Master's Thesis

Distributed Duplicate Removal

Sebastian Schlag

10th July 2013

Advisors: Prof. Dr. Peter Sanders
Dipl. Inform. Ingo Müller

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

Name:	Sebastian Schlag
Student ID:	1612224
Editing Time:	6 Months
Pursued Degree:	Master of Science

Contact Information:

Author:
Sebastian Schlag
mail@sebastianschlag.de

University:
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
Phone: +49 721 608-0
Fax: +49 721 608-44290
E-Mail: info@kit.edu
`http://www.kit.edu`

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.

Buchklingen, den 10. Juli 2013

Abstract

The distributed duplicate removal problem is concerned with the detection and subsequent elimination of all duplicate elements in a given multiset that is distributed over several computers connected by a network.

Sanders et al. [48] outline a communication efficient algorithm solving this problem. It uses distributed compressed single shot Bloom filters to identify distinct elements using minimal communication effort. The filter however produces false positive results. Thus, all elements passing the filter are post processed using a traditional hash-based distributed duplicate removal algorithm in order to distinguish real duplicates from false positives.

This thesis complements the theoretical analysis with an experimental evaluation. We transform the high-level description of the algorithm into an efficient implementation than runs on a shared-nothing system and exploits the shared memory parallelism capabilities of its nodes for all computationally intensive operations.

The results of our analysis substantiate the benefits predicted by theory. Our implementation outperforms the best-suited traditional algorithm up to the point where the input data contains 50% duplicates. When executed on datasets that contain less than 10% duplicates, our implementation achieves a communication volume that is more than one order of magnitude smaller than that of its competitor.

Zusammenfassung

Ziel der verteilten Duplikaterkennung ist die Identifikation von Elementen, welche mehrfach in einer großen, über mehrere Rechenknoten verteilten Datenmenge vorkommen.

Sanders et al. [48] präsentieren einen verteilten Algorithmus, welcher dieses Problem in einer besonders kommunikationseffizienten Art und Weise löst. In einer Vorverarbeitungsphase werden mit Hilfe eines verteilten, platz-effizienten Bloom Filters zunächst möglichst viele distinkte Elemente als solche identifiziert und somit die Gesamtmenge der noch zu betrachtenden Elemente stark reduziert. Da hierbei jedoch auch falsch positive Ergebnisse auftreten, müssen alle als potentiell nicht distinkt erkannten Elemente in einer zweiten Phase noch einmal überprüft werden. Hierzu wird ein klassischer Hash-basierter Algorithmus zur verteilten Duplikaterkennung angewendet.

Die vorliegende Arbeit ergänzt die theoretische Analyse durch eine praktische Evaluation. Wir erarbeiten hierzu eine effiziente Implementierung für Shared-Nothing Systeme. Besonders rechenintensive Schritte des Algorithmus werden zusätzlich durch Shared-Memory-Programmierung innerhalb eines Knotens parallelisiert.

Die Ergebnisse unserer experimentellen Untersuchung untermauern die durch die Theorie vorhergesagten Vorteile des Algorithmus. Unsere Implementierung ist signifikant schneller als der am besten geeignete klassische Ansatz solange die Eingabedaten zu weniger als 50% aus Duplikaten bestehen. Wird der Algorithmus auf Datensätzen ausgeführt, die zu weniger als 10% aus Duplikaten bestehen, so ist das gesamte Kommunikationsvolumen zudem mehr als eine Größenordnung kleiner als das des klassischen Konkurrenten.

Contents

List of Figures

List of Algorithms

List of Tables

1. Introduction

The existence of duplicate records is a severe problem within databases. These duplicate records may be caused by errors during data entry, inconsistent conventions for recording information, unstandardized data formats, or missing foreign key constraints.

Duplicates have to be *identified* and *removed* in order to achieve the data quality required by Online Analytical Processing (OLAP) and data mining applications [66]. This issue becomes even more severe, whenever different datasources have to be integrated to form a data warehouse. The problem of *data heterogeneity*, i.e., *different* representations of the same real-world objects across multiple heterogeneous databases, is therefore thoroughly studied in research literature [11]. In this context, the identification and removal of duplicate database entries is referred to as *duplicate record detection*, *record linkage* or *record matching* [18].

The *distributed duplicate removal* problem, which is at the heart of this thesis, is concerned with the detection and the subsequent elimination of all duplicate elements within a given *multiset* that is distributed over p processing elements (PEs). In contrast to the previously introduced record linkage / duplicate record detection problem, we do not deal with data heterogeneity. Instead, we assume the contents of *one* database to be distributed over the PEs. Thus, a duplicate in our case does not refer a different representation of an already existing real world object, but rather to an *equal* representation, i.e. the *exact same* record exists on more than one system - for example due to inconsistencies of the dataset or a projection.

We are therefore concerned with duplicate removal as a fundamental operation in query processing, as relational theory imposes the requirement that a relation must not contain any duplicate records, i.e., it has to be a *set* in the mathematical sense. Consider, for example, the projection operation $\pi_{c_2,...,c_n}(R)$ on a relation R with columns $c_0, c_1, ..., c_n$. After R is reduced to the subset of columns $c_2, ..., c_n$, it is necessary to apply a *distinct* operator to remove any now duplicate tuples that differed only in the projected columns c_0, c_1.

Duplicate removal is a computationally expensive operation because it is necessary to perform pairwise comparisons on the input dataset. However, it can be done efficiently (in terms of *memory consumption* and *execution time*) using sort- or hash-based approaches, as long as the whole dataset is stored on one single system. While today's machines can have several hundred GB up to a few TB of RAM and several TB of disk space, at some point a single machine will not be able to handle the data anymore - solely due to hardware

limitations. Under these circumstances, the only option is to scale-out, i.e., to use multiple machines connected by a high-speed network. In this case however, the duplicate removal problem becomes more challenging, since the traditional sort- or hash-based approaches quickly become unfeasible due to the huge load they put on the network.

In this distributed setting, the total *communication volume* has a significant impact on the overall execution time of an algorithm. Thus it is necessary to develop *communication-efficient* algorithms that minimize the communication volume. Lint and Agerwala [33] note that the complexity of communication depends on four major criteria: the *number of processors*, the *structure* of the network connecting these processors, the initial *distribution* of the input data as well as the *algorithm* used to solve the specific problem at hand. In the next section, we will fix the first *three* of these factors by establishing our problem setting.

1.1 Problem Setting

In this thesis, we consider the duplicate removal problem in the context of parallel, shared-nothing, in-memory database systems.

A *parallel database system* can be defined as *one single* database system that is implemented on top of a *tightly coupled* parallel computer. In contrast, a *distributed database system* consists of *several, locally autonomous* database systems, which are *loosely coupled* over a computer network [61].

Shared-nothing refers to the underlying machine architecture and is one of three basic architectures of parallel database systems (see Figure 1.1). In a shared-nothing system, both memory and disks are private to each processor, whereas a *shared-memory* system shares memory modules and disks among the processors. Thus each processor has access to every memory as well as to every disk location. A *shared-disk* system resides in the middle of these two extremes: Processors have their own private memory and only share the disks between each other. For a detailed discussion about the costs and benefits of each of these approaches in the context of database systems we refer to [54, 16, 68].

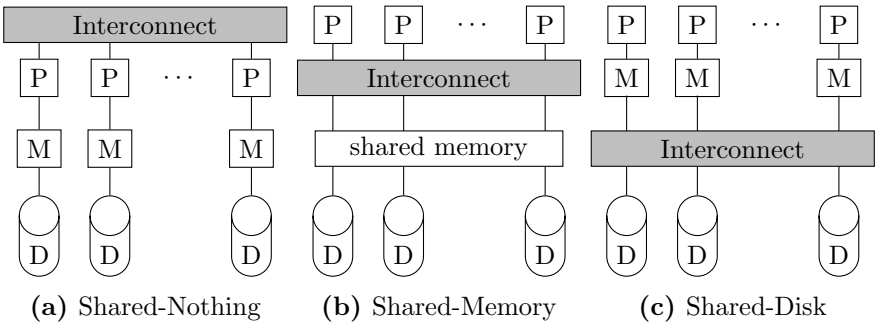


Figure 1.1: Architectures of parallel database systems.

Finally we restrict our focus to *in-memory* databases [51, 19, 10]. As a consequence, our implementation is concerned with main-memory data structures and corresponding algorithms. This puts us in the context of in-memory databases such as SAP HANA.

1.2 Problem Statement

Our work builds upon previous research efforts concerned with the communication-efficient detection of duplicates within a large, distributed multiset of data elements. In order to minimize the total amount of data that has to be transferred over the network, Sanders

et al. [48] outline a multi-pass filtering algorithm that uses *distributed* compressed *single shot Bloom filters* (dSBFs) to identify most *distinct* elements with minimal communication effort. Only elements that passed this filter cascade actually have to be transferred over the network and processed in a second phase. The communication volume of the filtering phase is minimized by performing batched insertions into the distributed filter data structure. By reordering insertion operations via sorting, it is possible to compress each batched insertion message using an information-theoretically optimal compression scheme. A detailed theoretical analysis shows that the communication volume of the filtering phase is *sublinear* in the input size.

It is the goal of this thesis to complement this theoretical analysis with an experimental evaluation that analyzes the practical impact of communication efficiency in the context of distributed duplicate removal. We therefore transform the high-level description of the multi-pass dSBF-based duplicate removal algorithm into an actual implementation that runs on a parallel, shared-nothing compute cluster. As each node of this system is a shared-memory parallel computer itself, the implementation is expected to exploit these shared-memory parallelism capabilities as well.

1.3 Contributions

Our key contributions can be summarized as follows:

Duplicate Removal using Distributed Single Shot Bloom Filters: We present an implementation of the distributed duplicate removal algorithm [48]. It is capable of both single- and multi-pass dSBF-filtering and consistently outperforms a traditional hash-based repartitioning algorithm. The significant performance advantages substantiate the theoretical benefits of designing *communication-efficient* algorithms. In summary, our experimental evaluation leads to the following observations:

- Both variants outperform the traditional hash-based algorithm up to the point where the dataset contains 50% duplicates. For duplicate-free datasets, the *total* running time of our algorithms is smaller than the time it takes the competitor to *communicate* the input data.
- The communication volume of our algorithms is more than one order of magnitude smaller than that of the traditional algorithm for datasets containing less than 10% duplicates.
- Our experiments were performed on a system with a high-performance interconnect. We therefore expect the gains of dSBF-based duplicate removal to be even larger on systems that employ less sophisticated network hardware.

All-to-All Personalized Communication: Collective communication operations are essential to both our algorithm as well as the traditional hash-based repartitioning algorithm. We therefore evaluate the implementation of the all-to-all personalized communication operation in OpenMPI and reveal several deficiencies along with their impact on the overall performance of the algorithm. Motivated by our analysis, we contribute an implementation of the 1-Factor algorithm [47] that can be used as a drop-in replacement for the respective library algorithm. Our implementation consistently outperforms the library counterpart for large message sizes. We further present a variant of this algorithm that *compresses* each message in parallel using Golomb compression. Furthermore, encoding, transmission, and decoding of each message are interleaved in a *pipelined* fashion to overlap computation and communication as much as possible. This operation is a key component of the dSBF, as it is used for batched insertions/queries on the filter.

Parallel Radix Sort In order to improve the initial sorting phase of our algorithm, we contribute a parallel radix sort implementation, which is based on the concepts of Wassenberg and Sanders [64]. Our implementation consistently outperforms the parallel sorting routine of the GNU C++ library for 32- as well as 64-bit integer keys. We plan to integrate this sorting routine into the SAP HANA database.

1.4 Outline

Chapter 2 starts with a review of traditional local and distributed duplicate removal algorithms. As the algorithm of Sanders et al. [48] builds upon a compressed, distributed Bloom filter replacement, we briefly present the employed compression scheme by summarizing the key results of Moffat and Turpin [37] and give a short introduction to Bloom filters. Having discussed all relevant prerequisites, we proceed with the presentation of the *distributed single shot Bloom Filter* and discuss its usage as a single- and multi-pass preprocessing stage in the distributed duplicate removal algorithm.

Chapter 3 then transforms these high-level ideas into a parallel algorithm and discusses the major design decisions regarding the actual implementation. We substantiate each decision with a dedicated experimental evaluation.

Putting all pieces together, we evaluate our distributed duplicate removal implementation in Chapter 4. In order to prove the practical impact of a *communication-efficient* implementation, we implemented a classical hash-based duplicate removal algorithm. The chapter therefore starts with relevant implementation details concerning this competitor before continuing with the actual experimental evaluation.

We close this thesis in Chapter 5 with a summary of our efforts as well as a discussion of current limitations. We also give several proposals for future work based on the insights gained over the course of this thesis.

2. Preliminaries and Related Work

We start by giving an overview about traditional duplicate removal algorithms on single-node and distributed systems. As the distributed duplicate removal algorithm of Sanders et al. [48] uses *compressed* distributed Bloom filters to identify distinct elements in a preprocessing phase, we introduce the employed compression method in Section 2.2 and also give a brief introduction to Bloom filters in Section 2.3.

Having discussed these preliminaries, we then present the *distributed* compressed *single shot Bloom filter* and explain how it is used in the distributed duplicate removal algorithm.

2.1 Traditional Duplicate Removal Algorithms

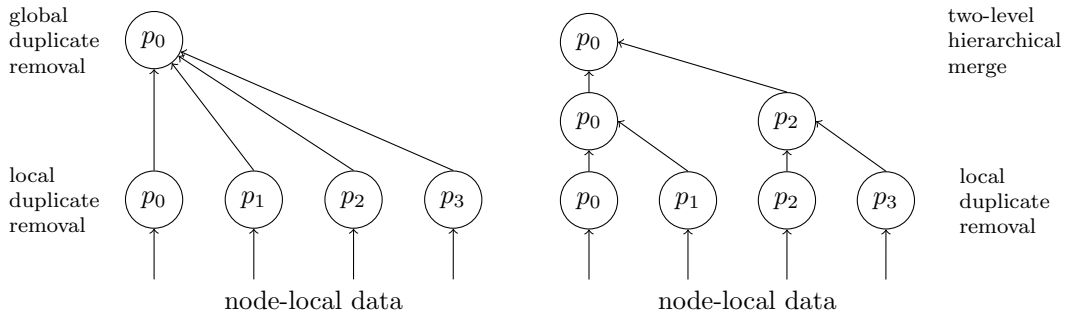
This section gives a brief overview over related work concerning sequential and parallel/distributed algorithms for duplicate detection and removal mainly motivated in the context of database systems.

Sequential/Single-system Algorithms

Traditionally, duplicate removal is a two-step process: First, the input relation is sorted, followed by a scan of the sorted data [1, 4]. Because sorting groups equal items together, duplicates can be easily identified and eliminated during the scan pass by comparing adjacent tuples. Huang and Langston [26] give a stable, in-place duplicate extraction algorithm on sorted records that requires $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ additional space.

The traditional approach can be improved by detecting and removing duplicates early as part of the sorting operation [22]. Munro and Spira [39] devise an information-theoretic lower bound on the number of comparisons that is required to sort a multiset with such an early duplicate removal technique. A quicksort algorithm for equal keys is given by Wegner [65]. Farzan et al. [20] adapt the cache-oblivious lazy funnelsort algorithm of Brodal and Fagerberg [7] to perform early duplicate elimination. A hash-based approach that requires linear time on the average and $\mathcal{O}(1)$ extra space is presented in [59].

All of these contributions consider duplicate removal in main memory. Databases however often reside in external memory (EM). Therefore several publications devise IO efficient external memory algorithms. Bitton and DeWitt [4] improve on an external merge-sort and subsequent scan approach by presenting and analyzing a modified external merge-sort procedure that performs early duplicate elimination. Teuhola [58] develops a two-pass



(a) `Merge-All` **(b)** `Hierarchical-Merge`

Figure 2.1: Traditional parallel duplicate removal algorithms with 4 nodes. `Merge-All` was first used in Gamma [15]. `Hierarchical-Merge` was proposed by Bitton et al. [5] to reduce the load on the root node.

external duplicate deletion algorithm based on hashing. The algorithm first generates mutually duplicate-free subsets that fit into main memory and deletes the duplicates from each of these subsets in the second pass. Helmer et al. [24] propose a new external hash-based algorithm for duplicate elimination that outperforms several sort-based algorithms as well as the hybrid-hashing approach of DeWitt et al. [17]. Larson [31] analyzes the benefits of applying early duplicate elimination to six different EM algorithms, two of which are based on repeatedly scanning the input, two on sorting as well as two on hashing.

Lehman and Carey [32] compare the performance of a sort-scan approach [5] to the hybrid-hashing algorithm in the context of an in-memory database and identify the hashing algorithm as the clear winner. More recently, Graefe [23] proposed an algorithm that combines the ideas of traditional aggregation and duplicate removal algorithms based on sorting and hashing into one, universally applicable algorithm.

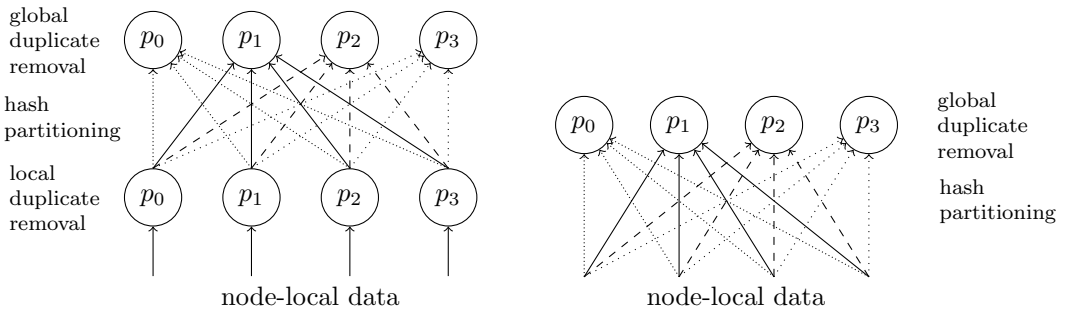
This thesis focuses on distributed duplicate removal. We therefore do not consider datasets that contain local, i.e., intra-node duplicates. However, in such cases one of the algorithms presented in this section could be used in an additional preprocessing phase in order to remove any local duplicates.

Parallel/Distributed-system Algorithms

In parallel query processing literature, duplicate removal algorithms are often adaptations of parallel algorithms for aggregate processing [56] due to the algorithmic similarity of both operations. Most of the following algorithms consist of two distinct steps [22]: *local duplicate removal* and *global duplicate removal*. First each node eliminates duplicates on its local partition of the relation. In the second step the partial results are then merged - thereby removing the inter-node duplicates.

In the traditional `Merge-All` approach, which was first implemented in the Gamma database machine [15], merging is done by a single node that is chosen as the coordinator (see Figure 2.1a). This approach only works well if the number of duplicates is high, otherwise the coordinator-node will become a bottleneck. In order to overcome this bottleneck, Bitton et al. [5] introduce the `Hierarchical-Merge` algorithm. Instead of leaving the final merging of the partial results up to the root node, merging is done pairwise in a hierarchical binary-tree manner (see Figure 2.1b). However, if the number of duplicates is sufficiently small compared to the input relation, the coordinating node again becomes a bottleneck, since it has to merge almost the complete input relation.

Shatdal and Naughton [49, 50] propose two algorithms that aim to achieve better load balancing and avoid the single-node bottleneck. The `Two-Phase` algorithm uses hash-



(a) `Two-Phase` **(b)** `Repartitioning`

Figure 2.2: Algorithms proposed by Shatdal and Naughton [49] to overcome the problems of the traditional approaches. The algorithms differ only in that `Repartitioning` does not perform local duplicate removal.

partitioning [14] to further parallelize the *global duplicate removal* phase: After local duplicates are removed, the partial results are distributed across all participating nodes. All nodes then perform the global duplicate removal step in parallel. While this approach achieves better load balancing than `Merge-All` and `Hierarchical-Merge` by using all available nodes, it may lead to duplication of work if there are only a few duplicates. In this case, the first step fails to significantly reduce the input relation and therefore both phases operate on almost the complete input. This problem is avoided by the `Repartitioning` algorithm, which does not perform any kind of local preprocessing. Instead, the input relation is directly hash-partitioned across all nodes, which then perform only the *global duplicate removal* phase. Both algorithms are depicted in Figure 2.2.

Bitton et al. [5] describe a duplicate removal algorithm as part of a projection operation. The method relies on a hardware broadcasting facility and is based on a shared disk cache architecture. Given a *page* as the basic unit of transfer within this memory-hierarchy, the algorithm operates in multiple rounds. Using p processors to eliminate duplicate tuples in $n = m \times p$ pages will take m distinct rounds. First each processor p_i receives one page P_i. The remaining $n - p$ pages are then broadcasted one after another and used by each processor to remove any duplicates from P_i. After this step, the p pages are duplicate-free with respect to the remaining $n - p$ pages, but not with respect to each other. Therefore the processors successively (starting from PE $p - 1$) broadcast their page to all PEs with lower IDs, which then remove any remaining duplicates from their page. The procedure is repeated until all duplicates are removed. See Figure 2.3 for an example of the broadcasting process.

