after a sorting algorithm has been executed

communicate messages and when they do it. An example for a communication pattern is a binomial tree based communication [22]. A collective operation only describes input and output conditions, but does not enforce a specific communication pattern. Many collective operations send data from a single PE to all other PEs or take data from all PEs and stores a result on a single PE. The single PE is called the *root* PE. We describe several commonly used collective operations. MPI supports even more collective operations that we will not list here.

- **Broadcast**: Take data from the root PE and send the data to all other PEs.
- **Gather**: Collect data from all PEs on the root PE.
- **Reduce**: The data from all PEs gets combined by an associative operator $\oplus$. Let d_i be the data stored on the PE with rank i. The result $d_0 \oplus d_1 \oplus ... \oplus d_{p-1}$ is stored on the root PE.
- **Scan**: An associative operator $\oplus$ is used to combine data from multiple PEs. Let d_i be the data stored on the PE with rank i. The result of the Scan on PE i is $d_0 \oplus d_1 \oplus ... \oplus d_i$.

Model of Computation

We use the single-ported message passing model to analyze the communication cost. This model abstracts all communication to single messages. To calculate the time for the communication in the algorithm, we sum up all individual send messages. A message of size l machine words takes time $\alpha + l\beta$ to send. The parameter α models the startup overhead and β the time to communicate one machine word. For simplicity reasons, we equate the size of a data element with the machine word size. We assume that the time taken is independent on how much PEs communicate at the same time and that the communication links are bidirectional.

3 Related Work

There exist various adaptations of the sequential Quicksort algorithm [14] for parallel sorting algorithms. On distributed memory systems, all PEs collectively select a pivot element (or multiple pivot elements). Each PE partitions their local data. Then the data is redistributed.

Many algorithms include collective operations. Quicksort based algorithm usually use collective communication. If a pivot element is selected depending on the input, a collective operation is necessary to so that each PE receives the same pivot element. For small inputs, the collective operations dominate the communication time. An inefficient implementation of a collective operation thus significantly increases the running time of algorithm in specific circumstances.

3.1 Sorting Algorithms for Distributed Systems

3.1.1 Parallel Quicksort

Parallel Quicksort [10, 23, 30] adapts sequential Quicksort [14]. Parallel Quicksort performs four steps on each level of recursion. Firstly, all PEs collectively select one pivot element. Then all PEs partition their data into small (elements smaller than the pivot) and large elements (elements large than the pivot). Next the group is split into two subgroups, one subgroup for the small elements and one subgroup for the large elements. The size of the subgroups is chosen so that the data imbalance is minimized. Then we redistribute the data such that PEs of the left subgroup only receive small elements and the PEs of the right subgroup only receive large elements. The necessary coordination between the PEs to exchange the data correctly can be achieved by using the collective operations prefix sum and broadcast. Parallel Quicksort is then called recursively on both subgroups. If the algorithm is called on a group that only contains one PE, the elements are sorted locally and the algorithm terminates.

Sanders and Hansch [25] propose multiple improvements to the algorithm. They propose that the PEs sort their local elements at the beginning. This reduces the impact of load imbalance and also simplifies the pivot selection. To keep the invariant that all local elements are sorted, the sorted data sequences received during the data redistribution step are merged. The time for the merging is offset because the partitioning can be done in logarithmic time. To reduce the recursion depth compared to the sequential Quicksort, three pivots are used instead of only one. Each PE determines the $\frac{1}{4}, \frac{1}{2} and \frac{3}{4}$ quantile of its local data. The elements are gathered to one PE and the median of each quantile is chosen as one of the pivots. The three pivot elements are then broadcasted and each PE splits the data into four partitions. By using a hypercube algorithm for the data redistribution, the communication patterns can be simplified. This reduces the communication but only works when p is a power of two.

3.1.2 Samplesort

Samplesort [9] selects $p - 1$ splitters and then sorts these splitters. The data is partitioned into p partitions. All PEs send the data from partition i to PE i. Then each PE sorts its data locally.

Samplesort can be seen as an adaptation of Quicksort with $p-1$ pivot elements. In contrast to most other algorithms based on Quicksort, Samplesort is no recursive algorithm.

To select the splitters, ps samples are taken randomly from the input keys. s is called the oversampling rate. Samplesort uses oversampling to increase the quality of the splitters. An optimal set of splitters divides the input into p equally sized partitions. When choosing a high oversampling rate, there is a high probability that the imbalance is minimal [17, 24]. Blelloch et al. show that Samplesort performs better than Bitonic Sort and Radix Sort on large inputs [5]. Samplesort is not efficient for small inputs because $O(p)$ samples have to be sorted. Sibeyn proposes an optimized deterministic splitter selection. In theory, a Samplesort implementation using this splitter selection performs similar to implementations that use a randomized splitter selection [29].

3.1.3 Hypercube Quicksort

Hypercube Quicksort, also called Hyperquicksort [34] is an adaptation of Quicksort. The PEs are arranged in a k-dimensional hypercube. In each recursion, a pivot element is selected collectively on the current hypercube. Each PE partitions its local data according to the pivot. In the i-th recursion, each PE communicates with the partner in the i-th dimension. The PE with the lower rank sends all large elements to its partner. The PE with the higher rank sends all small elements to its partner. The hypercube is then split into two hypercubes with one dimension less. Each hypercube executes the algorithm independently. The number of PEs per groups is halved in every recursion. The two subgroups thus have exactly the same size. A bad pivot introduces a strong imbalance. Axtmann et al. have presented a robust variant of Hypercube Quicksort that sorts efficiently for all input distributions [2]. The data is reshuffled before the sorting begins to create a randomly distributed input. Reshuffling the data increase the robustness of the algorithm but requires additional communication.

3.1.4 Schizophrenic Quicksort

Schizophrenic Quicksort is based on Quicksort and is similar to Parallel Quicksort. Schizophrenic Quicksort was introduced in the lecture 'Parallele Algorithmen' from Prof. Dr. Peter Sanders [12, 25, 30]. Algorithm 1 shows pseudo code of the procedure. The input of the algorithm is a data structure called *Quicksort Interval*. A Quicksort Interval contains the global index of the first and last element that will be sorted in the call of Schizophrenic Quicksort. At first a pivot element v is selected collectively by the group. Then all PEs partition their local data with the pivot element into two partitions. A prefix sum and broadcast are performed over the number of small elements on the PEs. The result of the prefix sum on PE i is number of small elements on all PEs with a rank smaller or equal to i. The results of the collective operations is used to calculate where each PE needs to send its data. Then the data is exchanged in such a way that all PEs receive the same amount of data that they send. The number of local element on a PE is constant and equals $\frac{n}{p}$. Let L_i be the local data of PE i. The function Φ is defined as

$$\Phi(L_i, v) = \begin{cases} -1, & \text{if } \forall e \in L_i : e \leq v \\ 0, & \text{if } \exists e_1 \in L_i : e \leq v \ \wedge \exists e_2 \in L_i : e > v \\ 1, & \text{if } \forall e \in L_i : e > v \end{cases}$$

If $\Phi(L_i, v) = 0$, then the local data of PE i contain elements larger than the pivot and elements smaller than the pivot. The group is then split into two subgroups, one group that contains the

Algorithm 1: schizophrenicQuicksort

```
Data: i, j: Data Interval
if i and j in local data then
    localSort(i, j)
    return
if j - i < 10 then
    sequentialSort(i, j)
    return
v ← selectPivot(i, j)
partitionData(v, i, j)
x ← calculateDataAssignments(i, j)
exchangeData(x, i, j)
< (i, k), (l, j) > ← splitInterval(x, i, j)
if Φ(L_i, v) = 0 then
    In parallel do
        schizophrenicQuicksort(i, k)
        schizophrenicQuicksort(l, j)
else if Φ(L_i, v) = -1 then
    quicksort(i, k)
else
    quicksort(l, j)
```

small elements and one group that contains the large elements. If $\Phi(L_i, v) = 0$ then PE i is part of both subgroups. In this case we call PE i a *schizophrenic* PE. We then call Schizophrenic Quicksort recursively on both subgroups. A schizophrenic PE communicates simultaneously on both groups in the next recursion. If Schizophrenic Quicksort is called with less than 10 elements, a sequential algorithm is called. If Schizophrenic Quicksort is called on a group that contains only one PE, the data is sorted locally. The algorithm guarantees that each PE can only work in two groups at most. Figure 3.1 illustrates the possible ways the two groups of a schizophrenic PE can be split. The figure shows that the PE is at most in two groups at the start of the next recursion. As mentioned before, the number of elements on each PE is constant throughout the recursions of Schizophrenic Quicksort. Because the input is distributed evenly, the output is also distributed evenly. All PEs thus sort the same number of elements locally. This increases the robustness compared to Parallel Quicksort.

The student Tobias Heuer has implemented Schizophrenic Quicksort as a practical part of a lecture [12]. The implementation was only tested on one machine with up to 48 cores. The implementation has several problems that make it impractical for massive number of PEs on a distributed memory system. Firstly, in each recursion two new MPI communicators are created. Creating these MPI communicators is not mandatory for Schizophrenic Quicksort but severely impacts the running time. Secondly, the implementation uses blocking collective operations. When using blocking collective operation, a schizophrenic PEs can not communicate on two groups simultaneously. Instead, the schizophrenic PE communicates sequentially on the two groups.

3.2 Collective Communication

A collective operation is an operation that involves communication via message-passing between multiple PEs. Collective operations are often used in parallel algorithms if data has to be exchanged between a group of PEs. Each collective operation is defined by an initial state or input and a final state or output. The (point-to-point) communication used to achieve the output are not defined and may vary between different implementations.

Hoefler et al. [16] show advantages of non-blocking compared to blocking collective operations. Non-blocking collective operations generate more overhead than blocking collective operations. By using non-blocking collective operations, up to 90% of the idle CPU time can be freed and used for computations while the collective operation is performed. If an algorithm can do computations while waiting for the completion of the collective operation, the running time of the algorithm can be reduced by using non-blocking instead of blocking collective operations. Hoefler et al. [15] also propose syntax for non-blocking collective operations for MPI.

Chan et al. [6] give lower bounds for the communication and computation time for multiple collective operations. Algorithms based on a minimum-spanning tree (MST) communication patters are commonly used to implement collective operations. MST algorithms are optimal for collective operations Scatter and Gather. MST algorithms are also optimal in α for the collective operations Broadcast and Reduce. For small input sizes, MST algorithms are optimal in practice.

Vadhiyar et al. [33] show different algorithms for implementing a broadcast. A binomial tree based algorithm performed the best overall, excluding a hybrid implementation that combines multiple algorithms.

Thakur and Gropp [28] implemented collective operations by using multiple algorithms for each collective operation. The optimal algorithm is selected based on the message size. Their implementations outperformed previous implementation of MPI.

Blelloch [4] presents the *up-sweep/down-sweep* algorithm that executes the collective operation Scan. The implementation uses a binomial tree communication pattern.

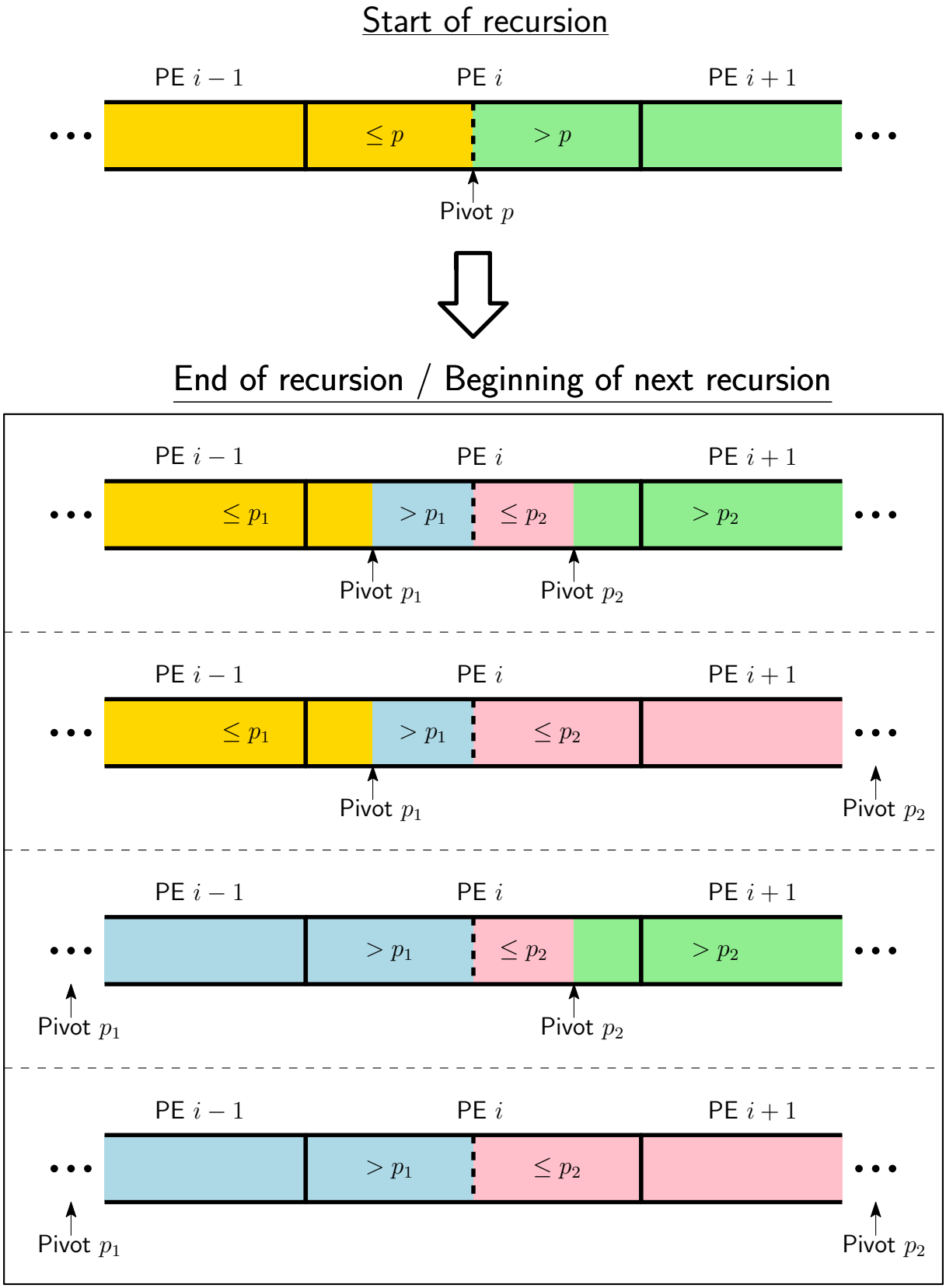

Figure 3.1: The figure depicts the four possible ways the two groups of a schizophrenic PE can be split in one level of recursion of Schizophrenic Quicksort. The pivot p_1 is the pivot in the left group, p_2 is the pivot in the right group. Each group is represented by one color.

4 Communicators and Collective Operations in Existing MPI Implementations

MPI is a standardized message-passing system. The MPI standard defines syntax and semantics of the library methods. Communication with MPI bases on MPI communicators, each communicator includes a group of PEs. When transmitting a message with point-to-point communication, the sending and receiving PE have to call the send and receive method with the same communicator. This allows the programmer to encapsulate communication in different communicators. Many commonly used MPI implementations use collective communication to create a new communicator [8]. The time to create a communicator thus scales in the number of PEs. Creating a communicator is significantly more expensive than performing a fast collective operation, for example a broadcast with input size of 1. In our tests, splitting a MPI communicator into two disjoint sub-communicators was more than a factor 100 slower than broadcasting one element to all PEs of this communicator. The time to create an MPI communicator is in $\omega(\log p)$ while the time to perform the operation Broadcast is in $O(\log p)$. The relative difference between the running time of the broadcast and the running time of the communicator creation therefore increases even further if we increase the number of PEs. See Section 8.2.2 for detailed results.

MPI provides method interfaces for various collective operations. A collective operation of MPI can only be executed on all PEs that are in the group of the MPI communicator, meaning all PEs have to call the same method.
If we want to execute a collective operation on a subgroup, we have to create a new communicator that includes only the PEs of the subgroup. Then we execute the collective operation in the new communicator. If an algorithm splits the group in each recursion and is then recursively executed on the subgroups, e.g. Parallel Quicksort, the algorithm uses each communicator only for a short time. We already mentioned that splitting a communicator is significantly slower than executing a fast collective operation on the same communicator. If an algorithm uses a communicator only for a few collective operations, it is possible that the cost of creating the communicator is higher than the cost of the collective operations. In this is the case, the running time of the communicator split dominates the communication time of the algorithm. We classify algorithms that split the group of PEs recursively into two categories. The first category of algorithms splits the groups independent of the input, e.g. Hypercube Quicksort. We create the communicators once in the beginning and then reuse them for multiple executions of the algorithm. We amortize the initial cost amortize by executing the algorithm multiple times. The second category of algorithms, e.g. Schizophrenic Quicksort, splits the groups non-deterministically into subgroups. We have to create the communicators dynamically, thus they can not be reused for multiple executions of the algorithm. To estimate the running time of such an algorithm, we also have to include the running time of the communicator split. Lower bounds of a theoretical algorithm might not be achievable in practice.

MPI provides blocking and non-blocking communication operations. Blocking operations block the program flow until the operation is completed. In contrast, non-blocking operations return the program flow immediately. While the non-blocking operations is executed, we can start additional communication or do local work. The completion of the non-blocking operation has to be explicitly checked with a test operation. Hoefler et al. show that up to 90% of idle CPU time can be freed when using non-blocking instead of blocking collective operations.[16]. Non-blocking collective operations also allow a PE to execute two collective operations simultaneously on two communicators.
Some, but not all MPI implementations support non-blocking collective operations. The IBM MPI compiler on the Juqueen [19] does not support non-blocking collectives. An implementation that uses non-blocking MPI collective operations is therefore not portable to all supercomputers.

5 Non-blocking Communication on Range Based Communicators

We present a library for communication on distributed memory systems based on message passing. The key feature of the library is that communicators are created in constant time without communication. The library provides point-to-point communication operations as well as a selection of collective operations. We provide blocking and non-blocking methods for all operations. Communication is based on *range based communicators*. Each range based communicator is based on a MPI communicator. A range based communicator includes an interval of the PEs that are connected by the MPI communicator. The included PEs are implicitly stored by the ranks (on the MPI communicator) of the first and last PE. If we talk about a MPI communicator in context of a range based communicator of our library, we mean the MPI communicator that the range based communicator is based on.

The library is based on MPI and we provide an interface that is very close to MPI. Because the interface is nearly identical to MPI for most operations, our library can easily be integrated to replace existing MPI code.

We implement point-to-point communication operations in the range base communicator by invoking their MPI equivalent in the MPI communicator. We adjust some parameters before calling the MPI operation. To execute a collective operation of MPI in a MPI communicator, all PEs in the MPI communicator have to call the collective operation. Because a range based communicator may include a subgroup of PEs, we have to reimplement all collective operations. We use the point-to-point communication of our library to implement collective operations. The algorithms that we use to perform the collective operations are all based on binomial tree communication patters [4, 33]. The communication patters are generic and not optimized for a specific network topology. Binomial tree based patterns are theoretically optimal for small input sizes [6]. Each communication operation requires a tag. Tags are used to distinguish between simultaneously executed communication.

Our library provides a blocking method and a non-blocking method for each communication operation. When we invoke a blocking method, we guarantee that the communication is completed when the method returns. When we invoke a non-blocking method, we do not guarantee that the communication is completed when the method returns. We have to invoke the method *`RBC::Test`* to check if the communication is completed. Each non-blocking method returns a request (*`RBC::Request`*). The request is given as a parameter to the method `RBC::Test`. `RBC::Test` returns true if the communication is completed and returns false if the communication is not completed. Our library also provides three additional operations to test non-blocking operations. The method `RBC::Wait` takes a request and blocks until the operation is completed. The method `RBC::Testall` takes an array of requests and returns true if all operations are completed. The method `RBC::Waitall` takes an array of requests and blocks until all operations are completed.

Each non-blocking method returns a request. To make progress, we have to call the method `RBC::Test` on the corresponding request (or one of the three other test methods). In Table 5.1

Collective Operation	Blocking Operation	Non-blocking Operation
Broadcast	`RBC::Bcast`	`RBC::Ibcast`
Reduce	`RBC::Reduce`	`RBC::Ireduce`
Scan	`RBC::Scan`	`RBC::Iscan`
Scan-and-Broadcast	`RBC::ScanAndBcast`	`RBC::IscanAndBcast`
Gather (same input size on each PE)	`RBC::Gather`	`RBC::Igather`
Gather (variable input size)	`RBC::Gatherv`	`RBC::Igatherv`
Gather (variable input size, merge the input)	`RBC::Gatherm`	`RBC::Igatherm`
Barrier	`RBC::Barrier`	`RBC::Ibarrier`

Table 5.1: Names of the operations that execute collective operations in our library

we give the names of the blocking and non-blocking methods for the supported collective operations.

We show example code how to use our library in Figure 5.1. The example executes the operation Broadcast on a subgroup of all PEs. Firstly, we create a communicator from the global MPI communicator. Then we create a second communicator that includes the ranks $[0, \frac{p}{2}]$. Then we execute the operation Broadcast in the new communicator.

We restrict the usage of tags if two communicators overlap on more than one PE, meaning multiple PEs are part of both communicators. To distinguish between operations in different communicators, all simultaneously executed communication operations have to use unique tags. If at most one PE is part of both communicators, the communication on both communicators does not interfere. In this case we do not restrict the usage of tags.

We will now describe the communicators of our library and how to create them. Then we describe the methods that are used for point-to-point communication. Lastly we will describe the collective operations that our library provides.

```
int size, v, root = 0, tag = 10;
RBC::Comm global_comm, new_comm;
RBC::Create_Comm_from_MPI(MPI_COMM_WORLD, &global_comm);
RBC::Comm_rank(global_comm, &rank);
RBC::Comm_size(global_comm, &size);
RBC::Comm_Create(global_comm, 0, size / 2, &new_comm);
if (rank <= size / 2) {
      if (rank == root)
            v = 42;
      RBC::Bcast(&v, 1, MPI_INT, root, tag, new_comm);
}
```

Figure 5.1: We create a subcommunicator including half of the PEs and then perform the operation Broadcast on the subcommunicator.

5.1 Range Based Communicators

A range based communicator is defined by a MPI communicator M and an interval $[m_k, m_l]$ of ranks. Let m_i be the rank of PE i in the MPI communicator. All PEs with rank m_j, $m_k \leq m_j \leq m_l$, are included in the range based communicator. The size of a range based communicator is equal to the number of PEs it contains. The size can be calculated with $m_l - m_k + 1$. We assign each PE in the communicator a unique rank. Let m_i be the rank of PE i in the MPI communicator. The rank in the range based communicator is defined as $r_i := m_i - m_k$. The MPI rank of a PE with rank r_i in the range based communicator can be calculated with $m_i = r_i + m_k$.

