Master thesis

Bloom Filters for ReduceBy, GroupBy and Join in Thrill

Alexander Noe

Date: 12. January 2017

Supervisors: Prof. Dr. Peter Sanders
Dipl. Inform. Dipl. Math. Timo Bingmann

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

Abstract

Thrill is a prototype of a high-performance general purpose big data processing framework. In the *Reduce* operation, which is similar to *Reduce* in MapReduce, Thrill performs an all-to-all hash based data shuffle of large amounts of data. Elements only occuring on a single worker could however be reduced locally without shuffling them. To find these elements, we propose a detection algorithm based on distributed single-shot Bloom filters to efficiently detect a large portion of these elements.
Additionally, we implemented an SQL-style *InnerJoin* operator for Thrill. For the *InnerJoin* operator it is not possible to perform local reduction before the hash based data shuffle. Therefore we implemented an augmented version of our detection algorithm, which detects the worker with the highest number of total occurences for each key. In order to reduce the amount of total network traffic, this worker is determined as the shuffle target for that key. We use multiple algorithmic micro-benchmarks on the AWS EC2 computing cluster to benchmark the performance of our implementations in the Thrill framework.
In communication-heavy benchmarks, such as median computation and the *TPC-H4* database join query, we can improve the running time by a factor of 1.5 up to 10 by using duplicate detection. In the *WordCount* and *PageRank* algorithms performed on real world data we can lower the amount of network traffic while keeping a comparable running time.

Acknowledgments

We would like thank the AWS Cloud Credits for Research program for making the experiments in Chapter 7 possible.

Contents

1 Introduction

1.1 Motivation

In the last decades, the amount of data generated in many applications increased exponentially. In order to process this ever-increasing amount of data, parallelized computation quickly became essential. One of the programming paradigms to handle parallel computing on commodity clusters is *MapReduce* [14], popularized by Google in 2004. The *MapReduce* paradigm offers a limited user interface with only two user-defined operations: *Map* and *Reduce*. Frameworks implementing *MapReduce* aim to provide scalability and fault tolerance while being easy-to-use. The first step of *MapReduce*, the *Map* operation, maps each input element to a set of key-value pairs. The second step, the *Reduce* operation, then groups all elements with equal key and reduces them to a single value.
The widely-used open source framework *Apache Hadoop* [47] implements the *MapReduce* paradigm. While the framework scales very well, Hadoop has low performance especially in iterative algorithms [9]. Many frameworks, such as *Apache Spark* [49], *Apache Flink / Stratosphere* [3] and *Thrill* [6] try to enhance the performance of Hadoop while offering a superset of operations to the user. In sufficiently large clusters, data communication between machines can become the main performance bottleneck for many applications. Examples for such communication-bound applications are sorting [36] and *PageRank* [37] [6].
In applications, where the bottleneck is the communication volume, performance enhancement is mainly possible by lowering this amount of communication. In *Reduce* this can be done by trying to detect keys that only appear on a single worker. These keys can be reduced locally and do not have to be sent to another worker in the network.
A possible approach to detect non-duplicate keys to reduce locally is *Bloom filtering* [7]. A Bloom filter is a space efficient data structure to detect duplicated elements in a data set. Bloom filters have a false positive rate but do not contain false negatives. In parallel computing, *distributed single-shot Bloom filters* (dSBF) [40] implement a single Bloom filter spread over multiple computation nodes. This dSBF data structure can be used to detect duplicate elements in a parallel environment.

1.2 Contribution

In this work, we use a dSBF in the *Reduce* operation of Thrill to probabilistically detect keys which only occur on a single worker with a small communication overhead. When the

amount of non-duplicate keys is large, the total amount of communication for a program can be reduced.
As a part of this work, we also wrote a *InnerJoin* operation in Thrill, which performs an SQL inner join to join two datasets as defined by the user.
In the operations GroupBy and InnerJoinWith, multiple elements with equal keys all have to be sent without being able to pre-reduce them locally. Therefore it is beneficial to find the worker with the highest amount of elements with a certain key. We use an extended dSBF to find this worker for each key in the datasets and additionally filter out keys which can't generate any join results.

1.3 Structure of Thesis

Chapter 2 of this thesis lists related work for big data processing frameworks and Bloom filters. Chapter 3 gives an overview of the current version of the Thrill framework, which is basis for this work. Chapter 4 describes the original Bloom filter data structure and multiple Bloom filter variants which are useful for our purpose. Chapter 5 describes the *InnerJoin* operation of Thrill in detail. Chapter 6 describes the application of dSBF to Thrill to detect duplicate elements. Chapter 7 shows experimental evaluation for *InnerJoin* and the duplicate detection with multiple algorithms. Chapter 8 discusses the experimental results and gives an outlook on potential future work.

2 Related Work

In recent decades, many different frameworks were written to handle very large amounts of data in a highly distributed environment [10]. One of the most heavily used frameworks is *Apache Hadoop* [47], which provides an open-source implementation in Java for the MapReduce [14] programming paradigm. The intended usage of Hadoop is scalable computation on clusters with a large amount of commodity machines. The Hadoop framework also incldues the *Hadoop Distributed File System* (HDFS) [42], which is a distributed file system storing files redundantly on multiple machines of the commodity cluster. HDFS is used in many big data frameworks.
As Hadoop has a very restricted interface and lackluster performance, many different frameworks tried to advance the idea. One of these big data frameworks is *Apache Spark* [49]. The central element of Spark's interface is a data structure called *Resilient Distributed Dataset* (RDD). An RDD is a distributed read-only multiset of elements. It can be generated from input or other RDDs using the transformations offered by Spark, which are a superset of MapReduce. Additionally, Spark offers libraries for machine learning (MLlib) [32], graph processing (GraphX) [48] and handling of data streams (Spark Streaming) [50].
Another notable big data processing framework is *Apache Flink/Stratosphere* [3]. Both Spark and Flink are implemented in Scala.
Pregel [28] offers a system for distributed graph processing. Multiple frameworks such as *GraphLab* [27] and *Google TensorFlow* [2] offer distributed machine learning.
A benchmark comparing Spark and Flink [29] shows that both frameworks have clear strengths and weaknesses. The benchmark shows that Spark is faster for large problems, but Flink is faster for batch and small graph workloads. The authors note that both frameworks have performance problems due to the limitations of the JVM, especially with garbage collection. Additionally, they observe a very complex optimization process for Spark.
HiBench [23] is a benchmarking suite that was originally written to benchmark the performance of Hadoop in a set of real-world algorithms. Recent versions of HiBench also benchmark Spark.

A Bloom filter [7] is a space-efficient data structure for probabilistic membership queries. A Bloom filter has false positives but no false negatives. There are many different variants of Bloom filters [8] [46], such as counting Bloom filters [17] and compressed Bloom filters [33]. Bloom filters can be compressed using Golomb encoding [38] [18]. It is possible to detect duplicate elements in a distributed dataset with little communication using a

Golomb encoded distributed Bloom filter [40].
Bloom filters are used in databases to improve performance by “preventing unnecessary searches” [21]. They can also be used to find duplicate entries in databases [30]. Bloom filters are also used to perform deep packet inspection [15]. Their usage can reduce communication volume in SQL joins in databases [34] [19]. This is especially true when the amount of elements not participating in the join is large.
Another usage for Bloom filters is the reduction of communication volume when performing a join in Hadoop [26]. In the TPC-H4 [12] benchmark on Hadoop, the running time is substantially lowered with usage of replicated Bloom filters.

Thrill [6] is a big data processing framework written in C++ -14. The central data structure of Thrill is the *Distributed Immutable Array* (DIA). In contrast to Spark and Flink, DIAs in Thrill are inherently ordered. In the Thrill introduction paper [6], the performance of the framework is compared to Spark and Flink on AWS EC2 machines. These machines are comparable to the machines used in this work. In most algorithms, Thrill can outperform both Spark and Flink.

3 An Introduction to Thrill

3.1 Overview

The amount of raw data generated in many scientific and commercial applications is increasing steadily. Large instances of data analysis problems such as machine learning, weather forecasting or webpage ranking require many processors or even a whole computing cluster to solve. Using a computing cluster introduces new problems, as programming on distributed systems is fundamentally different to single-core programming. Multiple threads need to be coordinated to ensure a correct result. On larger systems, the memory is distributed and the processors need to send messages to other processors. To solve these challenges of distributed programming, multiple different paradigms were introduced.
On the one end of the complexity spectrum, MPI [22] provides a complex and low-level message passing interface to allow communication between individual processing units. The user can send messages from one processor to another or perform collective functions. On the other end, MapReduce [14] provides an easy-to-use high level interface with only two user-defined procedures. The Map() function maps an input element to zero or more key-value pairs. This Map() function is applied to each element of the input data set. The resulting key-value pairs are then shuffled over the cluster and grouped by their respective key. The user-defined Reduce() function can then iterate through all values associated with a certain key and generate zero or more output values. MapReduce was popularized by Google in 2004 [14]. *Google Mapreduce* is their implementation of the MapReduce paradigm. It is a proprietary framework written in C++ . A widely used open source alternative is *Apache Hadoop* [47] written in Java.
While this MapReduce interface is useful for many applications, it has its limitations, especially in iterative algorithms [9]. *Apache Spark* [49] and *Apache Flink / Stratosphere* [3] are frameworks written in Scala which try to alleviate the limitations of MapReduce. They both have more generalized dataset transformation interfaces, which are supersets of the MapReduce interface. Both Spark and Flink generally outperform Hadoop on most use cases [49]. Their performance is however often still lacking [31], mostly due to limitations of the JVM. In many cases, the CPU performance is also the bottleneck for these frameworks [35].
Thrill is a framework, which provides an interface similar to these frameworks. Thrill tries to outperform Flink and Spark by being written in modern C++ 14 and using efficient low-level memory and data management. This chapter gives an introduction to the interface and the internals of the Thrill framework to give a basis for the chapters describing InnerJoin

and the duplicate detection.
There are many applications for big data frameworks such as Thrill, Flink, Hadoop or Spark. The paper introducing the MapReduce paradigm [14] contains two usage examples: a *grep* scan of records for a certain string pattern and a sorting example modeled after the *TeraSort* [36] benchmark. The *WordCount* algorithm, which counts the occurences of each word in a text, is often seen as the 'Hello World' example for such frameworks. Section 3.4.1 shows an implementation of *WordCount* in Thrill. Another popular algorithmic use is the website ranking algorithm *PageRank* [37]. The Thrill implementation of *PageRank* is detailed in section 3.4.2. Big data frameworks are also often used for machine learning [32] and for making database queries [4].

Thrill

Thrill is an open-source framework for distributed batch processing of large amounts of data. The Thrill core is written completely in C++ 14. Thrill user programs can be written in C++ 14 and compiled to a binary exectuable. The intended place of use for Thrill are commodity clusters with multiple commodity-grade hosts connected by a local area network. It is however also possible to use Thrill on high-performance clusters or single machines without needing additional configuration.
When a Thrill executable is executed, an equal program starts on each of the h machines allocated to this Thrill program. This program launches several worker threads, by default identical to the hardware concurrency c of the machine. Thrill currently only supports a uniform amount of workers per host.
The data from input files, by default assumed to be on a network file system, will be partitioned equally to all participating worker threads. To ensure good load balancing, the workers should be nearly identical in performance. The machines are interconnected by a reliable network backend. Currently supported backends in Thrill are fully-meshed TCP connections, a MPI backend and TCP/IP over Infiniband.
In the context of Thrill, each machine used is called a *host*. Each of the threads performing work is called a *worker*. The Thrill program has a total of $p = h \cdot c$ workers. Each worker has a unique identification handle enumerated from 0 to $p - 1$. On a unique host, the local workers are enumerated from 0 to $c - 1$. The hosts are enumerated from 0 to $h - 1$. In contrast to Spark and Hadoop, Thrill has no separate master node or driver.
The network subsystem of Thrill additionally provides a set of MPI-style network collectives like *reduce* and *broadcast*. These network collectives by default use worker 0 as the broadcasting source or reduce target. Internally, these collectives use efficient algorithmic solutions.
Thrill does not yet provide tolerance to hardware failures, however the design of the framework allows addition of snapshots for rollback.

3.2 Distributed Immutable Array

The central element of the API of Thrill is the *Distributed Immutable Array* (DIA). From the user's perspective, a DIA can be seen as an immutable array, which is distributed over all workers in the cluster. As indicated by the name, a DIA has an inherent global order of elements and each element in a DIA has equal type. A DIA can have an arbitrary type T and it is denoted as $DIA\langle T\rangle$. The element type is a template parameter of the DIA.
As a DIA is immutable, it is not possible to insert, delete or change elements in a DIA directly. Instead, a new DIA can be created by performing transformation operations to an existing DIA or input files. If we e.g. have a $DIA\langle Integer\rangle$, we can create a new $DIA\langle Integer\rangle$ by mapping each integer to its square. Most operations in Thrill keep the order of elements in a DIA intact. In most cases, the contents of a DIA are not actually materialized, they exist only conceptually. This allows multiple optimizations detailed in Section 3.3.5.
Section 3.3 gives an overview over all operations currently present in Thrill. Section 3.4 shows detailed implementations of different algorithms in Thrill. Section 3.3.6 describes the data, net and I/O layers of Thrill.