Su and Mikkilineni [55] propose a sorting algorithm that is also based on broadcasting and also relies on a specific hardware architecture. After a local sorting phase, the algorithm only broadcasts certain key values of each node that suffice to determine the global sort order and to detect and eliminate any global duplicates.

In the previously mentioned literature, duplicate removal algorithms were only considered as a by-product of either parallel sorting or projection and aggregation algorithms. Topkar et al. [60] propose and analyze three algorithms for duplicate removal on hypercube net-

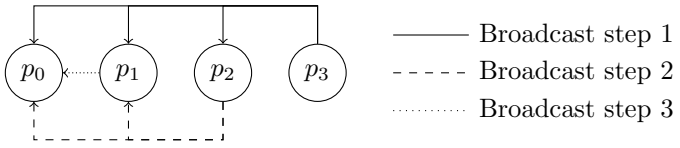
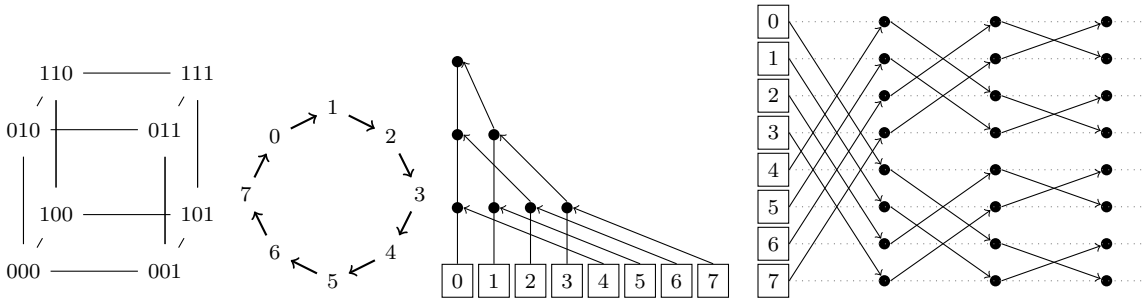


Figure 2.3: Broadcasting phase of the algorithm described by Bitton et al. [5]



(a) `Hypercube` **(b)** `Ring` **(c)** `Reduction` **(d)** `Bucket`

Figure 2.4: Hypercube (a) algorithms of Topkar et al. [60] on a cube with $p = 8$ nodes: (b) The `Multiple message ring` algorithm simulates a ring network. (c) The `Reduction` algorithm embeds the `Hiearchical-Merge` approach of Bitton et al. [5] into the hypercube. (c) The `Bucket` algorithm recursively partitions the nodes according to the current most significant bit of the node ID.

works (see Figure 2.4): The `Multiple-Message-Ring` algorithm simulates a ring network and removes duplicates by passing the locally duplicate-free tuples around that ring. The `Reduction` algorithm embeds the hierarchical-merge approach into the hypercube. Finally the `Bucket` algorithm exploits the a priori knowledge of the range of data values and recursively partitions the participating nodes into two groups based on most significant bit (MSB) of the node ID. In each group, the nodes divide their input data into two buckets depending on the range of the data values. One bucket is kept at each node, while the other bucket is sent to the corresponding node in the other group, i.e., the node whose node ID only differs in the MSB. The received bucket is then merged into the bucket that was kept at each node. During the process all occurring duplicates are removed. By distributing work more evenly among the nodes it thereby achieves better load balancing. In their experimental analysis, the `Bucket` algorithm was insensitive to the uniqueness of the input data and consistently outperformed both other algorithms.

All of the above mentioned algorithms (except `Repartitioning`) preprocess the input data by removing *locally duplicate* tuples in the first phase. This reduces the communication volume and the amount of work necessary in the global duplicate removal phase to a certain extent. Especially in the case where the dataset contains only a few or even no duplicates, all algorithms communicate almost the entire dataset. The distributed duplicate removal algorithm of Sanders et al. [48] explicitly avoids unnecessary data transfers by identifying distinct tuples in a preprocessing phase. Omitting the transfer of theses tuples significantly reduces the communication volume.

Abdelguerfi et al. [2] compare the performance of the `Two-Phase` and `Repartitioning` algorithm with an algorithm that uses a *global* preprocessing scheme. All three algorithms are implemented on a unidirectional, ring-connected, message-passing parallel database system and use the local duplicate removal algorithm of Teuhola and Wegner [59]. Instead of determining local duplicates, their approach creates node-specific bit-array filters that contain information about the tuples residing on all other nodes. This information is then used to identify tuples that are *globally distinct* and therefore do not need to be processed in the second phase.

The bit-array filtering works as follows: Before repartitioning the data for global duplicate removal, every node constructs a bit-array and passes it around the ring. As the filters travel around the ring, every PE hashes each of its local tuples and uses the hashes as an index into the bit-array. For each hashed tuple, the corresponding filter-bit is set. Once the filter arrives at its originating node, it is used to determine the global uniqueness of

local tuples. This is done by hashing the local tuples and inspecting the corresponding bit in the filter. A "0" indicates that the tuple is globally unique. A "1" indicates that there is a duplicate, or a tuple that produced the same hash value, on another node somewhere around the ring.

In their experiments, `Repartitioning` consistently outperformed both algorithms that use a dedicated preprocessing scheme, which showed that neither the local nor the global preprocessing scheme achieves a reduction in running time that is able to offset its additional cost.

The bit-array filtering approach is conceptually similar to ideas of Sanders et al. [48] in that it identifies distinct tuples in order to avoid unnecessary communication. The fundamental difference, however, is in the way this is achieved. In the approach of Abdelguerfi et al. [2], *each* PE gathers information about the tuples residing at *every other* PE. Thus, the bit-arrays that are passed around the ring contain a significant amount of redundant information. By creating *one* filtering data structure that is distributed over the PEs, the dSBF-based duplicate removal algorithm of Sanders et al. [48] eliminates these redundancies. Additionally, all messages that are necessary to build the filter are compressed using an information-theoretically optimal compression scheme.

2.2 Golomb Coding

The distributed duplicate removal algorithm of Sanders et al. [48] uses distributed *compressed* single shot Bloom filters to identify distinct elements. This filter can be constructed with minimal communication effort by performing batched insertion operations that can be compressed optimally using Golomb coding [21]. This section therefore briefly summarizes the relevant work of Moffat and Turpin [37].

As Golomb coding uses both *unary coding* and *truncated binary coding*, we start by defining both coding schemes:

Definition 2.1 (Unary coding)**.** Let $x \geq 1$ be a non-negative integer. The *unary code* $u(x)$ of x is defined as $u(x) = 1^{x-1}0$, where 0 marks the end of the codeword. A unary-coded codeword is decoded by counting the number of 1s up to the next 0.

Definition 2.2 (Truncated binary coding)**.** Assume a set of integers in the range $[1, n]$ and let $k = \log_2 n$ be the number of bits of one integer. Truncated binary coding assigns codewords of length $\lfloor k \rfloor$ bits to the first $2^{\lceil k \rceil} - n$ values in this range and the **last** $2n - 2^{\lceil k \rceil}$ codewords of length $\lceil k \rceil$ bits to the remaining $2n - 2^{\lceil k \rceil}$ values.

Example 1 (Unary coding, Truncated binary coding)**.** Assume we want to encode the set $S = \{1, 2, 3, 4, 5\}$. According to Definition 2.1 the corresponding unary codes are "0","10","110","1110","11110". Truncated binary coding assigns codewords of length $\lfloor \log_2 5 \rfloor = 2$ to the first $2^{\lceil \log_2 5 \rceil} - 5 = 3$ values: "00","01","10". The remaining $2 \times 5 - 2^{\lceil \log_2 5 \rceil} = 2$ values are then assigned the last 2 codewords consisting of $\lceil \log_2 5 \rceil = 3$ bits: "110","111".

Algorithms 1 and 2 show the encoding and decoding procedures based on [37, p. 31]. Function `write_int`(x, b) writes integer x to the output stream using b bits. Function `read_int`(b) interprets the next b bits as an integer. In case $\lceil k \rceil$ bits were used for encoding, function `get_bit`() is used to read the next bit from the input stream.

Based on these definitions, we can now define Golomb coding as follows:

Algorithm 1 Truncated Binary Encode

```
procedure TRUNC_BINARY_ENC(x, n)
    k ← ⌈log2 n⌉
    d ← 2^k − n

    if x > d then
        WRITE_INT(x − 1 + d, k)
    else
        WRITE_INT(x − 1, k − 1)
    end if
end procedure
```

Algorithm 2 Truncated Binary Decode

```
procedure TRUNC_BINARY_DEC(n)
    k ← ⌈log2 n⌉
    d ← 2^k − n
    x ← READ_INT(b − 1)
    if x + 1 > d then
        x ← 2 ∗ x+ GET_BIT( )
        x ← x − d
    end if
    return x + 1
end procedure
```

Definition 2.3 (Golomb coding)**.** Depending on a tuning parameter b, a value $x \geq 1$ is encoded in two parts. Let $q = \lfloor \frac{(x-1)}{b} \rfloor$ be the result of the division by the tuning parameter and $r = x - q \times b$ be the remainder. Then x is encoded by first issuing $q + 1$ using *unary coding*, followed by r, which is encoded using *truncated binary coding*. A Golomb codeword $c_{golomb}(x)$ is therefore decoded in three steps: First we decode the unary encoded quotient $q = unary\_decode() - 1$. Next we decode the remainder r using algorithm 2. The decoded value then can be calculated as $x = q \times b + r$.

Conceptually, Golomb coding divides the range of the input sequence into equally sized buckets of size b. The unary part of a codeword (the quotient q) then denotes the corresponding bucket id, while the truncated binary part (the remainder r) denotes the specific position within the bucket (see Figure 2.5).

In the next section we will use Golomb coding to compress a sorted sequence of n integer values chosen uniformly at random from a range $[1, nc]$. Let $\overline{S}$ denote the set of differences between consecutive values in S. Then $\overline{S}$ can be considered to be drawn from a geometric distribution with $p \approx \frac{1}{c}$ [44, 37, p. 38]. Compressing $\overline{S}$ using Golomb coding with $b = (\log_e 2)\frac{1}{p}$ yields a minimum-redundancy code, i.e., a code that (on average) requires a minimum number of bits per input element [37, p.36 ff., p. 51]. In this case, the code size is approximately $n(\log_2 \frac{1}{p} + 1.5)$ bits [37, p. 39 f.].

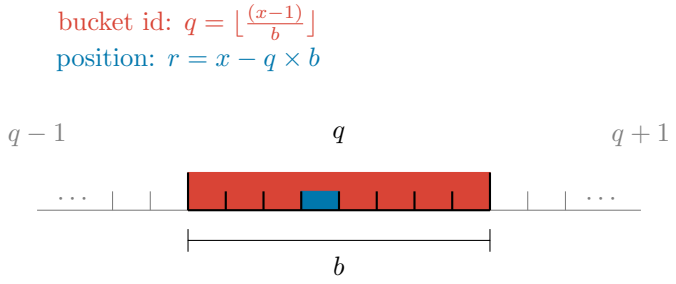


Figure 2.5: A Golomb code codeword conceptually consists of a *selector* part (q) that selects the corresponding bucket of size b and a *position* part (r) that specifies the position within the bucket.

2.3 Bloom Filter

In this section, we will focus on the traditional Bloom filter and a space-efficient alternative proposed by Putze et al. [44]: the Golomb-Compressed Sequence (GCS). We conclude this section with a review of some approaches to distributed Bloom filters. For a detailed

overview about the Bloom filter related research, we refer to the comprehensive surveys of Broder and Mitzenmacher [8] and Tarkoma et al. [57].

A Bloom filter is a space-efficient, probabilistic data structure for *approximate set membership queries*. It was introduced by Burton H. Bloom [6] in the 1970's and since then has been used in a variety of application domains such as databases [38, 62, 3, 45, 46, 36], peer-to-peer networks [9, 28, 12] or duplicate detection in data streams [35, 13, 30, 63] and numerous variants and replacements have been proposed [42].

We start by defining the problem solved by Bloom filters and related data structures:

Problem 1 (Data Structure for approximate set membership queries)**.** Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n elements from a universe U. Our goal is to represent S in a space-efficient manner (i.e., using m bits) and allow *set membership queries* of the form $y \in S?$. Furthermore we allow the data structure to report *false positives*, i.e., it is allowed to make a one-sided error by reporting an element as part of the set although it was not contained. However, we do not tolerate any *false negatives*. Thus, the data structure should never report that an element is not part of the set although it actually is. The answers to the set membership queries are therefore only *approximate* rather than *exact*. We denote the probability of reporting a false positive as the *false positive rate* (FPR) f [44].

By storing all n elements of S we would achieve a false positive rate of $f = 0$ at the cost of $m = n\lceil \log U \rceil$ bits. On the other hand, information theory gives us a lower bound of $m = n \log \frac{1}{f}$ bits that are necessary to represent all possible sets S of n elements from a universe U such that the false positive rate is bounded by f [8, 41].

We now introduce Bloom filters as a step closer to this lower bound:

Definition 2.4 (Bloom Filter)**.** A Bloom filter [6] representing S is an array of m bits. All bits in this Bloom filter array are initially set to zero. An element x is inserted into the filter by evaluating k independent hash functions $h_i(x), ..., h_k(x)$. Each hash function maps an element $x \in S$ to a random number in the range $[0, m-1]$ and sets the corresponding bit to 1. After all elements are inserted, a membership query $y \in S?$ for an element y can be answered by evaluating the k hash functions and determining the state of the corresponding filter bits. If any of the k bits is 0, then the element is *guaranteed* not to be a member of the set. However, if all bits are set, then the element *may* be part of the set. In some cases, the membership query will return *true* although the corresponding element is not part of the set. These false positives occur because of hash collisions (see Figure 2.6 for an example). Kirsch and Mitzenmacher [29] show that it is not necessary to actually evaluate k independent hash functions. Instead it suffices to evaluate just 2 hash functions and use a linear combination of both to generate the remaining $k-2$ hash values.

The FPR f of a Bloom filter can be controlled by adjusting the size m and thereby the number of hash functions k: Given the number of elements n we choose

$$m = \frac{n}{\ln 2} \log \frac{1}{f} \quad \text{and} \quad k = \frac{m}{n} \ln 2 = \log \frac{1}{f}$$

in order to achieve the desired FPR of f [57]. Using a Bloom filter to represent a set S of n elements from a universe U with a FPR of f thus only requires $m = \frac{n}{\ln 2} \log \frac{1}{f}$ bits. However, a Bloom filter still needs about $\frac{1}{\ln 2} \approx 1.44$ times as much space as the asymptotic lower bound of $n \log \frac{1}{f}$ bits [8, 41].

In order to achieve a near optimal space consumption, Putze et al. [44] propose a Bloom filter replacement named Golomb-Compressed Sequence (GCS), which can be regarded as a compressed version of a Bloom filter with $k = 1$ hash functions. We refer to this Bloom filter replacement as *single shot Bloom filter*:

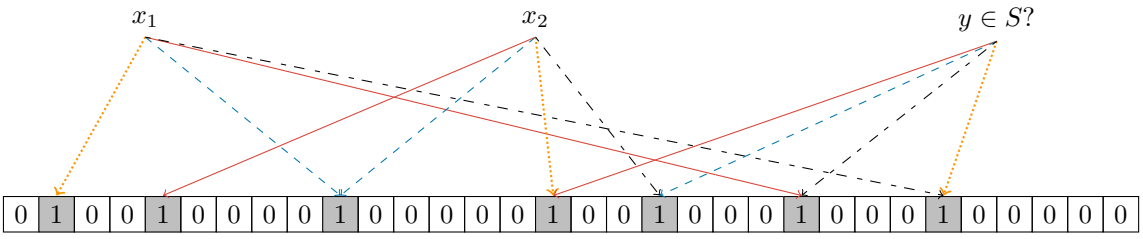


Figure 2.6: Bloom filter example. Assume $S = \{x_1, x_2\}$, $k = 4$ and $m = 32$ bits. After inserting x_1 and x_2, the Bloom filter is queried with element y. Although it is not part of the original set S the filter returns *true*, because all 4 corresponding bits are set to 1. Thus, y is reported as a *false positive*, because of hash collisions.

Definition 2.5 (Single Shot Bloom Filter)**.** Conceptually, a *single shot Bloom filter* (SBF), like a traditional Bloom filter, is an array of m bits - initially all set to 0. Inserting an element x into the filter corresponds to setting the bit at position $h(x)$ to 1 using a hash function h, while answering an approximate membership query $y \in S$? corresponds to computing $h(y)$ and determining whether or not the bit is set to 1. In order to achieve the same FPR f as a traditional Bloom filter, we have to choose $m = \frac{n}{f}$. In other words, the range of the hash function has to be $\{0, ..., nc\}$ to achieve an FPR of $f = \frac{1}{c}$. However, instead of storing the plain bit array, the SBF compresses the sorted sequence of hash values, i.e., the positions of the bits set to 1: Since the hashes were chosen uniformly at random, the differences of successive hash values follow a geometric distribution with $p = f = \frac{1}{c}$. As we have already seen in section 2.2, Golomb coding can be used in this case to create a minimum-redundancy code. Thus, the SBF has a size of approximately $n(\log \frac{1}{f} + 1.5)$ bits.

While the previously mentioned literature used Bloom filters as a local data structure, Koloniari et al. [30] propose two ways to create a distributed Bloom filter across several nodes to eliminate duplicates in high-volume, distributed event streams. In their setting, consumers are interested in events produced by a number of producers. In order to disburden both the consumers and the event system from processing and communicating duplicate events, a filtering component based on Bloom filters is used to eliminate duplicates. Since one central Bloom filter constitutes a single point of failure and a potential bottleneck, they propose two distributed approaches: *horizontal* and *vertical* Bloom filters.

A *horizontal Bloom filter* of size m that is distributed across p nodes consists of p distinct traditional filters of size $\frac{m}{p}$ (one at each node). Updates and queries first use an additional hash function h_p to determine which of the p traditional filters is responsible for the element and then use the corresponding k hash functions with range $\{0, ..., \frac{m}{p}\}$ to insert or probe that specific filter. A *vertical Bloom filter* partitions its m bits into p non-overlapping subsets that are distributed across the nodes. To update or query a *vertical* filter, the appropriate partitions that contain the corresponding k bits are located by applying the k hash functions to the element in question.

Analyzing both approaches theoretically as well as experimentally, the authors conclude that vertical Bloom filters achieve better load balancing, slightly better false positive rates and better fault tolerance at the cost of higher communication volume due to the distribution of the filter bits. In the worst case, k messages are necessary to update or query a vertical Bloom filter because each of the k bits is contained in a different partition on a different node. In contrast, one single message is always sufficient to perform the corresponding operation on the distributed single shot Bloom filter, which will be introduced in the following section.

2.4 Communication Efficient Distributed Duplicate Removal

Having discussed all fundamentals, we now present the *distributed* compressed *single shot Bloom filter* (dSBF) [48], a generalization of the previously discussed single shot Bloom filter (SBF) [44]. Section 2.4.2 then introduces the dSBF-based distributed duplicate removal algorithm that is implemented and evaluated over the course of this thesis.

2.4.1 Distributed Single Shot Bloom Filter (dSBF)

The basic idea is similar to the vertical Bloom filter (vBF) of Koloniari et al. [30]. However, instead of partitioning the m bits of a traditional Bloom filter into p non-overlapping subsets and assigning each partition to one PE, the dSBF partitions the bit array of a single shot Bloom filter over the PEs.

Recall that a SBF uses just *one* hash function h to set or probe the corresponding filter bit. As the filter is now distributed over p PEs, with PE i being responsible for the filter bits $i\frac{m}{p}...(i+1)(\frac{m}{p})-1$, an insert or query of an element x is performed as follows: The PE that stores x has to send the corresponding filter bit position $h(x) \bmod {}^{m}/_{p}$ to the respective PE that is responsible for this part of the dSBF (i.e., PE ${}^{h(x)p}/_{m}$). In case of an insertion, the receiving PE sets the corresponding bit to 1. In case of a query, it probes the corresponding bit and communicates the state of the bit back to the sender.

Thus, using the dSBF with a false positive probability f directly leads to a factor of $\Theta\left(\log {}^{1}/_{f}\right)$ less communication volume compared to the vBF of Koloniari et al. [30]: While the vBF approach needs to send and receive up to $k = \log {}^{1}/_{f}$ messages in the worst case in which each of the k bits is located on a different PE, *one* message always suffices in the case of the dSBF.