Creating a new range based communicator from an existing communicator (range based or MPI) has a constant running time and requires no communication. Let R be the existing range based communicator that is based on the MPI communicator M and includes the MPI ranks $[m_k, m_l]$. We want to create a new communicator R' that includes the ranks $[r_m, r_n]$ of the existing communicator. R' uses the same MPI communicator M. The interval of included MPI ranks $[m'_k, m'_l]$ can be calculated with $m'_k = m_k + r_m$ and $m'_l = m_k + r_n$. Creating the new communicator requires no communication and the local work does not depend on the size of the communicator.

5.2 Point-to-point Communication

In a point-to-point communication, a message is transmitted from a source PE to a destination PE. The source PE sends the message by invoking a send operation. The destination PE receives the message by invoking a receive operation. When we send a message, we have to specify the rank of the PE that should receive the message. To receive a message, we specify the source rank of the message. Instead of using a specific rank, we can also use a *wildcard*. When we use a wildcard, the receive operation will receive a message from any PE in the communicator. The third operation is the probe operation. The probe operation is similar to the receive operation. We specify a source rank or use the wildcard. The operation returns when a message from the source rank (or any rank if the wildcard was used) can be received. Instead of receiving the message, the probe operation returns a status object. The status object contains the source rank of the message. The status can be used to determine the size of a message. The probe operation is useful if we want to receive a message of unknown length. We can determine the size of the message and then start a receive operation that receive the message into a receive buffer of the correct size. Figure 5.2 gives the interfaces of the operations related to point-to-point communication. We now explain the methods in detail.

In this section, we use the rank in the Range communicator and the rank in the MPI communicator. Let r_i be the rank in the Range communicator and m_i be the rank in the underlying MPI communicator for a PE i. The ranks can be converted into each other like presented above. When we invoke an operation of MPI in our operations, we always use the underlying MPI communicator. If we invoke a MPI operation, we use the same tag that is the parameter of the operation of our library.

5.2.1 Send

The operation `RBC::Send` is a blocking operation that sends a message to a destination PE with rank r_i. We invoke the operation `MPI_Send`. Instead of r_i, we use the rank m_i as the destination

```
RBC::Send(const void *sendbuf, int count, MPI_Datatype datatype,
      int dest, int tag, RBC::Comm comm)
RBC::Isend(const void *sendbuf, int count, MPI_Datatype datatype,
      int dest, int tag, RBC::Comm comm, RBC::Request *request);
RBC::Recv(void *buffer, int count, MPI_Datatype datatype, int source,
      int tag, RBC::Comm comm);
RBC::Irecv(void *buffer, int count, MPI_Datatype datatype, int source,
      int tag, RBC::Comm comm, RBC::Request *request);
RBC::Probe(int source, int tag, RBC::Comm comm, MPI_Status *status);
RBC::Iprobe(int source, int tag, RBC::Comm comm, int *flag,
      MPI_Status *status);
```

Figure 5.2: Interface for the operations used in point-to-point communication

rank. All other parameters stay unchanged.

5.2.2 Isend

The operation `RBC::Isend` is a non-blocking operation that sends message to a destination PE with rank r_i. We invoke the operation `MPI_Isend`. Instead of r_i, we use the rank m_i as the destination rank. All other parameters stay unchanged. The operation `MPI_Isend` returns a MPI request. We store the MPI request in our request. When the test operation is invoked, we invoke `MPI_Test` on the stored MPI request. We return the return value of the operation `MPI_Test`.

5.2.3 Recv

The operation `RBC::Recv` is a blocking operation that receives a message from a source PE with rank r_i. The wildcard `MPI_ANY_SOURCE` can be used instead of a specific rank. If the wildcard is not used, we convert r_i to m_i. If the wildcard is used, we invoke the operation `RBC::Probe`. The operations `RBC::Probe` returns a status that contains the source rank of the message that is ready to be received. We define m_i as the source rank of the message. We invoke the operation `MPI_Recv` with the source rank m_i. All other parameters stay unchanged.

5.2.4 Irecv

The operation `RBC::Recv` is a blocking operation that receives a message from a source PE with rank r_i. The wildcard `MPI_ANY_SOURCE` can be used instead of a specific rank. If the wildcard is not used, we convert r_i to m_i. If the wildcard is used, the test method invokes the operation `RBC::Iprobe` to test if a message is ready to be received. If a message can be received, we define m_i as the source rank of the message. If no message is ready to be received, the test method returns. We invoke the operation `RBC::Iprobe` in the test method until a message is ready to receive. When we have defined m_i, we invoke the `MPI_Irecv` with the source rank m_i. All other parameters stay unchanged. The operation `MPI_Irecv` returns a MPI request. We store the MPI request in our request. When the test operation is invoked, we invoke `MPI_Test` on the stored MPI request. We return the return value of the operation `MPI_Test`.

5.2.5 Iprobe

The operation `RBC::Iprobe` is a non-blocking operation that tests if a message from a source PE with rank r_i is ready to be received. The wildcard `MPI_ANY_SOURCE` can be used instead of a specific rank. In contrast to the other non-blocking operations of our library, the operation `RBC::Iprobe` does not return a request. If the wildcard is not used, we invoke the operation `MPI_Iprobe` and return the return values of the operation. If the wildcard is used, we invoke the operation `MPI_Iprobe`. If a message is ready to be received, we test if the source PE of the message is in the Range communicator. We return true if the source PE is in the Range communicator, otherwise we return false. If the operation `RBC::Iprobe` returns true, it also writes the status that was returned by the operation `MPI_Iprobe` in the output parameter (`status`).

5.2.6 Probe

The operation `RBC::Iprobe` is a blocking operation that blocks until a message is ready to be received. We invoke the operation `RBC::Iprobe` with the same parameters. We call the operation `RBC::Iprobe` in a loop until the operation returns `true`. We return the status that was returned by the operation `RBC::Iprobe`.

5.3 Collective Operations

The communication of collective operations involves all PEs in the range communicator. This means that all PEs have to call the operation or the collective will not be executed completely. We implement several common collective operations. All implementations use binomial tree based communication patterns. We describe how we implement the non-blocking methods for all collective operations. When the blocking method is invoked, we invoke the non-blocking method and then use `RBC::Wait` to block until the operation is completed. For each collective operation, we implement a method `test`. When the method `RBC::Test` is invoked on a request, we invoke the corresponding method `test`. The algorithm for a collective operation is executed when the method `test` is invoked. We execute the algorithm until we have to communicate. We start all send or receive operations with non-blocking methods and then return from the method `test` with `false`. If the method `test` is called again, we invoke the operation `RBC::Testall` to check if all communication operations are completed. If not all operations are completed, we return with `false`. When all communication operations are completed, we continue with the execution of the algorithm until communication is required. We repeat the execution of the algorithm this way until the algorithm (and thus the collective operation) is completed. We then return with `true`. If the method `test` is invoked again, we always return `true`.

If we use the term *wait* in the following sections, that means that we wait for the completion of communication in a non-blocking way as described above. We describe the execution on a single PE. All PEs in the communicator invoke the same operation. If we use the terms *child node* and *parent nodes*, we mean the PE that is the child or parent of the PE in the binomial tree.

5.3.1 Broadcast

The operation `RBC::Ibcast` performs the operation Broadcast. We use a binomial tree communication pattern as shown in Figure 5.3a. The root of the binomial tree is the PE with rank 0. The input of the operation Broadcast is an arbitrary amount of data d on a PE with rank t. After

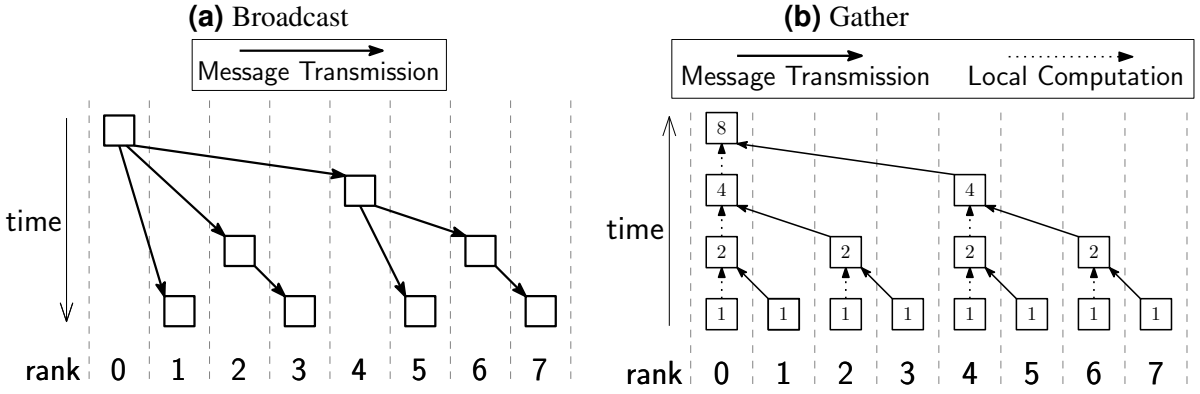

Figure 5.3: Communication patterns
Figure (b) shows the communication pattern for the operation `RBC::Igather`. The numbers in the squares are the numbers of local elements if the input of the operation is one element on each PE.

the operation Broadcast is completed, all PEs have a copy of d in the buffer. We shift the rank i of each PE in the communicator to the temporary rank $i' := i - t \mod p$. The reason we do this is to give the PE with rank t the temporary rank $t' = 0$, because we use this PE as the root of the binomial tree. We construct the binomial tree with the temporary ranks.

We now describe how the operation is executed when the method `test` is (repeatedly) invoked. Firstly, we invoke the operation `RBC::Irecv` on each PE to receive a message from the PE that is the parent node in the binomial tree. Then we wait until the message has been received. The root has d stored locally as the input of the operation. Thus, the root does not invoke the operation `RBC::Irecv`. When d has been received (or if the PE is the root), we send d it to other PEs according to the binomial tree. We invoke the operation `RBC::Isend` multiple times to send d to all PEs that are child nodes in the binomial tree. We first send to the PE with the highest rank i'. Then we wait until all send operations are completed. When d has been sent to all child nodes, the operation `RBC::Ibcast` is completed on this PE.

The binomial tree has a height of $O(\log p)$. Let m be the number of elements of d. The running time of the broadcast is in $O(\alpha \log p + \beta m \log p)$.

5.3.2 Gather

The local input for the operation Gather on PE i is an array a_i of size s_i. The operation Gather routes the content of the arrays to the root PE with rank t such that the output is an array $a_0, a_1, ..., a_{p-1}$. Our library provides three non-blocking operations to execute the operation Gather. The operation `RBC::Igather` executes the operation Gather with the condition that s_i is equal on all PEs. The operation `RBC::Igatherv` executes the operation Gather and allows s_i to be different on the PEs. The operation expects an array $s_0, s_1, ..., s_{p-1}$ on each PE that contains the size of the input on each PE. The operation also has another array $v_0, v_1, ..., v_{p-1}$ as input that allows to rearrange the output array. The content of array a_i is placed in the output array beginning by the index v_i. The operation `RBC::Igatherm` executes the operation Gather and allows s_i to be different on the PEs. The total size of the output array (the sum of all s_i) has to be specified. The operation expects a merge function $\odot$. The merge function is used to merge all input arrays into the output array. The output of the operation `RBC::Igatherm` is the array $a_0 \odot a_1 \odot ... \odot a_{p-1}$ [26].

We describe the implementation of the operation `RBC::Igatherm`. When the operations `RBC::Igather` or `RBC::Igatherv` are invoked, we invoke the operation `RBC::Igatherm` with changed parameters. The operation `RBC::Igatherm` expects the size of the output array and a merge function as additional parameters. If `RBC::Igather` is invoked, we compute the size of the output array as $s_i p$, because all s_i are equal. We pass a merge function that does not change the input arrays. If `RBC::Igather` is invoked, we compute the size of the output array as the sum over all s_i. We pass a merge function that does not change the input arrays. We use a buffer of the same size as the output array to receive the output of the operation `RBC::Igatherm`. When the operation `RBC::Igatherm` is completed, we copy the array from the buffer to the output array of the operation `RBC::Igatherv` according to the array $v_0, v_1, ..., v_{p-1}$.

We use a binomial tree communication pattern as shown in Figure 5.3b. The root of the binomial tree is the PE with rank 0. We shift the rank i of each PE in the communicator to the temporary rank $i' := i - t \mod p$. The reason we do this is to give the PE with rank t the temporary rank $t' = 0$, because we use this PE as the root of the binomial tree. We construct the binomial tree with the temporary ranks.

Firstly, we invoke the operation `RBC::Irecv` to receive an array from the child node with the lowest rank i'. We wait until the array has been received. Then we merge the received array with the local array by using the merge function $\odot$. We then invoke the operation `RBC::Irecv` to receive an array from the child node with the lowest rank i' from which we have not yet received an array. All received arrays are merged into the local array by using $\odot$ We repeat this until we have received an array from each child node. Then we invoke the operation `RBC::Isend` to send the local array to the parent node. When the send operation is completed, the operation `RBC::Igatherm` is completed. Because the root has no parent node, we do not send to any PE from the root. The operation `RBC::Igatherm` is completed on the root when all arrays from the child nodes have been received and merged.

The binomial tree has a height of $O(\log p)$. Let M be the total number of elements. If the elements are distributed evenly, the running time of the algorithm is in $O(\alpha \log p + \beta M)$. If all elements are one the last PE, the running time is in $O(\alpha \log p + \beta M \log p)$.

5.3.3 Reduce

The operation `RBC::Ireduce` performs the operation Reduce. The input of the operation Reduce on PE i is a vector a_i and a binary operator $\oplus$. The operation Reduce routes the content of the vectors to the root PE with rank t such that the output is a vector $a_0 \oplus a_1 \oplus ... \oplus a_{p-1}$. We call using $\oplus$ to combine two values u and v to a single value $u \oplus v$ a *reduction* (we *reduce* two values). Let $u = [u_0, u_1, \ldots, u_{m-1}]$ and $v = [v_0, v_1, \ldots, v_{m-1}]$ be two vectors of size m. The operator $\oplus$ is used componentwise to reduce two vectors. The result of the reduction $u \oplus v$ is defined as $[u_0 \oplus v_0, u_1 \oplus v_1, \ldots, u_{m-1} \oplus v_{m-1}]$.

We use a binomial tree communication patter as shown in Figure 5.4a. The root of the binomial tree is the PE with rank $p - 1$. We shift the rank i of each PE in the communicator to the temporary rank $i' := i - t - 1 \mod p$. The reason we do this is to give the PE with rank t the temporary rank $t' = p-1$, because we use this PE as the root of the binomial tree. We construct the binomial tree with the temporary ranks.

Firstly, we invoke the operation `RBC::Irecv` multiple times to receive a vector from each child node. We wait until all vectors have been received. Then we reduce all received vectors and the local vector with the operation $\oplus$. The result of the reduction is the new local vector. If the PE has no child nodes, we do not invoke the operation `RBC::Irecv`. When we have received

a vector from each child node, we invoke the operation `RBC::Isend` to send the local vector to the parent node. When the operation `RBC::Isend` is completed, the operation `RBC::Ireduce` is completed. Because the root has no parent node, we do not send to any PE from the root. The operation `RBC::Ireduce` is completed on the root when all vectors from the child nodes have been received and reduced.

The binomial tree has a height of $O(\log p)$. When the input of the operation are vectors of size m, each message has length m. The running time of the algorithm is in $O(\alpha \log p + \beta m \log p)$

5.3.4 Scan

The operation `RBC::Iscan` performs the operation Scan. The input of the operation Scan on PE i is a vector a_i and a binary operator $\oplus$. The operation Scan routes the content of the vectors such that the output on PE i is the vector $a_0 \oplus a_1 \oplus ... \oplus a_i$. We call using $\oplus$ to combine two values u and v to a single value $u \oplus v$ a *reduction* (we *reduce* two values). Let $u = [u_0, u_1, \ldots, u_{m-1}]$ and $v = [v_0, v_1, \ldots, v_{m-1}]$ be two vectors of size m. The operator $\oplus$ is used componentwise to reduce two vectors. The result of the reduction $u \oplus v$ is defined as $[u_0 \oplus v_0, u_1 \oplus v_1, \ldots, u_{m-1} \oplus v_{m-1}]$.

We use the up-sweep/down-sweep algorithm [4]. This algorithm uses communication patterns based on a binomial tree. The last PE, the PE with the highest rank, is the root of the tree. The algorithm consists of two phases, the *up-sweep* and the *down-sweep* phase. In the up-sweep phase, a reduction from the leaves to the root is performed as shown in Figure 5.4a. Firstly, we invoke the operation `RBC::Irecv` multiple times to receive a vector from each child node. We wait until all vectors have been received. Then we reduce all received vectors and the local vector with the operation $\oplus$. The result of the reduction is the new local vector. If the PE has no child nodes, we do not invoke the operation `RBC::Irecv`. When we have received a vector from each child node, we invoke the operation `RBC::Isend` to send the local vector to the parent node and wait for its completion. When the operation `RBC::Isend` is completed, the up-sweep phase is completed. Because the root has no parent node, we do not send to any PE from the root. The up-sweep phase is completed on the root when all vectors from the child nodes have been received and reduced. Before the down-sweep phase begins, the vector on the root is set to 0. All other PEs retain the local vector from the up-sweep phase. In the down-sweep phase, the communication is done from the root to the leaves as shown in Figure 5.4b. We use the same binomial tree as in the up-sweep phase. The algorithm proceeds from the highest to the lowest level. Firstly, we invoke the operation `RBC::Irecv` to receive a vector from the parent node and we also invoke the operation `RBC::Isend` to send the local vector to the parent node. We wait until both operation are completed. Then we replace the local vector with the received vector. We then communicate with the child nodes. We first communicate with the child node with the highest rank. We invoke the operation `RBC::Irecv` to receive a vector from the child node and we also invoke the operation `RBC::Isend` to send the local vector to the child node. We wait until both operation are completed. Then we reduce the received vector and the local vector with $\oplus$. The result of the reduction is the new local vector. We repeatedly communicate like described above with all child nodes, from the child node with the highest rank to the child node with the lowest rank. The operation is completed when we have communicated with all child nodes.

The binomial tree has a height of $O(\log p)$. The messages exchanged in the up-sweep and the down-sweep phase have the length m. The running time of the algorithm is in $O(\alpha \log p + \beta m \log p)$.

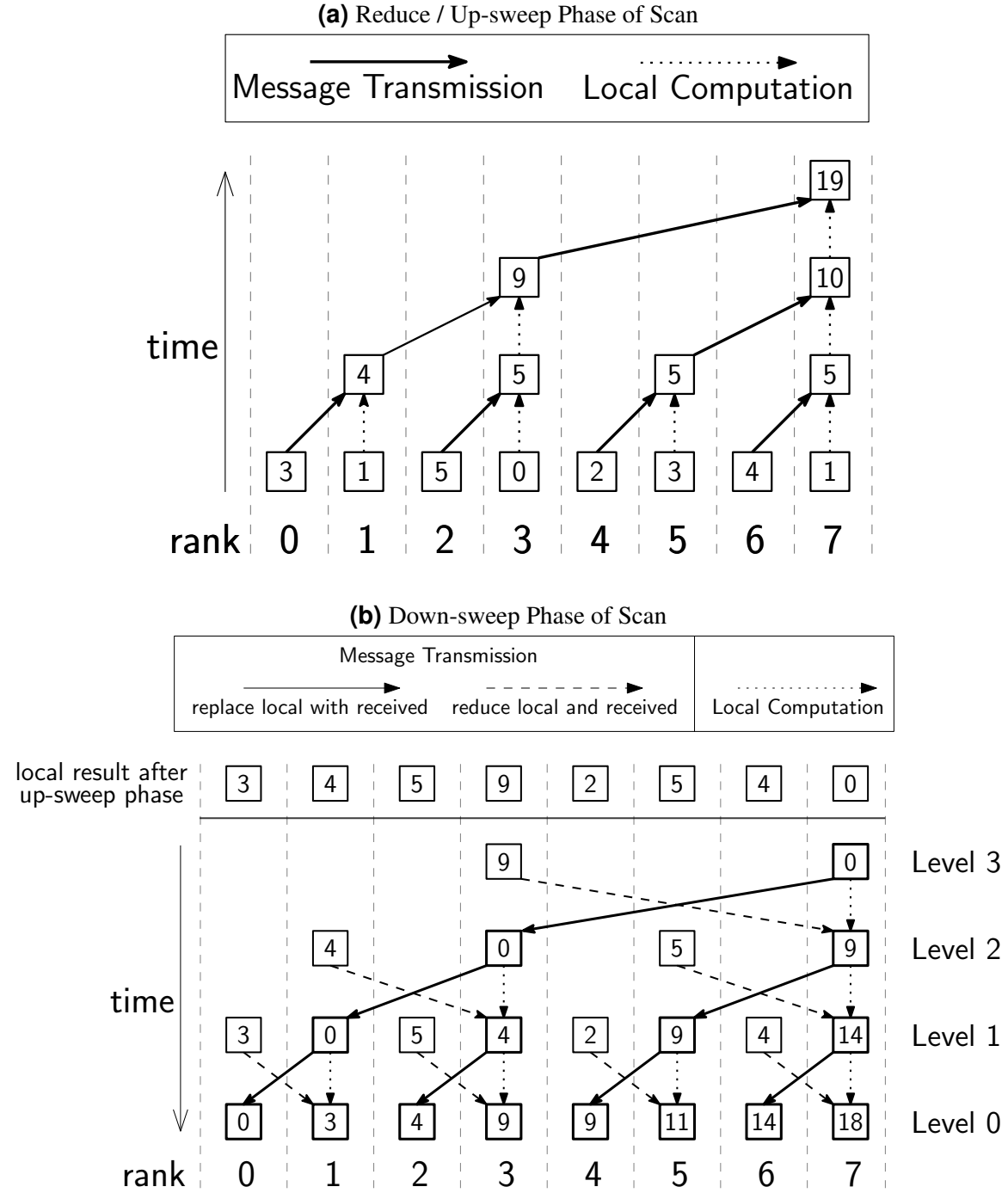

Figure 5.4: Figure (a) shows the communication of the operation `RBC::Reduce` and the communication in the up-sweep phase of the operations `RBC::Scan` and `RBC::ScanAndBcast`. Figure (b) depicts the communication in the down-sweep phase of the algorithm of the operation `RBC::Scan`. The numbers in the squares show the local values for an example input with vector size 1.