3.3 Operations in Thrill

An operation in Thrill creates a new DIA from either another DIA or an input file. In Thrill, most operations can be specified by one or multiple *User Defined* C++ Lambda *Functions* (UDFs). The operations in Thrill can be classified in four groups.
Source operations generate a new DIA either from input files or from a generator function. All other operations have at least one DIA as their input.
Local operations (LOps) are operations, which can be performed without communication between workers. The UDF further specifying the LOp is applied to every element of the input DIA. Multiple consecutive LOps are chained into a single function, further denoted as the *LOp Chain*.
Distributed operations (DOps) have at least one global communication step and therefore comprise a global barrier. A DOp is divided into three parts: the *Link* phase handles the finalizing local work by applying a function to every element. Usually, the *Link* phase stores elements locally or inserts them into a asynchronous network stream. The *Main* phase can perform global communication and computes the result DIA of the operation. The *Push* phase emits output elements into the following LOp Chain. When the result of a DIA is required more then once, only the *Push* phase is called in the subsequent calls.
Actions do not generate a new DIA but return a computed result, for example the sum of all elements in a DIA, to each worker. Actions consist of a *Link* phase and a *Main* phase. The roles of the phases are similar to the phases in a DOp. The *Push* phase is not needed as there are no resulting DIA and therefore no elements to push to the subsequent chain.

Table 3.1: Source Operations of Thrill, from [6]

Operation	User Defined Functions
Generate(n) : $[0, \ldots, n-1]$	n : DIA size
Generate(n, g) : $[A]$	g : **unsigned** $\to A$
ReadLines$()$: files $\to$ [**string**]	
ReadBinary$\langle A \rangle()$: files $\to [A]$	A : data type

3.3.1 Source Operations

Table 3.1 shows a set of source operations implemented in Thrill. These operations generate DIAs from an input. Available inputs are files from disk and generator functions.

Generate generates a DIA of unsigned integers in the range $[0, n)$. It is also possible to add a lambda function g, which maps the indices $[0, n)$ to the elements of a generated DIA.
ReadLines takes a file path and creates a *DIA*$\langle$`std::string`$\rangle$. In this DIA, each line from the input is one element. The operation internally uses the Linux tool glob to find all files for the path. The elements in the resulting DIA are ordered by the lexicographical order of the file names and the order of lines in a file. Thrill generally assumes a distributed file system, in which every worker can access all files and distributes the data equally on a global level. If there is no distributed file system, each host reads all of its data locally. Equal distribution is realized on a byte level, which means that every worker gets roughly equal amount of text bytes, even if the files or lines have unequal sizes. For an input size n, worker i reads all lines which start in $[\frac{n \cdot i}{p}, \frac{n \cdot (i+1)}{p})$.
It is also possible to access compressed files with *ReadLines*. As compressed files generally do not allow seeking, worker i completely reads all files, where the byte exactly in the middle of the file is in $[\frac{n \cdot i}{p}, \frac{n \cdot (i+1)}{p})$. *ReadLines* can also read files from the AWS S3 storage cloud. These files are read using the libS3 library.
ReadBinary reads binary files from disk. ReadBinary takes the desired element type as a template parameter.

3.3.2 Local Operations

Table 3.2 shows a set of LOps implemented in Thrill. These operations transform a DIA into a new DIA without communication.

Map applies a transformation UDF to each element of the input DIA. The type of the output DIA is hereby - like in most other operations - inferred from the return type of the mapping UDF.
FlatMap is a generalization of Map, in which the transformation UDF takes an input element and maps it to zero or more output elements. For this purpose, the UDF has an emitter function as a second parameter.
Filter filters a DIA according to a filtering UDF. This filter function maps each input ele-

Table 3.2: Local Operations of Thrill, from [6]

Operation	User Defined Functions
$\textbf{Map}(f) : [A] \to [B]$	$f : A \to B$
$\textbf{FlatMap}(f) : [A] \to [B]$	$f : A \to \textbf{list}(B)$
$\textbf{Filter}(f) : [A] \to [A]$	$f : A \to \textbf{bool}$
$\textbf{BernoulliSample}(p) : [A] \to [A]$	p : success probability
$\textbf{Union}() : [A] \times [A] \cdots \to [A]$	
$\textbf{Collapse}() : [A] \to [A]$	
$\textbf{Cache}() : [A] \to [A]$	

ment to a `bool`. The output DIA contains all elements, which were mapped to `true`.
BernoulliSample performs an independent Bernoulli Sample with success probability p on each element of the input DIA. The output DIA contains all sampled elements, the other elements are discarded.
Union unites multiple DIAs of equal type. In this process the order of elements is disregarded.
Cache saves a DIA to avoid recalculation when the DIA is used multiple times.

3.3.3 Distributed Operations

Table 3.3 shows a set of DOps implemented in Thrill. These operations create a new DIA and have at least one global communication step.

ReduceByKey groups all DIA elements by their key and reduces all elements with equal key to a single output element. The key extractor function k maps an element to its key. The associative reduce function r merges two elements to a single element. This function r is applied to elements with equal key until only a single element with this key remains.
ReducePair is a variant of ReduceByKey, which assumes that the input type is a `std::pair` of the key and value. Therefore, *ReducePair* does not need a key extractor function. The reduce function in *ReducePair* reduces two values to a single one. This reduction is performed on elements with equal key until only a single result is left for this key.
GroupByKey performs an equivalent reduction. In contrast to *ReduceByKey*, the group by function in *GroupByKey* maps an iterable of elements to a single element. This lowers performance and increases network load, but it allows changing element types and computations, which need all elements at once, for example median. *ReduceByKey* and *GroupByKey* disregard the order of the DIA and do not give any guarantees about the ordering of the output DIA.
Both *ReduceBy* and *GroupBy* also have a *ToIndex* variant, in which the key extractor maps elements to indices in $[0, n)$, where n is an integer defined by the user in the operation. The output DIA has exactly n elements and is ordered by those indices.
InnerJoinWith performs an inner join of two DIAs. The operation extracts the key of each

Table 3.3: Distributed Operations of Thrill, from [6]

Operation	User Defined Functions
ReduceByKey(k, r) : **ReducePair**(r) : $[(K, A)] \to [(K, A)]$ **ReduceToIndex**(i, r, n) : $[A] \to [A]$ **GroupByKey**(k, g) : **GroupToIndex**(i, g, n) : $[A] \to [B]$	$k : A \to K$ $r : A \times A \to A$ g : **iterable**$(A) \to B$ $i : A \to$ **unsigned**
InnerJoinWith(k_1, k_2, j) : $[A] \times [B] \to [C]$	$k_1 : A \to K$ $k_2 : B \to K$ $j : A \times B \to C$
Sort(c) : $[A] \to [A]$	$c : A \times A \to$ **bool**
Merge(c) : $[A] \times [A] \cdots \to [A]$	$c : A \times A \to$ **bool**
Concat$()$: $[A] \times [A] \cdots \to [A]$	
PrefixSum(s, i) : $[A] \to [A]$	$s : A \times A \to A$ i : initial value
Zip(z) : $[A] \times [B] \cdots \to [C]$	$z : A \times B \cdots \to C$
ZipWithIndex(z) : $[A] \to [B]$	z : **unsigned** $\times A \cdots \to B$
Window(k, w) : $[A] \to [B]$ **FlatWindow**(k, f) : $[A] \to [B]$	k : window size $w : A^k \to B$ $f : A^k \to$ **list**(B)

element and joins all pairs of elements with equal key. As *InnerJoinWith* was part of this thesis, the internals of the operation are described in Chapter 5 in detail.
Sort sorts a DIA according to a given comparison function. The operation uses *Super Scalar Sample Sort* as the distributed sorting algorithm [41]. To enable external memory sorting, the local sorting uses external merge sort.
Merge performs a merge step to merge multiple ordered input DIAs while keeping the element order intact.
Concat concatenates two DIAs while keeping the order. In contrast to the LOp *Union*, *Concat* keeps the order of elements. Therefore the data needs to be shuffled between workers.
Zip zips two DIAs similar to the zip operation in functional programming languages. *ZipWithIndex* zips each element of a DIA with its global index.
Window and *FlatWindow* are variants of *Map* and *FlatMap*. In contrast to *Map*, the user-defined function in *Window* has a sliding window of k elements as it's input.

3.3.4 Actions

Table 3.4 shows a set of actions implemented in Thrill. Actions do not result in a DIA but in an equal result value on each worker. All previously described operations are exe-

Table 3.4: Actions of Thrill, from [6]

Execute$()$	
Size$()$: $[A] \to$ **unsigned**	
AllGather$()$: $[A] \to$ **list**(A)	
Sum(s, i) : $[A] \to A$ **Min**(s) : $[A] \to A$ **Max**(s) : $[A] \to A$	$s : A \times A \to A$ i : initial value
WriteLines$()$: [**string**] $\to$ files **WriteBinary**$()$: $[A] \to$ files	

cuted lazily, only actions actually start computation. This process is further explained in Subsection 3.3.5, which describes the data flow graph of Thrill.

Execute only executes the previous operations and does not return anything.
Size returns the total number of elements in a DIA.
AllGather returns a DIA as an `std::vector` on each worker. This is useful in tests but generally not advisable for large datasets.
Sum returns the sum of all element values. The Actions *Min* and *Max* return the smallest or the largest element in the DIA.
WriteLines writes a DIA to files on disk. In default configuration WriteLines writes multiple files, it is also possible to write the whole output in a single file. *WriteBinary* writes a DIA to binary files. The DIA can then be recreated identically by the Source Operation *ReadBinary*. By default, WriteLines and WriteBinary write individual files with a size limit of 64MiB. The files are in global lexicographical order.

3.3.5 Data Flow Graph

The data flow of the operations in a Thrill program form a directed acyclic graph. Figure 3.1 shows an example data flow graph for the *WordCount* algorithm. This algorithm counts the number of occurences for each word in a text and is explained in detail in Section 3.4.1.
In the Thrill data flow graph, source operations are root nodes, e.g. nodes with an indegree of 0. An edge in this graph denotes that data flows directly from parent to child and thus the parent node needs to be computed before the child can be.
All non-source nodes have an indegree of at least 1, as all operations need at least one input DIA. Figure 3.1 was created from the Thrill JSON output using a Python script included in Thrill and the GraphViz [16] *dot* tool.

The Thrill data flow graph is executed lazily. In Thrill, only actions actually trigger computation. When an action is found in the program, the framework performs a reverse breadth-first search to find all operations which need to executed. These operations are then executed in topological order.
The result of an action usually is identical on each worker. This is important to ensure

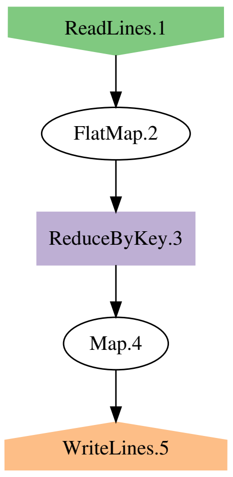

Figure 3.1: Data Flow Graph for WordCount Algorithm

program integrity, as action results are often break conditions for loops. Otherwise it could be possible to leave a loop on some workers while other workers reenter the loop resulting in undefined program flow.
In the example in Figure 3.1, computation is triggered as soon as the *WriteLines* action is encountered. The reverse breadth-first search finds all upstream operations, which are then executed from top to bottom.

Function Chaining

For the end user, a DIA can be seen as an actual array. In reality, the DIA exists only conceptually and the data is usually not materialized. This data flow concept makes some run time optimizations possible. In some LOps (*Map*, *FlatMap*, *Filter*, *BernoulliSample*), a lambda function is applied to each element of the DIA. Multiple consecutive LOps therefore form a chain of functions where the output of one function is the input of the next function.
Thrill combines this *LOp chain* to a single function which is applied to each element. The *Link* phase of the subsequent DOp or action is also inserted to the LOp chain. This chaining reduces the total number of operations and reduces memory overhead, as each element is only accessed once. Additionally this also renders storage for the intermediate results unnecessary.
In the example in Figure 3.1, the *FlatMap* node is chained with the *Link* phase of the *ReduceByKey* node. Each input element of the *FlatMap* node, which is a line of text, is splitted into words and each of these words is directly inserted into the hash table used for *ReduceByKey*. Therefore the words do not need to be stored intermediately. If we were to filter the words, e.g. only count proper names, the *Filter* node between *FlatMap* and *ReduceByKey* would be also a part of the LOp chain.

This chaining is performed by the C++ compiler on an assembly level to reduce the number of indirections to a minimum. In the resulting binary, a LOp chain is only a single function call, regardless of the number of chained LOps.

Template Types

The function type of a DIA consists of the element type and the *function stack*. This function stack contains all functions in the LOp chain up to this DIA and their types. As this quickly becomes a very complex type, DIAs should therefore be instantiated with C++ - 11 *auto* function type instead of using $DIA\langle T, f1, f2\rangle$. Additionally, if the type is added manually, the lambda functions have to be wrapped in `std::function`, which is an unnecessary indirection. The *auto* type of the DIA can be inferred by the compiler.
The function chain being in the DIA type is a problem in iterative or recursive algorithms, as DIAs need to updated, e.g. link to new data after an iteration.
As the function chain of the updated DIA might be different, Thrill has the LOp *Collapse*, which flushes the LOp chain and creates a new DIA node with an empty function stack. As this interrupts assembler-level optimization of the LOp chain, it induces additional overhead and should therefore only be used when necessary. A missing *Collapse* will result in compilation errors due to type mismatch.

3.3.6 Data, Network, and I/O Layer

Under the hood of the Thrill framework, the data is not actually handled in form of a materialized distributed array for each DIA. Multiple layers handle data storage, communication and operation execution. The lower layers of Thrill consist of the *data*, *net* and *io* layers. Additionally Thrill also has *common* and *core* layers which offer useful program utilities.