Batched Insertions

The data structure also supports *batched* insertions and queries, in which case the communication volume necessary to perform the operations can be reduced even further by applying the compression technique of the SBF. Consider a PE i has to perform x_i dSBF operations. After sorting the hashes corresponding to the x_i tuples that are either inserted or queried, we can compute the differences of successive hashes and use Golomb coding to create a minimum-redundancy code as discussed in Section 2.2. These compressed dSBF messages are then sent to the PEs managing the respective dSBF partitions, where they are decoded and processed. Thus, batched dSBF operations are optimally compressible. Indeed, Sanders et al. [48] prove that n batched dSBF operations on a filter of size $m = \frac{n}{f}$ with a false positive probability of f can be executed with an expected communication volume of

$$V = n_{max}\left(\log\frac{mp}{n} + \mathcal{O}(1)\right) = n_{max}\left(\log\frac{p}{f} + \mathcal{O}(1)\right) \text{ bits}$$

where n_{max} is the maximal number of operations performed on any PE.

2.4.2 The dSBF-based Duplicate Removal Algorithm

The goal of the algorithm is to identify and remove all duplicate tuples from a relation R using minimal communication effort. We assume the input relation R consisting of n tuples to be distributed evenly over the p PEs of a shared-nothing system, i.e., each PE has $\frac{n}{p}$ tuples. Similar to the bit-filtering approach of Abdelguerfi et al. [2], the algorithm is separated into a *global* filtering phase and a finalization phase, which only works on the remaining tuples that could not be identified as distinct during the filtering phase:

The Filtering Phase

In order to minimize the number of tuples that actually have to be compared to each other and therefore have to be transferred over the network, the algorithm uses a dSBF to identify *distinct* tuples with minimal communication effort:

Each PE inserts its $\frac{n}{p}$ tuples into the dSBF using *one* batched insertion operation. As the dSBF is not used for any successive queries, it is not necessary to materialize the corresponding bit array on each node (i.e., to build the corresponding SBF). Instead, each PE checks which hash values occur more than once (i.e., which dSBF positions would have been set to 1 multiple times) and communicates this information back to the corresponding source PEs. Thus, after having received all dSBF responses, each PE is able to filter out all tuples that could be identified as distinct. The remaining tuples will be passed on to the finalization phase.

Sanders et al. [48] prove that in case of a single filtering pass, the overall communication volume V is minimized, if the dSBF is built using a false positive probability of $f = \frac{1}{u}$, where u is the size of an input tuple in bits. This low FPR is necessary to significantly reduce the number of tuples that have to be considered in the second phase. However, it is also possible to use more than one dSBF-filtering pass before proceeding to the finalization phase. In case of a two-pass filtering approach, it is shown that the first dSBF can be built using a FPR of $f_1 = \frac{1}{\log up}$. The second one, which is used to identify more distinct tuples within the remaining elements of the first pass, then again uses an FPR of $f_2 = \frac{1}{u}$.

The Finalization Phase

After the preprocessing phase, most distinct tuples have been identified as such and can therefore be excluded from the finalization phase. However, it is still necessary to post process the tuples that passed the filter (i.e., that could not be identified as *distinct*) in order to tell the *real* duplicates apart from the false positive results reported by the dSBF. This is done by applying the `Repartitioning` algorithm introduced in Section 2.1 to the remaining tuples.

The detailed theoretical analysis of Sanders et al. [48] shows that single-pass filtering can be performed using an overall communication volume, which is *logarithmic* in the size u of the input tuples, while multi-pass filtering even achieves a communication volume, which is *double-logarithmic* in the tuple size.

3. Engineering the dSBF-based Duplicate Removal Algorithm

In their theoretical analysis, Sanders et al. [48] outline the high-level structure of the dSBF-based distributed duplicate removal algorithm. Goal of this thesis is to complement this abstract, theoretical description with an actual implementation that runs on a shared-nothing compute cluster and exploits the shared-memory capabilities of the multiprocessors contained in each cluster node for all computationally intensive operations.

We start by decomposing the algorithm into distinct computation and communication phases. Sections 3.3 through 3.6 then detail selected steps of our implementation that are considered crucial for the overall performance of the algorithm. We substantiate our design decisions with dedicated experimental evaluations. Section 3.2 therefore introduces the experimental setup that is used throughout this thesis.

3.1 The Algorithm: From Theory to Practice

We again assume the input relation consists of n tuples. These tuples and the m bits of the dSBF are distributed evenly over the p PEs, i.e., $\frac{n}{p}$ tuples and $\frac{m}{p}$ bits per PE. Without loss of generality, we further assume the dataset of each node to be duplicate-free, i.e., the input relation only contains inter-node duplicates and no intra-node duplicates. In case of intra-node duplicates, one of the algorithms described in Section 2.1 can be used to detect and remove them, before our algorithm is executed.

Figure 3.1 presents the algorithm from the perspective of alternating computation and communication rounds. For brevity, we outline the *single-pass* algorithm and detail the differences of the multi-pass variant at the end of this section. We distinguish 9 distinct phases:

1. **Preprocessing:** The first phase amounts to the preparation of the batched dSBF insertion messages. Every tuple of the input relation has to be preprocessed in order to determine the PE that is responsible for the corresponding part of the dSBF where the tuple will be inserted. Thus every PE processes its input tuples and creates p messages - one for each PE, containing information about which filter positions should be set to 1. Further details on the preprocessing phase will be given in the paragraph following this enumeration.

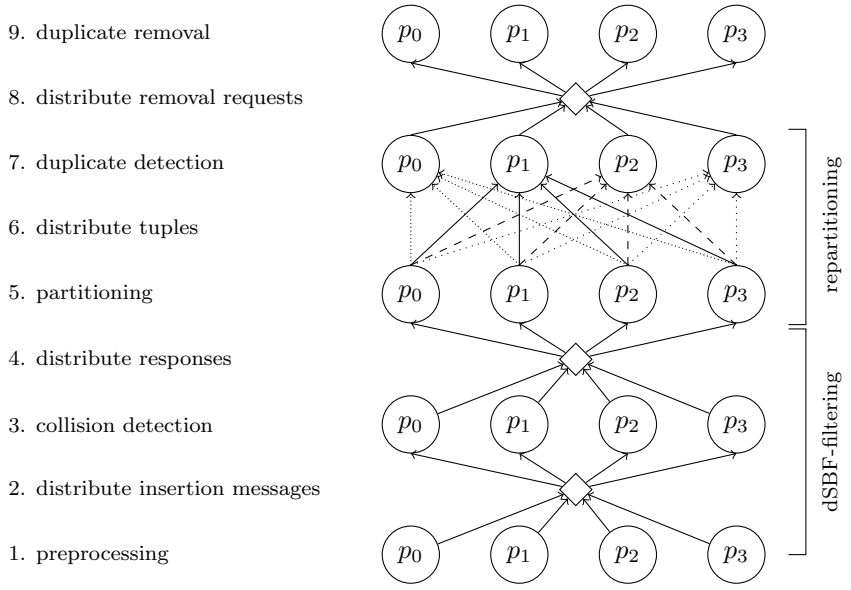


Figure 3.1: Computation and communication rounds of the dSBF-based duplicate removal algorithm.

2. **Distribute dSBF insertion messages:** Once the preprocessing is finished, all batched insertion messages are distributed to their respective destinations.

3. **Collision detection:** The dSBF is used to detect *distinct* tuples. In terms of the filter, a tuple is distinct if the respective filter bit is set to 1 only once. Whenever a bit is set to 1 multiple times, the corresponding input elements cannot be regarded as distinct. Since we use the dSBF only to detect filter-bit collisions and do not perform any subsequent membership queries, it is not necessary to actually materialize the SBF at each PE. Instead, every PE analyzes the insertion messages it receives, discards all insertion positions that only occur once and keeps track of all positions that occur multiple times.

4. **Distribute filter responses:** For every received batched insertion message, a corresponding response is sent back to the source PE. For each insertion position, the response indicates if the respective position produced a collision.

5. **Partitioning:** Using the dSBF responses of phase 4, each PE is now able to filter those tuples that did not produce a collision. As these tuples are now known to be distinct, they do not need to be processed any further and can be discarded from all subsequent phases. Since the dSBF however is a probabilistic data structure that produces false positive results, tuples that correspond to filter positions that occurred multiple times cannot be assumed to be duplicates. In order to distinguish false positives from real duplicates, the PEs now execute the `Repartitioning` algorithm on the remaining tuples. The partitioning phase partitions these remaining tuples into p messages.

6. **Distribute input tuples:** Each remaining tuple is sent to the PE that detected the corresponding dSBF collision in phase 3. The sole goal of the dSBF-based filtering is to minimize the amount of data that has to be transferred in this phase.

7. **Duplicate detection:** The duplicate detection phase of the `Repartitioning` algorithm solves the problem of false positive filter results. By applying, for example, one of the algorithms described in Section 2.1, this phase compares the received input tuples and therefore distinguishes tuples that were reported to be duplicates because of false positive filter results from those tuples that really occurred multiple times across the

nodes. For each tuple occurring more than once, one PE may keep it. The remaining PEs have to remove it from their part of the input relation.

8. **Distribute duplicate removal requests:** For every received input tuple, an indicator is sent back to the corresponding source PE, signaling either to keep the corresponding tuple or to remove it from the input relation.
9. **Duplicate removal:** Based on the received duplicate removal requests, each PE removes the corresponding input tuples, which were identified as duplicates. Afterwards, the input relation is free of any inter-node duplicates.

The algorithm executes distinct computation and communication phases. Communication phases thereby follow a regular pattern: In every communication round, each PE exchanges a (potentially empty) message with every other PE that participates in the duplicate removal process. This type of communication is known as *all-to-all personalized communication*. As the performance of this operation is crucial for the overall performance of the duplicate removal algorithm, Section 3.4 details our efforts in tuning this kind of communication operation.

Among the computation rounds, the *preprocessing* and *collision detection* phases deserve further attention. We therefore detail these phases from the perspective of one of the PEs, as all nodes perform the exact same steps - only on a different part of the input dataset.

3.1.1 The Preprocessing Phase

The preprocessing phase itself can be divided into 5 different steps as shown in Figure 3.2.

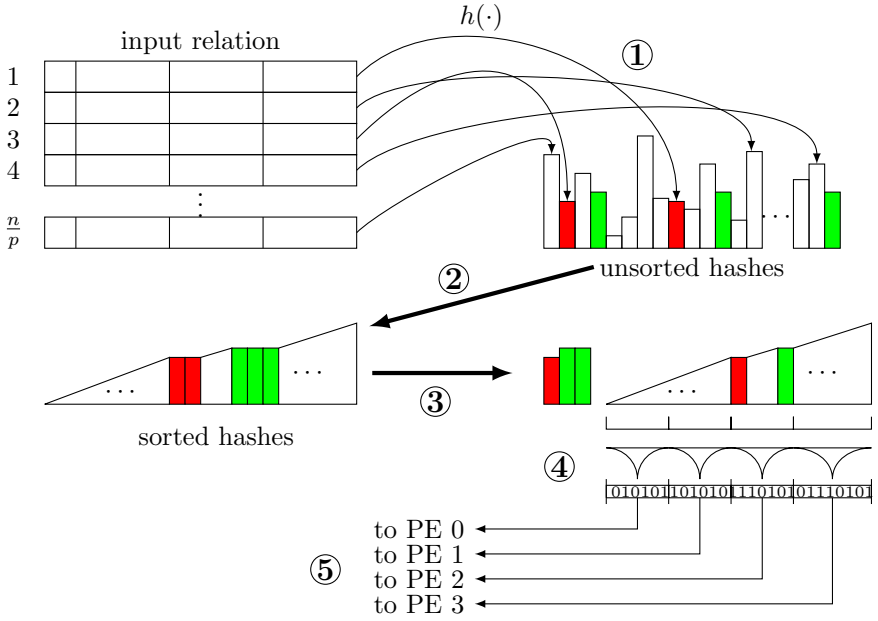


Figure 3.2: The preprocessing phase of the dSBF-based duplicate removal algorithm: 1.) Hashing, 2.) Sorting, 3.) Collision Extraction, 4.) Compression, 5.) Distribution.

1. **Hashing:** A tuple is inserted into the dSBF by setting the corresponding filter bit to 1. The position of this bit is determined by a hash function h. Thus it is necessary to hash all input tuples. As this operation is independent for each tuple, it can be performed in parallel using the shared-memory capabilities of the multiprocessors contained in each node.

2. **Sorting:** As described in Section 2.4.1, we do not actually materialize the SBF bit array. Instead, the hash values are used to describe the bit positions that are set to 1. In order to be able to compress the hashes using Golomb coding, it is necessary to sort them. Sorting can also be done in parallel using black-box comparison-based sorting routines like the parallel version of `std::sort` provided by the GNU C++ library [52]. However, we want to exploit the fact that we deal with (random) integer values. We therefore implement a parallel radix sort based on the concepts of Wassenberg and Sanders [64]. Section 3.3 covers the related details.
3. **Collision Extraction:** Hashing the input tuples may result in local hash collisions. As we want the communication volume to be minimal, we send every hash value only once. Duplicate hash values are therefore extracted in a sequential scan operation and only one instance of each hash is kept for distribution. However, it is not possible to discard the duplicate hash values, since each of them was produced by applying a hash function to a *different* tuple of the input relation. For each hash value that could not be identified as distinct after the dSBF-filtering phase, it is therefore necessary to distribute all input tuples that produced the same hash value in order to distinguish false positives from real duplicates in the `Repartitioning` algorithm.
4. **Compression:** The key to the minimal communication volume implied by the distributed single shot Bloom filter is the possibility to compress the logical filter bit array (i.e., the sorted sequence of hash values that indicate the positions of the one-bits in the filter). In Section 3.5 we therefore detail our engineering efforts regarding this phase.
5. **Distribution:** Finally, the compressed messages are distributed to the respective destination PEs. In order to overlap compression and communication, we implemented a pipelined communication algorithm, which will also be discussed in Section 3.5.

3.1.2 The Collision Detection Phase

As described in Section 2.4, the dSBF allows batched insertions to build up the distributed data structure. Subsequent batched membership queries can then be used to probe the filter. In the duplicate removal algorithm however, we do not need to query the filter. Moreover, we do not even need to construct the corresponding SBF on each node, as we are only interested in the filter positions that occur multiple times in the received batched insertion messages. Figure 3.3 provides a conceptual overview of this process. A description of the individual steps follows.

1. **Decoding:** The dSBF insertion messages arrive as Golomb compressed bit strings. The first step therefore is to decompress the messages in order to retrieve the original hash values. Section 3.5 discusses the relevant details.
2. **Collision Detection:** Each decompressed message contains a sorted sequence of hash values corresponding to the dSBF bit positions that should be set to 1. By using a parallel multiway-merging routine, it is therefore possible to detect colliding hash values. In Section 3.6 we describe our approach to collision detection.
3. **Distribution:** Signaling back hash collisions can be done by sending one bit per received hash value. A 1 indicates a hash collision, while a 0 indicates that the corresponding hash value occurred only once.

The distribution of the filter responses concludes the dSBF-filtering phase as depicted in Figure 3.1. In the single-pass variant of the distributed duplicate removal algorithm, the `Repartitioning` algorithm is now used to resolve the false-positive filter results and to determine the real duplicates.

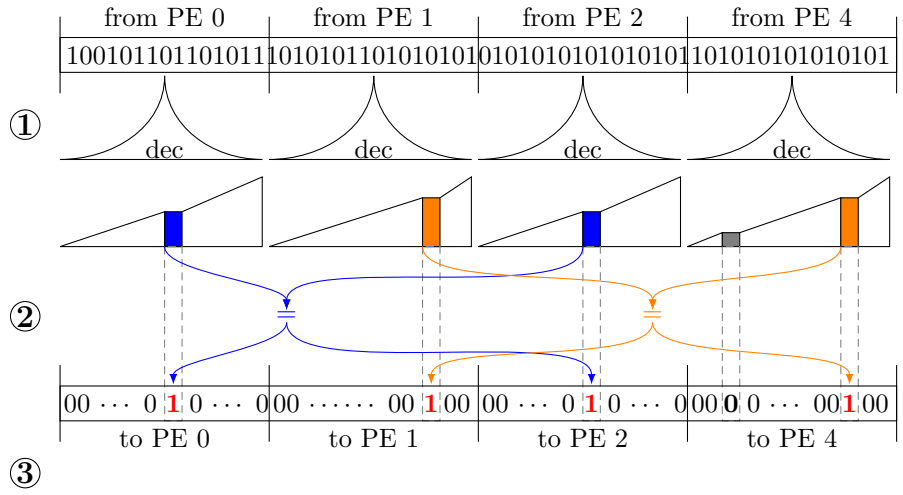


Figure 3.3: The collision detection phase of the dSBF-based duplicate removal algorithm.

3.1.3 Multi-Pass Filtering

In the multi-pass variant, the dSBF-filtering phase is repeated using the remaining tuples of the previous pass as input. As described in Section 2.4.2, it is necessary to choose the size m of the dSBF optimally, in order to minimize the overall communication volume. The size of the filter depends on the optimal false positive rate and the number of elements that could not be identified as distinct. For filtering pass i, the filter size therefore is $m_i = \frac{n_i}{f_i}$. Thus in order to proceed with the next dSBF-filtering phase, it is necessary to count the remaining tuples over all PE: $n_i = \sum_{p=0}^{p-1} n_p$ and distribute this value to all nodes. Together with the corresponding optimal false-positive rate f_i, the next filtering pass can than be started.

Once all filtering passes are finished, the multi-pass variant continues like its single-pass counterpart by using the `Repartitioning` algorithm to process all remaining tuples that could not be identified as distinct by any of the dSBF passes.

3.2 Experimental Setup

Our test environment is a distributed memory cluster system consisting of 400 dedicated compute nodes. Each node consists of two Octa-Core Intel Xeon E5-2670 CPUs (Sandy Bridge) clocked at 2.6 GHz and 64 GB of main memory and is running Suse Linux Enterprise (SLES) 11 SP2 with kernel version 3.0.42. Each core has a private 64 KB L1- and a private 256 KB L2-cache in addition to a 20 MB L3-cache, which is shared by eight cores on each socket. The nodes are connected by an InfiniBand 4x QDR interconnect with a theoretical point-to-point bandwidth of 4 GB/s.

In order to communicate between the nodes we use the message passing [34] library OpenMPI 1.6.4 compiled with `--enable-mpi-thread-multiple` to enable multithreading support. All programs are compiled using GCC 4.7.2 with `-O3 -mtune=native -march=native -std=c++11 -fopenmp`. We use both OpenMP [40] and the Intel® Threading Building Blocks (Intel® TBB) library [27] in version 4.1 to exploit shared-memory parallelism of the multiprocessors of each node.

3.3 Implementing a Parallel Radix Sort

The computationally most intensive step in the preprocessing phase of the dSBF-filtering process is sorting the hash values. The sorting phase is necessary in order to be able to

extract local hash collisions using a single pass over the hash values. Furthermore, it is a prerequisite for Golomb compression. While it is possible to use a black-box comparison-based sorting routine like the parallel version of `std::sort` provided by the GNU C++ library, we want to exploit the fact that we deal with (random) integer keys and even know the range in which the hash values are distributed. Since the hash values denote bit positions in the dSBF that are set to 1 and the dSBF logically consists of m bits, the range is $[0, m)$ and the hash values consist of $k = \lceil \log m \rceil$ bits.

Using these restrictions of the hash values, we are able to break the lower bound of $\Omega(n \log n)$ of comparison-based sorting algorithms by using a radix sort. Given n input keys of k bits each, radix sort recursively sorts the data by examining d-bit digits and partitions the keys according to these digits. Depending on the digit that is examined first, one distinguishes between least significant digit (LSD) and most significant digit (MSD) radix sort. LSD radix sort starts by examining the rightmost digit and partitioning the input sequence into buckets based on that digit. This process is repeated $\frac{k}{d} - 1$ times for the remaining digits. MSD radix sort proceeds analogously, starting from the leftmost digit.

Wassenberg and Sanders [64] propose a parallel radix sort for 32-bit integer keys that exploits several details of modern micro-architectures. Our implementation is a straightforward generalization of their algorithm that also works on 64-bit integer keys and uses the knowledge of the maximum possible key value to determine the maximum number of digits that have to be processed.

Algorithm 3 outlines our approach. Using the information of the maximum hash value, we determine the number of digits that have to be analyzed. Choosing $d = 8$ allows us to extract d-bit digits without masking by addressing an entire byte and partitions the input into 256 buckets in each pass. Like in [64] we use *one* MSD-pass to partition the input into a set of buckets (lines 3-6). This step is performed in parallel. Using a prefix sum, we then calculate the output indices for the first item of each MSD bucket.