5.3.5 Scan-and-Broadcast

We define the collective operation *Scan-and-Broadcast*. The input on each PE is a vector d_i with length m and a binary associative operator $\oplus$. The operation has two output vectors s and b. The result of the operation Scan-and-Broadcast on PE i is $s = d_0 \oplus ... \oplus d_i$ and $b = d_0 \oplus ... \oplus d_{p-1}$. The operation `RBC::IscanAndBcast` performs the operation Scan-and-Broadcast. We modify the up-sweep/down-sweep algorithm to integrate the operation Broadcast. The algorithm is identical to the algorithm for the operation `RBC::Iscan` for the most part. The up-sweep phase is exactly the same as shown in Figure 5.4a. Before we set the vector on the root to 0, we copy that vector in the output buffer for the operation Broadcast. After the up-sweep phase, the vector on the root is equal to b. We integrate the operation Broadcast into the down-sweep phase so that no additional messages are required. Figure 5.5 shows the communication pattern in the down-sweep phase. In the down-sweep phase of the operation `RBC::Iscan`, we send a message containing the local vector l to all child nodes. In the operation `RBC::IscanAndBcast`, we instead send a message that contains l and also b. When the child node has received the message, the child node replaces its local vector with the received vector l. The child node also replaces the vector in the output buffer for the operation Broadcast with the received vector b.

The binomial tree has a height of $O(\log p)$. When the inputs are vectors of size m, the messages have a length of m in the up-sweep phase and a length of $2m$ in the down-sweep phase. The running time of the algorithm is in $O(\alpha \log p + \beta m \log p)$.

5.3.6 Barrier

The operation `RBC::Ibarrier` performs the operation Barrier. The operation Barriers notifies all PEs when all PEs have invoked the operation. We implement the operation `RBC::Ibarrier` by invoking the operation `RBC::IscanAndBroadcast` with a dummy element. The operation `RBC::Ibarrier` is completed when the operation `RBC::IscanAndBroadcast` is completed.

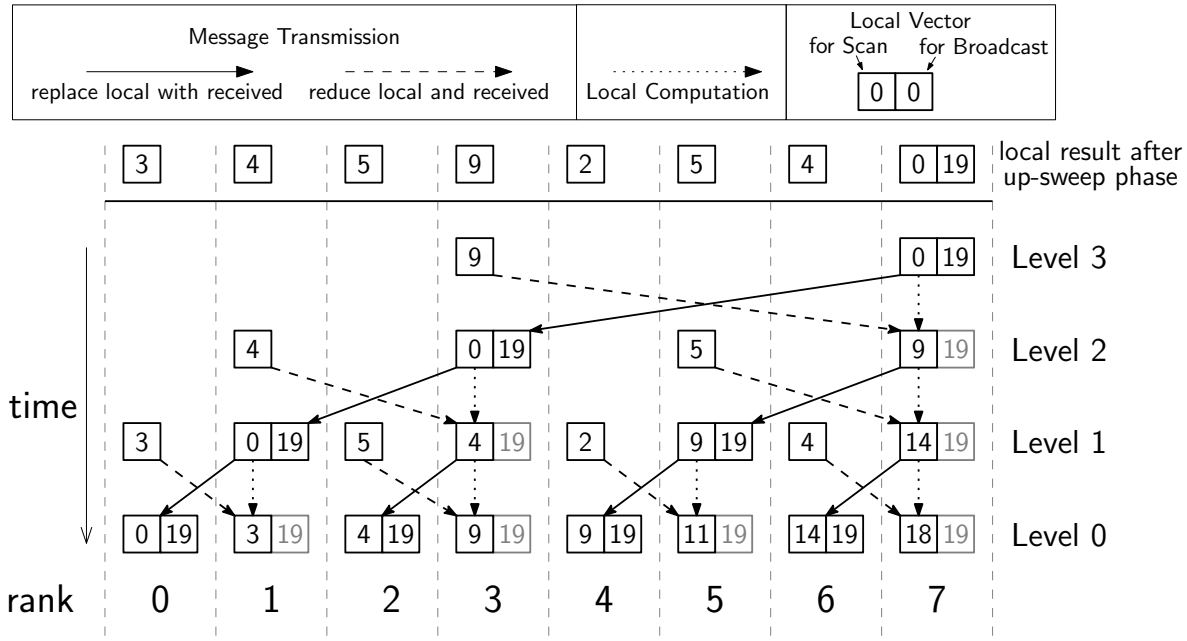

Figure 5.5: down-sweep phase for Scan and Broadcast

6 Efficient Schizophrenic Quicksort

In this section we present Efficient Schizophrenic Quicksort (*ESQ*). Algorithm 2 gives pseudocode. We now describe ESQ step by step. Firstly, we collectively select a pivot element. We pick a number of sample elements randomly from the input. We use the collective operation Gather to collect all samples to the PE with rank 0 in the current group. Each sample is defined as a tupel (x, z) where x is the value of the element and z is the global position in the input. We use lexicographic ordering to make the pivot selection robust against duplicate elements. The pivot is selected as the median of all samples. We use the operation Broadcast to send the pivot to all PEs. Then each PE partitions its local data according to the pivot element. Next we perform the collective operation Scan with the number of local small elements on each PE as the input and the collective operation Broadcast to send the total number of small elements to all PEs. With the result of both collective operations, we calculate where each PE needs to send its local data. Then we exchange the data so that all small elements are in one subgroup and all large elements are in another subgroup. Finally, we create new communicators for the two subgroups and invoke ESQ recursively on both subgroups. If ESQ is called on a group with one or two PEs, we break the recursion and execute a sequential algorithm. A schizophrenic PE is part of two groups. We use non-blocking communication operations to perform communication on both groups simultaneously.
We extract 6 subalgorithms that differ from the algorithm Schizophrenic Quicksort as described in Section3.1.4 which we call *SchizoQS*. Our implementation of the subalgorithms improves the performance of ESQ compared to SchizoQS (see our evaluation in Section 8.3). The subalgorithms are:

- **Base Cases**: If the current call of Schizophrenic Quicksort is a Base Case then use a sequential algorithm to sort the input and afterwards terminate Schizophrenic Quicksort.
- **Pivot Selection**: Select a pivot element and send it to all PEs in the group.
- **Partitioning**: Partition the local data according to the pivot element in small and large elements.
- **Exchange Calculation**: Calculate where each PE has to send its data.
- **Data Exchange**: Exchange the data.
- **Communicator Creation**: Create communicators for the two subgroups that contain the small and large elements.

We now describe how we improve SchizoQS by using our communication library instead of MPI. Then we describe the subalgorithms in detail and compare them to their counterparts in SchizoQS.

6.1 Integration of the Range library

SchizoQS uses MPI to perform communication. In Section 4 we explained restrictions of MPI. When splitting the group into two subgroups, the MPI communicator has to be split as well. Splitting the MPI communicator is expensive for large groups and significantly increases the

Algorithm 2: efficientSchizophrenicQuicksort

```
Data: i, j: Data Interval
1 if BaseCase(i, j) then
2     return
3 v ← selectPivot(i, j)
4 partition(v, i, j)
5 x ← calculateDataAssignments(i, j)
6 exchangeData(x, i, j)
7 < (i, k), (l, j) > ← splitInterval(x, i, j)
8 callRecursively(i, k, l, j)
```

running time of Schizophrenic Quicksort for small input sizes. We use our the communication library presented in Chapter 5 to reduce the cost of the communicator split. SchizoQS uses blocking collective operations, which can lead to cascade like execution of multiple groups. See Section 8.3.1 for a more detailed explanation. We prevent these cascades by using the non-blocking collective operations provided by our library.

6.2 Base Cases

We stop executing ESQ recursively when *Base Case* is reached. A Base Case is a call of Schizophrenic Quicksort that fulfills a certain condition. We use two different Base Cases:

- Base Case 1: The group of the current recursion consist of one PE
- Base Case 2: The group of the current recursion consist of two PEs

We use one sequential algorithm for each Base Case to sort the remaining unsorted elements. When the first Base Case is reached, we know that all remaining elements are stored locally on one PE. We sort the elements locally. If the second Base Case is reached, we know that the remaining elements are distributed on two PEs. The sequential algorithm has three steps. Firstly, each PE sends all its data to the other PE. Both PEs have all remaining elements stored locally. Then we partition the data such that the number of small elements equals the number of local elements that have to be stored on the left PE. The number of large element is then equal to the number of local elements that have to be stored on the right PE. To achieve such a partitioning, we use Quickselect [13]. Quickselect takes an unordered list U and an index k. The output of Quickselect is a permutation of U such that the element v with index k in U is in the same place it would be if the list was ordered. Additionally, all elements on the left of v are smaller or equal to v and all elements on the right of v are greater or equal than v. We choose U as the remaining elements and k as the number of small elements. Quickselect outputs a partitioning with the desired characteristics. Because both PE have all remaining data, the partitions are identical on both PEs. Lastly, the left PE sorts the small elements locally and the right PE sorts the large elements locally. We discard the unsorted data on each PE. When a Base Case is reached, we store the call of Schizophrenic Quicksort in a list. After all recursive calls on a PE have ended, the list is processed. At first any instances of the second Base Case are solved. Then the instances of the first Base Case are solved. The reason for this approach is that sorting the elements in a Base Cases is expensive. If we solve a Base Case immediately, the PE is blocked until the elements are sorted. Schizophrenic PEs do not communicate on the other group in this time. It is better to first continue the recursion and solve the Base Cases at the end.

The condition for the Base Cases have a direct impact on the depth of recursion of Schizophrenic Quicksort. In the following, we assume that the pivot element is always the median of all elements. When the pivot element is the median, then two subgroups for the small and large elements have the same size. The number of elements is then halved in each recursive call of Schizophrenic Quicksort. If we would recursively call Schizophrenic Quicksort until each call has only 1 element, the depth of recursion would be $\log_2 n$. We guarantee that a call of Schizophrenic Quicksort with $\frac{n}{p}$ or fewer elements is a Base Case. If we execute Schizophrenic Quicksort on a group of 3 or more PEs, the minimal number of elements is $\frac{n}{p} + 2$ (1 element one the first PE, $\frac{n}{p}$ elements on the second PE and 1 element on the third PE). The recursion depth with optimal splitting is therefore $\log_2 n - \log_2 \frac{n}{p} = \log_2 p \in O(\log p)$.

SchizoQS has two Base Cases. The first Base Case is identical to the first Base Case of our implementation. The second Base Case is reached if the input size of the call to Schizophrenic Quicksort is less than 10 elements. If the second Base Case is reached, the remaining elements are stored on two or more PEs. The depth of recursion with good splitting is $\log n - \log 10 \in O(\log n)$. For $n > p$, the depth of recursion of SchizoQS is higher than the depth of recursion of our implementation. Each recursion increases the running time of Schizophrenic Quicksort. Our implementation of the Base Cases thus reduces the running time compared to SchizoQS.

6.3 Pivot Selection

Schizophrenic Quicksort is executed recursively until a Base Case is reached. Each level of recursion of Schizophrenic Quicksort involves local work and communication. When we reduce the depth of recursion, we therefore also reduce the total running time of Schizophrenic Quicksort. Schizophrenic Quicksort splits the problem into two subproblems in each recursion. The lowest recursion depth is achieved if the two subproblems have the same size. The elements are partitioned into two partitions of the same size if the pivot is the median of the elements. Our goal is to find a pivot element that is as near to the median as possible.

Our implementation selects k random samples from the data and broadcasts the median of the samples as the pivot element. The Pivot Selection consists of five steps. Firstly, we distribute the number of samples on all PEs in the group. We adopt the sampling algorithm from Sanders et al. [26] to distributes the number of samples. The algorithm follows a binomial tree from the root to the leaves. At each node, the number of samples is distributed to the left and right subtree. We use a binomial distribution to split the samples according to the number of elements in each subtree. After the algorithm reaches the leaves of the binomial tree, we know how much samples each PE has to pick. Then we select the samples on each PE. We use a uniform distribution to pick the samples from the local data with replacement. Then we use the operation `RBC::Gatherm` to collect all samples at PE 0. The output of the operation is a ordered list of all samples. Then, we select the median of the samples as the pivot element. Finally, the pivot element is broadcasted to all PEs.

We want to find the median of the elements, so we call an element that is near the median a good pivot. When we increase the number of samples k, we expect the pivot selection to return a better pivot. The operation to gather the samples has a running time of $O(\log p)\alpha + O(k)\beta$. Increasing k increases the communication volume and the time needed to pick the samples. There exists a tradeoff between the cost of finding the pivot and the reduced cost because of fewer recursions. We try to find a k as high as possible to get a good pivot. At the same time, the effort for the pivot selection should not be asymptotically worse than the cost of the other operations in the recursion. We choose the number of samples as $k = \max(k_1 \log p, k_2 \frac{n}{p}, k_3)$.

$k_1 \log p$ increases the number of samples for a large p. The broadcast of the pivot has a cost of $\Omega(\log p)\beta$ so the cost can be justified. $k_2\frac{n}{p}$ increases the number of samples for large input sizes. The cost can be afforded because we exchange $O(\frac{n}{p})$ elements each recursion. k_3 is the lower bound of the number of samples for small n and p. The parameters k_1, k_2 and k_3 can be tuned to find the optimal tradeoff for the pivot selection. We achieved good experimental results with $k_1 = 16, k_2 = \frac{1}{50}, k_3 = 9$. The algorithm to distribute the number of samples originally uses communication along a binomial tree to accumulate the number of elements and then distribute the number of samples. Since the number of elements on each PE is constant, we omit the communication and calculate the number of samples locally. We synchronize the random generators on all PEs so that all PEs in a group calculate the same distribution of samples.

In SchizoQS, each PE takes c random samples and calculates their arithmetic mean. The mean values are gathered to PE 0 and the pivot is selected as the mean of the gathered values. The problem with this method is that the mean might not be defined for certain data types. SchizoQS can not sort all data types while our implementation works with all data types that can be sorted.

Tie-Breaking

We adapt the scheme from Axtmann et al.[1] to implement tie-breaking. In the scheme, each element x is identified by the triple (x, y, z) where y is the PE number that stores the element and z the position in the data. Using lexicographic ordering, each triple is unique. We simplify the triple to the tuple (x, g), where x is a data element and g the position in the global data (*global index*). The global index g can be calculated with $g = y\frac{n}{p} + z$. The number of elements $\frac{n}{p}$ on each PE is constant throughout the whole execution of Schizophrenic Quicksort. We can calculate the PE y from a global index g with $y = g/\frac{n}{p}$. The tuples are not explicitly stored and are only used in the pivot selection. The pivot element is a tuple (v, g_v).

Without any sort of tie-breaking, Schizophrenic Quicksort might not terminate for some input instances. Assume a group that contains only elements with the same value. If no tie-breaking is implemented, all elements are partitioned into the small elements. The next recursion thus contains the same elements. Because all elements are identical, the elements will be partitioned the same way as before, thus the recursion will never end. Another reason for an effective tie-breaking is that we want to achieve an optimal depth of recursion. Suppose the pivot is selected optimally, meaning it is the median of all elements. If some elements are identical to the pivot, the elements are not split exactly in the middle. When the data is not split in the middle, the recursion depth can be higher than the optimal depth. By using tie-breaking, we can guarantee the optimal recursion depth if the pivot is exactly the median of all elements.

SchizoQS uses a different approach to implement tie-breaking. SchizoQS performs the operation Reduce twice, one with the minimum operator and once with the maximum operator. If both result values are equal, then all elements are equal and Schizophrenic Quicksort terminates. This approach does not solve the problem that the recursion depth is most likely higher than the optimal depth if many duplicate elements exist. Our tie-breaking approach involves no communication operations.

6.4 Partitioning

We partition the data according to the pivot tuple (v, g_v). We do not create tuples for all elements and then compare them with the pivot tuple. Instead, we look at the global index g_v of the

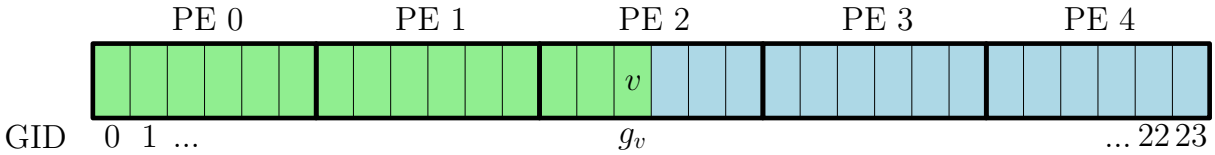

Figure 6.1: We divide the elements depending on if the global index is smaller (green) or larger (blue) than g_v. From the green elements, all elements with $x \leq v$ are small elements. From the blue elements, all elements with $x < v$ are small elements.

pivot element and calculate the PE y where the pivot element is stored with $y = g_v / \frac{n}{p}$. We compare each element to the pivot to decide if the element is partitioned into the small elements. Depending on the rank i of a PE, we use three different approaches to partition the local data. Figure 6.1 shows the approaches schematically for a group of size 5. If $i < y$, then all local elements have a global index smaller than g_v. An element e is partitioned into the small elements if $e \leq v$. If $i > y$, then all local elements have a global index greater than g_v. An element e is partitioned into the small elements if $e < v$. If $i = y$, then the pivot element has the local index $l_v := g_v \mod \frac{n}{p}$. We split the local elements into two parts at the local index l_v. We compare the elements of the left part with $e \leq v$ and the elements of the right part with $e < v$. After we partitioned both parts, we combine both small partitions and both large partitions to a single partition each.

6.5 Exchange Calculation

We perform the collective operation Scan with the sum operator and the number of local small elements as the input on each PE. The result of the operation on the PE with rank i is the number of small elements on all PEs with a rank smaller than i. We also perform the operation Broadcast with the total number of small elements on all PEs as the input. We use the operation `RBC::ScanAndBcast` from our communication library to simultaneously perform both collective operations.

We now describe how each PE computes where it has to send its local data. Let p_s be the result of the operation Scan, t_s be the result of the Broadcast, g_f be the global index of the first element in the current call of ESQ and g_l be the local index of the last element in the current call of ESQ. Let s be the global index of the first local small element before the data exchange and s' the index after the exchange, l and l' the index of the first local large element before and after the exchange. We calculate the new indexes with $s' = g_f + p_s$ and $l' = g_f + (z - t_s) + (s - g_f - p_s)$. Then we calculate the rank r_s of the PE where the element with the global index s' is stored on. Let l_s be the number of local small elements. If $\frac{n}{p} - (s' \mod \frac{n}{p}) \leq l_s$, we send all elements to PE r_s. If not, we send $\frac{n}{p} - (s' \mod \frac{n}{p})$ elements to PE r_s and the remaining elements to the PE with rank $r_s + 1$. We use the same procedure for the large elements.

SchizoQS performs the operation Scan twice, once for the number of small elements and once for the number of large elements. Afterwards, the total number of small and large elements are sent to all PEs by using the operation Broadcast twice. Our implementation saves the communication cost for executing the operation Scan once and execution the operation Broadcast twice.

6.6 Data Exchange

Each PE sends its small data to the left subgroup and its large data to the right subgroup. Both the small and large elements have to be either send to one PE or two PEs. We use the non-blocking operation `RBC::Isend` to send the elements to the PEs as calculated previously. Then we repeatedly invoke the operation `RBC::Iprobe` to test if a message is ready to be received. We receive the message with the operation `RBC::Irecv` if possible. We repeat this procedure until we have received $\frac{n}{p}$ elements.

SchizoQS uses the collective operation All-to-all. We do not use any collective operations and thus save communication cost.

6.7 Communicator Creation

When we have exchanged the data, we split the group of PEs into two subgroups. The left subgroup contains all PEs that have small elements in their local data. The right subgroup contains all PEs that have large elements in their local data. We create one communicator for each subgroup. Because we use our Range library, we can create the communicators in constant time. Then we call Schizophrenic Quicksort recursively on both groups.

SchizoQS uses MPI communicators. The running time for creating the communicators is in $\omega(\log p)$.

6.8 Time Complexity

We now give an explanation of the time complexity of ESQ. Table 6.1 shows the asymptotic running times of the phases of Schizophrenic Quicksort. Let ρ be the group size in the current recursion. We now describe the running time of the phases of ESQ in detail.

Each PE (except first and last in the group) has the same number of local elements. In the Pivot Selection, we use a binomial distribution to determine the samples, so the samples should be distributed nearly evenly. Gathering k elements to rank 0 costs $O(\alpha \log \rho + \beta k)$ time. We choose k such that βk is negligible compared to the total time of the level of recursion. The cost to gather the samples is thus $O(\alpha \log \rho + \beta)$. Broadcasting the pivot costs $O(\alpha \log \rho + \beta \log \rho)$. The total cost for the pivot selection is $O(\alpha \log \rho + \beta \log \rho)$. In the Partitioning, Each PE partitions $\frac{n}{p}$

Description	α	β	**Local Work**
Pivot Selection	$\log \rho$	1	1
Partition	-	-	$\frac{n}{p}$
Calculate Exchange	$\log \rho$	1	1
Data Exchange	1	$\frac{n}{p}$	1
Recursive Call	-	-	1
One Recursion	$\log \rho$	$\frac{n}{p}$	$\frac{n}{p}$
Base Case	1	$\frac{n}{p}$	$\frac{n}{p} \log \frac{n}{p}$
Total time	$\log^2 p$	$\frac{n}{p} \log p$	$\frac{n}{p} \log n$

Table 6.1: Runtime, implicit $O()$

local elements in time $O(\frac{n}{p})$. In the Exchange Calculation, the operation `RBC::IscanAndBcast` is performed with a vector size of 1 element. Because α is much larger than β, the time to send 1 element is negligible. The running time is in $O(\alpha \log \rho + \beta \log \rho)$. In the Data Exchange, each PE sends at most 4 messages with $\frac{n}{p}$ total elements, which costs $O(\alpha + \beta \frac{n}{p})$. Creating new communicators for the subgroups has a constant running time. Base Case 1 sorts $O(\frac{n}{p})$ elements which costs $O(\frac{n}{p} \log \frac{n}{p})$ time. In Base Case 2, one message is sent and received containing $O(\frac{n}{p})$ elements in $O(\alpha + \beta \frac{n}{p})$ time. The elements are split using Quickselect in $O(\frac{n}{p})$ time and then sorted in $O(\frac{n}{p} \log \frac{n}{p})$ time. The total running time of the Base Cases is thus $O(\alpha + \beta \frac{n}{p} + \frac{n}{p} \log \frac{n}{p})$. We assume that the selected pivot is of good quality. For good pivots, the algorithm has $O(\log p)$ recursions. Schizophrenic Quicksort executes $O(\log p)$ recursions and the Base Cases. Then the total running time of Schizophrenic Quicksort is

$$\begin{aligned} T &= O\left(\Sigma_{i=0}^{\log p}(\alpha \log p + \beta \frac{n}{p} + \frac{n}{p}) + (\alpha + \beta \frac{n}{p} + \frac{n}{p} \log \frac{n}{p})\right) \\ &= O(\alpha \log^2 p + \beta \frac{n}{p} \log p + \frac{n}{p} \log n) \end{aligned}$$

7 Implementation Details

We give implementation details about the communication library. We also give details about the implementation of Schizophrenic Quicksort and how we integrated the communication library. We implemented both our communication library and Schizophrenic Quicksort using C++11 and MPI.

7.1 Range-Based Communicators (RBC) Library

The class `RBC` contains all operations for the point-to-point communication, for the collective operations and to create communicators as static member functions. The class also contains the classes for the communicators (`RBC::Comm`) and requests (`RBC::Request`). The communicator `RBC::Comm` is a struct that contains a MPI communicator as well as the indexes of the first and last rank that is included in the Range communicator.