Data Storage

Internally, items in Thrill are stored as a serialized byte-sequence. In the Thrill serializer, fixed-length elements such as basic types or fixed-length structs of basic types are stored with no overhead as the sequence of their data bytes. Variable length elements such as strings or vectors are prepended with their size. Thrill serialization has an API which allows users to define the serialization for their own types when there is no way to automatically serialize them. Elements can also be serialized using the cereal serialization framework [20], which is an open-source serialization framework written in C++ .
In the lower levels of Thrill, data is always handled as serialized byte sequences. In order to be handled by the user-defined functions from the API, the stream of data is deserialized into actual elements. Directly after leaving the user-defined function, the data is serialized again.

The stream of items in a DIA is stored without separators or other overhead in *Blocks*. These Blocks have a default size of 2MiB. Apart from the raw data a Block contains only four integers of overhead. These integers are a pointer to the begin, a pointer to the end, a pointer to the first element in the Block and the number of total elements.
The pointer to the first element is necessary, as single elements can span over multiple Blocks. The start of the Block therefore does not need to be equal to the first element actually starting in the Block. In debug mode, every item is preceded with it's typecode. This typecode is verified on deserialization.
Blocks are organized in a *File*. A File is a sequence of Blocks and can be either stored in memory or disk. API operations can open as many Files as they deem necessary. Operations can also request data readers from a File. Offered reader types are *ConsumeReader* and *KeepReader*. The ConsumeReader destroys all elements in the File and frees the memory. KeepReader keeps the File intact and thus does not free the memory. A usage example for both reader types can be seen in the *PageRank* algorithm in Section 3.4.2.
Blocks in Thrill are handled by the *Block Pool*. The *Block Pool* manages *Block* reference counting and automatically deletes *Blocks* which are not referenced by any *File*. The memory consumption for data is also handled by the *Block Pool*. When a user-defined memory soft limit is exceeded, the least recently used *Blocks* are asynchronously written to disk. It is possible for the program to *pin Blocks* in order to keep them in memory or prefetch them from disk. When a hard limit is exceeded, the *Block Pool* does not allow new *Block* allocation.

Network

Elements can be transmitted between workers asynchronously using *Streams*. A *Stream* is a fully meshed net of communication streams between all workers. It allows a communication step between all workers, in which every worker can asynchronously send data to all other workers.
A *Stream* offers a vector of *Writers*, one *Writer* per worker. The *Writers* for workers on other hosts send elements in bulk through the network backend. The local *Writers* have the same API, but they do not send data through the network. The intra-host communication is instead executed by using shared memory.
Thrill has two types of *Streams*. In a *CatStream*, the items are delivered in worker rank order. In a *MixStream*, the order of recieved *Blocks* is arbitrary. A *MixStream* has potentially better performance but a *CatStream* guarantees element order and allows access a vector of individual *Readers* for each worker.
By default, the available memory is splitted in three equally large parts: *BlockPool* memory, DIA operation memory and heap memory for user objects. A DIA operation can request memory space by overloading *MemoryUse* functions. Thrill allocates space fairly between all operations which request them. If an operation exceeds the amount of memory granted, it can transfer data to external memory.
Thrill offers a multitude of stats. When logging is enabled, each *File* and *Stream* prints

performance metrics such as running time and size in JSON format on destruction. The JSON log files can be used to create profiles.

File I/O

In read and write Operations, Thrill opens a virtual file, which offers a basic file interface with reading, writing and seeking. The data read and written is a Block of uncompressed data with a defined size. Internally, the virtual file actually reads and writes the data and potentially calls a compression algorithm to compress or uncompress the data.
When the file is stored in Amazon S3 Cloud Storage, Thrill uses the libS3 [1] C library to read and write data from S3. Thrill reads data from S3, when the file name starts with the pattern “s3://”.

3.4 Algorithms

The following section describes algorithms and applications for Thrill. The main focus point are the algorithms used for experiments and benchmarks in Chapter 7. This section shows Thrill API code and shows some implementation details.

3.4.1 WordCount

WordCount is an algorithm which counts the number of occurences for each word in a text. This algorithm often serves as a “Hello World!” example for big data frameworks [25]. The complete Thrill code for *WordCount* is shown in Figure 3.2.

In the *WordCount* algorithm, we use the input and output file path as string parameters. In line 3 of the code example in Figure 3.2, we read the input from disk into a DIA of `std::string` elements.
We map these lines to single words using the *FlatMap* operation in lines 3-9. As *FlatMap* can’t infer the output type from the UDF, we need to define the output type as seen by the template parameter in line 3. The output type of this *FlatMap* is a `std::pair<std::string, size_t>`. In the *FlatMap* function in lines 5-8, we use `common::Split` provided by the common layer in Thrill. `common::Split` splits a line into single words. For each of these words, the lambda function in line 7 is called. For each word, we emit a `std::pair` of the word and a 1. The DIA `word_pairs` consists of pairs of the form `(word, 1)`.
We reduce the pairs with `ReduceByKey` in lines 10-16. In the *ReduceByKey* operation, we define two lambda functions: the key-extractor function is shown in line 12. This function defines that the key of each element is the word, which is the first element of the pair. The reduce function in line 15 defines how we reduce two pairs with equal key (same word). To perform the reduction, we add the counters of the words. In *ReduceByKey*, we first perform a local reduction of all pairs with equal words. Afterwards, we shuffle all pairs

```
 1 void WordCount(thrill::Context& ctx, std::string input, std::string output) {
 2   using Pair = std::pair<std::string, size_t>;
 3   auto word_pairs = ReadLines(ctx, input).template FlatMap<Pair>(
 4     // flatmap lambda: split and emit each word
 5     [](const std::string& line, auto emit) {
 6       common::Split(line, ' ', [&](std::string_view sv) {
 7         emit(Pair(sv.to_string(), 1));
 8       });
 9     });
10   word_pairs.ReduceByKey(
11     // key extractor: the word string
12     [](const Pair& p) { return p.first; },
13     // commutative reduction: add counters
14     [](const Pair& a, const Pair& b) {
15       return Pair(a.first, a.second + b.second);
16     })
17   .Map([](const Pair& p) {
18     return p.first + ": " + std::to_string(p.second); })
19   .WriteLines(output);
20 }
```

Figure 3.2: Complete WordCount Example in Thrill, from [6]

by a hash of their key and reduce the local counters. In the *Push* phase of *ReduceByKey*, we emit pairs of words and their global counter.
In order to store the result on disk, we map each of these pairs to a `std::string` in lines 17-18. In line 19, we write the result strings to disk.
The data flow graph for *WordCount* can be seen in Figure 3.1.

3.4.2 Page Rank

Google PageRank [37] is an algorithm famously used by Google to rank the importance of websites in the internet. The algorithm is based on the model of the random web surfer who starts surfing on a random site and then switches sites by following a random link on this site.
PageRank is an iterative algorithm. In each iteration of the algorithm, the rank of a page is evenly "pushed" through all outgoing edges. The rank of a page after the step is the sum of ranks from all ingoing edges. After each step, a dampening factor is used to represent new users surfing to a random page.

Implementation in Thrill

Figure 3.3 shows the complete implementation in C++ for the PageRank algorithm in Thrill. Algorithm 1 depicts PageRank in pseudo code. Each line of Algorithm 1 is annotated with corresponding code lines in Figure 3.3.

In lines 5-10 of the algorithm in Figure 3.3, we read the input from disk. We map each line of the input to a link, which is equal to a directed edge. A link consists of the source and the target of the edge. We assume that the input file has one line per link, which are in the format `"SourceID TargetID"`. We could also use different graph storage formats by replacing the *Map* function in lines 6-10, which creates the links.
In lines 11-13 of the algorithm we compute the total number of pages. By default, a Thrill operation consumes the data of it's parent DIA. As we still need the input, we use *Keep* in line 11 to increment the consume counter from 1 to 2. This counter disables consuming if it is larger than 1 at the begin of the operation. In the Map function, we map each link to the maximum of it's source and target. Due to this function, the consume counter is decremented to 1.
With the *Max* action we compute the maximum of all page IDs. As the pages start at 0, the number of total pages is higher than the maximum index by 1.
With the *GroupByKey* operation in lines 14-20, we create pairs of a page and a `std::vector` of all outgoing links from this page. In contrast to the previous *Map*, this operation consumes the DIA input, as the consume counter is now 1. The key extractor groups all links by their source. The *GroupBy* function has an iterator for all elements with equal key. This iterator implements the functions `Next()` and `HasNext()`. In the GroupBy function, we form a `std::vector` of all outgoing links and return a pair of the source and this `std::vector`. We cache this DIA and set its consume counter to infinite with the Thrill operation *KeepForever*.
We then generate the initial ranks for all pages with the *Generate* operation in lines 21-23. Initially, each page has the rank of $\frac{1}{num\_pages}$.
Lines 23-43 of the algorithm are the iterative main part. First, in lines 24-29, we join the linked pages with the ranks, which we previously generated. Both key extractors each extract the page ID. We join all pairs with identical keys - in this algorithm one rank link pair per page - by the join function. The join function returns a pair of the link list with the according page rank.
With the *FlatMap* LOp in lines 30-36 we distribute the page rank evenly to all outgoing links. When there is any outgoing link (line 32), we compute the contribution per link and emit a pair of target ID and rank contribution per outgoing link (line 35).
We reduce these contributions with the *ReducePair* operation in lines 37-39. In this operation, we group all target contribution pairs by their target ID and reduce the contributions by adding them up. Following this reduction, we apply the dampening factor, which represents new users who go to a random page. We start iteration steps with the *Execute* action in line 42.
We repeat this iteration for a fixed amount of iterations, by default 10. Afterwards, the DIA `ranks` contains the page rank for each page in the network. In lines 43-45, we map each of these ranks to an `std::string` and write the strings to disk at a user defined output path.

```
using PagePageLink = std::pair<size_t, size_t>;
using LinkedPage = std::pair<size_t, std::vector<size_t>>;
using RankedPage = std::pair<size_t, double>;
auto input = ReadLines(ctx, input_path)
  .Map([](const std::string& input) {
    char* endptr;
    size_t src = std::strtoul(input.c_str(), &endptr, 10);
    size_t tgt = std::strtoul(endptr + 1, &endptr, 10);
    return std::make_pair(src, tgt); });
size_t num_pages = input.Keep(/*1*/).Map([](const PagePageLink& ppl) {
    return std::max(ppl.first, ppl.second);
  }).Max() + 1;
auto links = input.GroupByKey<LinkedPage>(
  [](const PagePageLink& p) { return p.src; },
  [all = std::vector < size_t > ()](auto& r, const size_t& p) mutable {
    all.clear();
    while (r.HasNext()) { all.push_back(r.Next().second); }
    return std::make_pair(p, all);
  }).Cache().KeepForever();
  DIA<RankedPage> ranks = Generate(ctx,[num_pages](size_t page) {
    return std::make_pair(page, 1.0 /(double) num_pages);
  }, num_pages);
for (size_t iter = 0; iter < iterations; ++iter) {
  auto outs_rank = links.template InnerJoinWith(
    ranks, [](const LinkedPage& lp) { return lp.first; },
    [](const RankedPage& r) { return r.first; },
    [](const LinkedPage& lp, const RankedPage& r) {
      return std::make_pair(lp.second, r.second);
    });
  auto contribs = outs_rank.template FlatMap<PageRankPair>(
    [](const std::pair<std::vector<size_t, double>& p, auto emit) {
      if (p.first.size() > 0) {
        double rank_contrib = p.second / (double) p.first.size();
        for (const PageId& tgt : p.first)
          emit(std::make_pair(tgt, rank_contrib));
      }
    });
    ranks = contribs.ReducePair([](const double& p1,
      const double& p2) { return p1 + p2; })
      .Map([num_pages](const PageRankPair& p) {
        return std::make_pair(p.first, dampening * p.second +
          (1 - dampening) / (double) num_pages);
      }).Execute();
    };
ranks.Map([](const RankedPage& rp) {
  return std::to_string(rp.first) + ": " + std::to_string(rp.second);
}).WriteLines(output_path);
```

Figure 3.3: PageRank Example in Thrill

Algorithm 1: Pseudo code for PageRank Algorithm in Thrill

```
1  function PageRank(input, output, iterations)
2      $S :=$ ReadLines(input)                                   // Read input files (4)
3      $I := S$.Map("i0 i1"$\mapsto (i_0, i_1)$)                 // Map each line to an edge (5-9)
4      $n := I$.Keep().Map($(i_0, i_1) \mapsto max(i_0, i_1)$).Max() $+ 1$      // Find max index (10-12)
5      $L := I$.GroupByKey($(i_0, i_1) \mapsto i_0, (i,$iter$(r)) \mapsto (i, [r])$)      // Create link vector (13-18)
6      $L$.Cache().KeepForever()      // Cache Links and set consume counter to infinite (19)
7      $R :=$ Generate($p \mapsto (p, \frac{1}{n}), n$).Collapse()      // Generate initial ranks (20-22)
8      for $i \leftarrow 0$ to iterations do
9          $O := L$.InnerJoinWith($R, (i, [l]) \mapsto i, (p, r) \mapsto p, ((i, [l]), (p, r) \mapsto ([l], r))$)
           // Join links and ranks (24-29)
10         $C = O$.FlatMap($([l_0, l_1, \ldots], r) \mapsto \frac{\frac{l_0}{r}}{vec\_len}, \frac{\frac{l_1}{r}}{vec\_len} \ldots$)      // Compute contributions (30,36)
11         $R1 = C$.ReducePair($(r_0, r_1) \mapsto r_0 + r_1$)      // Add up contributions (37-38)
12         $R = R1$.Map($(p, r) \mapsto (p, r \cdot \alpha + \frac{1}{n} \cdot (1 - \alpha))$).Execute()      // Use dampening factor $\alpha$ (39-42)
13     $R$.Map($(p, r) \mapsto$"p r").WriteLines(output)      // Write ranks to disk (43-45)
```

Example

Figures 3.4 shows an example graph with 5 nodes and 7 undirected edges. The dampening factor is set to 0.25. The number of iterations is 2.