These MSD buckets are then sorted in parallel in LSD order. Each thread is responsible for a range of MSD buckets produced in the first pass. Note that because the MSD pass was performed in parallel, every thread has a complete set of MSD buckets and the input is now partitioned over all of these buckets. In order to sort one MSD bucket, a thread therefore has to accesses the corresponding buckets of all other threads during the first LSD pass (see lines 15-18).

It then processes all keys by examining the remaining digits and partitioning the keys into its bucket structure. The second to the last pass also computes the histogram for the last digit. After calculating the bucket indices using a prefix sum, the input elements can then be written directly to the correct output position in the last pass (lines 35-43).

Note that the algorithm uses $|\text{passes}| \times |\text{threads}| \times 2^d$ buckets. To make the number of buckets independent of the number of passes, we only use 3 sets of buckets. The first set of buckets is used to contain the elements of the MSD pass. The first LSD pass then partitions its element into a new set of buckets. As each thread processes one MSD bucket at a time, we cannot reuse the MSD buckets for the second LSD pass. We therefore use a third set of buckets. All further passes then reuse these two sets of LSD buckets interchangeably.

While this reduces the number of buckets to $3 \times |\text{threads}| \times 2^d$, managing these buckets explicitly would still incur a certain overhead. Wassenberg and Sanders [64] therefore propose to preallocate each bucket with a size of n (i.e., each bucket is big enough to contain the entire input). This is efficient, because modern operating systems map physical memory only on the first access of a memory page. While the algorithm therefore uses a

significant amount of virtual memory address space, it only causes a constant size memory overhead per bucket, which is limited by the page size.

Each pass of the algorithm reads each key once and writes it to one of the 2^d buckets depending on the current digit. In order to avoid cache pollution, Wassenberg and Sanders [64] advocate to use *non-temporal* writes that write directly into memory by bypassing the cache. As single memory accesses are expensive, Wassenberg and Sanders [64] make use of *software write-combining*. Input elements are first written to temporary buffers of the size of a cache line. Once such a buffer becomes full, a sequence of non-temporal writes is used to copy the buffer to the corresponding destination in the memory. Our implementation also adopts this approach.

Algorithm 3 Parallel Radix Sort

```
1: procedure Parallel Radix Sort
2:     numDigits ← ⌈log(m − 1)⌉/digitSize
3:     parallel foreach item do                                   ▷ MSD partitioning
4:         d ← firstRelevantDigit(item)
5:         buckets_msd[d] ← buckets_msd[d] ∪ {item}
6:     end parallel foreach
7:     Barrier
8:     foreach i ∈ [0, 2^d) do
9:         bucketSizes[i] ← Σ_thread |buckets_msd[i]|
10:    end for
11:    outputIndices ← PrefixSum(bucketSizes)
12:    numDigits ← numDigits − 1
13:    parallel foreach bucket_msd ∈ buckets_msd do
14:        currentDigit ← 0
15:        foreach item ∈ bucket_msd ∀threads do                  ▷ first LSD pass
16:            d ← Digit(item,currentDigit)
17:            buckets_currDigit[d] ← buckets_currDigit[d] ∪ {item}
18:        end for
19:        currentDigit ← currentDigit + 1
20:        numDigit ← numDigits − 1
21:        for currentDigit < numDigits − 1 do    ▷ LSD passes exept for the last one
22:            foreach bucket_prevDigit ∈ buckets_prevDigit do
23:                foreach item ∈ bucket_prevDigit do
24:                    d ← Digit(item,currentDigit)
25:                    buckets_currDigit[d] ← buckets_currDigit[d] ∪ {item}
26:                    // If this is the last loop iteration, we create the histogram
27:                    // otherwise these lines are not executed
28:                    d ← firstRelevantDigit(item)
29:                    histogram[d] ← histogram[d] + 1
30:                end for
31:            end for
32:            currentDigit ← currentDigit + 1
33:        end for
34:        bucketIndices ← PrefixSum(histogram)
35:        foreach bucket_prevDigit ∈ buckets_prevDigit do        ▷ final LSD pass
36:            foreach item ∈ bucket_prevDigit do
37:                d ← Digit(item,currentDigit)
38:                d1 ← firstRelevantDigit(item)
39:                i ← outputIndices[d1] + bucketIndices[d]
40:                bucketIndices[d] ← bucketIndices[d] + 1
41:                output[i] ← item
42:            end for
43:        end for
44:    end parallel foreach
45: end procedure
```

Experimental Evaluation

The performance of our radix sort implementation depends on the size of the keys. Using 32-bit integers and a digit size of $d = 8$ bits requires 4 passes through the input data, while up to 8 passes can be necessary for 64-bit integers, depending on the size of the maximum input key.

We therefore evaluate both cases and compare the performance of our implementation to the parallel comparison-based version of `std::sort` provided by the GNU C++ library. We use the mersenne twister random number generator of the Boost library to generate uniformly distributed random integers as input. As an additional experiment to sorting keys, we measure the running time for sorting 32- and 64-bit (key, value) pairs (i.e., a key paired with a value of the same size as payload). All reported timing values are averages over 25 iterations. Each experiment is performed on one dedicated node of the cluster described in Section 3.2. Both algorithms are configured to use 16 threads.

Figure 3.4 shows the results for sorting 32-bit integer keys as well as 32-bit (key, value) pairs. Our radix sort consistently outperforms the library algorithm. For plain 32-bit keys our implementation is up to 4 times faster than the comparison-based parallel sorter. For 32-bit (key, value) pairs it remains more than a factor of 2 faster. This behavior is expected, as radix sort is memory-bound: Doubling the amount of partitioned data approximately doubles the corresponding running time.

Figures 3.5 and 3.6 show the results for the same experiments, but using 64-bit keys and 64-bit (key, value) pairs. The number of passes of our radix sort implementation depends on the maximum size of the keys. We therefore choose the range of the input keys such that five to eight passes are necessary to sort the input and report the running time for each configuration. For example `radix (8 pass)` denotes the running time of our radix sort implementation for problem instances where all 8 digits have to be evaluated in order to sort the input. As the comparison-based sorting routine is insensitive to the size of the input keys, we only report its running time once.

Our implementation again outperforms the library algorithm. Its advantage however decreases by a constant factor for each additional pass over the input. Adding an equally sized value as payload to the keys (see Figure 3.6) again approximately doubles the running time per element.

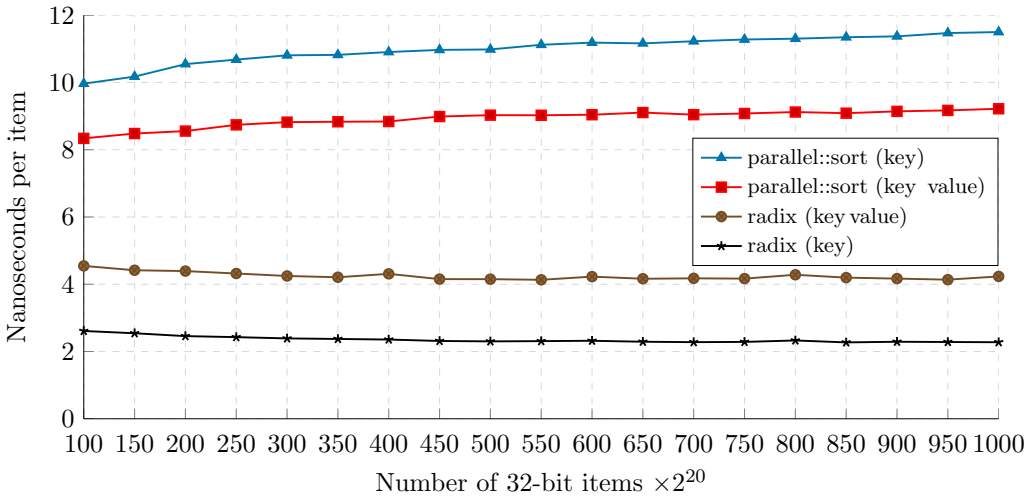


Figure 3.4: Running time comparison of parallel radix sort and parallel::sort for 32-bit keys and 32-bit (key, value) pairs. Both implementations use 16 threads.

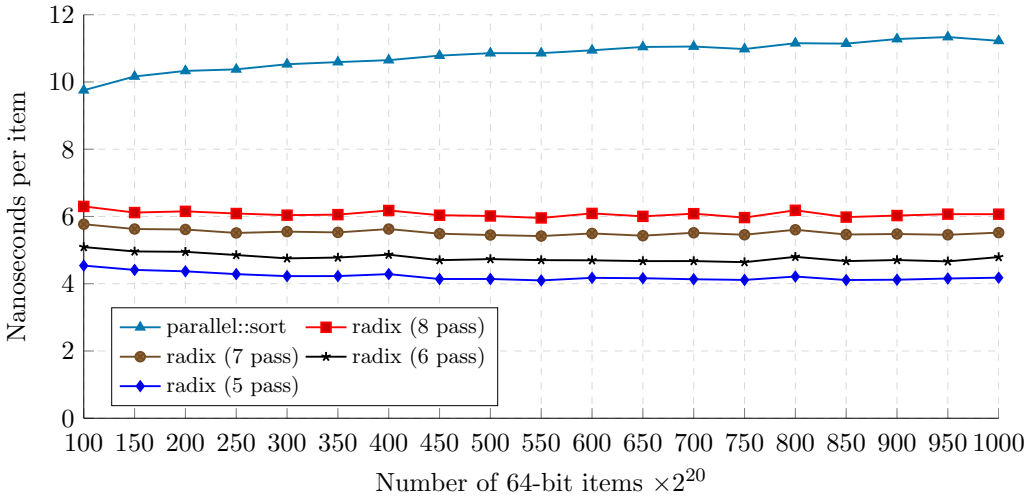


Figure 3.5: Running time comparison of parallel radix sort and parallel::sort for 64-bit keys. Keys are generated such that five to eight passes are necessary to sort the input using a digit size of $d = 8$ bits. Both implementations use 16 threads.

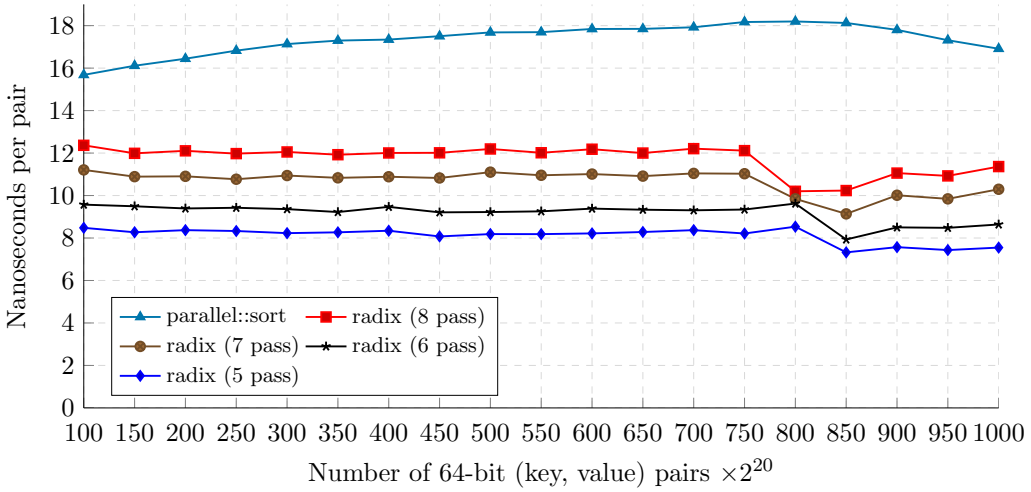


Figure 3.6: Running time comparison of parallel radix sort and parallel::sort for 64-bit (key, value) pairs. Keys are generated such that five to eight passes are necessary to sort the input using a digit size of $d = 8$ bits. Both implementations use 16 threads. Further investigation is necessary to explain the performance increase for sorting more than 750×2^{20} pairs.

3.4 Engineering All-to-All Communication

Fast and efficient all-to-all communication is crucial in both the traditional approaches like `Two-Phase` and `Repartitioning` as well as in the dSBF-based duplicate removal algorithm. While the former use one communication step to distribute the *entire* dataset across all nodes, the latter includes several calls to this collective communication operation, distributing Golomb-compressed hash values, bit arrays and input tuples.

In the terminology of the Message Passing Interface (MPI) [34] standard, this kind of operation is referred to as all-to-all communication. Depending on the flexibility of the operation, one further distinguishes MPI_Alltoallv (variable-length messages of the same data type) and MPI_Alltoallw (variable-length messages of potentially different data types). Both operations are crucial for the communication operations of the dSBF-based duplicate removal algorithm.

In Subsection 3.4.1 we analyze the performance of the standard MPI_Alltoallv implementation of OpenMPI in order to determine a baseline of the general network throughput of our high performance cluster described in Section 3.2. The results of this analysis clearly indicate the need for a different All-to-Allv implementation. Subsection 3.4.2 therefore introduces the 1-Factor algorithm [47], which is then experimentally evaluated in 3.4.3. The throughput of this implementation will serve as a baseline, against which the effectiveness of transferring Golomb-compressed hash values rather than plain hashes will be measured.

3.4.1 Analyzing the Library Algorithm

OpenMPI uses a pairwise message-exchange algorithm as default implementation of MPI_Alltoallv. Before communicating with other nodes, each PE copies the message destined for itself to the corresponding location within the receive buffer. Afterwards, message-exchanges with the other participants are performed in several communication rounds. In each of these rounds, messages are passed in a ring-like manner with an increasing stride (see Figure 3.7 for an example). The pseudo code of this algorithm is shown in Algorithm 4.

From a theoretical point of view, the algorithm is optimal for large messages in which case the startup overhead of a communication operation T_{start} is negligible compared to the time nT_{byte} it takes to exchange a message of size n. In each of the $p-1$ rounds, every PE directly sends the data destined for its *sendTo*-partner and directly receives the data destined for itself from the *recvFrom*-partner. Thus the complexity of the library algorithm is

$$p(T_{start} + nT_{byte}),$$

which is optimal for $n \to \infty$. However, the actual implementation of this default algorithm exhibits three shortcomings:

1. **Implementation Inefficiency:** Taking a closer look at the pseudo code together with the example reveals an inefficiency in the current implementation: Before entering the communication-loop, local data is copied from the send buffer to the receive buffer to avoid unnecessary overhead. Therefore only $p-1$ communication operations remain to be performed in the pairwise fashion. However, the loop condition actually enforces p pairwise exchanges - the last one being an additional exchange of node-local data. This exchange already happened before entering the loop and therefore leads to one unnecessary memcpy-operation.

[1] `http://www.open-mpi.org/hg/hgwebdir.cgi/ompi-svn-mirror/file/8ec1aa94dba9/ompi/mca/coll/tuned/coll_tuned_alltoallv.c`

Algorithm 4 Default All-to-Allv Algorithm in OpenMPI [1]

```
procedure OMPI_COLL_TUNED_ALLTOALLV_INTRA_PAIRWISE
    // PE index i ∈ 0, ..., p - 1
    receiveBuffer[i] ← sendbuffer[i]
    for j := 1 to p do                          ▷ Bug: should be j := 1 to p − 1
        sendTo ← (i + j) mod p
        recvFrom ← (i + p − j) mod p
        SENDRECV(sendTo, recvFrom)
    end for
end procedure
```

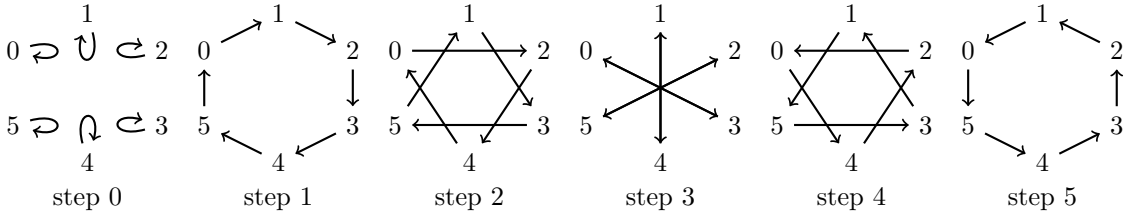


Figure 3.7: Communication pattern of OpenMPI's pairwise All-to-Allv algorithm with 6 nodes. Note that this example reflects the message exchanges in a correct implementation without the inefficiency. The current implementation performs another local exchange (step 0) at the end.

2. **Missing Optimization:** While other communication operations do not send messages if the corresponding message size is zero, `ompi_coll_tuned_alltoallv_intra_pairwise` uses a `Sendrecv` implementation that does not perform this optimization.[2]

3. **Synchronization Overhead:** Each round uses a *blocking* point-to-point operation for pairwise message exchange. Thus PEs have to wait for each communication operation to complete before they can initiate the next one. Rather than issuing all operations at once and then only wait for completion *once* at the end of the algorithm, the implementation has to wait for each of the p exchanges to complete. It thereby introduces an additional synchronization overhead that scales linearly with the number of nodes that participate in the collective communication operation.

All duplicate removal algorithms heavily rely on fast and efficient all-to-all personalized communication operations. Having identified these shortcomings, it is therefore necessary to evaluate the performance of OpenMPI's pairwise algorithm in order to answer the following questions:

- Given a fixed amount of data to distribute to all PEs. How does an increasing number of nodes affect the throughput of the MPI_Alltoallv implementation?
- Given a fixed number of nodes. How does increasing the message sizes affect the throughput?
- Do the identified deficiencies negatively affect the overall performance?

On a cluster that employs a high performance network one would expect to achieve a nearly constant network throughput regardless of the number of nodes and the overall transfer volume. To verify this assumption, we performed the following two experiments

[2] `http://www.open-mpi.org/hg/hgwebdir.cgi/ompi-svn-mirror/file/8ec1aa94dba9/ompi/mca/coll/tuned/coll_tuned_util.c`

# Nodes (p)	2	4	8	16	32	64
Throughput (Gb/s)	2.95	3.09	0.98	0.57	0.42	0.84

Table 3.1: Throughput of OpenMPI's default All-to-Allv algorithm (averaged over 50 iterations). Each node sends 1 GB of data. With increasing number of nodes the throughput decreases significantly.

Transfer Volume (Gb/Node)	1	1.41	2	2.82	4	5.65	8
Throughput (Gb/s)	1.02	0.88	0.72	0.84	0.83	0.55	0.79

Table 3.2: Throughput of OpenMPI's default All-to-Allv algorithm with 64 PEs and increasing communication volume (averaged over 50 iterations). The throughput is more than a factor of 3 apart from the possible network bandwidth.

on our distributed memory cluster: In the first experiment each node is given a fixed input size of 1 GB. For an increasing number of nodes, we measure the running time of the MPI_Alltoallv operation used to distribute the data evenly among the participants. The results are shown in table 3.1. While the throughput remains stable for 2 and 4 nodes, it decreases significantly as more and more nodes participate in the operation.

Table 3.2 shows the result of our second experiment, in which we kept the number of PEs constant (e.g., $p = 64$) and consistently increased the overall message transfer volume by a factor of $\sim \sqrt{2}$. The throughput remains almost constant.

However, given the fact that our cluster employs a high-speed InfiniBand interconnect with a theoretical bandwidth of 4 GB/s, the pairwise algorithm shows poor performance. Increasing the number of nodes results in a throughput that is way off the possible network bandwidth. Although this throughput remains relatively stable given a fixed number of PEs, the algorithm is still more than a factor of 3 away from peak performance for $p = 64$.

3.4.2 The 1-Factor Algorithm

In order to verify whether the resulting poor performance of the default MPI_Alltoallv algorithm is due to the current implementation or due to a general problem of the cluster configuration, we implemented an alternative algorithm for all-to-all communication: The 1-Factor algorithm [47]. Like the pairwise algorithm, it breaks down one collective communication operation into several rounds - each of which consists of distinct pairwise message exchanges.

The algorithm models a complete all-to-all operation as a communication graph $G = (V, E)$ with $V = \{0, ..., p-1\}$ and $E = \{\text{all pairwise message exchanges}\}$. Since every PE communicates with every other PE (including itself), G is a complete graph with a self-loop on each vertex. Sanders and Träff [47] prove the existence of a 1-factorization of this communication graph.

Each Edge $e \in E$ represents a pairwise message exchange that has to be performed in one of the communication rounds. In order to schedule these message exchanges, the algorithm exploits the existence of a 1-factorization. It performs p communication rounds - one for each 1-factor of G. In each round the nodes perform a pairwise message exchange along the edges of the current 1-factor.

Algorithm 5 shows the pseudo code for an *odd* number of PEs. In each round, one node communicates with itself (self-loop) while the others perform pairwise exchanges (see Figure 3.8 for an example).