The interface of our library is very similar to MPI. One difference is that each collective operation of our library has a tag as an input parameter. We use `RBC::Comm` instead of `MPI_Comm` and `RBC::Request` instead of `MPI_Request`. Our library provides a blocking and a non-blocking function for each collective operation. We implement the blocking function by invoking the non-blocking function and then invoking the operation `RBC::Wait` to wait until the collective operation is completed. For example, the function `RBC::Bcast` is implemented as follows:

```
void RBC::Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
      int tag, RBC::Comm comm) {
  RBC::Request request;
  RBC::Ibcast(buffer, count, datatype, root, tag, comm, &request);
  RBC::Wait(&request, MPI_STATUS_IGNORE);
}
```

Each non-blocking communication operation returns a request. The request is an object of the class `RBC::Request`. To check if the operation is completed, we have to invoke the operation `RBC::Test` with the request as a parameter. The operation `RBC::Test` returns `true` if the operation identified by the request is completed, otherwise it returns `false`. Figure 7.1 depicts the class structure of a request of type `RBC::Request`. Each collective operation has to store a number of variables for its execution. Because we want to separate variables from different collective operations, we use one request class per collective operation. We define the request classes in the class `RBC_Requests`. For example, the class `RBC_Requests::Ibcast` stores all variables for the operation `RBC::Ibcast`:

```
class Range_Requests::Ibcast : public Range::R_Req {
public:
      Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
            int tag, Range::Comm comm);
      int test(int *flag, MPI_Status *status);
```

```
private:
      //private variables
};
```

Each request has an operation `test` that executes the collective operation in a non-blocking way and then returns. The externally visible class contains a pointer to one of the request classes. To allow us to point to different requests, we define an abstract base class `RBC::R_Req`. The requests for the different collective operations all inherit from the class `RBC::R_Req` that only contains the function

```
virtual int test(int *flag, MPI_Status *status) = 0
```

The class `RBC::Request` contains a pointer to an object of the class `RBC::R_Req`. We overload the access operators `*` and `->` to allow access to the request object. The operator `=` is used to assign a request object to the pointer.

```
class Request {
  friend class Range;

public:
  Request();

private:
  R_Req& operator*();
  R_Req* operator->();
  void operator=(std::unique_ptr<R_Req> req);

  std::unique_ptr<R_Req> req_ptr;
};
```

When a non-blocking collective operation is called, we create a new request of a subclass of `RBC::R_Req` and then let the given request point to the new request.

```
void RBC::Ibcast(void *buffer, int count, MPI_Datatype datatype,
        int root, int tag, RBC::Comm comm, Range::Request *request) {
```

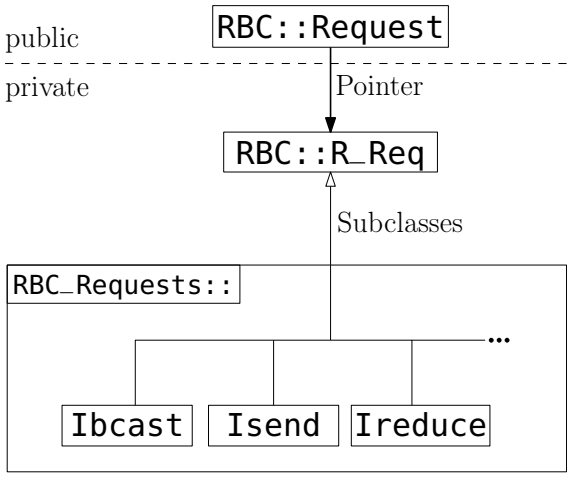

Figure 7.1: Class structure of the requests classes

```
        *request = std::unique_ptr<R_Req>(new Range_Requests::Ibcast(buffer,
                count, datatype, root, tag, comm));
    };
```

Each subclass of `RBC::R_Req` implements the function `test`. The implementation of each collective operation is executed when the function `test` is invoked. When the function `test` is invoked, we perform the collective operation until we wait for the completion of a communication operation. We then return with the value `false` from the function. The current state of a operation is stored in the request object so the execution can resume where the function was exited last time. The function `test` returns `true` if the collective operation has been completed and `false` otherwise.

When the function `RBC::Test` is invoked, we invoke the `test` function of the (sub-class of) `RBC::R_Req` that the passed `RBC::Request` points to:

```
int RBC::Test(RBC::Request *request, int *flag, MPI_Status *status) {
  *flag = 0;
  return (*request)->test(flag, status);
}
```

We use the MPI datatypes (`MPI_Datatype`) and the operation `MPI_Type_size` to determine the size of each datatype. We use the operation `MPI_Reduce_local` and MPI operators (`MPI_Op`) to reduce values, for example in `RBC::Ireduce`.

The interface `RBC::` is the main interface of the communication library. We also provide a second interface with the class `CollectiveOperations`. The template class

```
CollectiveOperations<typename COMM, typename REQ>
```

allows the flexible integration of both MPI and the RBC library. The class contains all operations that our library provides. The class has two specializations `<RBC::Comm, RBC::Request>` and `<MPI_Comm, MPI_Request>`. The functions of the specialization `<RBC::Comm, RBC::Request>` invoke the corresponding function of the RBC library. The functions of the specialization `<MPI_Comm, MPI_Request>` invoke the corresponding function of MPI.

7.2 Schizophrenic Quicksort

We implement Schizophrenic Quicksort in the class `QuickSort`. Figure 7.2 gives the code skeleton of the class. The class `QuickSort` has three template parameters. The first template parameter is the data type of the elements that we want to sort. The second and third template parameters are the communicator and request classes for the communication. The communicator and request either have to be `RBC::Comm` and `RBC::Request` or `MPI_Comm` and `MPI_Request`. We use the class `CollectiveOperations` so that the algorithm can either be executed using MPI or RBC. However, in the later stages of the implementation we integrated the function `RBC::Igatherm` that has no counterpart in MPI. Because of this, we can only execute QuickSort with RBC, but not with MPI.

We use `long long` (64 bit) variables every time we use a global element index or count the number of elements. The reason is that the maximum value of the type `int` (32 bit) is not large enough. For example for 2^{15} PEs and 2^{20} elements per PE, the input (and thus the largest global index) has a total size of 2^{35}, which is larger than a `int` variable can handle.

The class has two functions `quickSort` and `schizophrenicQuickSort` that we call recursively. We call the function `schizophrenicQuickSort` if the PE is schizophrenic and `quickSort`

else. The function `schizophrenicQuickSort` executes one level of recursion on two groups simultaneously. If communication is required, we invoke non-blocking operations on each group. Then we use the functions `Testall` or `Waitall` test or wait for the completion of both operations. We use `std::sort` to sort locally.

Pivot Selection In the pivot selection, we calculate the number of samples that each PE has to pick. We do this by traversing a binomial tree from the root to the leaves and splitting the number of samples at each node. We use the Mersenne Twister 19937 generator `std::mt19937` in combination with the binomial distribution `std::binomial_distribution` to split the number of samples. Each PE calculates the sample distribution locally. We have to guarantee that all PEs in a group calculate the same distribution or else the number of samples might be too high or too low. At the same time, we do not different groups to generate the same random numbers. We seed the generator before calculating with the same seed on all PEs in one group. To create different seeds for the groups, we randomize the seed with every recursion using a Minimal Standard generator `minstd_rand`. When the group is split, the seed on the right subgroup is increased by 1 before randomizing so that each group generates a different seed.

We pick the samples on each PE using the uniform distribution `std::uniform_int_distribution` and a second `std::mt19937` generator. We seed the generator once with `seed + rank` when Schizophrenic Quicksort is started, where `seed` is an initial seed and `rank` is the rank on the global communicator. Each PE is therefore seeded differently. We sort the local samples and then invoke the operation `RBC::Igatherm` to accumulate all samples in a sorted array on the PE with rank 0. We provide `std::inplace_merge` as the merge function for the operation `RBC::Igatherm`. The function merges two sorted arrays into a single sorted array.

```
template<typename T, typename COMM, typename REQ>
class QuickSort {
public:
   using coll = CollectiveOperations<COMM, REQ>;
   void sort(T *first, long long split_size, MPI_Comm mpi_comm,
         int seed) {
      COMM comm;
      coll::Create_Comm_from_MPI(mpi_comm, &comm);
      ... //sort
   };
   ...
};
```

Figure 7.2: Code skeleton of the class `QuickSort`.

8 Experimental Results

We now present the results of our experiments. We divided our experiments into two sections. Firstly, we measure the performance of the operations of our library and the corresponding MPI operations. We compare the running time for each collective operation of our library with the counterpart of MPI. We also compare the running time of creating a RBC communicator with the running time of creating a MPI communicator. Secondly, we measured the running times of our implementation of Schizophrenic Quicksort (*ESQ*) and the implementation of Robust Hypercube Quicksort (*RQuick*). We measured the running times of the different phases of ESQ. We compare ESQ with variants of ESQ that use MPI communicators, blocking collective operations or different numbers of samples for the pivot selection. We present results of RQuick with and without the integration of RBC. Then we evaluate the running times of ESQ and RQuick for different input distributions on various numbers of cores.

8.1 Experimental Setup

We run our benchmarks on two different supercomputers, the *JUQUEEN* [19] and the *SuperMUC* [20]. The JUQUEEN is a IBM BlueGene/Q based system that consist of 28 672 compute nodes. Each compute node consists of a IBM PowerPC A2 16-core processor with a nominal frequency of 1.6 GHz and 16 GB main memory. The nodes are connected by a 5D Torus network with 40 GB/s and $2.5\,\mu s$ worst case latency. Part of the 5D Torus is a collective network and a barrier network. The hardware includes direct support for the MPI collective operations MPI_Bcast and MPI_Reduce [7]. We run our tests on up to 262 144 cores (16 384 nodes). We compile our code with the GCC compiler version 4.8.1 and the MPICH2 library version 1.5. On the SuperMUC we use the Thin Nodes of Phase 1 which consists of 9 216 compute nodes. Each compute node consists of two Sandy Bridge-EP Xeon E5-2680 8C processors with a frequency of 2.3 GHz (16 cores per node). Each node has 32 GB main memory. The nodes are arranged in Node Islands with 512 nodes each. The nodes within an Node Island are connected by a Infiniband FDR10 network with a non-blocking Tree topology. The Node Islands are connected by a Pruned Tree topology. The bandwidth ratio of intra-Island to inter-Island communication is 4:1. Thus, the communication between nodes from the same Island is much faster than the communication between nodes from different Islands. We run our tests on up to 65 536 cores (8 islands). We compile our code with the Intel C++ compiler version 16.0 and the IBM MPI library version 1.4.

Before we start a measurement, we use a global barrier to synchronize all cores. We then use `MPI_Wtime` before we start the benchmark and again after the benchmark is finished to measure the local running time one each core. We take the maximum of the running times of all cores as the running time of this run of the benchmark. We repeat each measurement multiple times and take the median of the measurements as the result of the benchmark.

In this section we focus on the results on the JUQUEEN to evaluate the performance of our implementations. We also give details about differences of the results on the SuperMUC.

8.2 RBC Library

We give measurements of the collective operations of our library and the corresponding collective operations provided by MPI. We also measured the running times of creating a Range communicator and the running times of creating a MPI communicator. Our implementations of the point-to-point operations directly use the MPI counterparts. The differences in the running time between point-to-point operations of our library compared to the MPI operations are negligible. We run experiments on 1 024 to 32 768 cores. We measure the running time of each operation 12 times for each combination of number of cores and input size. The first measurement of an operation is sometimes significantly slower than subsequent measurements. We attribute this to a 'warm-up' effect of the network. We ignore the first measurement and take the median of the remaining 11 measurements as the running time of the operation for the specific number of cores and input size.

8.2.1 Collective Operations

We compare the running time of the non-blocking collective operations of our library with their counterparts that are provided by MPI. To measure the running time of non-blocking collective operations, we firstly invoke the non-blocking function (e.g. `RBC::Ibcast` or `MPI_Ibcast`) and then invoke the operation wait (`RBC::Wait` or `MPI_Wait`) to wait until the collective operation is completed. As the MPI library on the JUQUEEN does not support non-blocking collective operations, we compare the blocking collective operations of MPI (e.g. `MPI_Bcast`) on the JUQUEEN.

We run measurements for different number of cores and input sizes. Let p be the number of cores and n the number of elements of the input. We use 64-bit floating-point elements as the input. We evaluate the running time of the collective operations for two different scenarios. Firstly, we analyze the running time for increasing number of cores with constant input size. We evaluate the results for one element input on 1 024 to 32 768 cores. Secondly, we analyze the running time for different input sizes of the collective operations on 32 768 cores and $n = 1$ to $n = 2^{18}$.

Broadcast

We now present results of the operation Broadcast on the JUQUEEN and SuperMUC. Figure 8.1a shows the running times when the root sends one element on 2^{10} up to 2^{15} cores. On the JUQUEEN, the operation `RBC::Ibcast` is slower than the operation `MPI_Bcast` by a factor of 25 on 2^{10} cores and by a factor of 30 slower on 2^{15} cores. The collective network of the JUQUEEN supports the operation `MPI_Bcast` and thus speeds up the execution of this operation. On the SuperMUC, the difference in running time between the operations `RBC::Ibcast` and `MPI_Ibcast` is much smaller than the difference between the operations `RBC::Ibcast` and `MPI_Bcast` on the JUQUEEN. The operation `MPI_Ibcast` outperforms the operation `RBC::Ibcast` by just a factor of 1.6 on 2^{10} cores and decreases to a factor of 1.12 on 2^{15} cores. Note that on the SuperMUC, the running times of both operations increase by a factor of 3 if we go from 1 island to 2 islands and by a further factor of 2 if we go from 2 islands to 4 islands.

Figure 8.1b depicts the results for different input sizes on 32 768 cores. On the JUQUEEN, the operation `RBC::Ibcast` is slower than the operation `MPI_Bcast` by a factor of around

30 for most input sizes. Note that on the SuperMUC, the difference in running time between the operations `RBC::Ibcast` and `MPI_Ibcast` is smaller than between the operations `RBC::Ibcast` and `MPI_Bcast` on the JUQUEEN. The operation `MPI_Ibcast` outperforms the operation `RBC::Ibcast` by just a factor of 1.26 for the smallest input and by just a factor of 2 for the larges input.

On the JUQUEEN, the collective network speeds up the operation `MPI_Bcast`. As expected, the operation `MPI_Bcast` outperforms the operation `RBC::Ibcast` by a large factor (25 to 30) for all inputs. On the SuperMUC, the operation `RBC::Ibcast` is at most 2 times slower than the operation `MPI_Ibcast`.

Reduce

We now present results of the operation Reduce with the sum operator (`MPI_SUM`) on the JUQUEEN and SuperMUC. Figure 8.2a shows the running times of the operation on 2^{10} up to 2^{15} cores with one element as the local input on each PE. On the JUQUEEN, the operation `MPI_Reduce` is faster than the operation `RBC::Ireduce` by a factor of 13 on 2^{10} cores and by a factor of 20 slower on 2^{15} cores. The collective network of the JUQUEEN supports the operation `MPI_Reduce` and thus speeds up the execution of this operation. On the SuperMUC, the operations `RBC::Ireduce` and `MPI_Ireduce` have almost identical running times for all numbers of cores (less than 10% difference). Note that on the SuperMUC, the running times of both operations increase by a factor of 3 if we go from 1 island to 2 islands and by a further factor of 2 if we go from 2 islands to 4 islands.

Figure 8.2b depicts the results for different input sizes on 32 768 cores. On the JUQUEEN, the operation `MPI_Reduce` outperforms the operation `RBC::Ireduce` by a factor of 20 for the smallest input and by a factor of 72 for the largest input. On the SuperMUC, the difference in running time between the operations `RBC::Ireduce` and `MPI_Ireduce` is smaller than between the operations `RBC::Ireduce` and `MPI_Reduce` on the JUQUEEN. The operation `RBC::Ireduce` is at most by a factor of 1.9 slower for small to medium inputs ($n \leq 2^{14}$) and 6 times slower for the largest input. We assume that the operation `MPI_Ireduce` on the SuperMUC implements a different algorithm for inputs larger than 2^{12}. This explains why the operation `MPI_Ireduce` outperforms the operation `RBC::Ireduce` by a larger factor for large inputs.

On the JUQUEEN, the collective network speeds up the operation `MPI_Reduce`. As expected, the operation `MPI_Reduce` outperforms the operation `RBC::Ireduce` by a large factor (25 to 30) for all inputs. On the SuperMUC, the operation `RBC::Ireduce` is at most 1.9 times slower than `MPI_Ireduce` for small and medium inputs.

Scan

We now present results of the operation Scan with the sum operator (`MPI_SUM`) on the JUQUEEN and SuperMUC. Figure 8.3a shows the running times of the operation on 2^{10} up to 2^{15} cores with one element as the local input on each PE. On the JUQUEEN, the operation `RBC::Iscan` is slower than the operation `MPI_Scan` by a factor of 4.5 on 2^{10} cores and by a factor of 4 slower on 2^{15} cores. On the SuperMUC, the difference in running time between the operations `RBC::Iscan` and `MPI_Ibcast` is much smaller than the difference between the operations `RBC::Iscan` and `MPI_Scan` on the JUQUEEN. The operation `MPI_Scan` outperforms the operation `RBC::Iscan` by just a factor of 1.7 on 2^{10} cores and by just a factor of 1.05 on 2^{15} cores. Note that on the SuperMUC, the running times of both operations increase by a factor of 1.9

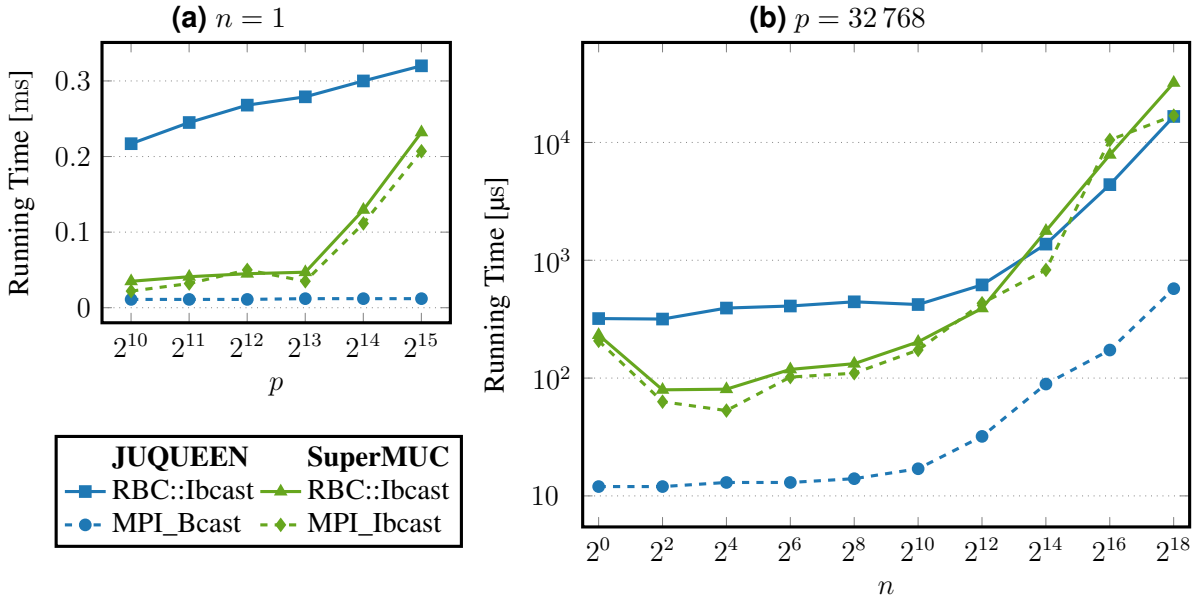

Figure 8.1: Figure (a) depicts the running times of the operation Broadcast for different numbers of cores with a input size of one element on the root. Figure (b) depicts the running times for different input sizes on 32 768 cores. Blue lines show measurements on the JUQUEEN and green lines show measurements on the SuperMUC.

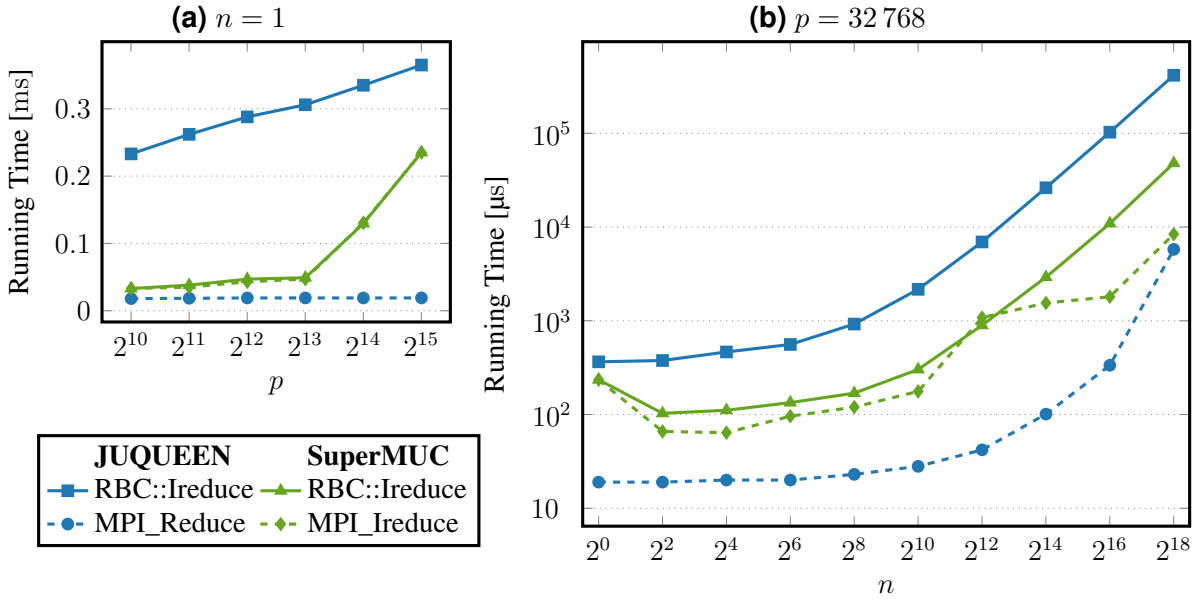

Figure 8.2: Figure (a) depicts the running times of the operation Reduce for different numbers of cores with a input size of one element on each PE. Figure (b) depicts the running times for different input sizes on 32 768 cores. Blue lines show measurements on the JUQUEEN and green lines show measurements on the SuperMUC.

if we go from 1 island to 2 islands and by a further factor of 1.9 if we go from 2 islands to 4 islands.