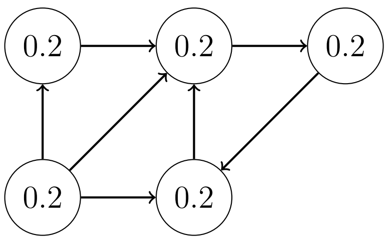

Figure 3.4: Example graph

Before the first iteration, each page has a rank of $\frac{1}{5} = 0.2$. We push these ranks evenly through all outgoing edges of a note. After the iteration, we add all up all incoming ranks on each of the nodes. The resulting rank graph can be seen in Figure 3.5.

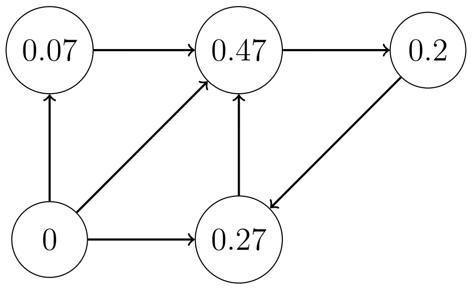

Figure 3.5: Ranks after Iteration 1

Afterwards we apply the dampening factor of 0.25. Therefore, we multiply each of the ranks with $1 - 0.25 = 0.75$ and add $0.25 \cdot \frac{1}{5}$. The resulting ranks can be seen in Figure 3.6.

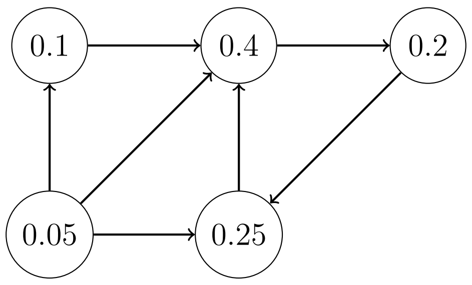

Figure 3.6: Ranks after applying dampening factor

In the next iteration, we push these ranks through the outgoing edges again. This results in Figure 3.7.

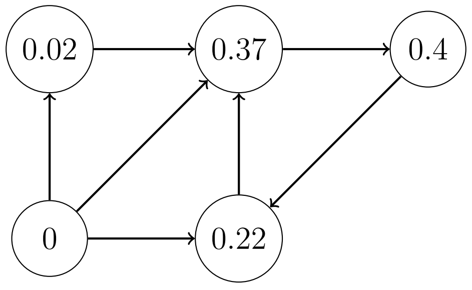

Figure 3.7: Ranks after Iteration 2

We apply the dampening factor again like in the previous step. This rank graph in Figure 3.8 is the end result of the algorithm. Generally, the number of iterations should be larger than 2. A possible number of iterations is 10. It is also a good idea to perform iteration steps until the ranks converge.

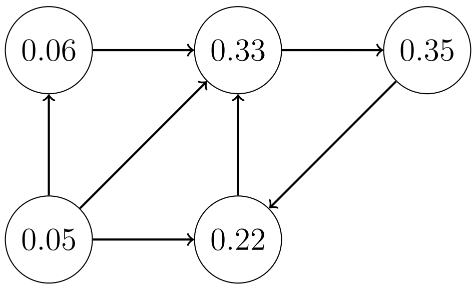

Figure 3.8: Final ranks

3.4.3 TPC H4

Algorithm 2: Pseudo code for modified TPC-H4 benchmark.

```
function TPCH4($input$)
    $L_i$ := ReadLines(input/lineitem.tbl)                                  // Read lineitems
    $L$ := $L_i$.Map("lineitem" $\mapsto$ $lineitem$).Cache().Execute()     // Map line to struct item
    $O_i$ := ReadLines(input/orders.tbl)                                    // Read orders
    $O$ := $O_i$.Map("order" $\mapsto$ $order$).Cache().Execute()           // Map line to struct lineitem
    $net.Barrier()$
    $timer.Start()$
    $J$ := $L$.InnerJoinWith($O, lineitem \mapsto id, order \mapsto id, (lineitem, order) \mapsto joined$)
      // Join all lineitems with orders where ID is equal
    $n$ := $J$.Size()
    $net.Barrier()$
    $timer.Stop()$
    return n
```

The Transaction Processing Performance Council (TPC) [11] is an organisation, which defined several database benchmarks. The TPC-H benchmark is comprised of a suite of decision support queries for a fictional business. One of the queries in the TPC-H suite, the TPC-H4 query, was used to benchmark a repliacted Bloom filter implementation for Hadoop MapReduce [26].

The TPC-H4 query joins a table of lineitems with a table of orders. Tables of any size can be auto-generated using a script offered by the TPC. Database specifications for the TPC-H database can be seen in the official documentation [12].

Lee et al. [26] use a slightly modified version of TPC-H4 to benchmark their Bloom filter implementation, which is also used as a benchmark in this thesis, see Algorithm 2.

The TPC-H database generation script creates a set of tables forming a database of orders

Algorithm 3: Pseudo code for Median in Thrill

```
1 function Median(size)
2     $I :=$ Generate(size, $i \mapsto ((i\%128),(i/719))$)        // Generate input data
3     $E$.Execute()                                                // Execute input before timer start
4     $net.Barrier()$
5     $timer.Start()$
6     $G := E$.GroupByKey($(k,v) \mapsto k, iter(v) \mapsto$ median$(v)$)   // Group pairs by key,
        compute median of values
7     $G$.Size()
8     $timer.Stop()$
```

with total size of n GiB. This database contains a table of $6.000.000 \cdot n$ lineitems and a table of $1.500.000 \cdot n$ orders. In the evaluation of TPC-H4 we read the database tables from AWS S3.
In lines 1-4 of algorithm 2, we read the tables from AWS and map each row to a fixed size struct item. As we do not want to benchmark reading from AWS, we *Execute* caching of these items. When all workers reached the following global barrier, we start the program timer.
This timer only measures the *InnerJoinWith* in line 6. In this join operation, we extract the id of each lineitem and order and join them by creating a joined struct, which contains of all data of both lineitem and order.
The size of a lineitem is 169 byte, the size of an order is 145 byte and the size of a joined element is 306 byte.

3.4.4 Percentiles / Median

A default benchmark for *GroupByKey* is median computation. The median of a data population is the value separating the higher half and the lower half of the sample. To compute the median of a population, all elements need to be available at the same time.
In the evaluation, we generate a large set of key-value pairs comprising of two integers. The first integer represents the key and the second integer represents the value of the pair. The generator creates 128 pairs for each key, all following each other.
Algorithm 3 groups the pairs by their key and day and computes the value median for each key.

In line 2 of Algorithm 3, we generate the input DIA with a user-defined number of elements. We cache this DIA and start the evaluation timer.
We group the elements in the input DIA by their key and compute the median for each of these groups. We insert all values with equal key to a `std::vector`, and we use `std::nth_element` to compute the median of this data vector.

Algorithm 4: Psuedo code for Triangle Counting Algorithm in Thrill

```
function TriangleCounting(input)
    $I$ := ReadLines(input)                                    // Read graph from input files
    $E$ := $I$.FlatMap("i0 i1 i2..." $\mapsto (i_0, i_2), \dots$)   // Map input line to edges, only
      emit edges where targetID > sourceID
    $E$ := $E$.Execute()                                       // Execute input before timer start
    $net.Barrier()$
    $timer.Start()$
    $E2$ := $E$.InnerJoinWith($E, (s,t) \mapsto t, (s,t) \mapsto s, ((s_0,t_0),(s_1,t_1) \mapsto (s_0,t_1))$)
      // Create all edges of length 2
    $T = E$.InnerJoinWith($E2, (s,t) \mapsto (s,t), (s,t) \mapsto (s,t), ((s_0,t_0),(s_1,t_1) \mapsto 1)$)
      // Count triangles
    $n = T$.Size()
    $timer.Stop()$
    return n
```

3.4.5 Triangle Counting

Triangle counting is an algorithm which counts the triangles in an undirected graph. The implementation of triangle counting uses *InnerJoinWith*. The general idea is using *Inner-JoinWith* twice to join the list of edges with the list of edges, which have a length of 2. Algorithm 4 shows pseudo code for our implementation to count triangles.

In line 2 of Algorithm 4, we read the graph from the input files specified by the user. The graph format in this algorithm is one line per node, starting with id of the node followed by a list of all nodes which are adjacent. Every edge is therefore in two lines of the input file. Following the reading process, we use *FlatMap* to map each line of the input into all edges in which the target id is larger than the node id. Each edge of the graph is emitted once, in the line of the node with lower id.
As the algorithm is used to benchmark *InnerJoinWith*, we execute this input operation and then perform a global barrier.
When every processor reaches the global barrier, we start a timer. In line 7 of the algortihm, we perform a self-join on the DIA containing all edges. One key extractor is the edge source and the other is the key target. The join result is an edge with the source of the first edge and the target of the second edge. The resulting DIA is a list of all edges with length two, in which the target id is larger than the source id.
Then we join these length 2 edges with the DIA of edges. In this join, the key extractor is the identity function and the join function returns 1. For every triangle in the original graph, there is one 1 in the output DIA of that join function. In line 8, we return the total number of triangles in the graph.
In order to enumerate triangles instead of just counting them, the InnerJoinWith in line

7 has to include the id of the intermediate node to the edge of length 2. In the second InnerJoinWith, the join function returns the ids of all three nodes present in the triangle.

Example

Figures 3.9 shows an example graph with 4 nodes and 5 undirected edges. We perform the Triangle Count Algorithm as detailed in Algorithm 4 on this graph. All intermediate DIAs are shown as lists of edges.

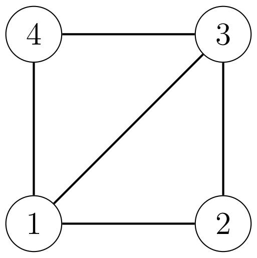

Figure 3.9: Example graph

After the map step in line 3, DIA E contains all edges with targetID > sourceID. Figure 3.10 shows these edges for the example graph. For every undirected edge in Figure 3.9, there is one directed edge in DIA E.

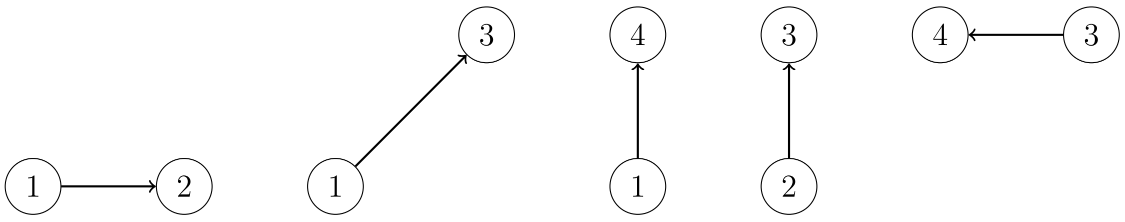

Figure 3.10: List of all directed edges with increasing index

The DIA E2 with all index-increasing edges of length 2 is seen in Figure 3.11. In total there are two edges in DIA E2. The source edges for the length-2-edges can be seen as dashed edges in the Figure.

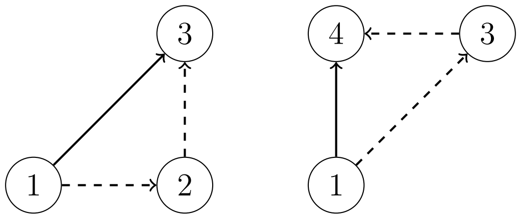

Figure 3.11: List of all directed edges of length 2 with increasing index

Figure 3.12 shows all triangles found by joining the DIAs E and E2.

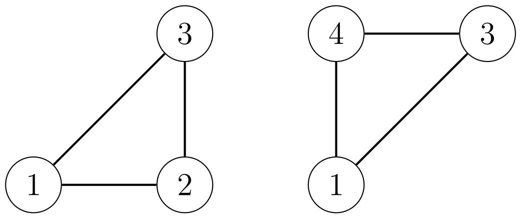

Figure 3.12: List of all triangles found in the example graph from Figure 3.9

4 Bloom Filters

A Bloom filter [7] is a space-efficient probabilistic data structure used to resolve membership queries. They were originally introduced by Burton H. Bloom in 1970. Bloom filters have a non-zero false positive rate but guarantee that there are no false negatives. A Bloom filter allows addition of new elements but no deletion.
On the data level, a Bloom filter is a bit array with a total size of m bits. When the Bloom filter is empty, each of the m bits is set to zero. The capacity of the Bloom filter is n elements $S_0, S_1, \dots, S_{n-1}$ for a small factor $c = \frac{m}{n}$. Additionally, the Bloom filter uses k independent hash functions $H_0, \dots, H_{k-1}$, each with hash ranges in $\{0, \dots, m-1\}$.
When a new value x is inserted into the filter, all hash result bits $H_0(x), \dots, H_{k-1}(x)$ are set to 1. To answer a membership query for a value y, all bits $H_0(y), \dots, H_{k-1}(y)$ are checked. If each bit is set to 1, the data structure has a positive answer, if at least one bit is 0, the answer is negative.