Algorithm 5 1-Factor Algorithm (p odd)

procedure 1-Factor-Alltoallv
  // PE index $i \in 0, ..., p - 1$
  for $j := 0$ **to** $p - 1$ **do**
    Exchange data with PE $(j - i) \bmod p$
  end for
end procedure

Algorithm 6 1-Factor Algorithm (p even)

procedure 1-Factor-Alltoallv
  // PE index $i \in 0, ..., p - 1$
  for $j := 0$ **to** $p - 2$ **do**
    $idle \leftarrow \frac{p}{2} j \bmod (p - 1)$
    if i = p - 1 **then**
      $commPartner \leftarrow idle$
    else if i = idle **then**
      $commPartner \leftarrow p - 1$
    else
      $commPartner \leftarrow (j - i) \bmod (p - 1)$
    end if
    Exchange data with PE $commPartner$
  end for
  Exchange data with PE i ▷ self-communication
end procedure

If the same algorithm was used for an even number of nodes as well, there would be 2 PEs communicating with themselves every second round although they could exchange messages with each other. Therefore Algorithm 6 excludes the last node $p-1$ and performs Algorithm 5 on the remaining odd number of PEs. As we have already seen, this results in one node that is paired with itself. This node is now chosen to communicate with the previously excluded PE $p - 1$. A final message exchange accounts for the remaining self-loop 1-factor. Figure 3.9 shows the communication pattern of 6 nodes as an example.

Like the pairwise algorithm used in OpenMPI, the 1-Factor algorithm is also optimal if the time it takes to exchange a message is large compared to the startup overhead (i.e., optimal for large messages). However, it is somewhat simpler in terms of the communication pattern: It only exchanges messages in a pairwise distinct fashion. In the library algorithm, this is only the case in one of the p rounds.

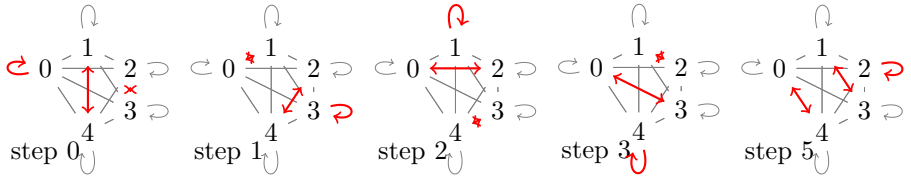


Figure 3.8: Complete Graph with 5 vertices and self-loops. The communication pattern used by the 1-Factor algorithm in each step is highlighted in red.

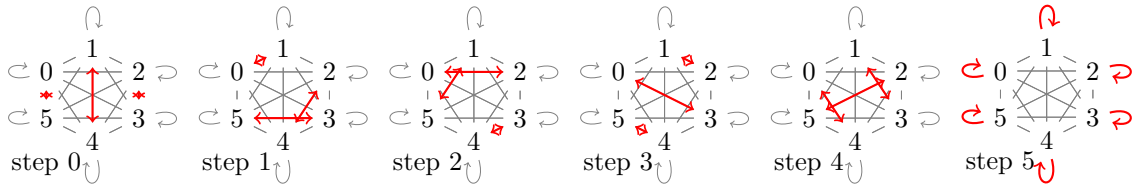


Figure 3.9: Complete Graph with 6 vertices and self-loops. The communication pattern used by the 1-Factor algorithm in each step is highlighted in red.

3.4.3 Experimental Evaluation

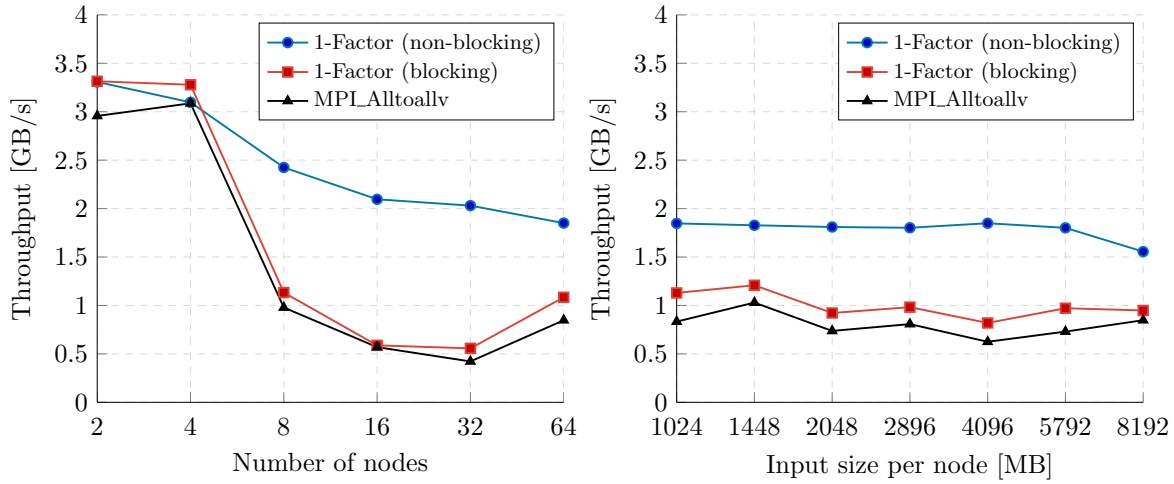
While most of the algorithm's peudocode can be directly translated into an implementation, there is one subtle detail that needs particular attention: The message exchange itself. In order to realize this pairwise communication, it is possible to use either *blocking* (e.g. MPI_Sendrecv) or *non-blocking* (e.g. MPI_Isend, MPI_Irecv) point-to-point communication primitives. Using blocking operations would potentially lead to the same inefficiencies as in the pairwise algorithm, whereas using non-blocking operations would allow to overlap the different communication rounds as much as possible. Rather than having to wait for *each* pairwise exchange to complete before another exchange can be started, a non-blocking implementation would put all outbound messages on the network and collectively wait *once* for all message transfers to complete.

To verify the hypotheses that the current implementation of the pairwise algorithm is deficient and that the 1-Factor algorithm can benefit from using non-blocking operations, we implemented both variants and measured their performance on the same experiments as described in Section 3.4.1. The results are shown in Figures 3.10a and 3.10b.

The results of our experiments clearly indicate the superiority of the non-blocking 1-Factor algorithm. In both experiments it met the expectations and outperformed the blocking variant as well as the pairwise algorithm. Increasing the number of PEs while holding the input size constant resulted only in a modest decrease in throughput. When increasing the input size for a fixed number of nodes, the throughput remained almost constant.

In Figure 3.10a the curves of the library algorithm and the blocking 1-Factor implementation progress in exactly the same manner. This is not surprising as there are only two minor differences between these two implementations. First, the blocking 1-Factor implementation does not copy local data twice, as it is done in the pairwise algorithm because of the inefficient implementation. Second, the communication patterns slightly differ: While in the blocking variant each node sends to and receives from exactly the same node, the library algorithm uses different communication partners for send and receive operations.

Key disadvantage of both approaches is the usage of blocking point-to-point operations. While the non-blocking implementation is able to overlap the p communication rounds and only has to wait once at the end for all operations to finish, both algorithms require each communication operation to be completed before the next operation can be started. Thus, the increase in performance and stability of the fastest implementation can be attributed to the usage of non-blocking communication operations. As can be seen in Figure 3.10b, the non-blocking 1-Factor algorithm remains more than a factor of 1.7 faster than OpenMPI's default implementation and more than a factor of 1.5 faster than its blocking counterpart. Since communication performance is crucial in all duplicate removal algorithms, we adopt the non-blocking 1-Factor algorithm as a drop-in replacement for OpenMPI's default All-to-Allv algorithm in all further experiments.



(a) Increasing number of nodes and fixed input size of 1024 MB per node.

(b) Fixed number of nodes $p = 64$ and increasing input size per node.

Figure 3.10: Throughput comparison of MPI_Alltoallv and both 1-Factor implementations (averaged over 50 iterations).

3.5 Integrating Golomb Compression

The distributed single shot Bloom filter introduced in Section 2.4.1 is built by distributing Golomb-compressed hashes across all nodes (batched insertion). Given the high performance networks of today's compute clusters and the results of the previous section, the question is whether the application of Golomb compression is actually able to reduce the overall communication time.

We therefore integrate the Golomb coder implementation of Putze et al. [44] into the 1-Factor all-to-all implementation and analyze the performance of this approach in Section 3.5.1. The results of this analysis show the need for parallelizing encoding and decoding in order to be competitive with uncompressed communication. Thus Section 3.5.2 devises three different parallelization approaches, which are then evaluated experimentally in Section 3.5.3.

3.5.1 Sequential Compression

The dSBF-based duplicate removal algorithm compresses a sorted sequence of hash values using Golomb coding before distributing them to the respective PEs that manage the corresponding part of the filter. This significantly reduces the amount of data that has to be sent over the network. However, it also introduces additional overhead since the hashes have to be encoded before and decoded after the send operation.

Encoding is done sequentially for each message (see Figure 3.11). For each PE i the corresponding hashes are encoded contiguously into the send buffer. Additionally we store some meta data for each Golomb code, which will be used for decoding by the receiving node. The meta data contains the Golomb tuning parameter b, the hash-offset that is needed due to difference encoding as well as the number of encoded hashes. In order to avoid an additional communication step to distribute this information, we store meta data and Golomb codes in consecutive memory locations. This allows us to distribute both using just one collective communication operation. The receiving node then uses the meta data information located at the beginning of each message to decode it sequentially.

Figure 3.12 shows the performance of sequential Golomb compression compared to directly sending the uncompressed hashes. Both algorithms use the 1-Factor All-to-Allv drop-in

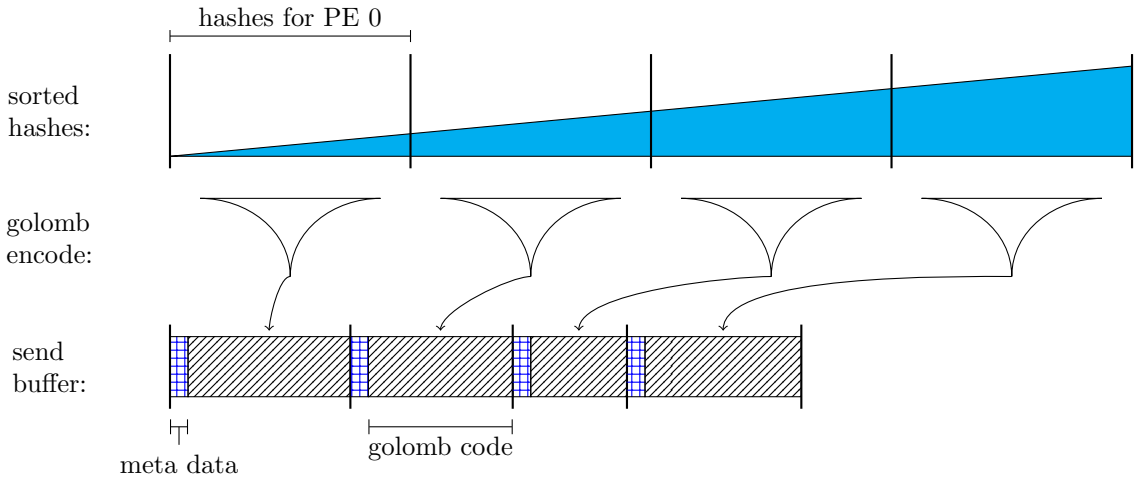
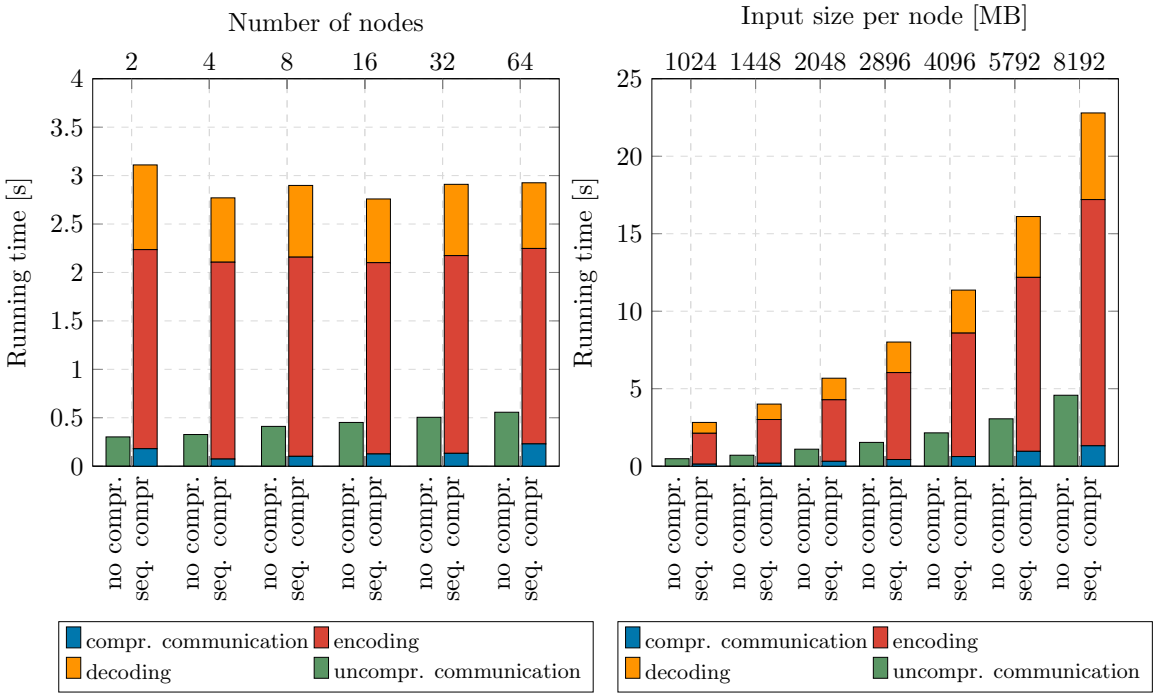


Figure 3.11: Sequential Compression: The messages are compressed sequentially starting with the message destined for PE 0. The resulting Golomb code along with the necessary meta data is put into consecutive memory locations. Therefore we can use a standard All-to-Allv call to distribute meta data and Golomb codes.



(a) Fixed input size of 1024 MB per node and an increasing number of nodes.

(b) Increasing the input size for a fixed number of $p = 64$ nodes.

Figure 3.12: Running time comparison: sending uncompressed and Golomb-compressed data using our 1-Factor implementation. Golomb compression reduces message transfer times. However, encoding and decoding also introduce a significant overhead.

replacement. The results clearly show that compression greatly reduces the communication time. Encoding and decoding however strongly dominate the overall running time. In the next Section we therefore devise several approaches to parallelize compression.

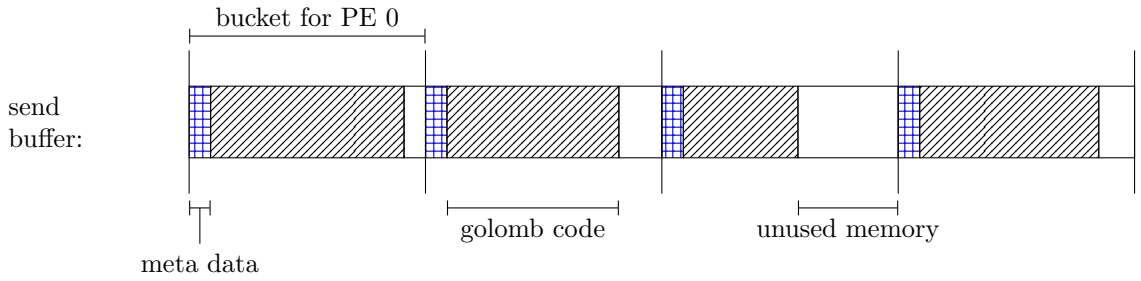


Figure 3.13: Send buffer layout for naïve parallelization approach assuming 4 PEs.

3.5.2 Parallelization Approaches

In order to speed up encoding and decoding it is necessary to parallelize both steps. In this subsection we assume a system with p nodes and t threads per node and present three parallelization approaches that build upon one another. All implementations avoid the overhead of encoding and decoding the bucket each PE sends to itself. Instead, these hash values are directly copied into the corresponding memory location of the receive buffer.

Naïve

Given t threads, it is possible to encode up to t messages in parallel. However, the sizes of each of the t resulting Golomb codes are not known until completion of the encoding process. Thus it is not possible to encode directly into contiguous memory locations of the send buffer in parallel. We therefore conceptually divide the send buffer into p buckets - one for each destination PE.

We again exploit virtual memory to account for the unknown Golomb code size of each bucket. Memory is allocated such that each bucket is big enough to contain the Golomb code of the entire input sequence. Thus it is ensured that each bucket is big enough to store meta and compressed data without overflowing into the neighboring bucket. Each of the t threads is therefore able to encode its corresponding hashes into the correct position of the send buffer without any synchronization with the other threads.

Figure 3.13 shows an example of a send buffer, which is conceptually divided into 4 buckets. Each bucket can further be subdivided into three different memory areas. At the beginning of each bucket the meta data is stored, followed by the actual Golomb code. Since we estimated the code sizes conservatively, each code is smaller than the bucket size - leaving some unused space at the end of each bucket.

This layout causes some fragmentation of the send buffer. Unlike in the sequential case, the compressed data does not reside in consecutive memory locations. It is therefore necessary to adjust the send displacements of the All-to-Allv operation accordingly to account for these offsets.

Decoding is done analogously to encoding. Using the meta data information at the beginning of each message, t threads concurrently decode their respective message content into the receive data structure.

Maximum Parallelism

If the number p of nodes is greater than the number t of threads per node, the naïve approach efficiently exploits parallelism. If, however, the number of nodes participating in the collective operation is less than t, it is not able to use all threads for encoding as well as decoding and therefore wastes potential for parallelization.

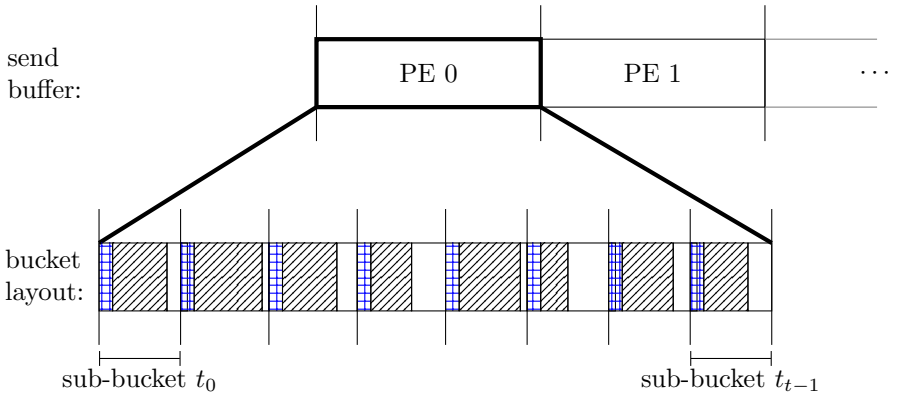


Figure 3.14: Send buffer layout for maximum parallelism approach. Each bucket is conceptually divided into t sub-buckets, which will be encoded and decoded in parallel.

To be able to always use the maximum number of threads (regardless of the number of nodes), we conceptually partition the send buffer layout of the naïve approach even further (see Figure 3.14). Each bucket is itself sub-divided into t sub-buckets that are big enough to contain the Golomb code of the entire input sequence.

Thus, it is possible to encode a bucket by encoding all sub-buckets in parallel, thereby always using all available threads. This approach however leads to further fragmentation of the send buffer. While the naïve approach has to account for p potentially unused memory locations, this approach needs to consider $p \cdot t$ gaps in the send buffer.

Due to this fragmentation it is not possible to use a standard All-to-Allv operation to send only those memory areas that contain actual data, because it does not allow for this kind of complex memory layout. We could just send the entire buffer (including the unused space), but devise a more elaborate approach to save the overhead caused by the fragmentation.

As described in the beginning of this section, the most general form of the all-to-all communication operation additionally allows the specification of a data type per bucket. We know the bucket sizes as well as the sub-bucket sizes before starting the encoding process. Furthermore, we know the actual sizes of the resulting Golomb codes once the encoding process is finished. It is therefore possible to create bucket-specific MPI data types that account for the individual sub-bucket structures of each bucket. Thus, we are again able to distribute all batched insertion messages using one collective communication operation and without having to transmit the data that resides in the unused memory locations.

Pipelined Compression

Both the naïve and the maximum parallelism approach treat the actual message exchange as a black box communication operation. This leads to an inherent sequentialization of the *encoding-transfer-decoding* process: The communication operation is only initiated once every bucket as been compressed and parallel decoding starts only after the collective communication operation is finished.

In order to prevent this implicit sequentialization, the pipelining approach exploits the fact that the all-to-all operation is implemented as p pairwise message exchanges. Instead of building encoding and decoding around the communication operation, it integrates compression directly into the 1-Factor algorithm described in Section 3.4.2: Messages are put into the compression pipeline (see Figure 3.15) in the order determined by the 1-Factor schedule.