Figure 8.3b depicts the results for different input sizes on 32 768 cores. On the JUQUEEN, the operation `MPI_Scan` outperforms the operation `RBC::Iscan` by a factor of 4 for the smallest input and by a factor of 1.25 for the largest input. On the SuperMUC, the operation `RBC::Iscan` is slower than the operation `MPI_Iscan` by just a factor of at most 1.5 for small to medium inputs ($n \leq 2^{12}$). For the largest input, the operation `RBC::Iscan` even outperforms the operation `MPI_Iscan` by a factor of 14. We assume that the operation `MPI_Ireduce` on the SuperMUC implements a different algorithm for inputs larger than 2^{12}. However, the new algorithm seems to slow down the execution by a factor of 15 to 20 instead of increasing the performance. This explains why the operation `RBC::Ireduce` outperforms the operation `MPI_Ireduce` for large inputs.

On the JUQUEEN, the operation `MPI_Iscan` outperforms the operation `RBC::Iscan` by a factor of up to 4. The factor decreases for large inputs. On the SuperMUC, the operation `RBC::Iscan` has a similar running time than `MPI_Iscan` for small and medium input sizes and outperforms the operation `MPI_Iscan` for large inputs.

Scan-and-Broadcast

We now present results of the operation Scan-and-Broadcast as defined in Section 5.3.5 on the JUQUEEN and SuperMUC. We use the sum operator (`MPI_SUM`) for the Scan. Our communication library provides the operation `RBC::IscanAndBcast` that performs the operation. MPI provides no such operation. To measure the running time of the MPI operations, we first invoke the operation `MPI_Iscan` and wait until the operation is completed. Then we invoke the operation `MPI_Ibcast` with the last PE as the root and wait for its completion. The results are extremely similar to the results of the operation Scan.

Figure 8.4a shows the running times on 2^{10} up to 2^{15} cores when the input on each PE is one element. On the JUQUEEN, the operation `RBC::IscanAndBcast` is 4 times slower than the operations `MPI_Scan/MPI_Bcast` on all numbers of cores. On the SuperMUC, the difference in running time between the operations `RBC::IscanAndBcast` and `MPI_Iscan/MPI_Icast` is much smaller than the difference between the operations on the JUQUEEN. The operation `RBC::Ibcast` is by a factor of 1.2 slower on 2^{10} cores and decreases to a factor of 0.95 on 2^{15} cores. Note that on the SuperMUC, the running times of both operations increase by a factor of 2.3 if we go from 1 island to 2 islands and by a further factor of 1.6 if we go from 2 islands to 4 islands.

Figure 8.4b depicts the results for different input sizes on 32 768 cores. On the JUQUEEN, the operation `MPI_Scan/MPI_Bcast` outperforms the operation `RBC::IscanAndBcast` by a factor of 4 for the smallest input and by a factor of 1.45 for the largest input. Note that on the SuperMUC, the operation `RBC::IscanAndBcast` is at most by a factor of 1.5 slower than the operations `MPI_Iscan/MPI_Icast` for small to medium inputs ($n \leq 2^{12}$). For the largest input, the operation `RBC::IscanAndBcast` even outperforms the operations `MPI_Iscan/MPI_Icast` by a factor of 11.

On the JUQUEEN, the collective network speeds up the operation `MPI_Scan`. However, the speed up is much smaller than the speed up for the operations `MPI_Bcast` or `MPI_Reduce`, especially for large inputs. On the SuperMUC, the operation `RBC::Iscan` has a similar running time than `MPI_Iscan` for small and medium input sizes and outperforms the operation `MPI_Iscan` for large inputs.

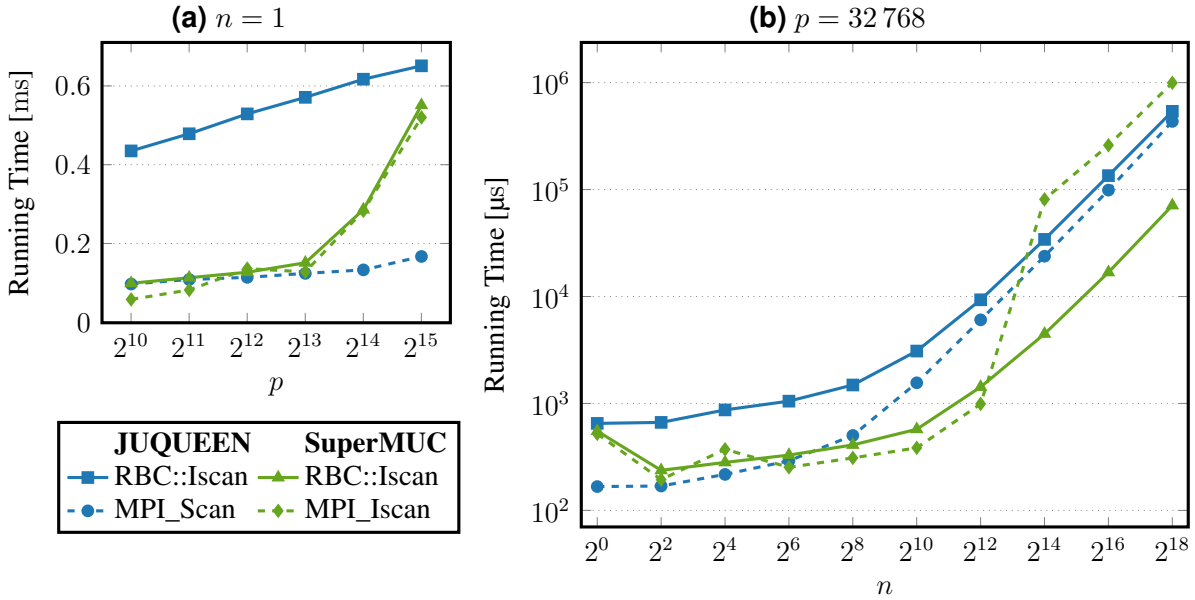

Figure 8.3: Figure (a) depicts the running times of the operation Scan for different numbers of cores with a input size of one element on each PE. Figure (b) depicts the running times for different input sizes on 32 768 cores. Blue lines show measurements on the JUQUEEN and green lines show measurements on the SuperMUC.

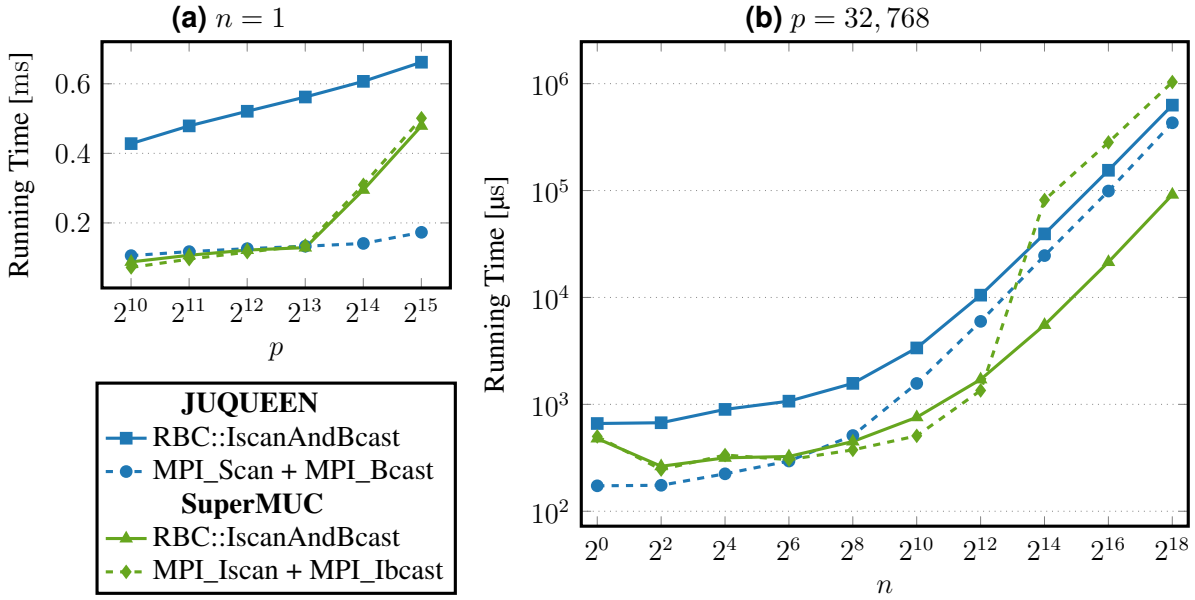

Figure 8.4: Figure (a) depicts the running times of the operations Scan and Broadcast for different numbers of cores with a input size of one element on each PE. Figure (b) depicts the running times for different input sizes on 32 768 cores. Blue lines show measurements on the JUQUEEN and green lines show measurements on the SuperMUC.

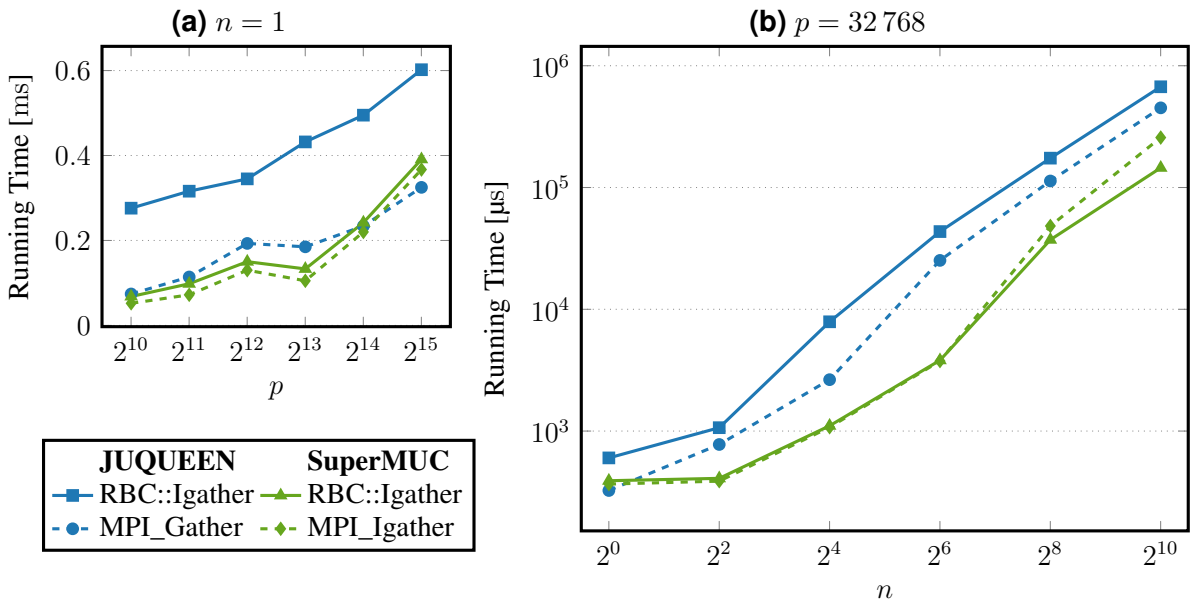

Figure 8.5: Figure (a) depicts the running times of the operation Gather for different numbers of cores with a input size of one element on each PE. Figure (b) depicts the running times for different input sizes on 32 768 cores. Blue lines show measurements on the JUQUEEN and green lines show measurements on the SuperMUC.

Gather

We now present results of the operation Gather on the JUQUEEN and SuperMUC. The local input of the operation on each PE has the same number of elements (n). Figure 8.5a shows the running times on 2^{10} up to 2^{15} cores with one element as the local input on each PE. On the JUQUEEN, the operation `RBC::Igather` is slower than the operation `MPI_Gather` by a factor of 4 on 2^{10} cores and by a factor of 1.9 on 2^{15} cores. On the SuperMUC, the difference in running time between the operations `RBC::Igather` and `MPI_Igather` is much smaller than the difference between the operations `RBC::Igather` and `MPI_Gather` on the JUQUEEN. The operation `MPI_Igather` outperforms the operation `RBC::Igather` by just a factor of 1.3 on 2^{10} cores and just a factor of 1.06 on 2^{15} cores. Note that on the SuperMUC, the running times of both operations increase by a factor of 1.8 if we go from 1 island to 2 islands and by a further factor of 1.6 if we go from 2 islands to 4 islands.

Figure 8.5b depicts the results for different input sizes on 32 768 cores. On the JUQUEEN, the operation `MPI_Gather` outperforms the operation `RBC::Igather` by a factor of 1.9 for the smallest input and by a factor of 1.5 for the largest input. On the SuperMUC, the difference in running time between the operations `RBC::Igather` and `MPI_Igather` is smaller than between the operations `RBC::Igather` and `MPI_Gather` on the JUQUEEN. For the smallest input size, the operation `RBC::Igather` is slower than the operation `MPI_Igather` by just a factor of 1.06. For the largest input, the operation `RBC::Igather` even outperforms the operation `MPI_Igather` by a factor of 1.8.

On the JUQUEEN, the operation `MPI_Gather` outperforms the operation `RBC::Igather` at most by a factor of 1.9. On the SuperMUC, the operation `RBC::Igather` has a similar running time as the operation `MPI_Igather` for most input sizes and even outperforms the operation `MPI_Igather` for the largest input.

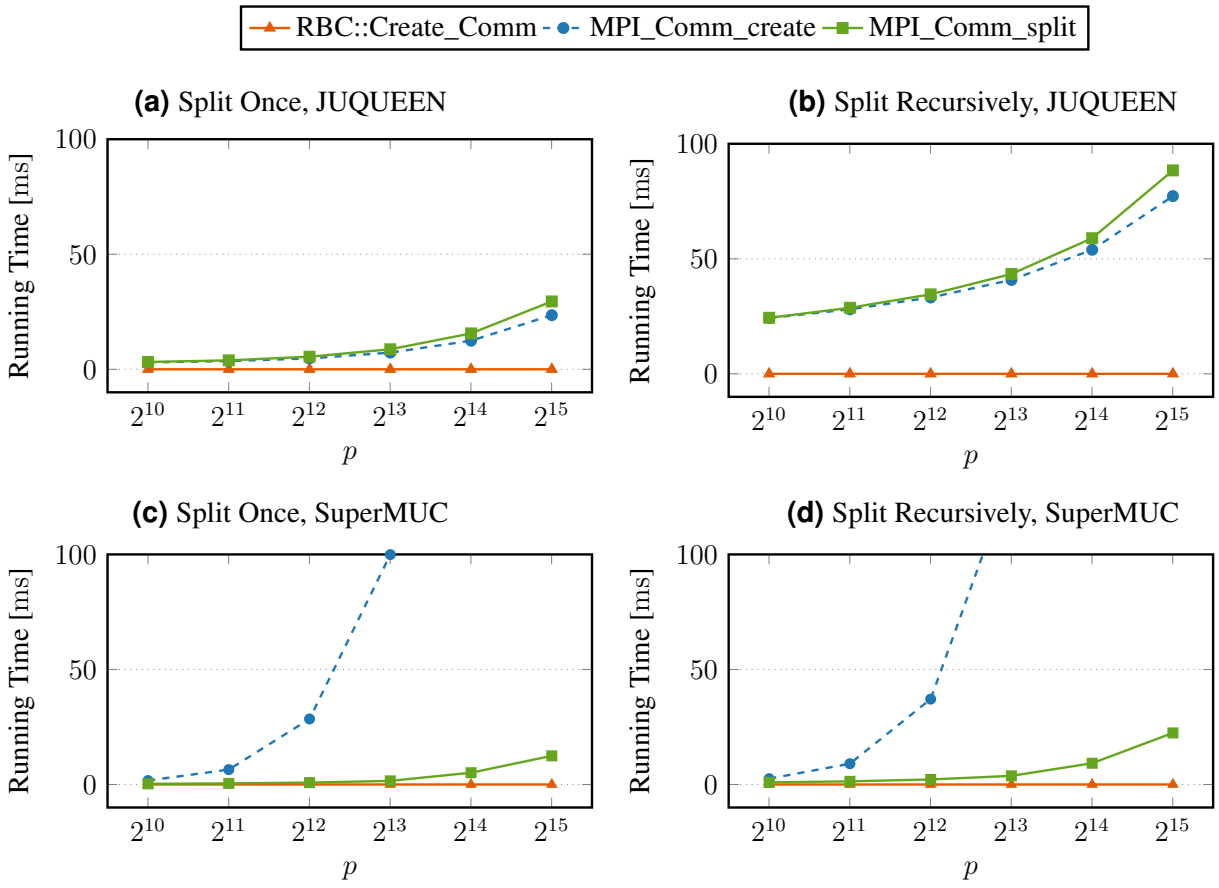

Figure 8.6: Figure (a) and (c) depict the results when we split the global communicator once. Figure (b) and Figure (d) depict the results when the global communicator is recursively split until the communicators have a size of 1. Figure (a) and Figure (b) show the results on the JUQUEEN, Figure (c) and (d) show the results on the SuperMUC.

8.2.2 Communicator Split

We measured the running times of splitting a Range communicator and splitting a MPI communicator. In this section, splitting a communicator means that we split a communicator of PEs with ranks $[0, p-1]$ into two subcommunicators with the ranks $[0, p/2-1]$ and $p/2, p-1]$. We measure two scenarios of communicator splitting. In the first scenario, we split the global communicator (containing all PEs) once. In the second scenario, we split the global communicator recursively until all communicators have a size of 1. We use the function RBC::Create_Comm twice to split a communicator of type `RBC::Comm`. MPI provides two operations to create new MPI communicators, `MPI_Comm_create` and `MPI_Comm_split`. To create a new communicator with `MPI_Comm_create` we first have to create a `MPI_Group` G from the original communicator. Then we create a new `MPI_Group` G_0 and invoke the operation `MPI_Group_range_incl` that includes an interval of ranks from G into G_0. Finally, we create a new communicator by calling `MPI_Comm_create` with the original communicator and G_0 as parameters. We invoke the operaton `MPI_Group_range_incl` with different parameters to split the communicator. To create a new communicator with the operation `MPI_Comm_split`, each PE provides an integer value (*color*) as the input of the operation. The operation creates one subcommunicator for each color. The subcommunicator for a specific color includes all PEs that invoked `MPI_Comm_split` with that color. We invoke `MPI_Comm_split` with color 0 on the PEs with rank $[0, p/2-1]$ and with color 1 on the PEs with rank $[p/2, p-1]$.

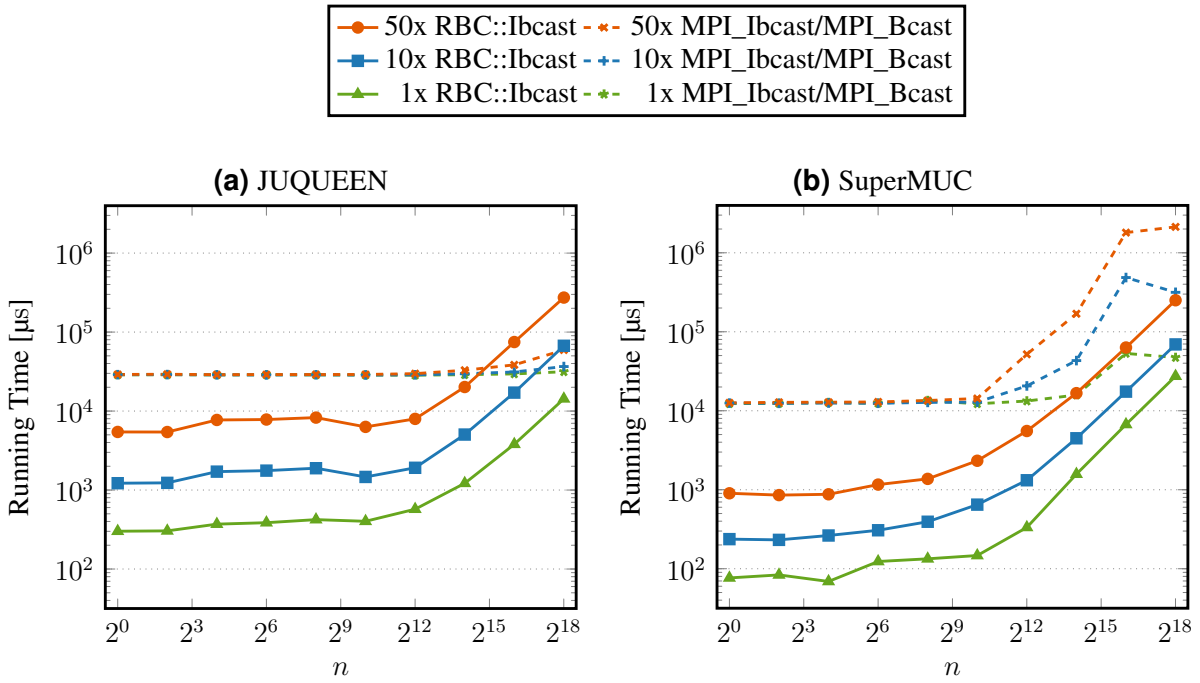

Figure 8.7: We split the global communicator and then perform the operation Broadcast multiple times on both subcommunicators. Figure (a) shows the running times on 32 768 on the JUQUEEN. Figure (b) shows the running times on 32 768 cores on the SuperMUC.

The running time of the operation `RBC::Create_Comm` is negligible compared to the other two operations of MPI (factor 10 000+). Thus, we now just compare the operations `MPI_Comm_create` and `MPI_Comm_split`. Figure 8.6a depicts the running times of a single split on 2^{10} up to 2^{15} cores on the JUQUEEN. The operation `MPI_Comm_split` is slower than the operation `MPI_Comm_create` by a factor of 1.05 on 2^{10} cores and by a factor of 1.25 slower on 2^{15} cores. Figure 8.6b shows the running times of a recursive split on 2^{10} up to 2^{15} cores on the JUQUEEN. Both the operation `MPI_Comm_create` and the operation `MPI_Comm_split` are slower than for a single split by a factor of 8 on 2^{10} cores and by a factor of 3 on 2^{15} cores.
Figure 8.6c depicts the running times of a single split on 2^{10} up to 2^{15} cores on the SuperMUC. The operation `MPI_Comm_create` scales significantly worse than the operation `MPI_Comm_split`. The operation `MPI_Comm_create` is slower than the operation `MPI_Comm_split` by a factor of 6 on 2^{10} cores and by a factor of 80 on 2^{14} cores. Figure 8.6b shows the running times of a recursive split on 2^{10} up to 2^{15} cores on the SuperMUC. Compared to a single split, the operation `MPI_Comm_Create` is 3 times slower on 2^{10} cores and 1.8 times slower for 2^{15} cores.

The operation `MPI_Comm_split` is much faster than the operation `MPI_Comm_create` for large numbers of cores on the SuperMUC. On the Juqueen, the operation `MPI_Comm_split` is a just few percentages slower than the operation `MPI_Comm_create`. Because we want to compare our communication library against the fastest operation that splits a MPI communicator, we use `MPI_Comm_split` for all following benchmarks.

We have shown that splitting a MPI communicator is much slower than splitting a `RBC::Comm` communicator. In contrast, collective operations of MPI outperform the collective operations of our library for most inputs. In practical applications, algorithms may firstly split communicators and then execute collective operations on the subcommunicators. For such algorithms, we expect that MPI outperforms our library if many collective operations are executed or if the input of the collective operations is very large. But for few collective operations or small inputs, we

expect our library to be faster than MPI. We want to estimate how many collective operations have to be executed so that MPI outperforms our library. We split the global communicator on 2^{15} cores and then perform the operation Broadcast k times on both subcommunicators simultaneously. The root PE of the operation Broadcast is the PE with rank 0 in each subcommunicator. We measure the running times of splitting the communicator and then executing the operation Broadcast once, 10 times or 50 times for different input sizes.