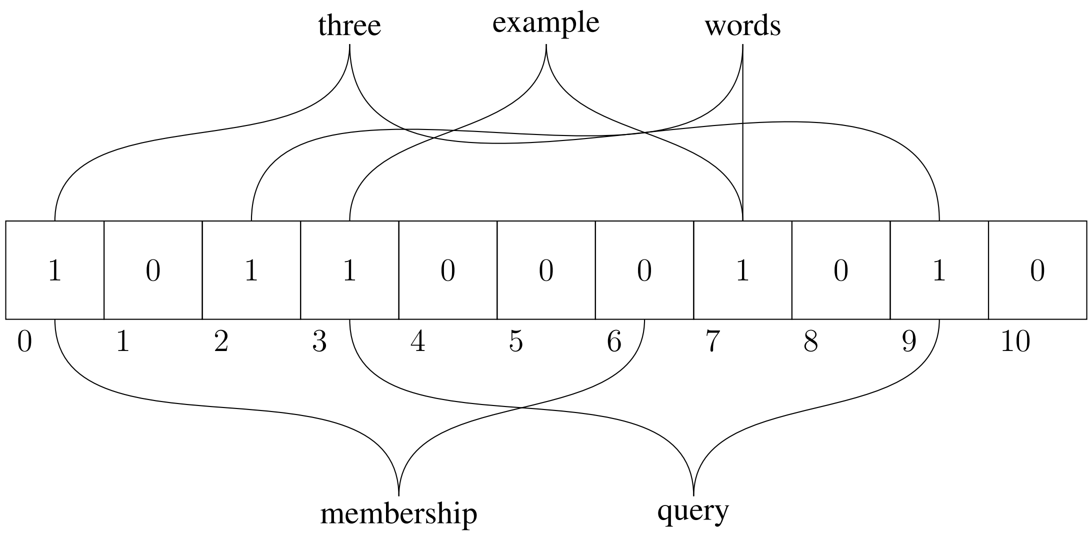

Figure 4.1: Bloom Filter Example with $m = 11, n = 3, k = 2$

Figure 4.1 shows an example Bloom filter with $m = 11$, $k = 2$ and $n = 3$ with three example words, which are “three”, “example”, “words”. The hash functions H_0 and H_1 are truly random functions.
In the beginning, all bits in the Bloom filter are set to 0. The hash functions hash “three”

to the values 0 and 9. After entering the word to the empty filter, bits 0 and 9 are set. “example” is hashed to the values 3 and 7, which are also set by inserting the word. “words” is hashed to 2 and 7. Bit 7 is already set to 1, so inserting “words” only sets bit 2.
In the example filter, there are two membership queries for the words “membership” and “query”. The word “membership” is hashed to the values 0 and 6. As bit 6 is not set, the query returns that the word is definitely not in the dataset. The word “query” is hashed to 3 and 9. Both bits are set, therefore the membership query returns that the word might be in the dataset. This is a false positive.
Multiple words being hashed to the same value is the reason why element removal is not possible in a standard Bloom filter. If the word “example” were to be removed from the filter and bits 2 and 7 are unset, a membership query of the word “words” would return false. As a Bloom filter guarantees a false negative rate of zero, element removal can not be allowed.

The *false positive rate* (FPR) p of a Bloom filter is the probability of a positive query response for a word not present in the data set. If the hash functions are assumed to be truly random functions, in a Bloom filter with m bits, n elements and k hash functions the probability for randomly chosen bits of the Bloom filter being set is approximately [33]:

$$(1 - \frac{1}{m})^{kn} \tag{4.0.1}$$

Equation 4.0.1 follows from the fact that k bits are set for each of the n input elements. In total the bitset has m bits. If $\frac{m}{n \cdot k}$ is too small, the amount of hash collisions is large and the amount of total bits set is lower. Equation 4.0.1 holds approximately true for a bitset with large enough size. In this case, the FPR for a Bloom filter is approximately:

$$p = (1 - (1 - \frac{1}{m})^{kn})^k \tag{4.0.2}$$

Each of the hash results for a non-member has a probability of $(1 - (1 - \frac{1}{m})^{kn})$ to be unset, which is one minus the rate of set bits from Equation 4.0.1. As the Bloom filter has k random hash functions in total, the FPR is as seen in Equation 4.0.2.
For set values for m and n, the FPR of an uncompressed Bloom filter can be minimized by using a total of $k = \ln(2) \cdot \frac{m}{n}$ hash functions [33], which results in a set bit rate of ≈ 0.5.
Multiple Bloom filters can be united by performing a bitwise OR on all Bloom filters.

4.1 Counting Bloom Filter

A variant of the Bloom filter is the counting Bloom filter [17]. A counting Bloom filter uses c bits for each of the m values. A Counting Bloom filter thus has a total size of $c \cdot m$. When inserting an element x to the Bloom filter, all hash results $H_0(x), \ldots, H_{k-1}(x)$ are incremented by one. When one of the hash results is already $2^c - 1$, the result value can not be incremented and remains at $2^c - 1$. Membership queries are equal to a standard Bloom filter: for an element y, the query has a positive result when each hash result $H_0(y), \dots, H_{k-1}(y)$ is at least 1. Alternatively membership queries can also be answered with the smallest integer in $H_0(y), \dots, H_{k-1}(y)$ to denote the upper bound of elements which are equal to y. This mostly makes sense in the case $k = 1$.

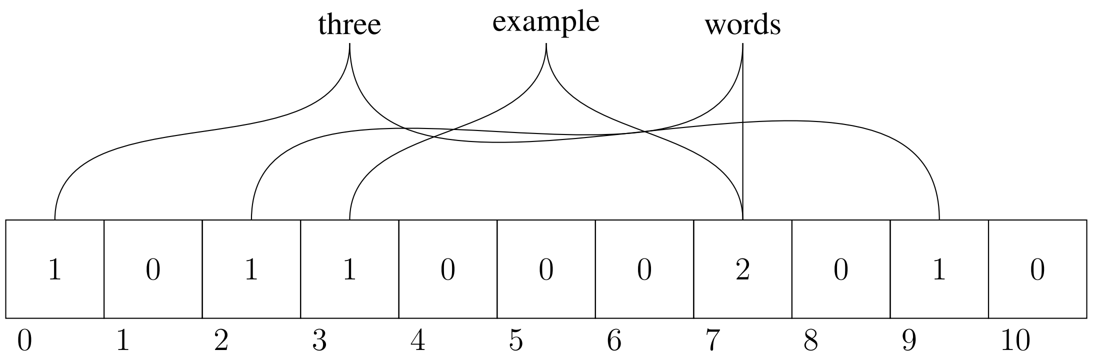

Figure 4.2: Counting Bloom Filter for Example from Figure 4.1

In a counting Bloom filter, removing elements from the filter is possible. When removing an element z, all hash results $H_0(z), \dots, H_{k-1}(z)$ are decremented by one. False negatives are only possible when a hash result has been incremented more than $2^c - 1$ times and the counter is afterwards decremented to zero. Fan et al. [17] deemed that scenario very unlikely in real world examples.

4.2 Compressed Bloom Filter

In standard Bloom filters, the number of hash functions k is optimized to achieve a set bit rate of approximately 0.5. This minimizes the FPR for a given bitset size m and capacity n. According to information theory, a bit rate of 0.5 maximizes the information rate to 1 bit per data bit of the Bloom filter. The bitset thus can not be compressed, as it already has the smallest possible size.

If the bitsets may be compressed, a lower information rate might result in a lower compressed size for a different k. Mitzenmacher [33] showed that the FPR for a given compressed size and n can be minimized by using 0 or infinite hash functions for optimal compression algorithms. As the number of hash functions needs to be an non-zero integer, $k = 1$ has the lowest FPR in the compressed Bloom filter.

Using only a single hash function also increases the performance of the filter, as only a single hash function needs to be evaluated for each insert or query.

On a bitset with uniformly distributed 1 bits, the distance between 1 bits is geometrically distributed. A possible compression algorithm for the bitset is Golomb encoding [18] of distances between set bits, described in Section 4.2.1. Golomb encoding is near-optimal for geometrically distributed integers and therefore well suited for the compression of Bloom filters [38]. The Golomb encoded bitset is static, it is not easily possible to insert new elements after compressing the bitset.

4.2.1 Golomb Encoding

Golomb encoding [18] is a variable length encoding for small integers. A Golomb code has a parameter M. To encode an integer N, it is split into two parts: $q = \lfloor \frac{N}{M} \rfloor$ and r, the remainder of the division $\frac{N}{M}$.
The Golomb encoding of N is the unary encoding of q followed by the truncated binary encoding of r.
Truncated binary encoding is a generalization of binary encoding for alphabet sizes $M \neq 2^{\mathbb{N}}$. When M is not a power of two, let $k = \lfloor \log_2 M \rfloor$ and let $u = 2^{k+1} - M$ be the distance to the next power of two. The truncated binary encoding of $x \in \{0, u-1\}$ is equal to the binary encoding of x. For $x \in \{u, M-1\}$, the truncated binary encoding of x is mapped to the last $M - u$ values of length $k+1$. The truncated binary encoding of x is equal to the binary encoding of $x + u$.

0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 4.3: Example Bitset for Golomb Encoding

Example In Figure 4.3, we have a bitset with $m = 22$ and 4 bits set. The Golomb Encoding of the bitset with parameter $M = 5$ can be found as follows:
The first set bit is bit 6. $q_1 = \lfloor \frac{6}{5} \rfloor = 1_{10}$(ternary encoding) $= 10_1$ (unary encoding) and $r_1 = 5 - 5 = 0_{10} = 01_{t2}$ (truncated binary encoding). The golomb encoding of 6 is 1001.
The next set bit is 10, therefore the delta is $10 - 6 = 4$. $q_2 = \lfloor \frac{4}{5} \rfloor = 0_{10} = 0_1$ and $r_2 = 4_{10}$. In the bitset, $u = 2^3 - 5 = 3$. The remainder 4 is $\geq u$, therefore the truncated binary encoding of $r_2 = 111_{t2}$. The encoding of 4 is 0111.
The next bit is 17, therefore the delta is $17 - 10 = 7$. $q_3 = \lfloor \frac{7}{5} \rfloor = 1_{10} = 10_1$ and $r_3 = 2_{10} = 10_{t2}$. The encoding of 7 is 1010.
The next bit is 18, therefore the delta is $18 - 17 = 1$. $q_4 \lfloor \frac{1}{5} \rfloor = 0_{10} = 0_1$ and $r_4 = 1_{10} = 01_{t2}$. The encoding of 001.
The whole bitset in Figure 4.3 is encoded with the bitset 100101111010001.

4.3 Distributed Bloom Filters

Most Bloom filter variants are sequential data structures which run on a single machine. A recent survey by Tarkoma et al. [46] shows a multitude of sequential Bloom filter variants but no real distributed Bloom filter structure. In several works that use distributed Bloom filters, each processor builds a local Bloom filter for all local data [26] [44]. The Bloom filters are combined by perfoming bitwise OR. This replication increases the amount of necessary communication and storage.
The distributed single-shot Bloom filter [40] (dSBF) is an actually distributed Bloom filter with a single hash function. The dSBF uses only a single hash function to minimize the structure size while compressed.
The basic structure of a dSBF with p processors is a Bloom filter with a total size of m bits, split into p equally large parts. Processor i is responsible for the range of bits from $\frac{i \cdot m}{p}$ to $\frac{(i+1) \cdot m}{p} - 1$. When the dSBF is initialized, all bits in the distributed bitset are unset.
In order to construct the dSBF data structure, each processor computes the hash value $H(x)$ for each local element x. These hash values are then sorted. In the communication step, each worker sends a compressed bitset to each other worker. These bitsets consist of the golomb encoded deltas of the hash values in the range of the receiving processor.
The received elements are decompressed and each processor can build the global Bloom filter for it's local range by inserting all hash values into a bitset. In a dSBF, membership queries for an element y can be answered by sending $H(y)$ to the responsible worker. The performance is higher when queries or inserts are performed in batches.
In the original paper [40], the dSBF data structure is used to perform distributed duplicate detection. For this purpose, each worker builds his local hash bitsets and sends them to the other workers. Each worker then checks the incoming bitsets for hash values which occur on more than one processor. These hash values are signalled back to their source processor.
Only the elements, which hash values were signalled back, are sent to the processor responsible for their hash value. This algorithm greatly reduces communication volume and running time for low replication factors compared to a hash repartition algorithm which repartitions all elements.