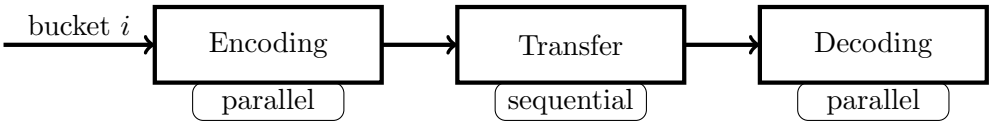
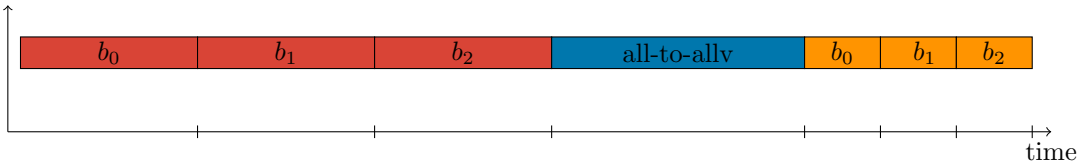


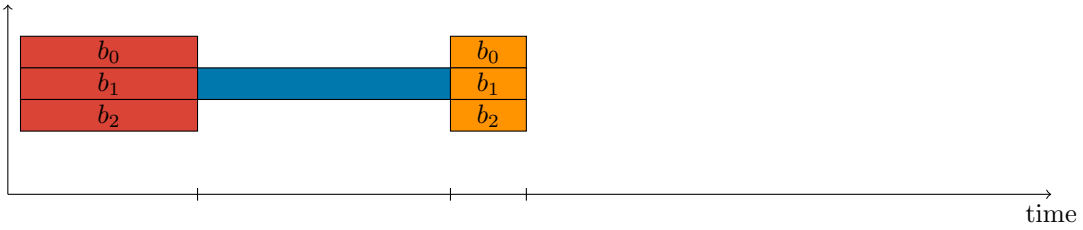
Figure 3.15: Three-stage compression pipeline. Encoding and decoding is done in parallel using t threads as described in the maximum parallelism approach.

As soon as a message is compressed, the corresponding message exchange with the destination PE is initiated and the next bucket is fed into the encoding stage. Likewise parallel decoding of a bucket is started as soon as a bucket leaves the transfer-stage (because it's pairwise communication operation is finished). By using this three-stage pipeline, this approach overlaps communication and computation as much as possible in order to further reduce the overall running time.

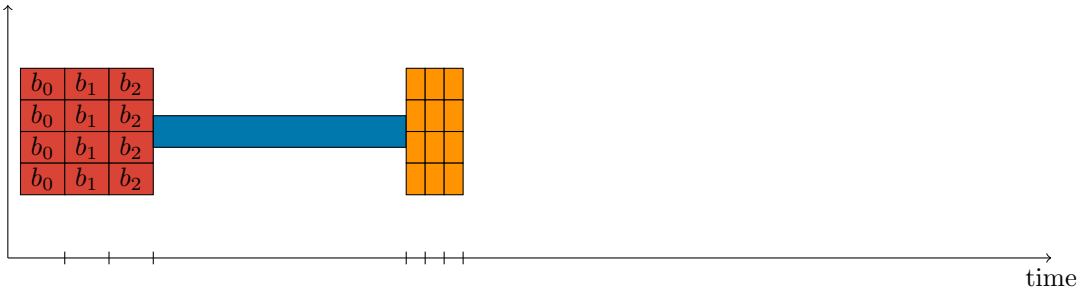
Figure 3.16 summarizes the different parallelization approaches and illustrates the implications on the total running time and the ability to efficiently exploit shared-memory parallelism.



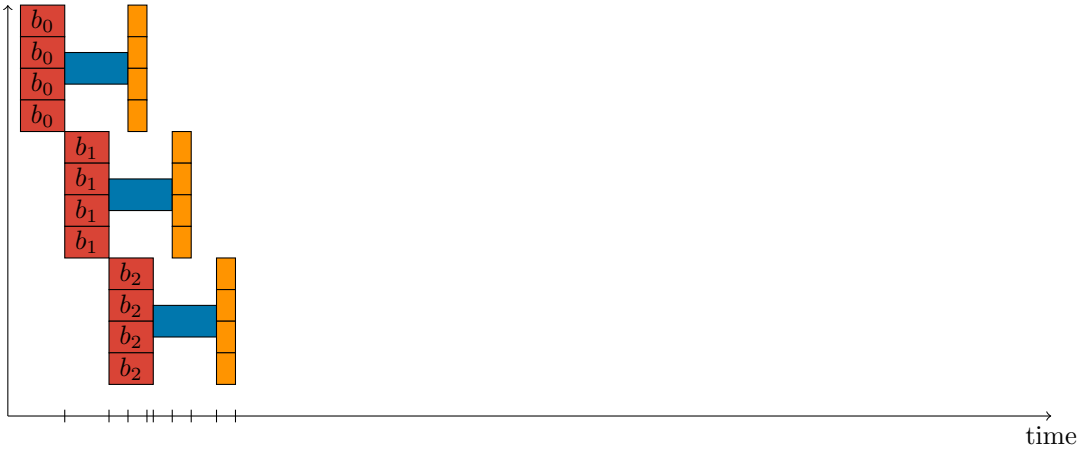
(a) Sequential compression: Buckets are encoded and decoded sequentially.



(b) Naïve parallelization: Buckets are encoded and decoded in parallel. If $p \geq t$ this approach efficiently exploits shared-memory parallelism.



(c) Maximum parallelism: Assuming a cluster configuration with $t = 4$ threads per node, the naïve approach would not be able to utilize all threads. By sub-dividing each bucket into sub-buckets, the maximum parallelism approach is able to always use all available system resources for encoding and decoding, even if $p < t$.



(d) Pipelined compression: Encoding, transfer and decoding of each bucket is interleaved in a pipelined fashion to overlap computation and communication as much as possible. This approach explicitly uses p pairwise message exchanges rather than one single collective communication operation.

Figure 3.16: Chronology of the different approaches to integrate compression into the all-to-all communication operation assuming a system configuration with $p = 3$ nodes and $t = 4$ threads per node.

3.5.3 Experimental Evaluation

We implemented all three approaches using Intel® Threading Building Blocks (Intel® TBB) 4.1 as well as OpenMP 3.1 for shared-memory parallelization. Since the TBB-based implementations consistently performed better, this section does not present any OpenMP-based results. In order to provide comparability to the results of the previous sections, we again use the benchmarks described in Section 3.4.1 to evaluate the performance of the three approaches experimentally. In the following, we refer to the implementation of the naïve parallelization approach as `naïve`, of the maximum parallelization approach as `max-par`, and of the pipeline approach as `pipe`.

Figure 3.17 shows the results (averaged over 50 iterations) for a fixed input size of 1024 MB per node and an increasing number of nodes. As predicted in the previous subsection, `naïve` is not able to exploit potential parallelism as long as $p < t = 16$. However, as soon as the number of nodes participating in the communication operation is greater than the number of threads per node, `naïve` shows better overall performance than `max-par`.

While the encoding- and decoding-performance of both approaches is comparable, the actual communication time of `max-par` is consistently higher than that of `naïve`. This difference can be explained with the increased complexity of the send buffer's memory layout: In `naïve` it is sufficient to use an All-to-Allv operation and standard MPI data types, because memory fragmentation can be addressed by adapting the send displacements. The `max-par` algorithm however needs to create a specific MPI data type for each bucket in order to be able to distribute the Golomb codes using one All-to-Allw operation.

The performance of the pipelined approach is not as good as anticipated theoretically. This can be explained by some implementation details: In order to achieve true overlapping of communication and computation, it is not sufficient to rely on non-blocking communication operations like `MPI_Isend` and `MPI_Irecv`. A first implementation of `pipe` that solely used p non-blocking communication operations did not perform any better than `max-par`. The reason for this is that according to the MPI standard [34], the usage of a non-blocking communication operation does not necessarily imply asynchronous execution of computation and communication. Thus MPI libraries do not have to account for asynchronous progress in order to be standard-compliant. The current version 1.6.4 of OpenMPI, for example, does not support this feature (called "progress threads")[3]. Instead, the actual communication is performed in the subsequent `MPI_Test` or `MPI_Wait` calls [25, 67].

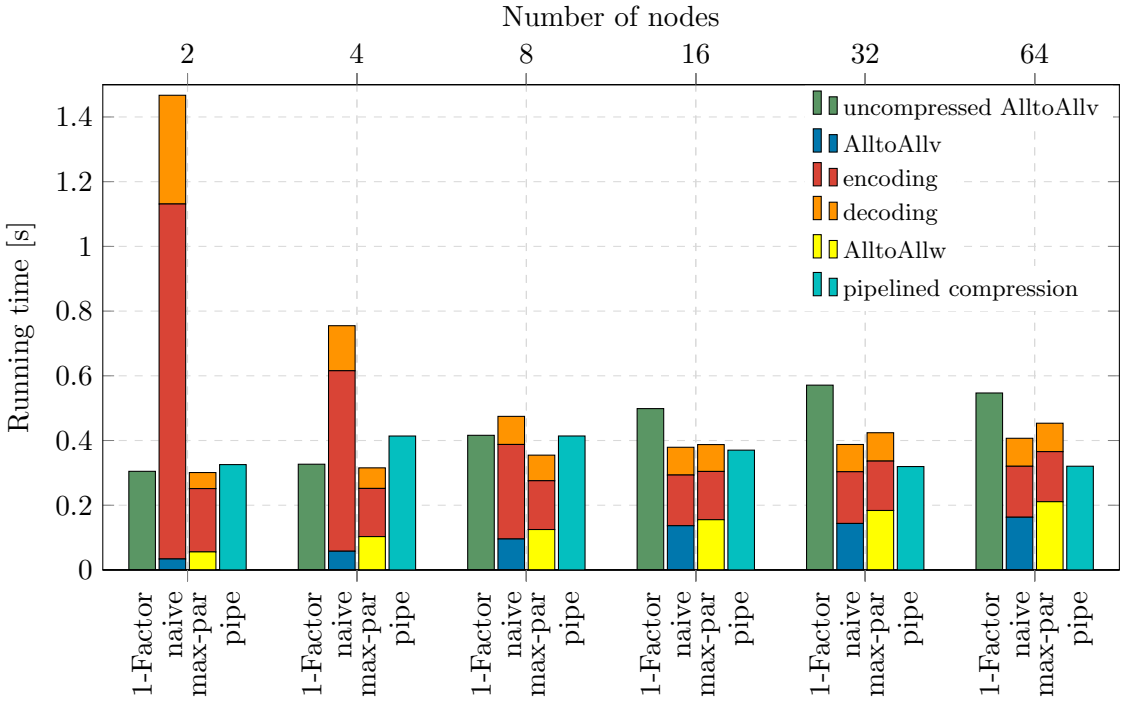
In order to be able to explicitly overlap encoding and decoding with the distribution of Golomb codes, we use a manual progression technique [25]: A special *progression task* is spawned at the beginning of the encoding process. On execution, it issues the `MPI_Irecv` calls in 1-Factor order and then repeatedly calls `MPI_Waitsome` to drive message progression and to check for completion. As soon as one or more transfers completed, the corresponding messages are decoded in parallel.

By sacrificing one thread for this task, it is possible to overlap compression and computation to a certain degree as can be seen in Figure 3.17. For $p \geq 16$ `pipe` performs better than both other parallel approaches and the uncompressed message transfer.

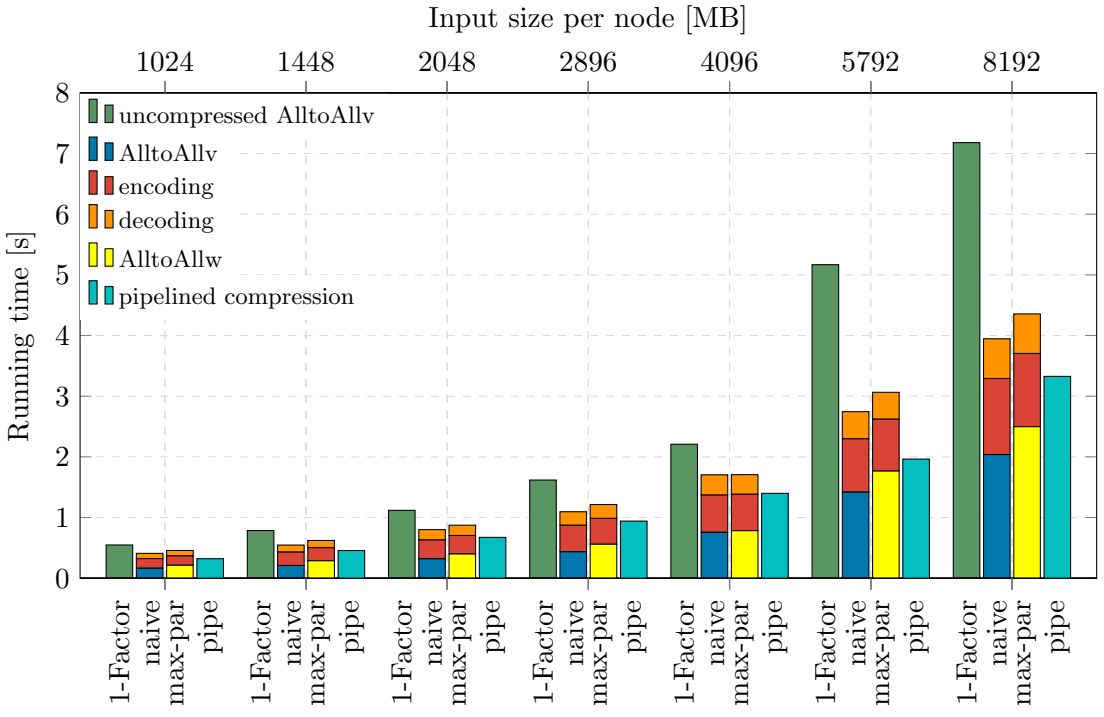
Figure 3.18 compares the throughput of our parallel Golomb compression implementations with the throughput of sending uncompressed data. For a sufficiently large number of PEs ($p \geq 16$), the pipelined compression algorithm outperforms its competitors. For $p = 64$ PEs, it achieves a throughput of more than 2.5 GB/s. Considering that the network of our cluster provides a theoretical point-to-point bandwidth of 4 GB/s, we conclude that for a reasonable amount of nodes, compression is still effective in reducing the overall

[3] `http://www.open-mpi.org/community/lists/users/2012/06/19526.php`

running time of collective communication operations - even on high-performance networks like InfiniBand.

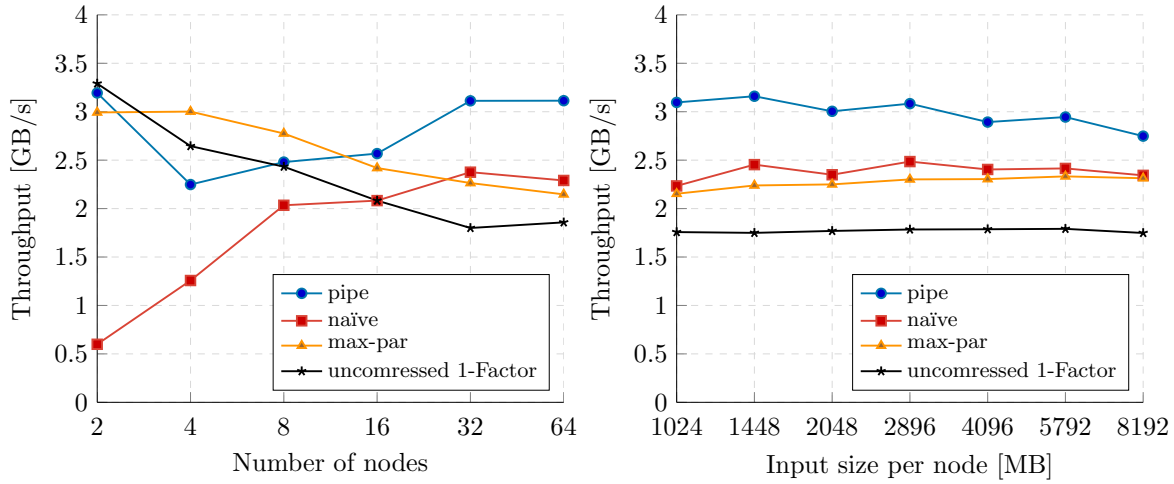


(a) Fixed input size of 1024 MB per node, increasing the number of nodes.



(b) Increasing input size, fixed number of nodes $p = 64$.

Figure 3.17: Running time comparison: sending uncompressed and Golomb compressed data. Compression is performed using the three parallelization approaches. Employing parallel compression reduces the overall communication time even on a high-performance interconnect.



(a) Fixed input size of 1024 MB per node and an increasing number of nodes.

(b) Increasing the input size for a fixed number of $p = 64$ nodes.

Figure 3.18: Throughput of our parallel Golomb compression approaches compared to sending uncompressed data using our 1-Factor implementation. For a sufficiently large number of nodes, all compressed communication operations outperform the uncompressed 1-Factor distribution.

3.6 Engineering Collision Detection

The dSBF-filtering algorithm as described in Section 3.1 consists of a local preprocessing step that creates the batched insertion messages, a distribution step that communicates these messages to their corresponding destinations and a collision detection step that identifies dSBF positions that would be set to 1 multiple times if the filter was actually materialized on each PE. Engineering a shared-memory parallel implementation of this step is the subject of this section.

Each received and decompressed batched insertion message consists of a sorted sequence of hash values. A parallel multiway-merging algorithm can be employed to identify hashes that occur multiple times. The identified duplicate hash values have to be signaled back to their corresponding source PEs. This can be done sending each source PE a dedicated response bit-vector of size n, where n is the number of hash values contained in the batched insertion message. A one-bit indicates that the hash occurred multiple times. A zero-bit represents hash values that occurred only once during the merging phase.

Section 3.6.1 introduces an implementation that builds upon the parallel multiway-merging algorithm [53] provided by the GNU C++ library. Section 3.6.2 then details an advanced parallel collision detection algorithm that reuses the key components of the parallel multiway-merger and that is specifically tailored to our problem setting.

Since both algorithms are tightly coupled to our distributed duplicate removal algorithm, we defer their experimental evaluation to Section 4.3.

3.6.1 Library-based Multiway-Merging

The pseudo code of our library-based multiway-merging collision detection approach is shown in Algorithm 7. Assume p received batched insertion messages and t threads that should perform the merging operation in parallel. To be able to set the bits in the bit-vector responses during the parallel merging operation, it is necessary to know the source of each hash value along with the corresponding position in the batched insertion message.

Algorithm 7 Multiway-merging collision detection

procedure DetectHashCollisions(received dSBF insertion messages)
  parallel foreach $hash \in receivedMessage\ \forall PE$ **do**
    $mergeTuples \leftarrow mergeTuples \cup \{SourcePE, Index, Hash\}$
  end parallel foreach
  foreach $i \in [0, t)$ **do**
    foreach $message \in receivedMessages$ **do**
      $threadLocalBitVector[i] \leftarrow threadLocalBitVector[i] \cup BitVector(|message|)$
    end for
  end for
  parallelMultiwayMerge($mergeTuples, CollisionComparator$)
  parallel foreach $i \in [0, p)$ **do**
    $filterResponseMessage[i] \leftarrow \bigvee_{threads} threadLocalBitVector[i]$
  end parallel foreach
end procedure

Algorithm 8 Multiway-merging Comparator

procedure CollisionComparator($MergeTuple\ a$, $MergeTuple\ b$)
  if $a.hash == b.hash$ **then**
    $threadLocalBitVector[threadID][a.SourcePE][a.Index] = 1$
    $threadLocalBitVector[threadID][b.SourcePE][b.Index] = 1$
  end if
  return $a.Hash < b.Hash$
end procedure

Unfortunately, it is not possible to pass this information directly into the parallel multiway-merging algorithm, as the only way to influence the merging process is the use of a custom comparator function. Because the parallel multiway-merging algorithm operates on copies of the input elements, it is not possible to infer either source PE or index from within the comparator.

The first step of our algorithm is therefore to create an augmented input sequence in parallel. Each hash value is augmented with the ID of the source PE and the index within the corresponding message. Using these triples in the comparator shown in Algorithm 8, it is possible to directly set the corresponding response bits in case two hash values are equal. Note that we do not set the bits directly in the response bit-vectors, since current bit-vector implementations like `boost::dynamic_bitset` can not handle concurrent writes to arbitrary bit positions. In order to avoid costly fine-grained synchronization operations, each thread has its own copy of the response bit-vectors. The final response messages are created after the merging operation by OR-ing the thread-specific bit-vectors for each message.