Figure 8.7a gives the results on the JUQUEEN. For $k = 1$, the operation `RBC::Ibcast` outperforms the operation `MPI_Bcast` by a factor of 95 for the smallest input and by a factor of 2.2 for the largest input. For $k = 10$, the operation `MPI_Bcast` is slower than the operation `RBC::Ibcast` by a factor of 23 for the smallest input, but outperforms the operation `RBC::Ibcast` by a factor of 1.8 for the largest input. The operation `RBC::Ibcast` is always faster for $n \leq 2^{16}$. For $k = 50$, the operation `MPI_Bcast` is slower than the operation `RBC::Ibcast` by a factor of 5.3 for the smallest input, but outperforms the operation `RBC::Ibcast` by a factor of 4.6 for the largest input. The operation `RBC::Ibcast` is always faster for $n \leq 2^{14}$. Note that the running times of the operation `RBC::Ibcast` increase just by a factor of around 18 on all input sizes if we go from $k = 1$ to $k = 50$. The running time to split the communicator is negligible compared the running times of the operation `RBC::Ibcast` for all inputs. Thus, we would expect the running times to increase by a factor similar to k. We assume that the implementation of the operation `RBC::Ibcast` together with the fact that all operations have the same root are the reason of this behavior. In the implementation, the root PE sends a message to a couple of PEs and then has completed the operation. We believe that the root completes the operation `RBC::Ibcast` faster than other PEs, e.g. the last PE in the group, and then immediately starts the next execution of the operation. Thus, the total time is only k times the running time of the operation `RBC::Ibcast` on the root plus the time until the operation is completed on the last PE.

Figure 8.7b depicts the results on the SuperMUC. For $k = 1$, the operation `RBC::Ibcast` outperforms the operation `MPI_Bcast` by a factor of 160 for the smallest input and by a factor of 1.7 for the largest input. For $k = 10$, the operation `RBC::Ibcast` outperforms the operation `MPI_Bcast` by a factor of 52 for the smallest input and by a factor of 4.6 for the largest input. For $k = 50$, the operation `RBC::Ibcast` outperforms the operation `MPI_Bcast` by a factor of 14 for the smallest input and by a factor of 8.5 for the largest input. Note that the running times of the operation `RBC::Ibcast` increase just by a factor of around 10 on all input sizes if we go from $k = 1$ to $k = 50$. The running time of the operation `MPI_Ibcast` however increases by a factor of 45 for the largest input, where the communicator split is only a small portion of the total running time. We assume that the operation `MPI_Ibcast` is implemented in such a way that it can not take advantage of the fact that the operation is executed multiple times from the same root. This explains why the operation `RBC::Ibcast` outperforms the operation `MPI_Ibcast` on all inputs even though we showed that a single execution of the operation `RBC::Ibcast` is slower than a single execution of the operation `MPI_Ibcast` for the largest input (Section 8.2.1).

On the JUQUEEN, our library outperforms the MPI implementation for small and medium inputs and $k \leq 50$. On the SuperMUC, our library outperforms the MPI implementation for all tested inputs and all tested k. The communication library outperforms MPI unless an algorithm performs a great number of collective operations after each communicator split or performs multiple collective operations with large inputs.

8.3 Sorting Algorithms

We measure the performance of our implementation of Schizophrenic Quicksort (ESQ) and the implementation of Robust Hypercube Quicksort (RQuick) on up to 262 144 cores on the JUQUEEN and on up to 65 536 cores on the SuperMUC.

We run experiments with input sizes from 1 to 2^{20} elements per PE. We execute each benchmark 7 times for each combination of number of cores and input size. We present the median of the 7 iterations unless stated otherwise. We execute each algorithm two times for each input size before we start our measurements. We do this because we noticed that the first few executions were slower than the following executions. On SuperMUC, the first 10 iterations are slower than later iterations even after the warmup. The first iteration is by a factor 10 slower, each following iteration is faster. After 10 iterations, the running times are relatively constant. We thus add 10 executions of each algorithm with the smallest input size to the end of the warmup phase.

This section is divided into three subsections. Firstly, we evaluate ESQ. We analyze how much the different phases of Schizophrenic Quicksort contribute to the total running time of ESQ. We show that the integration of our communication library increases the performance of ESQ. We also compare different approaches to select the number of samples for the pivot selection. Secondly, we present results of RQuick with and without our communication library. Finally, we compare the performances of ESQ and RQuick for uniformly distributed inputs on the JUQUEEN and SuperMUC. We also evaluate the running times of both ESQ and RQuick for different input instances on the JUQUEEN.

8.3.1 ESQ

Algorithm Phases

We divide each level of recursion of Schizophrenic Quicksort into five distinct phases: The Pivot Selection, the Partitioning, the Exchange Calculation, the Data Exchange the Communicator Creation. After we end the recursion of Schizophrenic Quicksort, we sort the remaining unsorted element in the Base Cases (the sixth phase). We measure the running times of all phases individually. In each level of recursion, we execute a barrier (`MPI_Barrier`) before each algorithm phase to synchronize all PEs. The running times of the phases are accumulated over all levels of recursion. We determine the maximum running time of each phase on all PEs as the result of the measurement. We run our experiments on 32 768 cores on both the SuperMUC and the JUQUEEN.

Figure 8.8a and Figure 8.8b show the running times of the different phases on 32 768 cores on the JUQUEEN. The time for Communicator Split is negligible for all input sizes. For the smallest input, the Pivot Selection and Exchange Calculation dominate the running time by representing 95% of the total running time. The Pivot Selection is around 50% slower than the Prefix Sum. For the largest input, the Pivot Selection and Exchange Calculation represent only 8% of the total running time. The slowest phase is the Data Exchange that represents 60% of the total running time. The Base Case represents 21%, the Partitioning 11% of the total running time. The running time of the Exchange Calculation is not dependent in the input size. The running time of the Pivot Selection increases only for large inputs, and increases slower than the total running time. The running time of the phases Data Exchange, Partitioning and Base Cases increases for larger inputs. As expected, the former dominate the running time for small input, while the latter dominate the running time for large inputs.

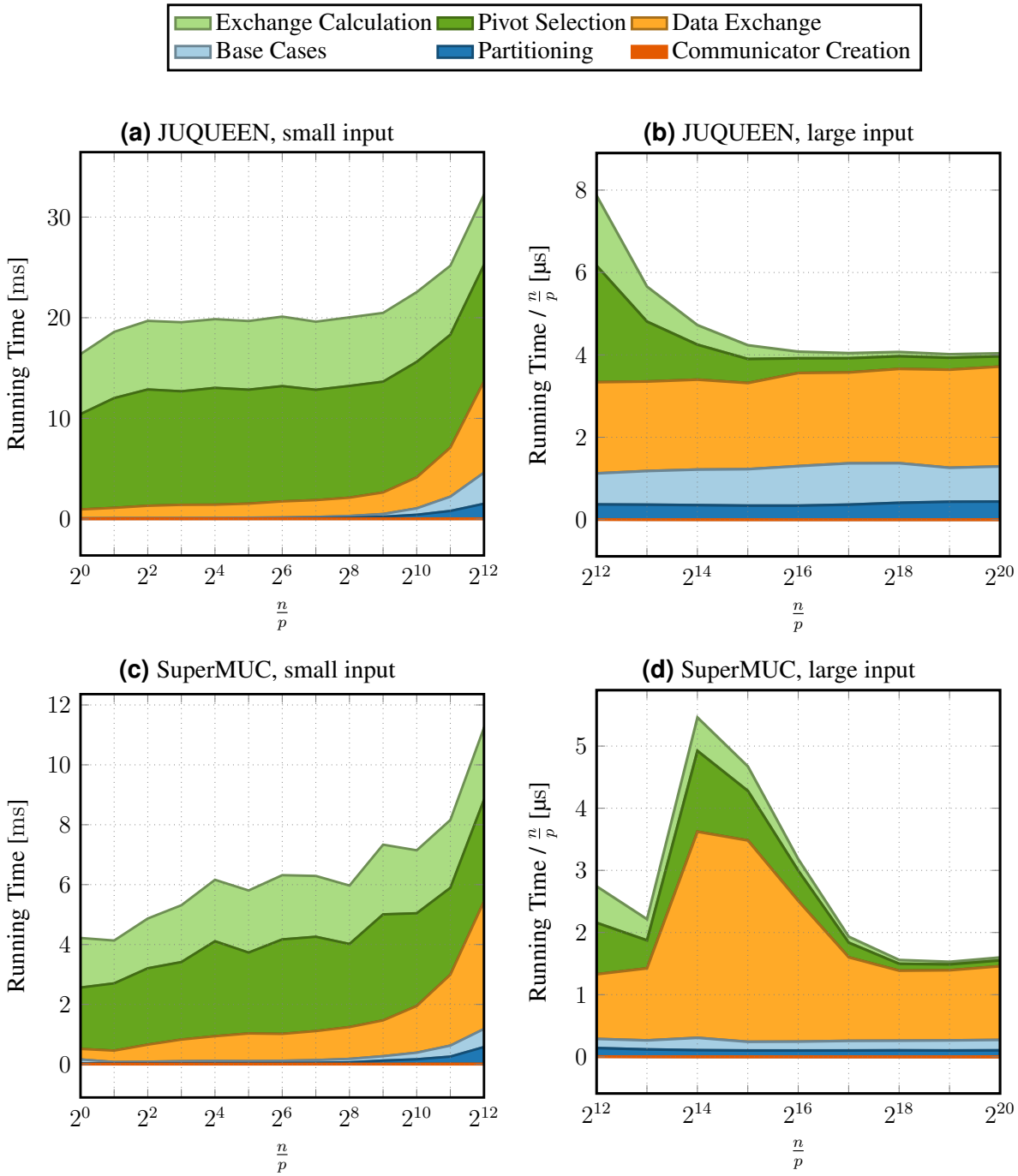

Figure 8.8: Figure (a) and (b) give the accumulated running times of the phases of Schizophrenic Quicksort on 32 768 cores on the JUQUEEN. Figure (c) and (d) depict the results on the 32 768 on the SuperMUC.

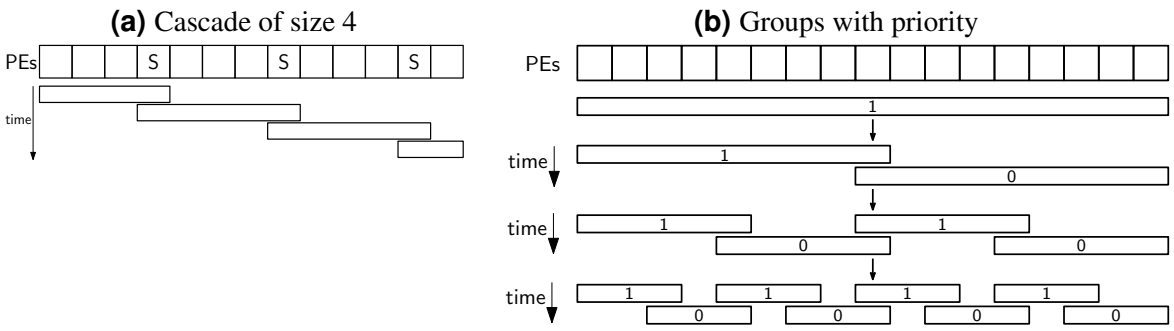

Figure 8.9: Figure (a) shows a cascade of size four. Each group is represented by one rectangle. The three schizophrenic PEs are marked with *S*. Figure (b) depicts our implementation that assigns each group a priority. The figure shows three levels of recursions from the top to the bottem. The priority of each group is indicated by the number in the rectangle.

Figure 8.8a and Figure 8.8b give the running times of the different phases on 32 768 cores on the SuperMUC. The Pivot Selection and Prefix Sum represent 85% of the total running time for the smallest input. For the largest input, the Exchange makes up 74% of the total running time. Note that running times of the communication phases (Pivot Selection, Exchange Calculation and Data Exchange) increases by a factor of 2.6 if we go from $\frac{n}{p} = 2^{13}$ to $\frac{n}{p} = 2^{14}$. We believe that the network of the SuperMUC performs worse when we send messages with size 2^{14} or larger, because we observe no similar effects on the JUQUEEN. Because the communication is slower, the running time per element of ESQ increases by a factor of 2.3 if we go from $\frac{n}{p} = 2^{13}$ to $\frac{n}{p} = 2^{14}$. For larger inputs, running time per element decrease to a valuer lower than for $\frac{n}{p} = 2^{13}$. We believe that the network of the SuperMUC handling of communication to be effective for large messages. However, the other approach seems to switch too early.

Blocking Collectives

We compare our best implementation of Schizophrenic Quicksort which uses non-blocking collective operations (ESQ) with an implementation that uses blocking collective operations (BCO-ESQ). We give results for 32 768 cores on the JUQUEEN and the SuperMUC. We can not execute two or more blocking collective operations simultaneously on a PE. Instead, we have to execute the operations sequentially. In SchizoQS, a schizophrenic PE first executes a collective operation on the left group, then executes the collective operation on the right group. This method causes a sequential execution on the groups from left to right, if the groups overlap on schizophrenic PEs. We call groups that are connected via schizophrenic PEs a *cascade*. Figure 8.9a shows a cascade of four groups with three schizophrenic PEs. In a cascade, the rightmost group has to wait for the schizophrenic PE on the left end of the group. The schizophrenic PE first communicates on its left group and then on its right group. This means that a group can only finish the communication after all groups to its left are already finished. Let g bet the number of groups in a cascade. The collective operations in a cascade are slowed down by a factor of g if the collective operations have the same running time in each group. We use a simple approach to prevent the occurrence of long cascades when using blocking collective operations. We assign a priority of 0 or 1 to each group. If the left group on a schizophrenic PE has priority 1, the collective operations are first executed on the left group and then on the right group. If the left group has priority 0, the collective operations are first executed on the

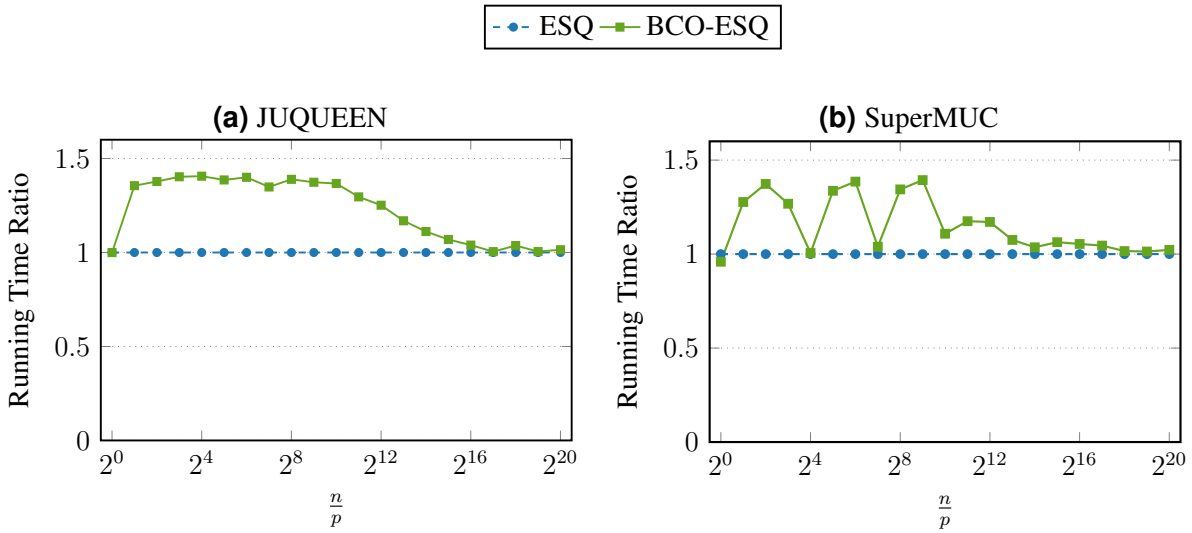

Figure 8.10: Figure (a) depicts the running time ratio of BCO-ESQ compared to ESQ on 32 768 cores on the JUQUEEN. Figure (b) shows the results on 32 768 cores on the SuperMUC.

right group. We assume that the pivot selection is good so that each group is split into two equally sized subgroups. When a group is split, we assign priority 1 to the left subgroup and priority 0 to the right subgroup. As seen in Figure 8.9b, the groups alternate in priority so that adjacent have a different priority. The groups with the same priority do not overlap, meaning no PE is part of two groups with the same priority. We execute simultaneously the collective operations on all groups with priority 1. Then we simultaneously execute the collective operations on all groups with priority 0. The total running time until the collective operations are completed on all groups is double the running time of one collective operation.

Figure 8.10a depicts the results on the JUQUEEN. If the input is only one element on each PE, no schizophrenic PEs occur. As expected, the running time of ESQ and BCO-ESQ is identical for the smallest input. For small inputs ($\frac{n}{p} \leq 10$), BCO-ESQ is around 40% slower than ESQ. For the largest input, ESQ and BCO-ESQ have a nearly identical running time (only 1.5% difference). Using blocking collective only increases the running time of the Pivot Selection and the Exchange Calculation. Because these two phases represent only a small part of the total running time for the largest input, the blocking collective operations have a small impact on the total running time.

Figure 8.10a gives the results on the SuperMUC. For most small inputs, the running time ratio of BCO-ESQ to ESQ is similar to the results on the JUQUEEN. Note that BCO-ESQ is nearly as fast as ESQ for $\frac{n}{p} = 2^4$ and $\frac{n}{p} = 2^7$. We believe that the network of the SuperMUC is the reason for these variations.

Communicator Split

We compare our implementation that splits communicators with `RBC::Split_Comm` (ESQ) with an implementation that splits the MPI communicator with `MPI_Comm_split` (MCS-ESQ). The communicator creation with `MPI_Comm_split` is a blocking collective operation. We use the method described in Section 8.3.1 to prevent cascades when creating the communicator.

Figure 8.11a shows the running time of MCS-ESQ and ESQ on 32 768 cores on the JUQUEEN. MCS-ESQ is about 0.7 seconds slower than ESQ for all inputs. ESQ outperforms MCS-ESQ by a factor of 40 for the smallest input and by a factor of 1.15 for the largest input.

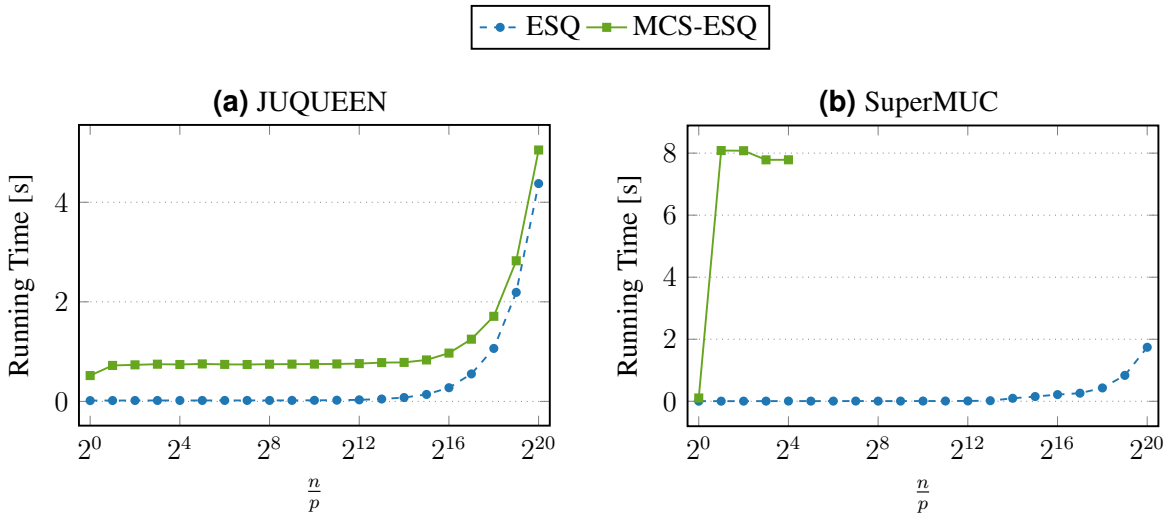

Figure 8.11: Communicator Split, Schizophrenic Quicksort, 32 768 cores
Figure (a) gives the running times of MCS-ESQ and ESQ on 32 768 cores on the JUQUEEN. Figure (b) shows the running time on 32 768 cores on the SuperMUC.

Figure 8.11b gives results on 32 768 cores on the SuperMUC. Note that MCS-ESQ is about 8 seconds slower than ESQ for $\frac{n}{p} \geq 2$. ESQ outperforms MCS-ESQ by a factor of 2000 in this case. We did not measure the running time of MCS-ESQ for larger inputs because MCS-ESQ is 4 times slower for small inputs than ESQ for the largest input. Note that the running time of MCS-ESQ is only 30 times slower than the running time of ESQ for an input of one element per PE. We believe that creating the communicators on schizophrenic PEs increases the running time significantly on the SuperMUC.

Pivot Selection

We test different strategies for choosing the number of samples in the pivot selection of Schizophrenic Quicksort. We always take 9 samples (*ESQ-9*) or $16 \log p$ samples (*ESQ-logp*). We also test an approach that picks $\frac{1}{50}\frac{n}{p}$ samples, but at least 9 samples. ESQ combines these three approaches in the pivot selection by picking the maximum of $\frac{1}{50}\frac{n}{p}$, $16 \log p$ and 9 as the number of samples (*ESQ-combined*).

Figure 8.12a shows the running time of the variants on 32 768 cores on the JUQUEEN. As expected, ESQ-9 is around 10% to 20% slower than ESQ-combined for most inputs. The running time for ESQ-logp and ESQ-combined is similar for all inputs. ESQ-n/p has an identical running time as ESQ-9 for small inputs and an identical running time as ESQ-combined for large inputs. Figure 8.12b shows the average depth of recursion on the JUQUEEN. As expected, the depth of recursion of ESQ-9 and ESQ-lopp is not dependent on the input size. Of the three simple approaches, ESQ-logp has the lowest depth for small inputs and ESQ-n/p has the lowest depth for large inputs. As expected, ESQ-combined has the lowest depth for all inputs. Note that ESQ-combined is slower than ESQ-logp by a factor of 1.05 for the largest input. However, we only run experiments with a uniformly distributed input. We assume that the lower depth of ESQ-combined increases the robustness compared to ESQ-logp.