5 InnerJoin in Thrill

5.1 Overview

InnerJoin is a *Distributed Operation* (DOp) in Thrill implemented as a part of this thesis. The operation performs an inner join on two input DIAs with arbitrary types. Inner join, also known as join, is a keyword in the SQL standard [13]. The inner join operation joins elements with matching keys from two database tables.
In Thrill, *InnerJoin* is called on two DIAs with types *DIA*$\langle A \rangle$ and *DIA*$\langle B \rangle$, where A can be equal to B. *InnerJoin* is further specified by three user defined functions with the following function types:

$$k_1 : A \to K \tag{5.1.1}$$

$$k_2 : B \to K \tag{5.1.2}$$

$$j : A \times B \to C \tag{5.1.3}$$

The key extractor functions k_1 and k_2 map each element from input DIAs to it's key of type K. For a certain key x, x_A denotes the list of elements from the first DIA with key x and x_B denotes the list of elements from the second DIA with key k.
The join function j is applied to each element of the cross product $x_A \times x_B$. The output DIA of type *DIA*$\langle C \rangle$ consists of all elements emitted by j.
As InnerJoinWith performs an inner join, no element is emitted for key x when at least one of the lists x_A and x_B is empty. In outer joins, the unmatched elements would be emitted as is. Outer joins would restrict the DIA types, as output and input need to be equal. They are not yet implemented in Thrill, but could be implemented similar to InnerJoinWith. It is also possible to implement left and right outer joins, which emit all unmatched elements in either DIA A or B.

5.2 Implementation

This Section shows the implementation of InnerJoinWith in Thrill. This is the implementation without Bloom Filters, the variant using Bloom Filters to reduce communication value is shown in Chapter 6. The different phases of the algorithm are also detailed in an example performing the InnerJoinWith of two small example DIAs. The types of these input DIAs are `A := std::pair<int, std::string>` and `B :=`

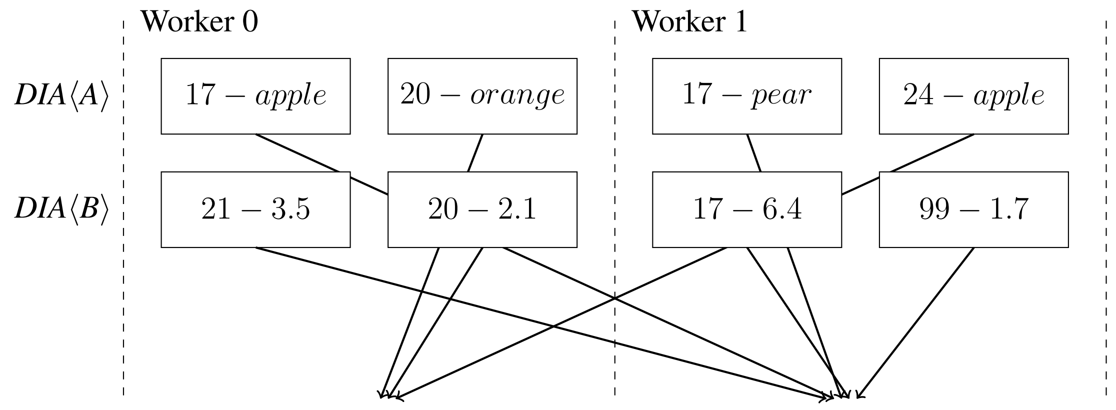

Figure 5.1: Link Phase of InnerJoinWith Example

`std::pair<int, double>`. The type C of the output DIA is `std::tuple<int, std::string, double>`.

5.2.1 Link Phase

The *Link* phase of InnerJoinWith is linked with the previous LOp chain. On construction of the JoinNode, we open two mix data streams and writers to these streams. We use these data streams to shuffle the input elements by their key.
In the *Link* function, we apply the according key extractor function k_1 or k_2 to each input element. We hash these keys and send each element to the worker with global worker ID $H(key) \pmod p$. As both the key extractor and hash function are deterministic, all elements with a certain key are sent to the same worker.
After all elements of an input DIA passed the *Link* phase, we close the stream writers.

Figure 5.1 shows the *Link* phase of a graphical example for InnerJoin.
The type A of *DIA*⟨*A*⟩ is a pair of integer and string. The key extractor function k_1 emits the integer. The type B of *DIA*⟨*B*⟩ is a pair of integer and floating-point number. The key extractor function k_2 also emits the integer.
To simplify the example, the hash function for target worker selection is just the identity function. Therefore, all elements with even key are sent to worker 0 and all elements with uneven key are sent to worker 1.

5.2.2 Main Phase

In the *Main* phase of InnerJoin, all elements of both DIAs are received on their target workers. For each DIA, we add all recieved elements to an `std::vector` until either the stream is empty or the memory soft limit is reached. Then we sort the `std::vector` using `std::sort` by key.

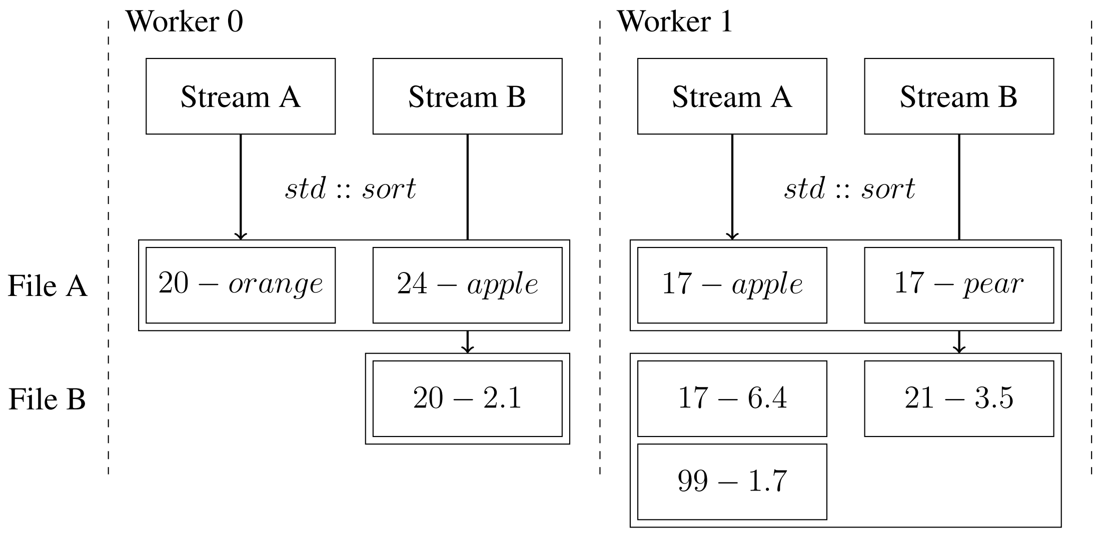

Figure 5.2: Main Phase of InnerJoinWith Example

The sorted `std::vector` is written to a Thrill *File*. This *File* is potentially written to external memory when the internal memory is exhausted. After the *Main* phase, each worker holds one or more sorted *Files* for each of the two input DIAs.
According to our execution model, we only perform this phase once. Reruns of the JoinNode node only call the *Push* phase.
The example from Figure 5.1 is continued in Figure 5.2. In the example, each worker has only a single *File* per input DIA. All elements received from the streams are sorted and inserted into *Files*.

5.2.3 Push Phase

After the *Main* phase, every worker has a list of *Files* sorted by element keys, potentially in external memory. In order to perform the actual join process, the *Files* need to be merged to a single sorted data stream. For this purpose, the lower layers of Thrill provides a multiway merge tree. This merge tree takes the list of all sorted *Files* and provides a single iterator for all data.
Internally, the smallest elements of all *Files* are stored in a *buffered loser tree* [39]. The loser tree is a data structure for k-way-merging. Removing the smallest element from the loser tree has a complexity of $\mathcal{O}(\log(m))$, where m is the number of *Files*. We read each *Block* only once from disk. The implementation of the loser tree was taken from the *Multicore Standard Template Library* [43].
In the actual joining process, we compare the smallest elements from both iterators. If they are not equal, we discard the smaller element and update the iterator to the next smallest element.

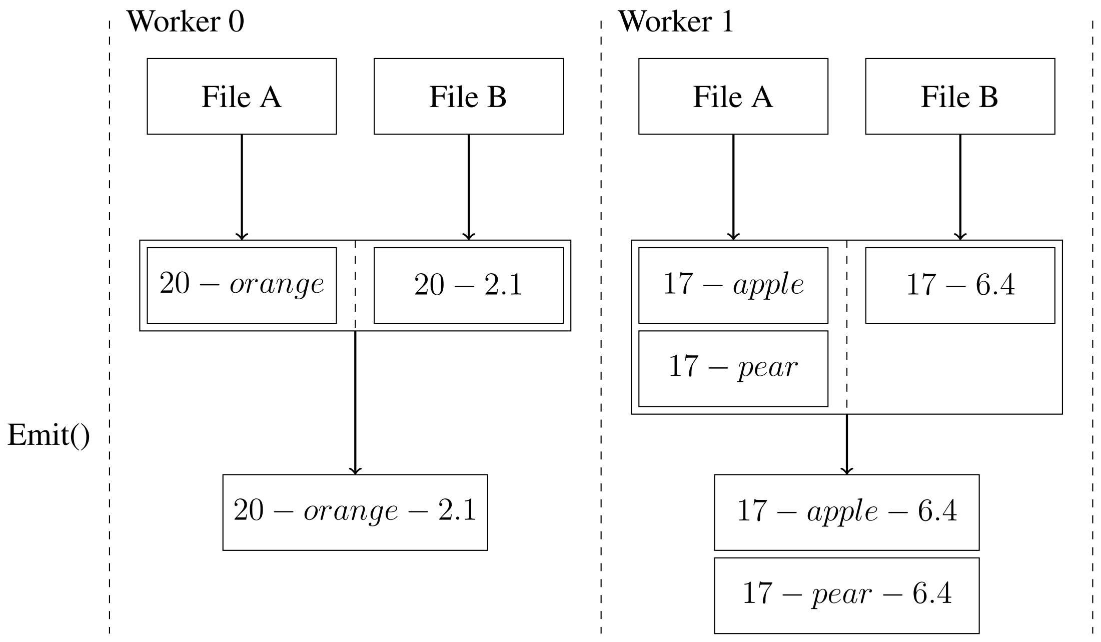

Figure 5.3: Push Phase of InnerJoinWith Example

If the elements are equal, we add all elements with equal key to two `std::vector`s, one per merge tree. After we gathered all elements, the join function j is called for every pair in the cross product of the `std::vector`. We emit each joined elements to the subsequent operations.

Figure 5.3 continues the example from Figure 5.1 and Figure 5.2. As the example is small, every worker only has a single *File* per input DIA. There is no need for merging as the loser tree only has a single input File. In the example, only the keys 17 and 20 occur in both input DIAs. For each pair of elements with that key, we emit one element. The join function j joins the pairs to a tuple of integer, `std::string` and floating point number. Note that in this example, elements with keys not equal to 17 and 20 do not result in emitted elements. It is preferable both in running time and communication volume to discard these elements as early as possible, if possible before data transmission in the *Main* phase. It is also beneficial to send elements to the worker which already has most elements with that key. For this purpose, Thrill uses dSBF to eliminate unique items. These Bloom filters and their use are described in Chapter 6.

6 Distributed Bloom Filters in Thrill

6.1 Overview

One of the possible bottlenecks for big data frameworks is the network bandwidth between individual computing nodes. If we can reduce the communication volume in our programs, this can increase the general performance of our framework. This is especially true in commodity clusters such as the *Amazon Elastic Compute Cloud* (AWS EC2) [24], as commodity clusters generally have lower communicaton bandwidth than super computing clusters.
In several DOps in Thrill (*Reduce*, *InnerJoin*, *GroupBy*), all data in the DIA is shuffled by hash in a global communication step. When there is no information about the data, this complete communication step is necessary. However, it is potentially beneficial to use some computation and communication to find elements which can be excluded from this communication step.
The optimization examined in this thesis uses a dSBF [40] as described in Section 4.3 in these operations to find elements which do not need to be sent to other workers.
In order to find these elements, a dSBF is distributed over all workers. Every worker hashes all available keys, sorts the hashes and builds a dSBF out of these hashes. The dSBF parts are shuffled between workers to find unique elements and the prevalent location of elements.
This process introduces additional work and communication but may yield a net positive for the whole program runtime and reduce total network traffic volume, depending on the algorithm and the input.

6.2 Duplicate Detection in Reduce

In the *ReduceByKey* DOp, elements are grouped by their key and each key group is reduced with an associative reduce function. The key extractor function returns the key of each element. The associative reduce function defines how two elements with equal key are reduced to one element.
In the *Link* phase of *ReduceByKey*, each worker creates a custom hash table to perform local reduction. This table contains the key and value of each element and immediately applies the reduce function on key collision. After the Link Phase is finished, each worker has a set of locally reduced key value pairs. By default, all of these pairs are shuffled by

Algorithm 5: Pseudo code for duplicate detection in Thrill

```
1  function DetectDuplicates([hashes])
2      $U := net$.AllReduce(hashes.size())   // Find global upper bound of distinct keys
3      $B := U \cdot \frac{1}{FPR}$   // Define dSBF size with U and false positive rate
4      $hashes \mapsto hashes \pmod{B}$   // Compute modulo B for each hash value
5      Sort(hashes)   // Sort the array of hash modulos
6      $GBF$ := WriteGolombCodes(hashes)   // Create golomb encoded Bloom filters
7      Shuffle($GBF$)   // Global shuffle of Bloom filter parts
8      $LT$ := CreateLoserTree($GBF$)   // Create loser tree with all incoming Bloom filter parts
9      $GU := []$   // Create empty Golomb encoded bitset for each worker
10     while $LT$.HasNext() do
11         if $LT$.Next() is unique then
12             $GU_{src}$.Insert(next)   // Insert unique element to bitset for source worker of element
13     Shuffle($GU$)   // Global shuffle of unique bitsets
14     return Concat($GU$)   // Return concatenated Bloom filter bitset
```

hash of their key.
Some elements only occur on a single worker. These elements do not need to be sent through the network in the shuffling process. The duplicate detection uses a dSBF to find most of these unique elements. When the amount of unique elements is high, the total communication volume can be reduced.