3.6.2 Parallel Tournament Tree Collision Detection

Using the parallel multiway-merging algorithm as a black box operation leads to significant administrative overhead, because it is necessary to augment the input sequences with meta data and provide thread-specific result data structures in order to compensate for the limitations of the library interface. In order to bypass these limitations, the tournament tree collision detection algorithm reuses the key components of the multiway-merging algorithm to create an implementation that is specifically tailored to our problem, i.e., detecting hash collisions and directly setting the corresponding response bits in parallel.

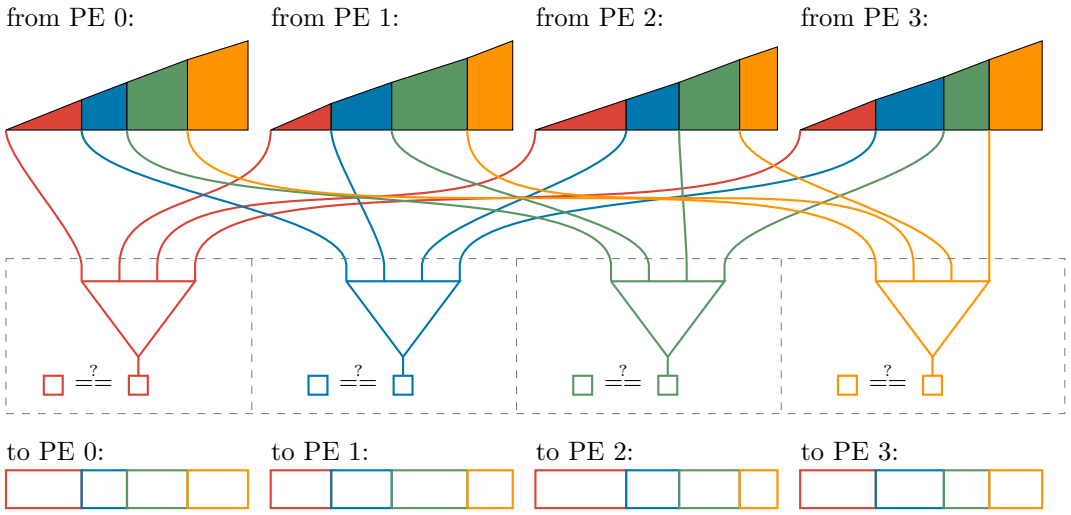


Figure 3.19: Parallel Tournament Tree Collision Detection assuming $p = 4$ received batched insertion messages and $t = 4$ threads to perform the collision detection in parallel. Each thread merges one partition and sets the corresponding bits in the response bit-vector in case of a collision.

Our approach is depicted in Figure 3.19. We reuse the multisequence partitioning implementation [53] to partition the input sequences such that they can be merged in parallel using t threads. The actual collision detection on each of these partitions is then performed as follows: Each thread manages a tournament tree data structure as a p-way merger. We again reuse the corresponding highly-tuned library implementation [52]. As we are not interested in the overall results of the merging process, i.e., the completely merged sequence of all received hash values, we do not materialize this output sequence. Instead, each thread only keeps track of two hash values: The current minimum and the previous minimum. Whenever these two hashes are equal, the thread sets the corresponding bits in the response bit-vectors.

Bit-vector data structures use integer data types as the underlying storage format for their bits. By carefully aligning the partition boundaries to the size of these data types, we can limit parallel random access to the bit-vector such that each integer is only accessed by one thread. This allows us to directly work on the response bit-vectors in parallel instead of having to create thread-specific temporary data structures.

By explicitly managing the merging process rather than relying on a black-box operation, we are able to use the information that is already inherent in the structure of the input sequences, i.e., the source PE of each hash value and the current index within the respective batched insertion message. Therefore, it is not necessary to store this information along with each hash value.

The tournament tree-based collision detection algorithm thus completely eliminates the additional administrative overhead of the approach that just reused the parallel multiway-merging algorithm.

4. Evaluation

It is the goal of this thesis to complement the theoretical analysis of Sanders et al. [48] with an experimental evaluation that analyzes the practical performance of the single-pass and two-pass dSBF-based duplicate removal algorithms. We therefore provide a detailed comparison of the running times of both implementations and the competitor that is best suited for our test setup, which will be introduced in Section 4.1.

The benefits of filtering distinct elements decrease as the total number of duplicates increases. Thus, the algorithm is expected to excel in situations where the input dataset is duplicate-free or only contains relatively few duplicates. By varying the number of duplicates, we determine the tipping point up to which dSFB-based filtering using our implementations is considered beneficial.

Finally, in order to evaluate the communication efficiency of our implementations, we analyze the total amount of data that is transferred over the network during the duplicate removal process.

4.1 Test Setup

We use the following test setup for all experiments presented in this section:

Our input relation consists of 2^{27} records per node and hence scales linearly with the number of nodes. The records have a fixed size of 104 Byte. Thus, the dataset of each PE resembles a database table of 13 GB in size. Since we are interested in the distributed duplicate removal problem, we generate these input records such that no intra-node duplicates exist, i.e., all duplicates are distributed over the nodes. We therefore omit an initial duplicate removal phase that would otherwise be necessary.

In order to identify the tipping point up to which dSBF-filtering is beneficial, we control the total number of duplicates using the duplication factor α. Let $n = 2^{27} \times p$ be the total number of input records of all p PEs participating in the duplicate removal operation. Then the input data is generated such that it contains αn duplicates and $(1 - \alpha)n$ unique records.

In the remainder of this chapter, we refer to the dSBF-based duplicate removal algorithm that uses a single filtering pass as `1dSBF`, while the algorithm that uses two filtering passes is referred to as `2dSBF`. The overall communication volume of both algorithms depends on the optimal choice of the false positive rate f of the employed distributed single shot

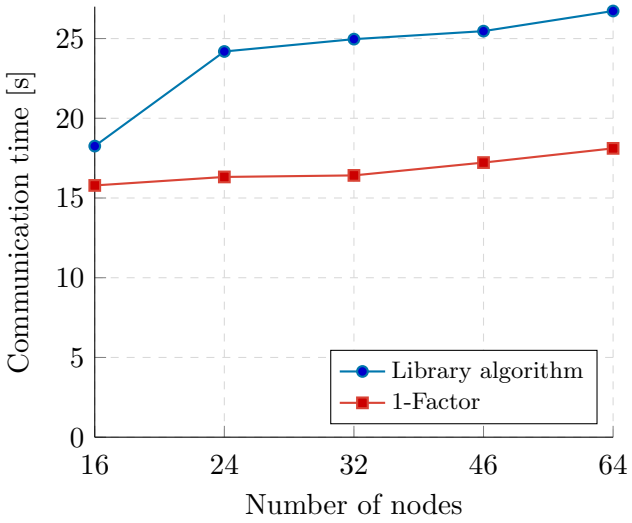


Figure 4.1: Communication time of the `Repartitioning` algorithm using our 1-Factor implementation as well as the OpenMPI's default algorithm. The input relation did not contain any duplicates, i.e., $\alpha = 0$.

Bloom filter. We choose f according to the theoretical analysis of Sanders et al. [48], i.e., $f = 1/u\ln(2)$ for `1dSBF` and $f_1 = 1/ln(2)ln(up) + 0.746$ and $f_2 = 1/u\ln(2)$ for `2dSBF`, where u denotes the size of an input record in bits and p is the number of PEs. In all our experiments the records have a fixed size of 104 Byte, therefore $u = 832$ bits.

We test our implementations on the same distributed memory cluster system using the same compiler and compile flags as described in Section 3.2. However, we had to refrain from using our self-compiled version of OpenMPI 1.6.4 with multithreading support, since it turned out to be instable when used in the cluster environment. We therefore use the library provided by the cluster system for all following experiments. The system currently provides OpenMPI version 1.6.3 compiled without multithreading support.

All reported timing values are averages of the elapsed wall-clock time of otherwise unloaded nodes. For each experiment, we average all timing values over all PEs and over 25 successive iterations.

4.2 Choosing the Competitor

As shown by our review of related work in Section 2.1, the hash-based `Repartitioning` algorithm is deemed to be the best-suited algorithm for datasets that do not contain any intra-node duplicates, because it avoids the (in this case unnecessary) local duplicate removal phase and achieves good load balancing by distributing work evenly over the PEs. We therefore choose this algorithm to serve as the competitor to our dSBF-based duplicate removal algorithm and provide a tuned reference implementation. The implementation is referred to as `RePart` throughout this chapter.

In the `Repartitioning` algorithm, each PE hashes its input records into p messages, distributes these messages to the p PEs participating in the distributed duplicate removal operation, and executes a single-system duplicate removal algorithm on the received messages.

We parallelized the hash-partitioning of the input tuples using OpenMP. As our analysis in Section 3.4 showed, the choice of an appropriate collective communication operation is crucial - especially in the case where huge amounts of data are to be transferred over the network. Because the input relation of our test setup is considerably larger than the data

we transferred in the experiments presented in Section 3.4, we confirmed our observations regarding the performance issues of OpenMPI's default algorithm. The results are shown in Figure 4.1 and validate the findings of our microbenchmarks. We therefore used the 1-Factor implementation to distribute input records in all experiments presented in this chapter.

The final duplicate removal on the received input tuples can be performed by one of the sequential/single-system algorithms described in Section 2.1. Our implementation uses the traditional *sort+scan* approach. The sorting phase is parallelized using the parallel version of `std::sort` provided by the GNU C++ library. A sequential scan then identifies any duplicates. We also experimented with approaches based on hash tables. However, none of these approaches was able to outperform the parallel sort and successive scan approach.

4.3 Implementation Choices

The dSBF-based duplicate removal algorithm executes the `Repartitioning` algorithm on all tuples that passed the filtering phase. We therefore settle the implementation choices regarding the communication of the input tuples and the final duplicate removal phase as described in the previous section. The filtering phase, however, offers additional potential for optimization as detailed in Sections 3.3 through 3.6.

The impact of each of our engineering efforts is depicted in Figure 4.2, which shows the normalized running time of the `1dSBF` algorithm executed on 64 PEs and a duplicate-free dataset. Other configurations as well as the `2dSBF` algorithm yield similar results and are therefore omitted. For each implementation choice, the left bar shows the total running time of the algorithm when all optimizations are switched on. The right bar shows the running time for the same configuration except that the optimization in question is turned off.

Using our parallel radix sort implementation reduces the running time of the sorting phase by a factor of 1.6 compared to the comparison-based parallel multiway-merging algorithm. This is consistent with the results in Section 3.3. The input of both algorithms consists of (*key*,*value*) pairs, where the *key* corresponds to the hash value produced by hashing the corresponding input tuple with index *value*. Based on our test setup and the optimal false positive rate, 43 bits are necessary to cover the range of hash values (i.e., dSBF bit positions). Thus, the parallel radix sort performs 6 passes over the data.

Transferring the batched insertion messages of the dSBF-filtering phase in Golomb-compressed format results in a $\sim 40\%$ decrease in the corresponding communication time. This confirms our assumption that compression is worthwhile even on systems that employ high-performance interconnects. When compared to the gains of compression reported in Section 3.5, this decrease in communication time seems to be less than expected. The reasons for this behavior are twofold: As the OpenMPI library provided by our cluster system does not offer multithreading support, we were not able to use our highly-tuned pipelined compression algorithm. Instead, all experiments reported in this chapter use the naïve algorithm. Furthermore, the sizes of the batched insertion messages are on the lower spectrum of the message sizes evaluated in Section 3.5. We therefore expect the benefits of compression to further increase with increasing size of the input relation.

Tournament tree-based collision detection turns out to be the most successful optimization, reducing the running time of the collision detection phase by a factor of more than 4 when compared to the library-based multiway-merging collision detection algorithm. This significant decrease can be attributed to the complete elimination of administrative overhead that was necessary when reusing the parallel multiway-merging routine, even though these administrative tasks were performed in parallel.

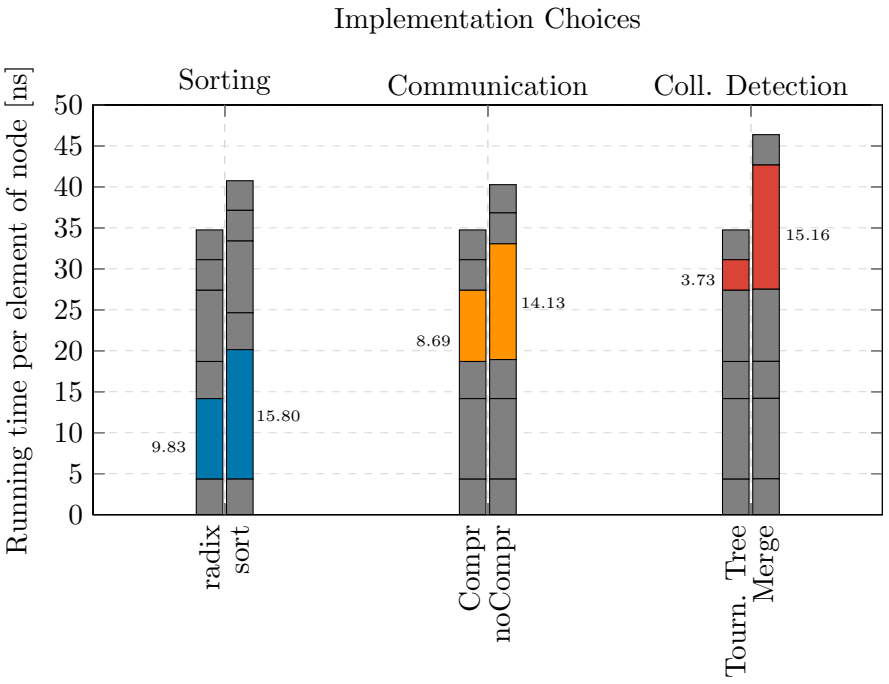


Figure 4.2: Impact of our engineering efforts on the total running time of the `1dSBF` algorithm, executed on 64 PEs and a dataset that does not contain any duplicates. For each implementation choice, the left bar shows the total running time with all optimizations turned on. The right bar shows the same configuration except that the optimization in question is turned off.

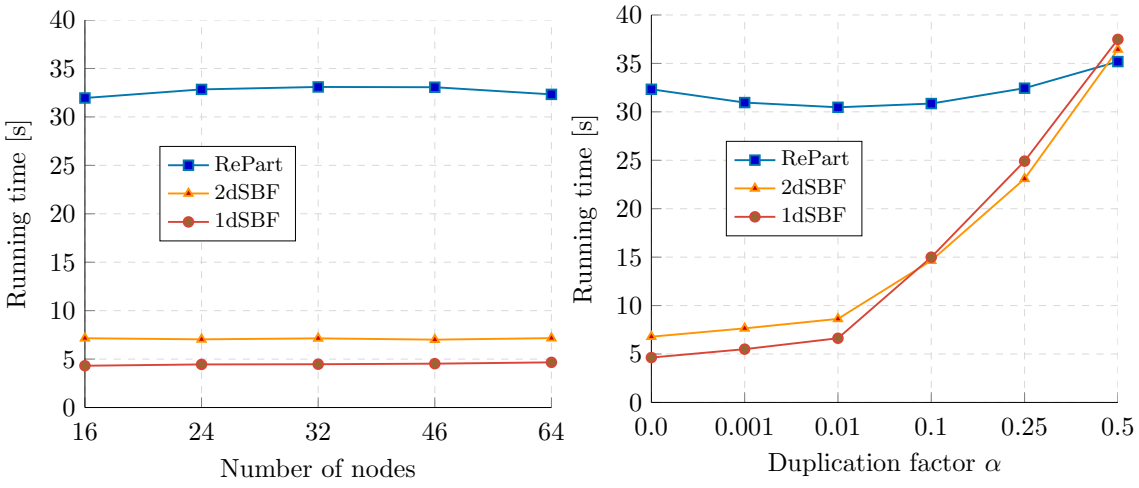
Based on this analysis, all following experiments use the parallel radix sort implementation in the sorting phase, Golomb compressed communication to distribute the batched dSBF insertion messages, and the tournament tree-based collision detection algorithm.

4.4 Experimental Results

The total running times of our dSBF-based filtering algorithms and the `Repartitioning` algorithm are shown in Figures 4.3a and 4.3b. The results confirm the practicability of the dSBF-based duplicate removal algorithms, as both `1dSBF` and `2dSBF` outperform `RePart`. For an increasing number of nodes, the running time of our algorithms remains constant, while the running time of our competitor slightly increases. As can be seen in Figure 4.3b our algorithms are faster than the traditional hash-based algorithm up to the point where the dataset consists of 50% duplicate records. Thus, dSBF-based duplicate removal is beneficial even in cases where the input dataset contains a significant amount of duplicates. The two-pass filtering algorithm is slightly slower than its single-pass counterpart except for datasets with more than 10% duplicates.

In order to verify that the performance benefits of our dSBF-based duplicate removal algorithms can be attributed to their communication efficiency and to provide further insights into the differences of `1dSBF` and `2dSBF`, we analyze the different phases of all our implementations in more detail.

As can be seen in Figure 4.4, the *total* running time of both dSBF-based algorithms is significantly smaller than the time it takes `RePart` to *communicate* the duplicate-free input dataset. This performance advantage is independent of any potential optimizations applied to the computation phases of the `Repartitioning` algorithm and confirms that communication efficiency is crucial for the overall performance of distributed duplicate removal algorithms.



(a) Total running time on a dataset that does not contain any duplicates.

(b) Total running time for $p = 64$ PEs and an increasing duplication factor.

Figure 4.3: Total running time of the repartitioning algorithm and the dSBF-based filtering algorithms. Our dSBF-based implementations outperform `RePart` for datasets with less then 50% duplicates.

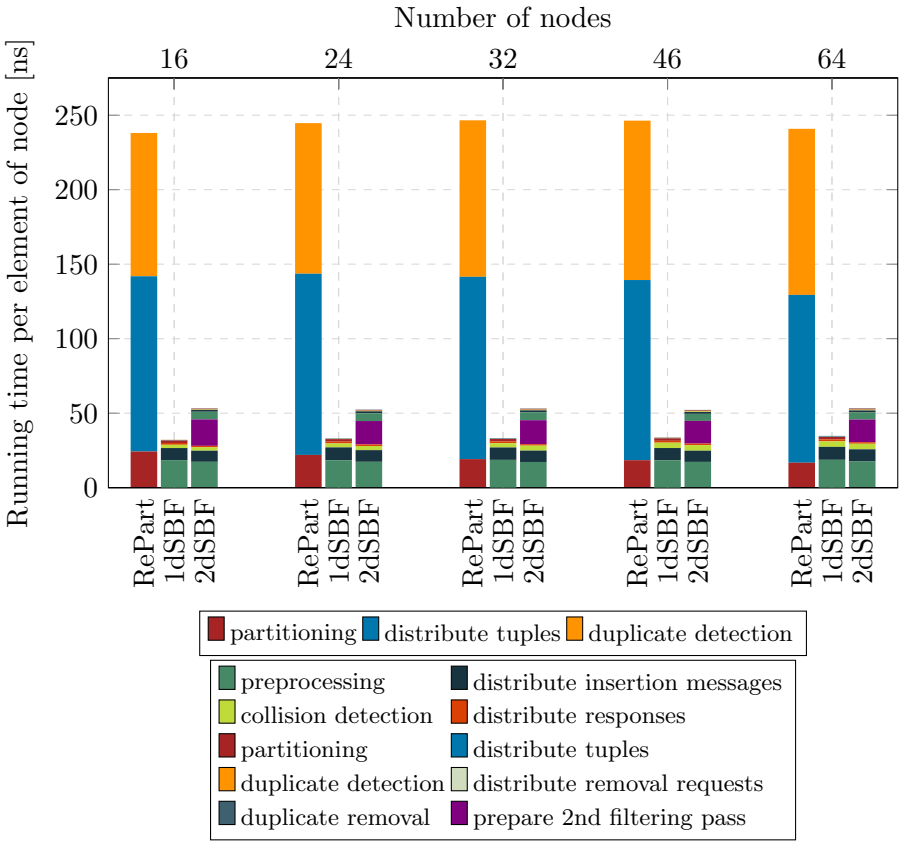


Figure 4.4: Breakdown of the running time of the different phases for a duplicate-free dataset. The *total* running times of `1dSBF` and `2dSBF` are significantly smaller than the time it takes `RePart` to *communicate* the input data.