Figure 8.12c and Figure 8.12d give the running time and recursion depth on 32 768 cores on the SuperMUC. Note that ESQ-9 and ESQ-n/p have a similar running time than ESQ-combined for small inputs. As expected, the depths of recursion are identical to the depths on the JUQUEEN

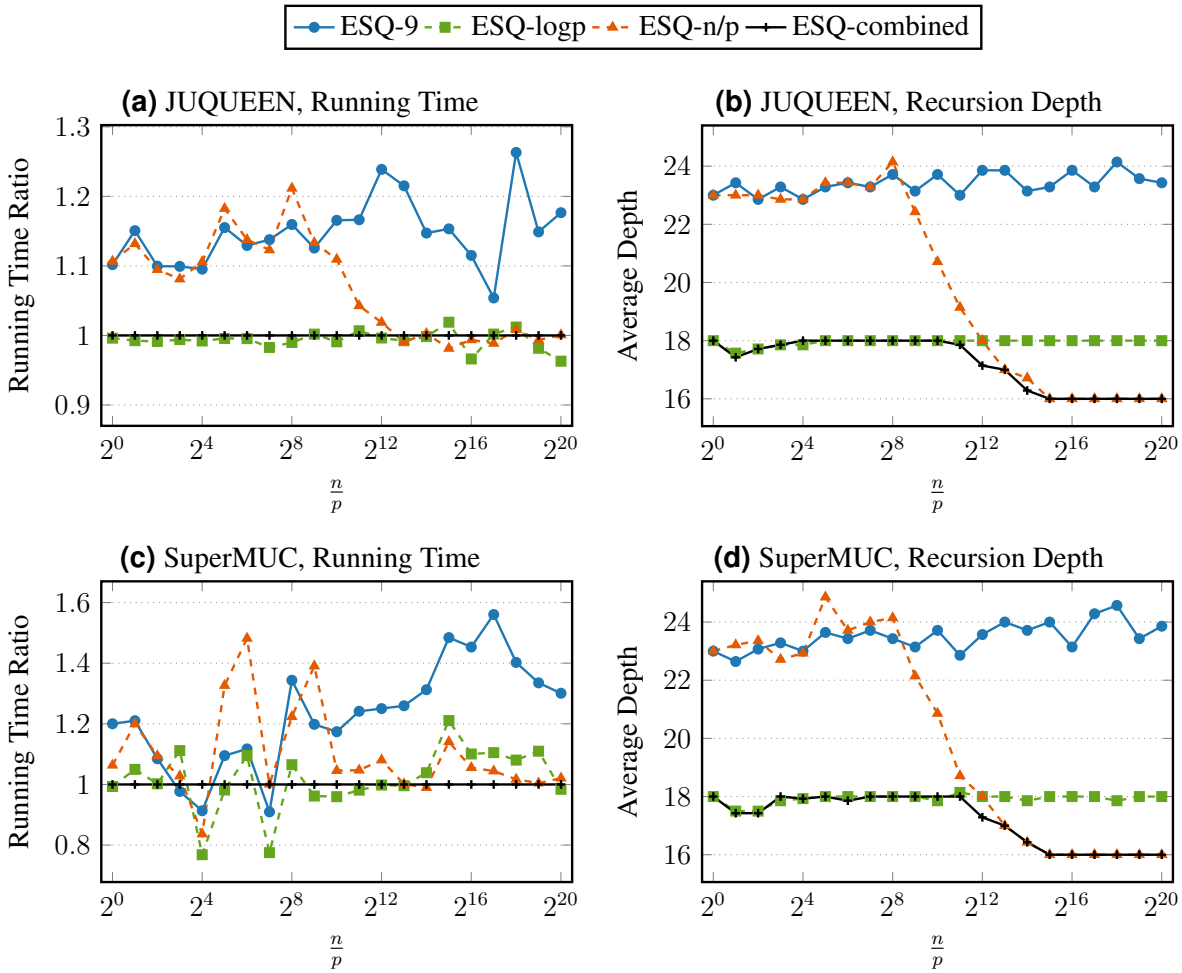

Figure 8.12: We compare different approaches for determining the number of samples in the Pivot Selection of Schizophrenic Quicksort. Figure (a) and (c) show the running time ratio of all approaches compared to the combined approach. Figure (b) and (d) show the average recursion depth.
Figure (a) and Figure (b) show the results on 32 768 cores on the JUQUEEN, Figure (c) and Figure (d) show the results on 32 768 cores on the SuperMUC.

for all strategies and inputs. ESQ-combined is at slower than the other strategies by a factor of at most 1.05, excluding two irregular slow results on the SuperMUC. Because the combined strategy is more robust and at most slightly slower than the other strategies, we consider it the best strategy.

8.3.2 RQuick

We have integrated our communication library into the implementation of Robust Hypercube Quicksort by Axtmann [2] (RQuick). RQuick uses only the point-to-point communication provided by our library. All collective operations are manually implemented. The subgroups that occur in Hypercube Quicksort are not dependent on the input and thus known before the algorithm is executed. In the previous implementation, all MPI communicators were created before the algorithm was executed the first time. The communicators were then reused for multiple executions of the algorithm. By integrating our library, the cost of the communicator creation can be nearly eliminated. We evaluate the running time of RQuick with our library. We com-

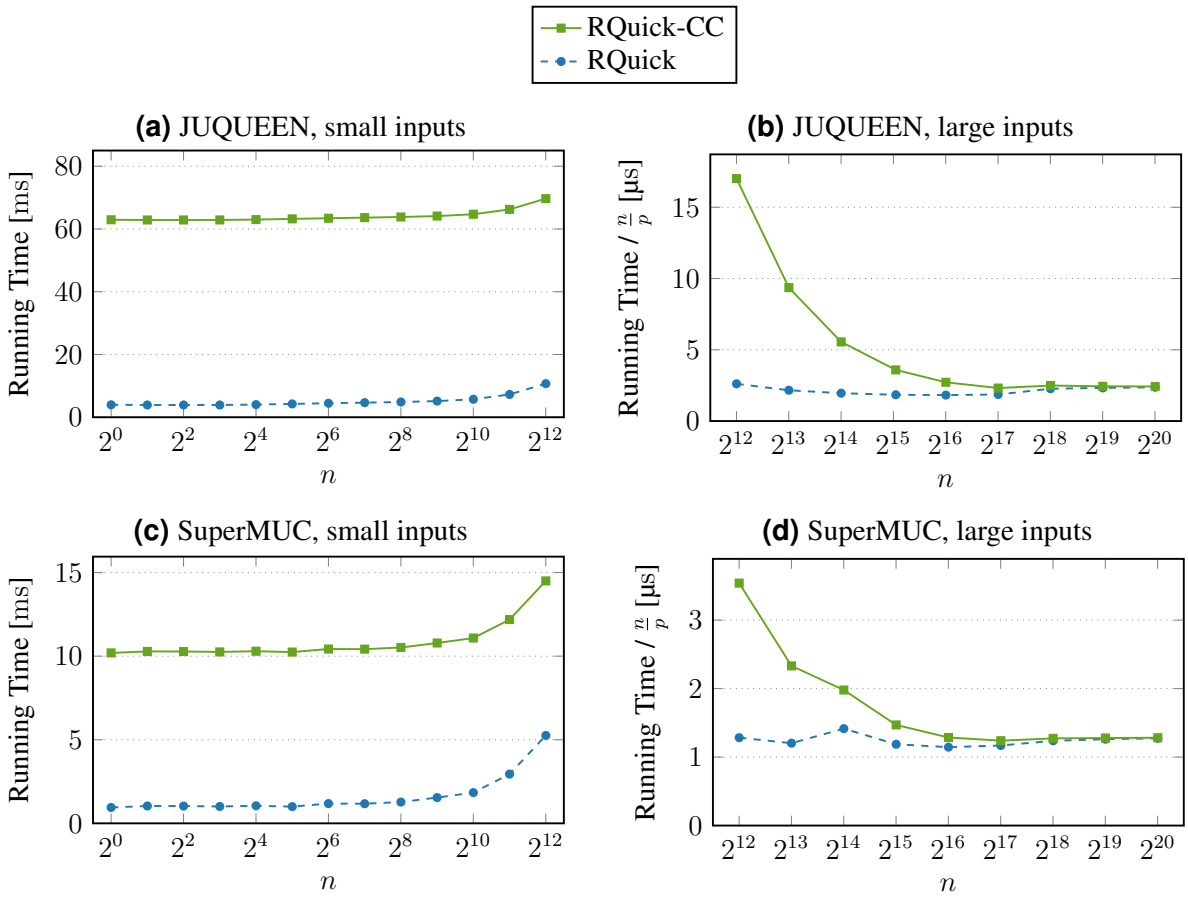

Figure 8.13: Figure (a) and Figure (b) show the running times of RQuick with and without MPI communicatior creation on 16 384 cores on the JUQUEEN. Figure (c) and Figure (d) give the running times on 16 384 cores on the SuperMUC.

pare this to the running time of RQuick plus the running time of the communicator creation (*RQuick-CC*). For the running time of the communicator split, we use our measurements from Section 8.2.2 for which we split the global communicator recursively. We compare only the running times of one execution of RQuick. If RQuick is executed multiple times on the same group, the communicators do not have to be created again. The average running time of one execution of RQuick-CC would decrease for multiple executions.

Figure 8.13a and Figure 8.13b give the running times on the JUQUEEN on 16 384 cores. RQuick outperforms RQuick-CC by a factor of 15 for the smallest input and by a factor of 1.02 for the largest input. As expected, the impact of the communicator creation on the total running time is smaller for large inputs. For $\frac{n}{p} = 2^{10}$, RQuick is still faster than RQuick-CC by a factor of 11.

Figure 8.13c and Figure 8.13d depict the running times on the SuperMUC on 16 384 cores. The results are similar to the results on the JUQUEEN. RQuick outperforms RQuick-CC by a factor of 10 for the smallest input and by a factor of 1.007 for the largest input.

The results show that we improved the performance of RQuick by integrating our communication library for small and medium inputs. In the following section we compare RQuick with ESQ. We use the implementation that integrates our communication library for all measurements.

8.3.3 Comparison of ESQ and RQuick

We now present the running times of ESQ and RQuick on the JUQUEEN and on the SuperMUC. The input consists of uniformly distributed random elements. Figure 8.14a and Figure 8.14b depict the running times on 265 144 cores on the JUQUEEN. RQuick outperforms ESQ by a factor of 3 for smallest input and by a factor of 1.7 for the largest input. Note that the running time of both RQuick and ESQ is nearly constant for all small input sizes ($\frac{n}{p} \leq 2^8$). For such inputs, the overhead portion of the communication dominates the running time. Figure 8.15a gives the running time ratios of ESQ compared to RQuick on 16 384, 65 536 and 265 144 cores. For small inputs, RQuick outperforms ESQ by a factor of around 4 for all numbers of cores. For the largest input, RQuick outperforms ESQ by a factor of 1.4 on 65 536 cores and by a factor of 1.55 on 16 384 cores. On lower numbers of cores, ESQ performs closer to RQuick. We assume that the local work (sorting, partitioning) has a similar running time for both algorithms. The reason why ESQ is slower than RQuick is that the communication is that the communication is slower. Because the amount of communication decreases on lower numbers of cores, the running time ratio of ESQ compared to RQuick decreases.

Figure 8.14c and Figure 8.14d depict the running times on 65 536 cores on the SuperMUC. RQuick outperforms ESQ by a lower factor than on the JUQUEEN. For the smallest input, ESQ is slower than RQuick by a factor of 2.7. For the largest input, ESQ and RQuick have a nearly identical running time (less than factor 1.05). Note that the running time per element of ESQ increases by a factor of 2.3 if we go from $\frac{n}{p} = 2^{13}$ to $\frac{n}{p} = 2^{14}$. We showed in Section 8.3.1 that the communication time increases because of the network of the SuperMUC. For $\frac{n}{p} = 2^{12}$, ESQ has a nearly identical running time as RQuick. We assume that if the network would not increase the communication time, ESQ might be able to outperform RQuick for larger inputs. Figure 8.15a gives the running time ratio of ESQ compared to RQuick on 16 384, 65 536 and 265 144 cores. The ratio on 65 536 cores is less stable than on fewer cores. We believe that the reason for this is the network of the SuperMUC because we do not observe similar tendencies on the JUQUEEN.

Figure 8.16a depicts the running time ratio of the slowest and fastest measurement of ESQ on 262 144 on the JUQUEEN, Figure 8.16b gives the running time ratio of RQuick. The median of the running times of ESQ is at most faster than the maximum by a factor of 1.05 and at most slower than the minimum by a factor of 1.05. The median of the running times of RQuick is at most faster than the maximum by a factor of 1.01 and at most slower than the minimum by a factor of 1.02. The running times of RQuick are more consistent than the running times of ESQ. However, a maximum derivation of 5% from the median is not unexpectedly high.

Figure 8.16c and Figure 8.16d show the running time ratio of ESQ and RQuick on 65 536 cores on the SuperMUC. The running time ratio of the maximum running times for small and medium inputs is much higher than on the JUQUEEN for both ESQ and RQuick. The maximum running time is slower by a factor of up to 3.2 for ESQ and by a factor of up to 5.2 for RQuick. We believe that the network of the SuperMUC is less reliable than the network of the JUQUEEN and thus communication is sometimes much slower than on average.

We run experiments with ESQ and RQuick with different input instances on the JUQUEEN. We use eight distributions from Helman et al.[11]. The instance *Uniform* generates uniformly distributed random input on each PE. If not explicitly mentioned, all following random elements are generated by a uniform distribution. The instance *Gaussian* generates random input according to the Gaussian distribution on each PE. We set all input elements to zero for the *Zero* instance. The instance *Bucket Sorted* splits the input on each PE into p buckets. The elements of bucket i are random values from the range $[i\frac{2^{31}}{p}, (i+1)\frac{2^{31}}{p} - 1]$. The instance *g-Group* first di-

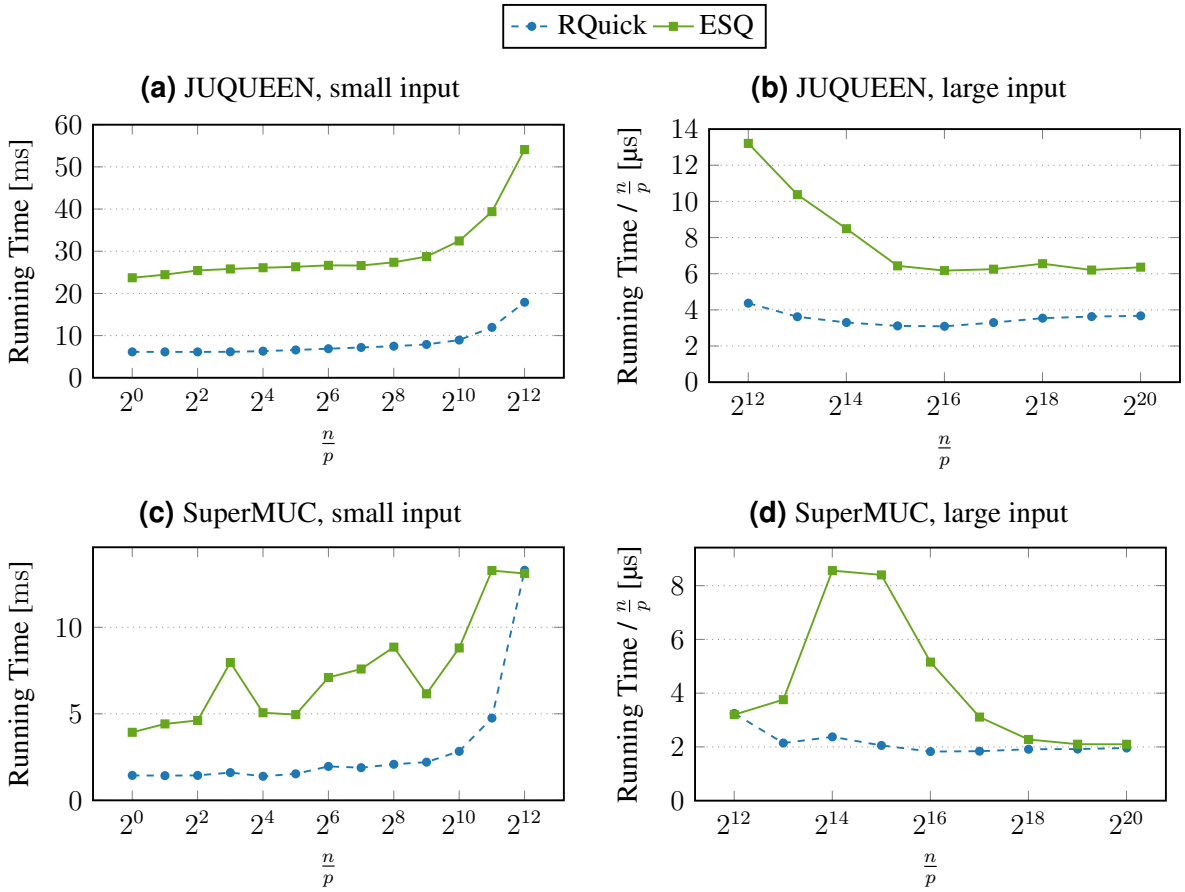

Figure 8.14: Figure (a) and (b) show the running time for Uniform input on 265 144 cores on the JUQUEEN. Figure (c) and (d) give the running time on 65 536 cores on the SuperMUC.

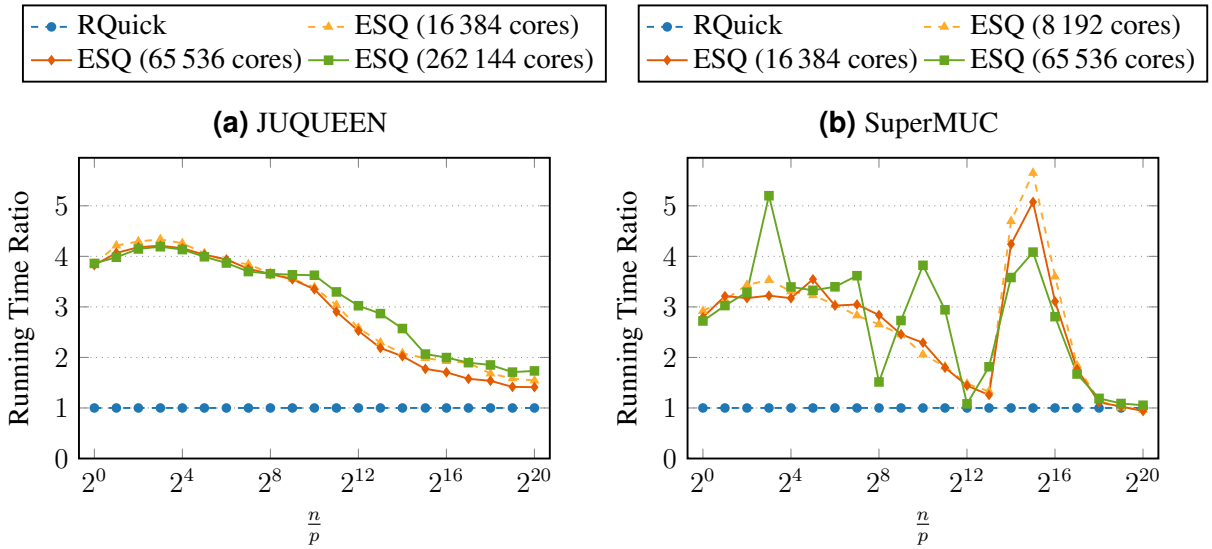

Figure 8.15: Figure (a) shows the running time ratio of ESQ compared to RQuick on different numbers of cores on the JUQUEEN. Figure (b) gives running time ratios for different numbers of cores on the SuperMUC.

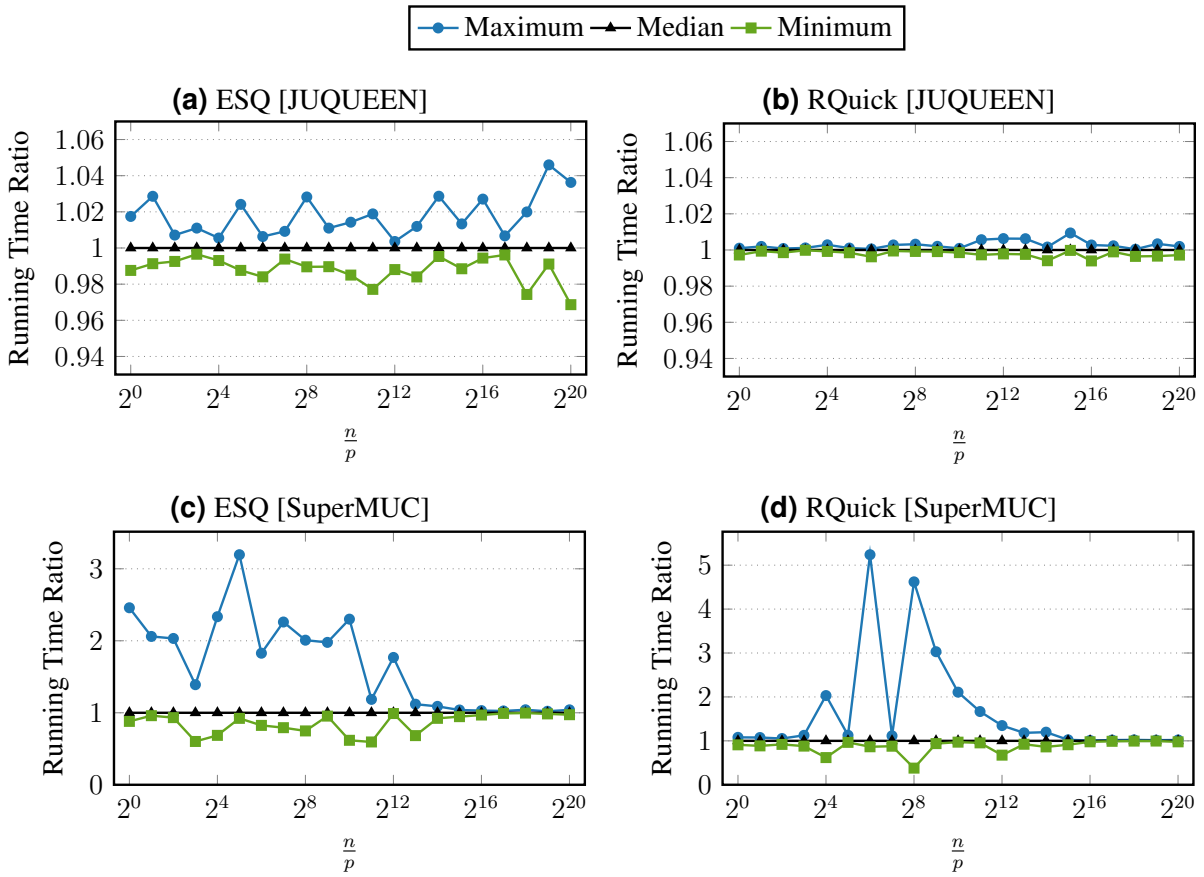

Figure 8.16: We show the running time ratio of the fastest and slowest measurement compared to the median of all measurements. Figure (a) depicts the results for ESQ, figure (b) the results for RQuick on 262 144 cores on the JUQUEEN. Figure (c) gives the results for ESQ, figure (d) for RQuick on 65 536 cores on the SuperMUC.