Algorithm 5 depicts the duplicate detection in *ReduceByKey* in pseudo code. The duplicate detection has a `std::vector` of all hashes of keys as its input parameter.
In the duplicate detection algorithm, we start with the global communication primitive *AllReduce* to find the upper bound of distinct keys. This upper bound is the sum of all local counts of distinct keys. The size of the dSBF is equal to the upper bound of distinct keys multiplied with the inverse of the desired FPR of the dSBF. The default value for the FPR is $\frac{1}{8}$.
Afterwards, we apply the modulo operator to each hash value with the modulus being the dSBF size and sort the hashes, which are now in $[0, B)$.
We can write the encoded Bloom filter parts from the elements in the `std::vector` of hashes. Each worker builds a golomb Encoded bloom filter part for each worker. The part for destination worker i contains all elements in $[\frac{B \cdot i}{p}, \frac{B \cdot (i+1)}{p})$. As the first element in the bitset is possibly very large, it is not encoded. The deltas of all subsequent elements are Golomb encoded.
We perform a global shuffle step, in which we interchange the Bloom filter parts. After this shuffle step, each worker has p Bloom filter parts with the hash values it is responsible for.

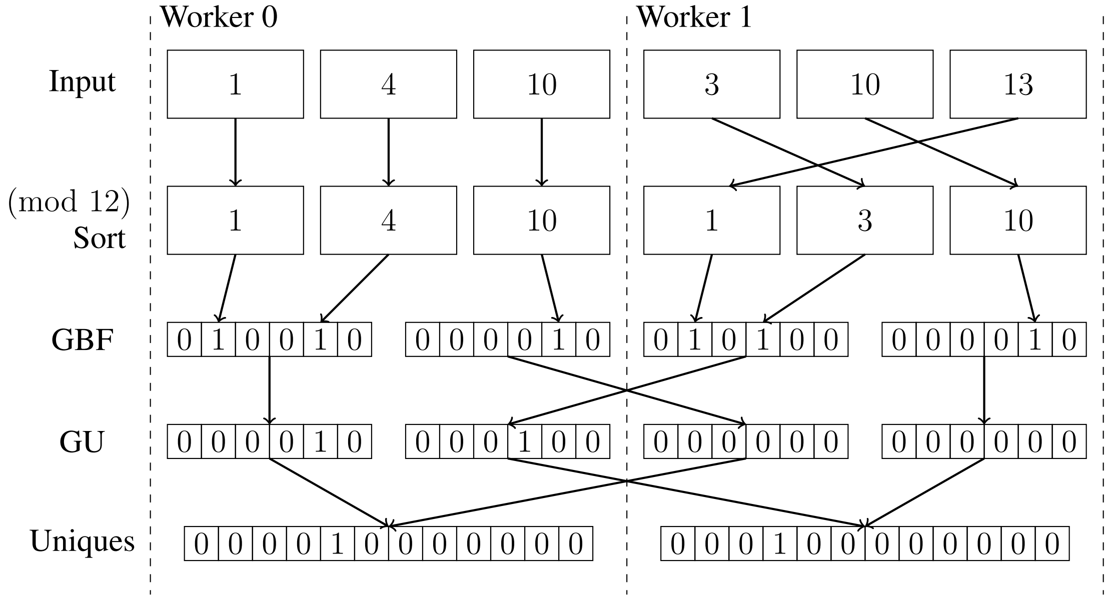

Figure 6.1: Example for Duplicate Detection

We insert these parts in a loser tree which provides an iterator to all elements, sorted from smallest to largest.
We iterate over all elements to find the elements which appear only on a single worker. If we find such a unique element, we insert it into a golomb encoded bitset for it's source worker. In a second global shuffling step, we again interchange these bitsets and each worker can concatenate its bitsets to a single bitset. A bit set to true in this bitset indicates that only this worker holds elements with this key.
These elements, which appear only on a single worker, do not need to be distributed but can be kept on this worker. Due to hash collisions in the Bloom filter it is however possible to overlook uniques. This FPR is in a trade off with the Bloom filter size.
Good applications for duplicate detection are those, in which the number of elements only occuring on a single worker and the size of elements is large.

Figure 6.1 shows an example for the duplicate detection. In the example, each of the 2 workers has an input of 3 hashed keys. To keep the example reasonably small, the target FPR is $\frac{1}{2}$. Thus the dSBF size is $6 \cdot 2 = 12$. We also omitted Golomb encoding of bitsets for ease of understanding. In reality, each of the bitsets communicated in GBF and GU is Golomb encoded as described in Section 4.2.1.
In the duplicate detection example, the dSBF correctly detects that 10 is a duplicate and falsely detects 1 as a duplicate. The output bitset for worker 0 shows that the value 3 only occurs on this worker, the bitset for worker 1 shows that the value 4 only occurs on this worker.

Algorithm 6: Pseudo code for Location Detection in Thrill

```
function DetectMaxLocation(HashTable ht)
    hashes := $ht$.Emit()                          // Emit all elements from table into array
    $U := net$.AllReduce(hashes.size())            // Find global upper bound of distinct keys
    $B := U \cdot \frac{1}{FPR}$                   // Define dSBF size with U and False Positive Rate
    hashes $\mapsto$ hashes $\pmod{B}$             // Compute modulo B for each hash value
    Sort(hashes)                                   // Sort the array of hash modulos
    $GBF$ := WriteGolombCodesOccurences($hashes$)  // Create golomb encoded Bloom
      Filters with number of occurences
    Shuffle($GBF$)                                 // Global shuffle of Bloom Filter parts
    $LT$ := CreateLoserTree($GBF$)                 // Create Loser Tree with all incoming Bloom Filter
      parts
    $GU$ := []                                     // Create one empty golomb encoded bitset
    while $LT$.HasNext() do
        while $LT$.Next().Key() is equal do
            $GU$.Insert(key, worker)               // Insert pair of key and optimal worker into
              bitset
    net.AllGather($GU$)                            // Broadcast location bitset, recieve other worker's bitsets
    return unordered_map($GU$)                     // Return concatenated Bloom Filter bitset
```

6.3 Location Detection

In contrast to *ReduceByKey*, local pre-reduction is not possible in *InnerJoin* and *GroupBy*. Therefore, it is not only important to detect whether an element occurs only on a single worker but also on which worker it occurs most. We choose the worker with the highest occurence count as the target worker for this element. As only the elements not already on this worker need to be communicated, this processor is the optimal target in terms of expected communication volume, if we assume all elements to have equal size.

The Algorithm 6 for location detection can be seen as an extended version of the Algorithm 5 for duplicate detection. When location detection is disabled in *InnerJoin* or *GroupByKey*, we shuffle elements by hash of the key in the *Link* phase.
When location detection is enabled, this is not possible, as the optimal location for the elements is not yet known at this point. Therefore we have to store incoming elements in a *File*. Additionally, the keys are inserted into a hash table. This hash table counts the occurences of each key in the local data set.
In Algorithm 6, we first have to emit all data from the table into a `std::vector`. This is done with a custom emitter function which emits data by pushing it to the `std::vector`. The following steps are similar to duplicate detection: we compute the dSBF size, apply modulo dSBF on size on each hash values and sort the results.

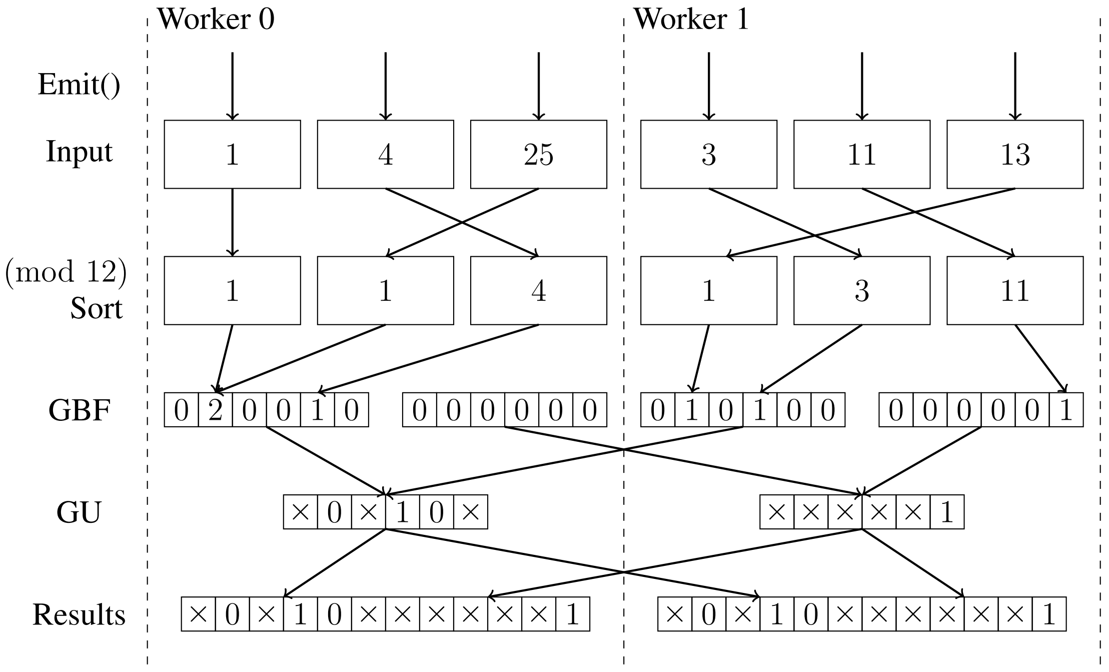

Figure 6.2: Example for Location Detection

In the Golomb code step, we use the bitset to store both the values and their number of occurences. Therefore we append a non-encoded 8 bit counter to each element. In our use case, this is preferable to a counting Bloom filter, as described in Section 4.1, because we use compressed Bloom filters. Therefore the fill rate, which is the rate of set bits in the bitset, is considerably lower than in non-compressed Bloom filters.
Let f be the fill rate of the Bloom filter and B be the number of buckets in the filter. We set the bucket sizes to 8 bits to be able to count up to 255 elements per key. A counting Bloom filter with bucket sizes of 8 bits has a total size of $8 \cdot B$. The Golomb encoded counting Bloom filter has a total size $<B + 8 \cdot f \cdot B$ bits. The first part is the non-counting Golomb encoded filter, the second is a 8 bit counter for each filled bucket. For fill rates $f < \frac{1}{2}$ the Golomb encoded variant is more space efficient. By default, we limit the bucket size to 8 bit and truncate values of 2^8 and higher to $255 (2^8 - 1)$.
In *InnerJoin*, we also use 2 additional bits to label whether a key appeared in DIA A and / or DIA B. This can be used to discard elements which do not appear in both DIAs, as they can't have any join partners.
The bitset GU is also a counting Golomb encoded bitset. For each key, we insert a pair of the key and the optimal worker into the bitset. For the worker ID we use $\log_2 p$ bits. After these bitsets are broadcast similar to MPI AllGather, each worker builds a `std::unordered_map` to map keys to their target worker.
In the *Main* phase of *InnerJoin* and *GroupByKey*, each element in the file is then sent to the worker indicated by the map. In *InnerJoin*, elements not appearing in the map are

discarded, as they did not appear in both DIAs. In *GroupByKey*, each hash of key should appear in the map.

Figure 6.2 shows an example for the location detection. In the example, each of the 2 workers has 3 elements, the desired FPR is $\frac{1}{2}$. For ease of understanding, the bitsets are not compressed. The resulting map is displayed as an array, in which integers denote the optimal processors and $\times$ denotes a value which did not appear in the bitset. In *InnerJoin*, the buckets 3, 4 and 11 would be discarded as they only appeared on a single worker. Only bucket 1 would be in the resulting map.

7 Experimental Evaluation

This chapter uses multiple algorithms described in Section 3.4 to evaluate the performance of duplicate and location detection in the Thrill framework. This evaluation is performed using the wall clock time of algorithms and algorithm parts. Additionally, the total amount of network traffic in the cluster is recorded.

7.1 Implementation

In order to evaluate the performance of our location and duplicate detection, we compare wall clock time and total network traffic of Thrill implementations with and without Bloom filters. All of these performance benchmarks are run on the AWS EC2 computing cluster [24].
We selected five different algorithmic micro-benchmarks: *WordCount*, *PageRank*, *TPC-H4*, *Triangle Counting* and *Medians*. For each of these micro benchmarks, we performed weak and strong scaling experiments with multiple workload sizes. The implementations of the micro-benchmarks are equal to the implementations in the Thrill introduction paper [6]. Each of the micro-benchmarks is performed on 1 to 32 nodes from the AWS EC2 cluster.

7.2 Experimental Setup

7.2.1 Environment

The benchmarks were performed on multiple nodes of the AWS EC2 computing cluster using r3.xlarge instances. Each of these instances contains 4 vCPU cores of an Intel Xeon E5-2670 v2 with 2.5 GHz, 30.5 GiB RAM and a local 80 GB SSD storage.
These machines are smaller versions of the machines used in the Thrill introduction paper. Therefore, the results from this work can be generalized to the results in that paper. It therefore allows comparison of the performance with the frameworks Apache Spark and Apache Flink, as the Thrill introduction paper evaluates the performance of Thrill compared against those frameworks.
We implemented an AWS S3 reader for Thrill as part of this work. This reader reads data directly from AWS S3 into a Thrill program without caching it on disk.