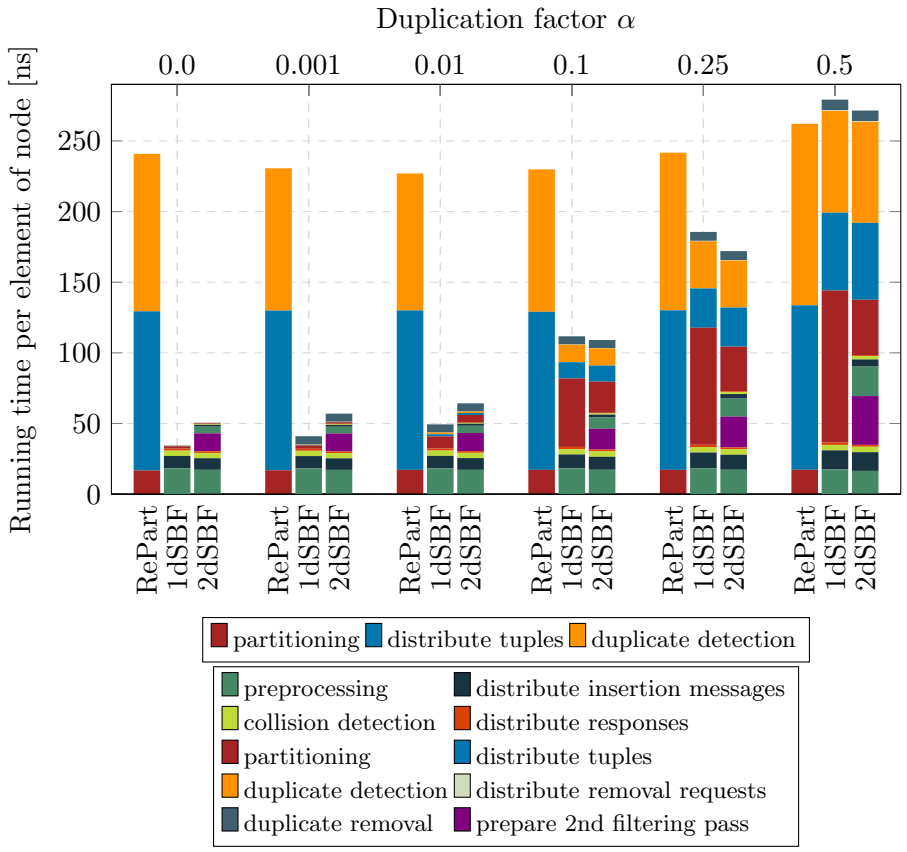


Figure 4.5: Breakdown of the running time of the different phases for a dataset with an increasing number of duplicates and $p = 64$ nodes. Both implementations offer potential for optimization in order to change the tipping point in favor of our dSBF-based algorithms for $\alpha = 0.5$.

The dSBF-based algorithms fall back to the `Repartitioning` algorithm on all records that could not be identified as distinct. As can be seen in Figure 4.5, this post-processing phase (■ partitioning, ■ distribute tuples, ■ duplicate detection) starts to dominate the overall running time for datasets that contain more than 10% duplicates.

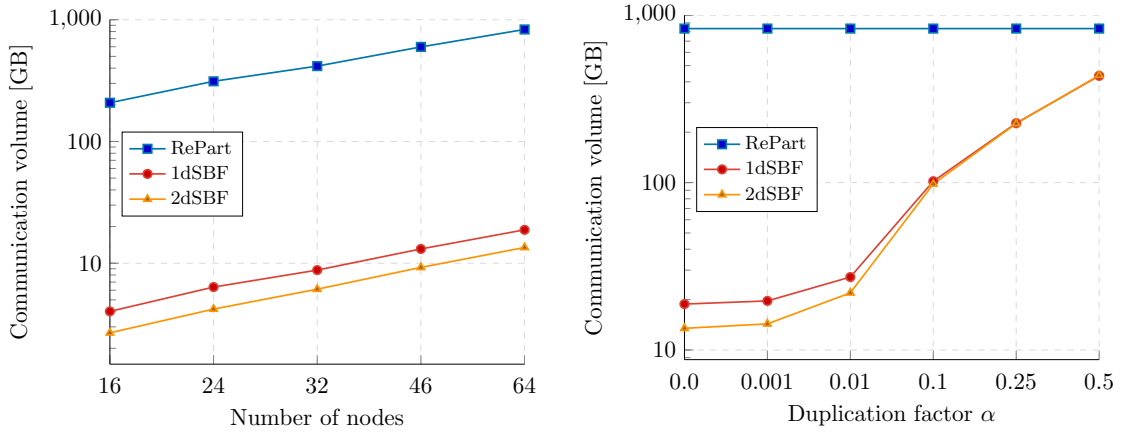
A detailed look at Figures 4.4 and 4.5 reveals potential for further optimizations of our implementations. While the running time of the partitioning phase of `1dSBF` and `2dSBF` remains unnoticeable for duplicate-free datasets, it significantly increases with an increasing number of duplicates. Both dSBF-variants post process the tuples that pass the filtering phase. Thus, one would expect a modest increase in the running times of the phases corresponding to the `Repartitioning` algorithm. While this is true for the distribution of the input tuples and the final duplicate detection phase, the partitioning phase exhibits unexpectedly longer running times - even longer that the corresponding phase of `RePart`. This is surprising, as `1dSBF` and `2dSBF` partition only those elements that passed the filtering phase, while `RePart` always partitions the entire input dataset. Furthermore, the running time of `2dSBF` currently is negatively influenced by a preparation phase for the second filtering pass.

Both of these shortcomings can be traced back to a particular implementation detail: The partitioning phase, as well as the 2nd pass preparation phase, operates on the bit-vector responses of the dSBF. Each 1-bit in the responses indicates that the hash value in the respective batched insertion message produced a hash collision. `1dSBF` therefore hash-partitions the input tuples corresponding to these hash values (■ partitioning). `2dSBF` uses these tuples as input for the second filtering pass (■ prepare 2nd filtering pass).

To find all 1-bits in the dSBF responses it is therefore necessary to iterate over the bits of the bit-vectors. In our current implementation, we use the `boost::dynamic_bitset`[1] implementation to represent these bit-vectors, which proved superior to alternative bit-vector implementations [43]. While iterating over these bitsets is cheap in case the bit-vector response is sparse (i.e., the dSBF reported only a few hash collisions), this operation becomes a bottleneck as the number of 1-bits in the dSBF responses increases. This, however, is exactly the case for an increasing number of duplicates and additionally slows down the `2dSBF` implementation, because the two-pass algorithm chooses a high false positive rate f for the first filtering pass, which leads to an increasing number of hash collisions that are reported back by the distributed single shot Bloom filter.

We therefore believe that by optimizing our implementation further, we can eliminate these bottlenecks and thus further reduce the running times of our dSBF-based duplicate removal algorithms on datasets that contain up to 50% duplicates. The optimizations might even change the tipping point, at which `RePart` is more beneficial than either `1dSBF` or `2dSBF`, in favor of our dSBF-based duplicate removal algorithms.

In our current implementation, `2dSBF` is slightly faster for datasets that contain more than 10% duplicates. Note, however, that this is only the case because of the previously described shortcomings of our implementations. If these overheads were eliminated completely, the two-pass variant would always be slightly slower than its single-pass counterpart. This result stands in contrast with the theoretical analysis of Sanders et al. [48] that assumes the two-pass pass algorithm to be faster, because it minimizes the overall communication volume even further than `1dSBF`. This reduction in communication volume comes at the cost of a second dSBF-preprocessing phase, in which all remaining input records that passed the first filter have to be hashed, sorted, and compressed again. The slight decrease in communication time, however, is not able to offset the additional computation costs that come with a second filtering pass - at least on systems that employ a high performance interconnect like our compute cluster.



(a) Overall communication volume for an increasing number of nodes and a duplicate-free dataset.

(b) Overall communication volume for $p = 64$ PEs and an increasing duplication factor.

Figure 4.6: Analysis of the overall communication volume. The total communication volume of the dSBF-based duplicate removal algorithms is more than one order of magnitude smaller than that of `RePart` for a duplicate-free dataset. The benefits of two-pass filtering become negligible as the number of duplicate records approaches 10%.

[1] `http://www.boost.org/doc/libs/1_52_0/libs/dynamic_bitset/dynamic_bitset.html`

The overall communication volume of our dSBF-based algorithms and that of the `Repartitioning` algorithm is shown in Figure 4.6. For duplicate free datasets, the communication volume of `1dSBF` and `2dSBF` is up to two orders of magnitude smaller than that of `RePart`. As can be seen in Table 4.1, the dSBF-filtering phase is able to identify almost all elements as distinct and thus only few input elements (i.e., the false positives) have to be distributed in the post-processing phase.

With an increasing number of duplicates it is inevitable that the communication volume increases, because all duplicates have to be transferred at least once. With increasing α, the communication volume caused by the filtering phase therefore gets dominated by the communication volume caused by the distribution of the input tuples. Nevertheless, the filtering phase still reduces the overall communication volume significantly.

As can be seen in Table 4.2 the second filtering pass becomes more expensive, as the total number of duplicates increases. In case of 50% duplicates, the communication volume caused by the dSBF-preprocessing phase is even larger than that of `1dSBF`. The second filtering pass operates on all tuples that could not be identified as distinct in the first pass. Thus, with an increasing number of duplicates, more and more tuples pass the first filter and therefore increase the communication volume of the second filtering pass. In case the dataset is known to contain a significant number of duplicates, `1dSBF` therefore can be considered a better choice than `2dSBF`.

P	Algorithm	dSBF-filtering [GB]	Tuple distribution [GB]	Total [GB]
16	`RePart`	0	208	208
	`1dSBF`	3.68	0.34	4.02
	`2dSBF`	2.18 | 0.46	0.04	2.68
24	`RePart`	0	312	312
	`1dSBF`	5.85	0.52	6.37
	`2dSBF`	3.42 | 0.72	0.07	4.21
32	`RePart`	0	416	416
	`1dSBF`	8.08	0.7	8.78
	`2dSBF`	4.98 | 1.04	0.09	6.12
46	`RePart`	0	598	598
	`1dSBF`	12.08	1.02	13.1
	`2dSBF`	7.74 | 1.39	0.12	9.25
64	`RePart`	0	832	832
	`1dSBF`	17.38	1.42	18.8
	`2dSBF`	11.3 | 2.01	0.16	13.47

Table 4.1: Communication volume of our dSBF-based duplicate removal algorithms and the repartitioning algorithm. dSBF-filtering contains the data of the batched insertion messages and the corresponding collision indicator responses. For `1dSBF` and `2dSBF` tuple distribution also contains the data of the removal requests. For `2dSBF` the first number corresponds to the first filtering pass and the second number to the second filtering pass.

α	Algorithm	dSBF-filtering [GB]	Tuple distribution [GB]	Total [GB]
$*$	`RePart`	0	832	832
0.0	`1dSBF`	17.38	1.42	18.8
	`2dSBF`	11.3 | 2.01	0.16	13.47
0.001	`1dSBF`	17.38	2.25	19.64
	`2dSBF`	11.3 | 2.03	1	14.33
0.01	`1dSBF`	17.38	9.73	27.12
	`2dSBF`	11.3 | 2.16	8.49	21.95
0.1	`1dSBF`	17.38	84.52	101.9
	`2dSBF`	11.3 | 3.47	83.41	98.18
0.25	`1dSBF`	17.38	209.19	226.57
	`2dSBF`	11.3 | 5.68	208.32	225.29
0.5	`1dSBF`	17.38	417.04	434.42
	`2dSBF`	11.3 | 9.46	416.53	437.29

Table 4.2: Communication volume of `1dSBF`, `2dSBF`, and `RePart` for an increasing duplication factor α. dSBF-filtering consists of the batched insertion messages and the corresponding collision indicator responses. For `1dSBF` and `2dSBF` tuple distribution also contains the data of the removal requests. For `2dSBF` the first number corresponds to the first filtering pass and the second number to the second filtering pass. While the dSBF-filtering communication volume remains stable for `1dSBF`, it consistently increases for `2dSBF`, because more and more elements pass the first filter and therefore also have to be considered in the second filtering pass.

5. Conclusion

This thesis presented an implementation of the distributed duplicate removal algorithm outlined in the theoretical analysis of Sanders et al. [48] and complemented this theoretical analysis with an experimental evaluation. Our implementation is capable of both single-pass and multi-pass filtering using the distributed single shot Bloom filter.

We demonstrated the practical performance impact of the communication efficiency achieved by our algorithms by comparing both variants to a tuned implementation of the best-suited traditional algorithm [49] in terms of overall running time and communication volume. Our implementations outperform the traditional algorithm up to the point where the dataset consists of 50% duplicates. For duplicate-free datasets and those that contain less than 10% duplicates, the communication volume of our implementations is more than one order of magnitude smaller than that of the competitor, which always distributes the entire dataset. As the dSBF-based algorithms cannot avoid to transmit each duplicate record at least once, the communication volume increases with the total number of duplicates. Nevertheless, the distributed single shot Bloom filter still minimizes the amount of data that actually has to be transferred, because it is able to indentify almost all distinct elements.

Our experiments were performed on a distributed memory cluster system that employed a high-performance interconnect. As the experimental results clearly demonstrate the benefits of communication efficient duplicate removal on such a system, we expect the gains of our dSBF-based filtering approach to become even larger on systems with less sophisticated network infrastructures.

Two side results of our engineering efforts are of independent interest. Motivated by the analysis of the default all-to-all communication algorithm provided by the OpenMPI message-passing library, we presented an implementation of the 1-Factor algorithm [47] that can be used as a drop-in replacement. It outperforms its library counterpart especially for a large number of nodes and large message sizes. Furthermore, we implemented a parallel radix sort based on the concepts of Wassenberg and Sanders [64] to exploit the fact that we deal with (random) integer keys in the dSBF-preprocessing phase.

Our current implementation still offers potential for further optimizations. The distributed single shot Bloom filter uses bit-vectors as responses to batched insertion messages. Using `boost::dynamic_bitset` as implementation for these bit-vectors turned out to introduce a bottleneck when processing datasets with an increasing amount of duplicates and furthermore negatively affected the performance of the two-pass variant. By eliminating this

bottleneck, we expect to be able to change the tipping point, at which the dSBF-based duplicate removal algorithm is slower than the traditional algorithm, in favor of our implementation.

Finally, Sanders et al. [48] propose additional tuning measures that have not been considered in the course of this thesis. This includes the usage of a parallel bucket sort to sort the hash values in the preprocessing phase and the usage of a hash-based collision detection algorithm to avoid the factor $\log p$ work overhead that comes with our approaches based on multiway-merging.

Bibliography

[1] M. Abdelguerfi and A. Sood. Computational complexity of sorting and joining relations with duplicates. *IEEE Transactions on Knowledge and Data Engineering*, 3(4): 496 –503, December 1991.

[2] M. Abdelguerfi, K. Grant, E. Murphy, W. Patterson, and J. Stelly. Duplicate deletion in a ring connected, shared-nothing, parallel database system. In *Database and Expert Systems Applications*, volume 720 of *Lecture Notes in Computer Science*, pages 146–153. Springer, 1993.

[3] E. Babb. Implementing a relational database by means of specialzed hardware. *ACM Transactions on Database Systems*, 4(1):1–29, March 1979.

[4] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2):255–265, June 1983.

[5] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[7] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2002.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[9] H. Cai, P. Ge, and J. Wang. Applications of bloom filters in peer-to-peer systems: Issues and questions. In *International Conference on Networking, Architecture, and Storage*, pages 97–103, 2008.

[10] S. K. Cha and C. Song. P*TIME: highly scalable OLTP DBMS for managing update-intensive stream workload. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB '04, pages 1033–1044, 2004.

[11] A. Chatterjee and A. Segev. Data manipulation in heterogeneous databases. *SIGMOD Record*, 20(4):64–68, December 1991.

[12] I. Dar, T. Milo, and E. Verbin. Optimized union of non-disjoint distributed data sets. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 12–23, 2009.

[13] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36, 2006.

[14] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[15] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44 –62, March 1990.

[16] D. J. DeWitt and J. Gray. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, December 1990.

[17] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 1–8. ACM, 1984.

[18] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[19] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[20] A. Farzan, P. Ferragina, G. Franceschini, and J. Munro. Cache-oblivious comparison-based algorithms on multisets. In *Algorithms – ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 305–316. Springer, 2005.

[21] S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12 (3):399–401, 1966.

[22] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993.

[23] G. Graefe. New algorithms for join and grouping operations. *Computer Science - Research and Development*, 27(1):3–27, 2012.

[24] S. Helmer, T. Neumann, and G. Moerkotte. Early grouping gets the skew, 2002.

[25] T. Hoefler and A. Lumsdaine. Message progression in parallel computing - to thread or not to thread? In *IEEE International Conference on Cluster Computing*, pages 213–222, 2008.

[26] B. Huang and M. Langston. Stable duplicate-key extraction with optimal time and space bounds. *Acta Informatica*, 26:473–484, 1989.

[27] Intel Corporation. Intel ®Threading Building Blocks, 2011.

[28] C. Jamard, G. Gardarin, and L. Yeh. Indexing textual xml in p2p networks using distributed bloom filters. In *Advances in Databases: Concepts, Systems and Applications*, volume 4443 of *Lecture Notes in Computer Science*, pages 1007–1012. Springer, 2007.

[29] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[30] G. Koloniari, N. Ntarmos, E. Pitoura, and D. Souravlias. One is enough: distributed filtering for duplicate elimination. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 433–442, New York, NY, USA, 2011. ACM.

[31] P.-A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. *Manuscript submitted for publication*, 1997.

[32] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 239–250. ACM, 1986.

[33] B. Lint and T. Agerwala. Communication issues in the design and analysis of parallel algorithms. *IEEE Transactions on Software Engineering*, SE-7(2):174–188, 1981.

[34] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf` (Sep. 2009).

[35] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate detection in click streams. In *Proceedings of the 14th international conference on World Wide Web*, pages 12–21, 2005.

[36] L. Michael, W. Nejd, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. In *21st International Conference on Advanced Information Networking and Applications*, pages 187–194, 2007.

[37] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Springer, 3 2002.

[38] J. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558 –560, May 1990.

[39] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM Journal on Computing*, 5(1):1–8, 1976.

[40] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[41] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Proceedings of the 16th annual ACM-SIAM symposium on Discrete algorithms*, pages 823–829, 2005.

[42] S. K. Pal and P. Sardana. Bloom filters & their applications. *International Journal of Computer Applications Technology and Research*, 1(1):25 – 29, 2012.

[43] V. Pieterse, D. G. Kourie, L. Cleophas, and B. W. Watson. Performance of c++ bit-vector implementations. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 242–250. ACM, 2010.

[44] F. Putze, P. Sanders, and J. Singler. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms*, volume 4525 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 2007.

[45] G. Z. Qadah. Filter-based join algorithms on uniprocessor and distributed-memory multiprocessor database machines. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '88, pages 388–413, London, UK, UK, 1988. Springer.

[46] S. Ramesh, O. Papapetrou, and W. Siberski. Optimizing distributed joins with bloom filters. In *Distributed Computing and Internet Technology*, pages 145–156. Springer, 2009.

[47] P. Sanders and J. L. Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, pages 799–803. Springer, 2002.

[48] P. Sanders, S. Schlag, and I. Müller. Communication efficient algorithms for fundamental big data problems. Manuscript submitted for publication, 2013.

[49] A. Shatdal and J. F. Naughton. Processing aggregates in parallel database systems. 1994. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.9494`.

[50] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 104–114. ACM, 1995.

[51] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 international conference on Management of Data*, pages 731–742, 2012.

[52] J. Singler and B. Konsik. The GNU libstdc++ parallel mode: software engineering considerations. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 15–22. ACM, 2008.

[53] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2007.

[54] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, pages 4–9, 1986.

[55] S. Y. W. Su and K. P. Mikkilineni. Parallel algorithms and their implementation in MICRONET. In *Proceedings of the 8th International Conference on Very Large Data Bases*, VLDB '82, pages 310–324, San Francisco, CA, USA, 1982.

[56] D. Taniar, C. H. C. Leung, W. Rahayu, and S. Goel. *High Performance Parallel Database Processing and Grid Databases*. Wiley, 1 edition, 10 2008.

[57] S. Tarkoma, C. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131 –155, 2012.

[58] J. Teuhola. External duplicate deletion with large main memories, 1993.

[59] J. Teuhola and L. Wegner. Minimal space, average linear time duplicate deletion. *Communications of the ACM*, 34(3):62–73, March 1991.

[60] V. Topkar, O. Frieder, and A. K. Sood. Duplicate removal on hypercube engines: An experimental analysis. *Parallel Computing*, 17(8):845 – 871, 1991.

[61] P. Valduriez. Parallel database systems: open problems and new issues. *Distributed and parallel Databases*, 1(2):137–165, 1993.

[62] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9(1):133–161, 1984.

[63] X. Wang, Q. Zhang, and Y. Jia. Efficiently filtering duplicates over distributed data streams. In *International Conference on Computer Science and Software Engineering*, volume 4, pages 631–634, 2008.

[64] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2011.

[65] L. M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, C-34(4): 362 –367, April 1985.

[66] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proceedings of the VLDB Endowment*, 1(2):1253–1264, 2008.

[67] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous mpi for the masses. *CoRR*, abs/1302.4280, 2013.

[68] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 3rd edition, 2011.