vides the processors into groups of consecutive processors of size g, where g can be any integer which partitions p evenly. For each group j, the input is divided into buckets with $\frac{n}{pg}$ elements each. The elements from bucket k are randomly picked from the range $[(((jg + \frac{p}{2} + k - 1) \mod p) + 1)\frac{2^{31}}{p}, (((jg + \frac{p}{2} + k) \mod p) + 1)\frac{2^{31}}{p} - 1]$. The instance *Staggered* sets the input on PE i to random values from the range $[(2i + 1)\frac{2^{31}}{p}, (2i + 2)\frac{2^{31}}{p} - 1]$ if $i < \frac{p}{2}$, else it sets the input to random values from the range $[(2i - p)\frac{2^{31}}{p}, (2i - p + 1)\frac{2^{31}}{p} - 1]$. For the instance *Deterministic Duplicates*, we set all elements on the first $\frac{p}{2}$ PEs to be $\frac{\log n}{2^0}$, all elements on the next $\frac{p}{2^2}$ PEs to be $\log \frac{n}{2^1}$ and so forth. At the last PE, we set the first $\frac{n}{2^1 p}$ elements to be $\log \frac{n}{2^0 p}$, the next $\frac{n}{2^2 p}$ elements to be $\log \frac{n}{2^1 p}$ and so forth. For the instance *Randomized Duplicates*, we first select an integer value r. Each PE fills an array T of size r with random values between 0 and r-1. Let S be the sum of all values. The first $\frac{T[1]}{S}\frac{n}{p}$ elements are set to a random value between 0 and r-1, the next $\frac{T[2]}{S}\frac{n}{p}$ elements are set to another random value between 0 and r-1, and so forth. Furthermore, we use three input instances from Axtmann and Sanders [2]. The instance *Reverse Sorted* generates an input that is reversely sorted. The instance *Mirrored Target* sets the input of PE i to random values between $\frac{2^{31}}{p}(m_i)$ and $\frac{2^{31}}{p}(m_i + 1)$, where m_i is the reverse bit representation of i. For the instance *AllToOne*, we set the first $\frac{n}{p} - 1$ elements on each PE to random values between $p + (p - i)\frac{2^{32}-p}{p}$ and $p + (p - i + 1)\frac{2^{32}-p}{p}$. The last element on PE i is set to the value $p - i$. We do not show results of the instance Gaussian because the running

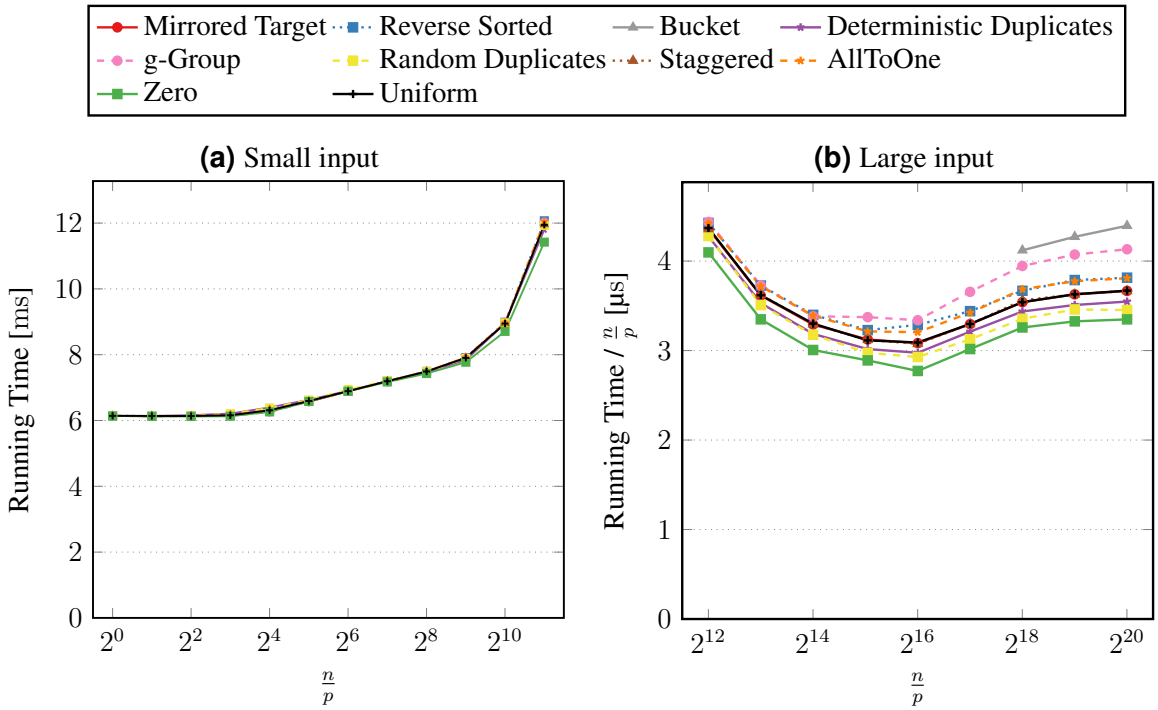

Figure 8.17: We give the running times of RQuick for multiple input distributions on 262 144 cores on the JUQUEEN. Figure (a) depicts the running times for small inputs. Figure (b) depicts the running times for large inputs.

time of Gaussian is identical to Uniform for both ESQ and RQuick for all input sizes.

Figure 8.18 shows the results for ESQ on 262 144 cores on the JUQUEEN. AllToOne is the slowest input instance for medium input sizes. For $\frac{n}{p} = 2^{13}$, AllToOne is 6 times slower than Uniform. However, AllToOne has almost the same running time as Uniform for the smallest and largest input. In the AllToOne, each PE has one small element in its local input. The first PE receives $\min(\frac{n}{p}, p)$ messages in the first recursion of ESQ. For small $\frac{n}{p}$, the number of received messages is low and thus has no noticeable impact on the total running time. If p is constant, the number of received messages increases for larger input sizes. For $\frac{n}{p} > p$, the number of messages does not increase anymore. Because the total running time increases but the communication time for the messages stays constant, the impact of AllToOne is very small for the largest input. Figure 8.19 gives the results for 16 384, 65 536 and 262 144 cores on the JUQUEEN. The running time of the AllToOne is identical to the Uniform for $\frac{n}{p} \geq 2^{17}$ on 16 384 cores and for $\frac{n}{p} \geq 2^{19}$ on 65 536 cores. AllToOne has a smaller impact on the running time on smaller numbers of cores because the number of messages that the first PE receives is at most one message from each PE. For the largest input, the slowest input instances are Mirrored Target and Reverse Sorted. Both instances are slower than Uniform by a factor of about 1.11 on 262 144 cores. The reason is that we exchange all elements in each recursion of ESQ. Zero is the fastest input instance for all input sizes. For the smallest input, the running time of Zero is nearly identical to the running time of the Uniform. For the largest input, Zero is faster than Uniform by a factor of 4.7. The reason is that we do not exchange data in ESQ for the instance Zero. Because the exchange is the slowest part of Schizophrenic Quicksort for large inputs, Zero is significantly faster than Uniform. All other instances are at most faster than Uniform by a factor of 1.35 and at most slower than Uniform by a factor of 1.13.

Figure 8.17 shows the results for RQuick on 262 144 cores on the JUQUEEN. For small inputs ($\frac{n}{p} \leq 2^{10}$), the running time of RQuick is nearly identical for all input instances. For the largest

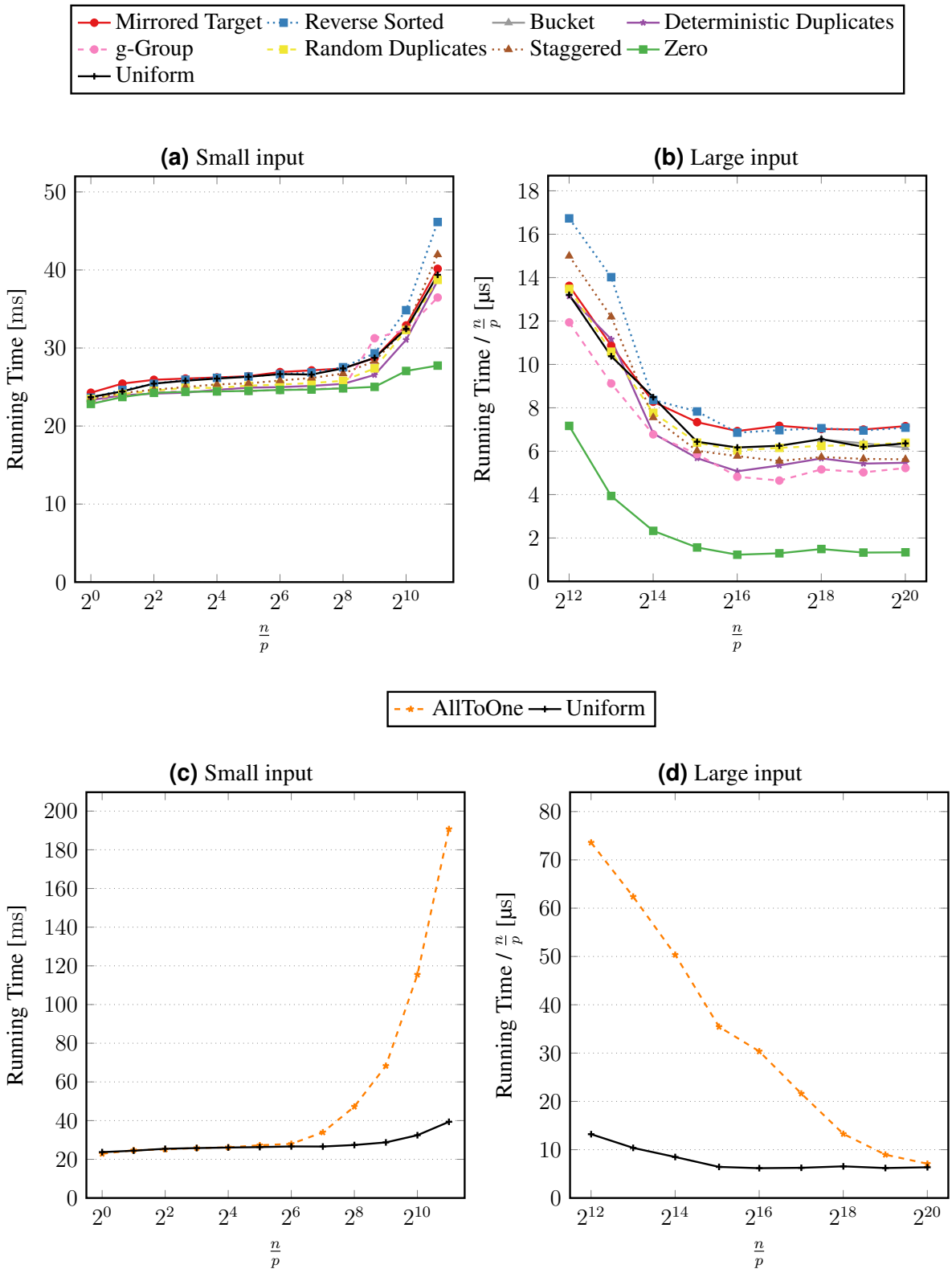

Figure 8.18: Figure (a) and (b) depict the running times of ESQ for multiple input distributions on 262 144 cores on the JUQUEEN. Figure (c) and (d) give the running times for the distributions AllToOne and Uniform.

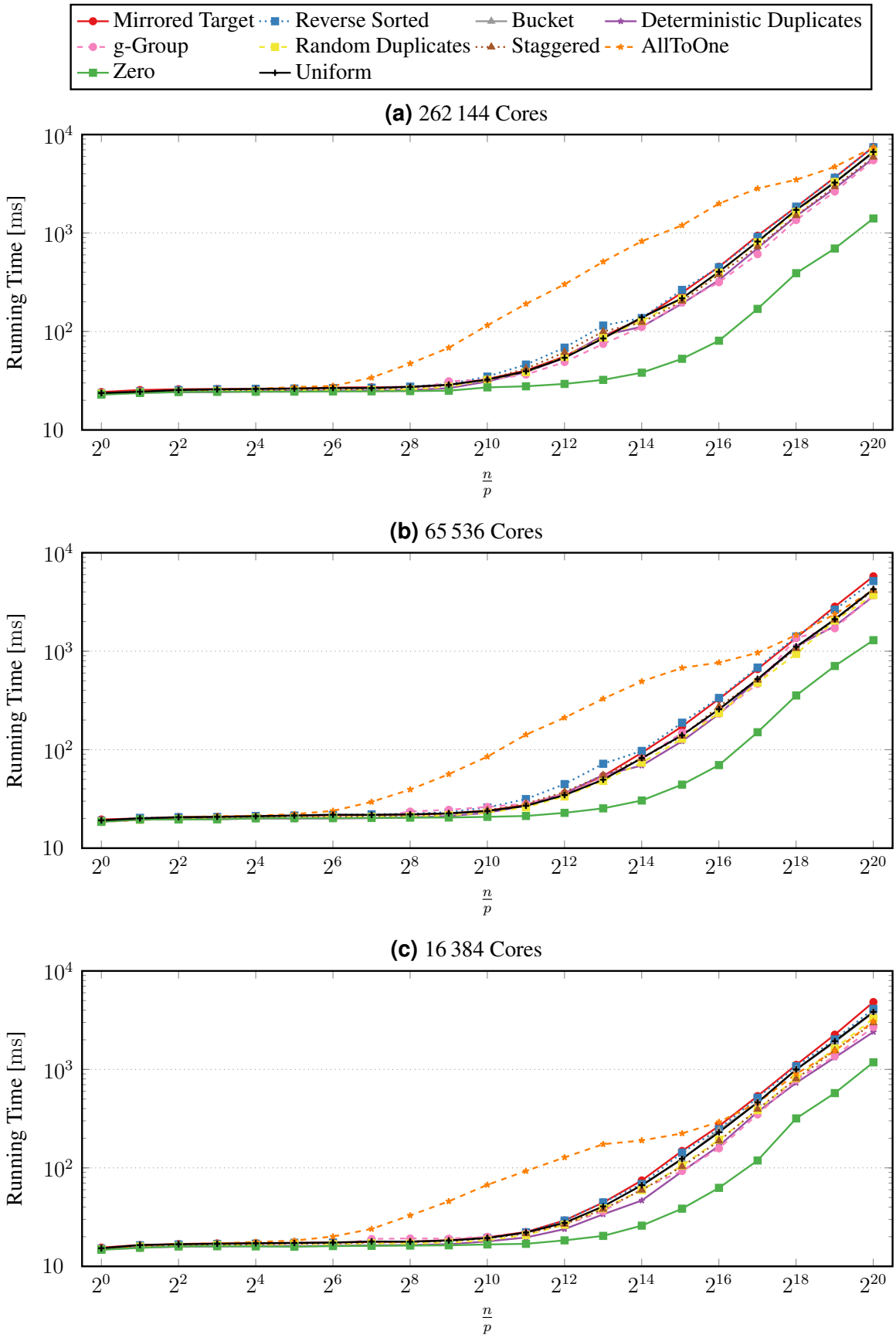

Figure 8.19: Figure (a) depicts the running time of ESQ for multiple input distributions on 262 144 cores on the JUQUEEN. Figure (b) gives the running times on 65 536 cores, Figure (c) gives the runnig times on 16 384 cores.

input, Bucket is the slowest input instance. Bucket is slower than Uniform by a factor of 1.2. Zero is the fastest input instance for the largest input that is faster than Uniform by a factor of 1.1. As expected, RQuick is robust than ESQ if we consider all input instances. However, ESQ is only much slower than Uniform for a single input instance (Uniform). All other input instances only have a comparable impact on ESQ as they have on RQuick. Note that even though ESQ is slower than RQuick for all input sizes with the input instance Uniform, ESQ outperforms RQuick for the largest input with the distribution Zero by a factor of 2.5.

We have shown that ESQ is multiple times slower than RQuick for small inputs. For large inputs, the difference in the running times of ESQ and RQuick decreases. ESQ has a nearly identical running time as RQuick for some input sizes. This indicates that even though RQuick outperforms ESQ overall, ESQ might be faster on different supercomputers. The network on the SuperMUC seems to be less reliable than the network on the JUQUEEN. The variance in the running time of both ESQ and RQuick is very high for some small and medium inputs. The network of the SuperMUC seems to also slow down ESQ for medium inputs. As expected, RQuick is more robust than ESQ. However, ESQ is only not robust against one of the input instances that we tested.

9 Conclusion & Further Work

We have presented a communication library that is based on communicators which can be created in very little time and without communication. The library provides point-to-point communication and five collective operations. We run tests on two supercomputers. Our library has an interface that is very close to MPI. It is thus simple to integrate our library into existing implementations to replace MPI as the communication library.
The time for creating MPI communicators differs strongly on the two systems. Our results show that on the SuperMUC, the operation `MPI_Comm_create` is very inefficient for large groups of PEs and should thus not be used. In comparison, the running time of creating communicators of our library is negligible. On the SuperMUC, the collective operations of our library perform similar to the counterparts of the MPI implementation for most inputs. On the JUQUEEN, the MPI implementation is much faster for some collective operations. The reason is that the network of the JUQUEEN supports the collective operations of MPI in the hardware and thus speeds up the execution. When we split a communicator and then perform the operation Broadcast multiple times, our library outperforms MPI even for 50 Broadcasts with small or medium inputs. A disadvantage of the library is that communication is not truly encapsulated on one communicator. If we send two messages on different communicators from PE i to PE j, we have to use different tags to identify the messages.
We presented an efficient implementation of Schizophrenic Quicksort (ESQ) that uses our communication library. By using our communication library instead of MPI, we speed up ESQ by a factor of up to 40. For small inputs, the collective operations dominate the total running time of ESQ. For large inputs, the exchange of data dominates the running time. ESQ is robust against all but one of the tested input distribution. We also integrated the communication library in RQuick and thus increased the performance by a factor of up to 15. In our experiments, the algorithms ESQ and RQuick behaved different on the two supercomputers. ESQ is not able to outperform RQuick for small inputs on both supercomputers. For large inputs, RQuick outperforms ESQ by a factor of less than 2 on the JUQUEEN. On the SuperMUC, the running times of ESQ and RQuick are nearly identical for the largest input. An advantage of ESQ is that it can be executed on any number of cores while RQuick has to be executed on 2^n cores, $n \in \mathbb{N}$. In the most unfavorable case (2^n-1 cores), RQuick can thus only utilize a bit more than half of the available PEs.
Our experimental results show that the MPI communicator creation does not scale well for large numbers of PEs. On the SuperMUC, the running times of the operation `MPI_Comm_create` are much slower than the running times of the operation `MPI_Comm_split`. If an implementation uses MPI communicators, there has to be additional effort to evaluate the communicator split.

The most obvious way to extend the communication library is by providing additional collective operations, e.g. Scatter, All-Gather, All-Reduce or All-to-All. The presented algorithms for the collective operations are optimized for small inputs. The library could be extended by algorithms that perform better for large inputs. If multiple algorithms for a collective operation are available, the optimal algorithm could then be chosen dependent on the input size.
The implementation of Schizophrenic Quicksort could be improved in different ways. The

phases Pivot Selection and Exchange Calculation are slower than we expected from theory. These phases could be measured in detail to identify possible optimizations. Maybe a different random generator could increase the performance of the Pivot Selection. The current implementation requires an equal input size on all PEs. It could be extended to allow varying input sizes, e.g. by redistributing the input evenly in the beginning and then execution the current implementation. ESQ is not robust against all input instances. A possible solution is to randomly reshuffle the data in the beginning like in RQuick. This does however increase the running time for uniformly distributed input.

Bibliography

[1] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 13–23. ACM, 2015.

[2] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. *arXiv preprint arXiv:1606.08766*, 2016.

[3] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[4] Guy E. Blelloch. Prefix sums and their applications. 1990.

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

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[7] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011.

[8] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In *European MPI Users' Group Meeting*, pages 282–291. Springer, 2011.

[9] W. Donald Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

[10] Jonathan C. Hardwick. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments*, pages 105–114. Citeseer, 1996.

[11] David R. Helman, David A. Bader, and Joseph JáJá. A randomized parallel sorting algorithm with an experimental study. *journal of parallel and distributed computing*, 52(1):1–23, 1998.

[12] Tobias Heuer. Schizophrenic Quicksort, 2016.

[13] Charles A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[14] Charles A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[15] Torsten Hoefler, J Squyres, George Bosilca, Graham Fagg, Andrew Lumsdaine, and Wolfgang Rehm. Non-blocking collective operations for MPI-2. *Open Systems Lab, Indiana University, Tech. Rep*, 8, 2006.

[16] Torsten Hoefler, Jeffrey M. Squyres, Wolfgang Rehm, and Andrew Lumsdaine. A case for non-blocking collective operations. In *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ISPA'06, pages 155–164, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] JS Huang and YC Chow. Parallel sorting and data partitioning by sampling. 1983.

[18] Mihai F. Ionescu and Klaus E. Schauser. Optimizing parallel bitonic sort. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 303–309. IEEE, 1997.

[19] Jülich Supercomputing Centre. JUQUEEN: IBM Blue Gene/Q Supercomputer System at the Jülich Supercomputing Centre. *Journal of large-scale research facilities*, 1(A1), 2015.

[20] Leibniz Supercomputing Centre. SuperMUC. `https://www.lrz.de/services/compute/supermuc/systemdescription/`, 2016.

[21] David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Computers*, 28(1):2–7, 1979.

[22] David Padua. *Encyclopedia of Parallel Computing*, volume 4. Springer Science & Business Media, 2011.

[23] Sanguthevar Rajasekaran and Sandeep Sen. Random sampling techniques and parallel algorithms design. *Synthesis of Parallel Algorithms*, pages 411–451, 1993.

[24] John H. Reif and Leslie G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM (JACM)*, 34(1):60–76, 1987.

[25] P. Sanders and T. Hansch. Efficient massively parallel quicksort. In Bilardi, G. and Ferreira, A. and Luling, R and Rolim, J., editor, *Solving Irregulary Structured Problems in Parallel*, volume 1253 of *Lecture Notes in Computer Science*, pages 13–24, 1997. 4th International Symposium on Solving Irregularly Structured Problems in Parallel.

[26] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient random sampling-parallel, vectorized, cache-efficient, and online. *arXiv preprint arXiv:1610.05141*, 2016.

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

[28] Jop F. Sibeyn. Improving the performance of collective operations in MPICH. In *European Conference on Parallel Processing*, pages 389–398. Springer, 1997.

[29] Jop F. Sibeyn. Sample sort on meshes. In *European Conference on Parallel Processing*, pages 389–398. Springer, 1997.

[30] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient algorithms for parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20(2):95–131, 1991.

[31] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[32] Hari Sundar, Dhairya Malhotra, and George Biros. Hyksort: A new variant of Hypercube Quicksort on distributed memory architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 293–302, New York, NY, USA, 2013. ACM.

[33] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 3. IEEE Computer Society, 2000.

[34] Bruce Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299, 1987.