The machines are interconnected using a fully-meshed TCP/IP backend. The binary, which contains the micro-benchmark, is placed on each of the machines before the program is called. All IP addresses and their according ports are given as parameters to the Thrill program.
Every experiment is performed three times, we plot the median of the three result values.

7.2.2 Tuning Parameters

We assume that the tuning parameters of the Thrill framework are already optimized well. In general, Thrill does not even have a lot of tuning parameters. The available tuning parameters, such as block sizes, hash table fill rates, etc. were optimized in previous development steps.
The location and duplicate detection each contain distributed single-shot Bloom filters. Most of the optimization parameters are inherited from the paper introducing the dSBF [40]. This section looks at the parameters for the golomb encoder and shows throughput and data structure size for different values of the golomb parameter M.

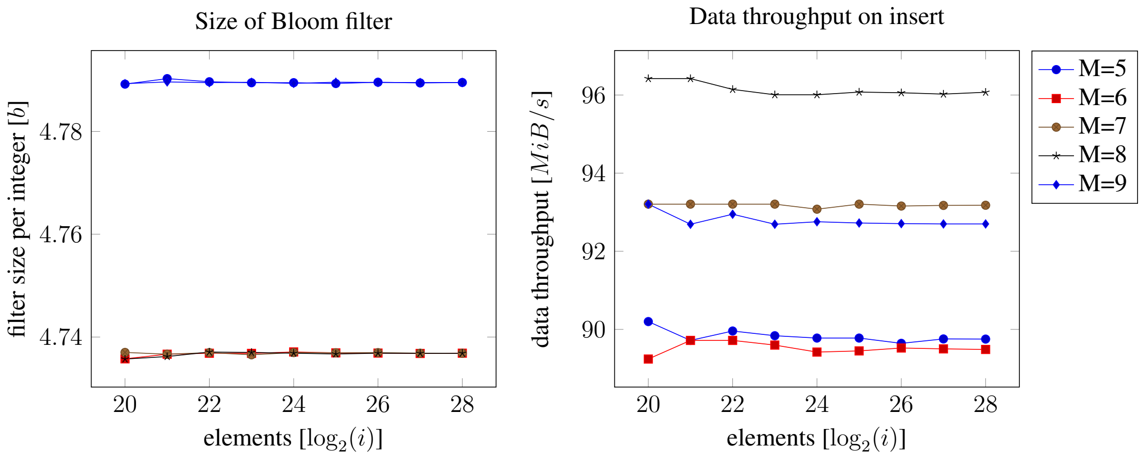

Figure 7.1: Size and throughput of golomb encoded bitset

Figure 7.1 shows the results for a micro-benchmark, in which small random integers are inserted. The benchmark generates random integers in range $[1, 19]$. We chose this range, as it is close to the expected range of deltas between hash values in the actual dSBF with a FPR of $\frac{1}{8}$. These integers are inserted into a golomb encoded bitset with FPR $\frac{1}{8}$. We plotted insert throughput and bitset size for golomb parameter(M) values from 5 to 9. The number of integers inserted are in a range of 2^{20} to 2^{28}.
For values $M = 6, 7, 8$, the average size of the bitset is $4.73b$ per element, for $M = 5, 9$ the size of the bitset is $4.79b$ per element. The data throughput is highest when $M = 8$, followed by $M = 7$ and $M = 9$, which have 3% less data throughput.
Therefore we chose $M = 8$, as it both has the lowest bitset size and the highest throughput.

Presumably the throughput is highest, because divisions by 8 can be performed as a simple bit shift.

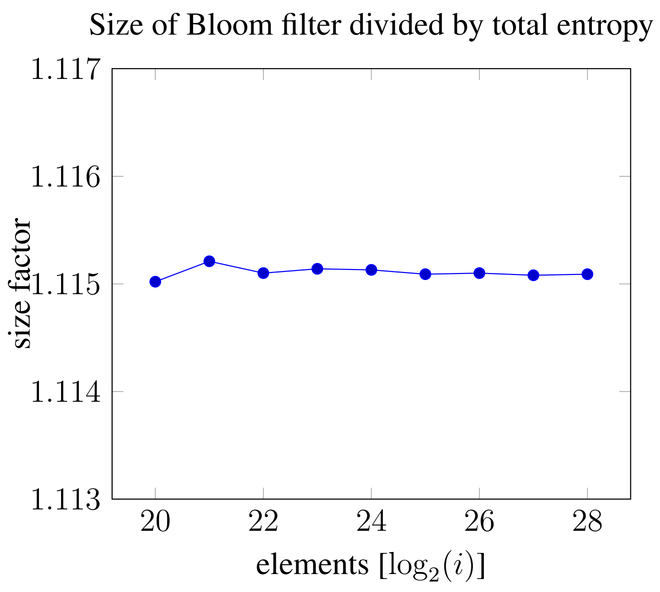

Figure 7.2: Factor of bitset size to total entropy

Figure 7.2 plots the factor of bitset size to total entropy, which is the theoretical minimum size for the bitset. The bitset has a golomb parameter of $M = 8$. The size factor is plotted for 2^{20} to 2^{28} random integers in range $[1, 19]$.
The size factor of the bitset is between 1.115 and 1.116, independent of the bitset size. All following experiments are performed on bitsets with $M = 8$ and FPR $\frac{1}{8}$.
The results of this synthetic micro-benchmark are very positive, as the factor of bitset size to total entropy is lower than $\frac{9}{8}$.

To check real-world data, we also performed the same benchmark with the *WordCount* algorithm on data from the *CommonCrawl* data corpus.
Figure 7.3 shows this size factor with real-world data on a single AWS r3.xlarge node with 4 CPU cores. The duplicate detection in the reduce step in *WordCount* uses Golomb encoded bitsets on two separate occasions. In step 1, all elements are shuffled. In step 2, only the elements unique to a single worker are shuffled. Therefore the deltas between elements in step 2 are larger than in step 1.
When the number of files is 1 or 2, not every processor has an input file, as the *CommonCrawl* corpus is in compressed data files. When each processor has input, the size factor in step 1 is approximately 1.17 and the size factor in step 2 is approximately 1.51. It is larger in step 2, as the values inserted are substantially larger on average.

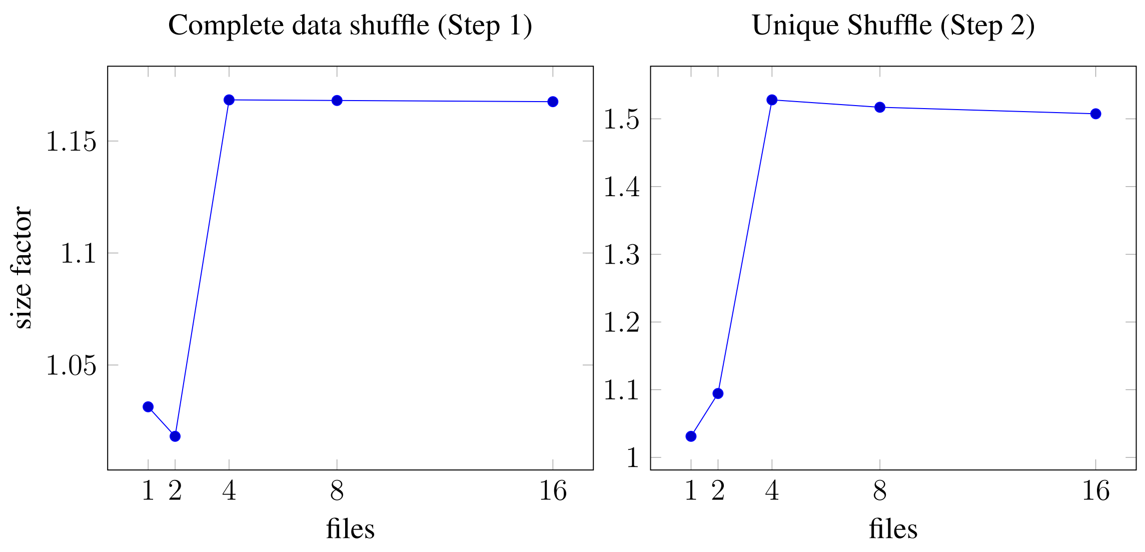

Figure 7.3: Factor of bitset size to total entropy in *CommonCrawl WordCount*

7.3 Experimental Results

We performed each micro-benchmark with detection set to on and off on 1 to 32 AWS EC2 r3.xlarge nodes, so in total with a cluster size of 4 to 128 CPU cores. In the results, we show data throughput and total net traffic in the Thrill program, as well as speedup and communication volume ratios. We performed each experiment 3 times and used the median result.

7.3.1 Word Count

In the *WordCount* benchmark, we used data from the *CommonCrawl* data corpus. Each of the files in the *CommonCrawl* data corpus contains 392MiB of text data, which is stored on the AWS S3 storage.
The upper half of Figure 7.4 shows weak scaling results as well as speedup for WordCount for 4, 16 and 32 files per host. We can see that the use of duplicate detection only yields a small speedup when the amount of files is large and the total amount of hosts is 16 or lower.
In the lower half of Figure 7.4, which depicts the total amount of net traffic in the program, we can see the reason for this problem. Due to the large amount of processors, the upper bound for the number of unique elements in the Duplicate Detection becomes very large. Therefore communication of the large bloom filters yields a large amount of additional traffic, even more than the traffic for the words which can be marked unique.
While the volume ratio of communication is below 0.4 for 2 hosts and 32 files per host, it increases on larger host sizes and reaches 1.32 with 32 hosts and 32 files per host.

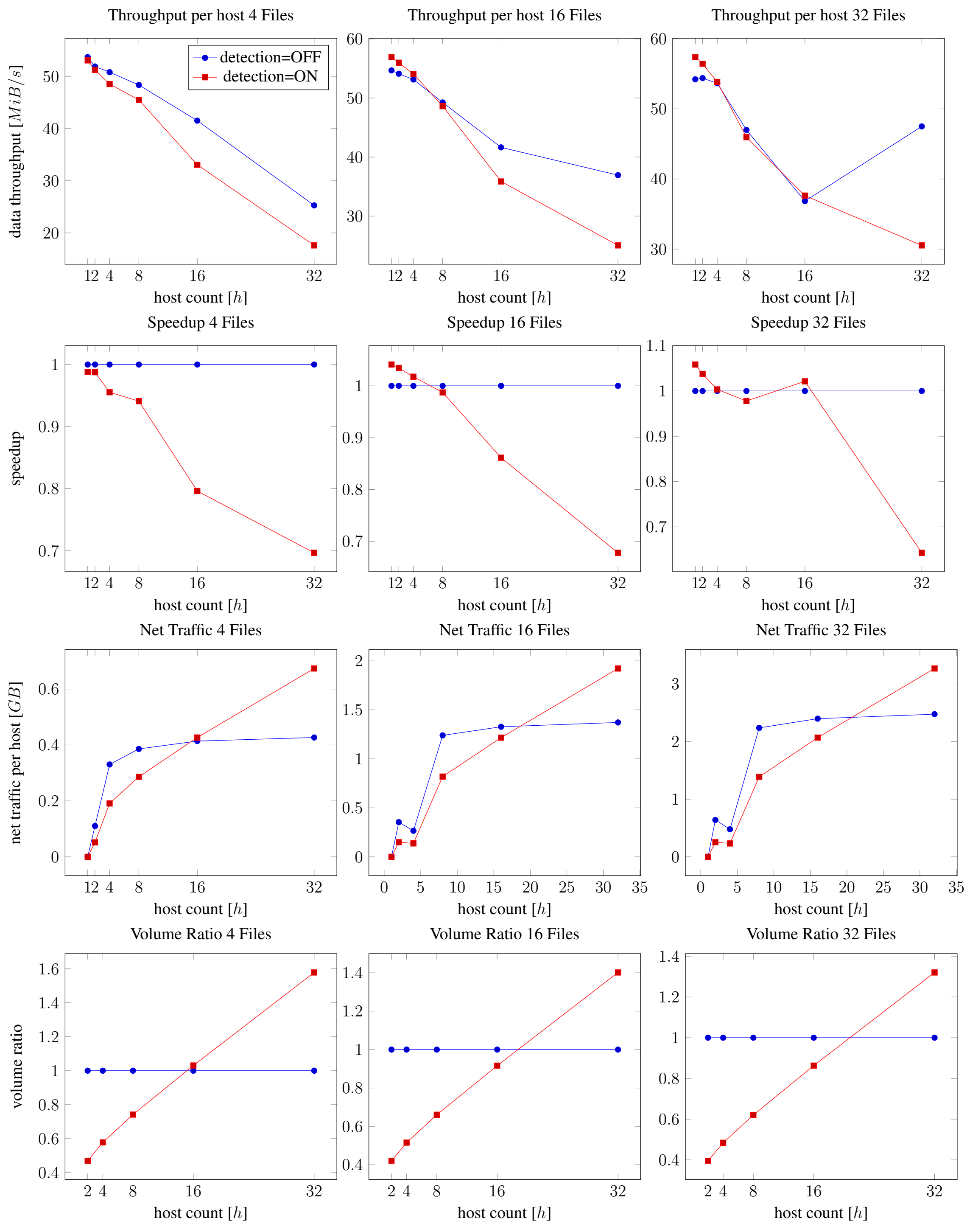

Figure 7.4: Weak Scaling of WordCount algorithm for 4,16 and 32 files per host

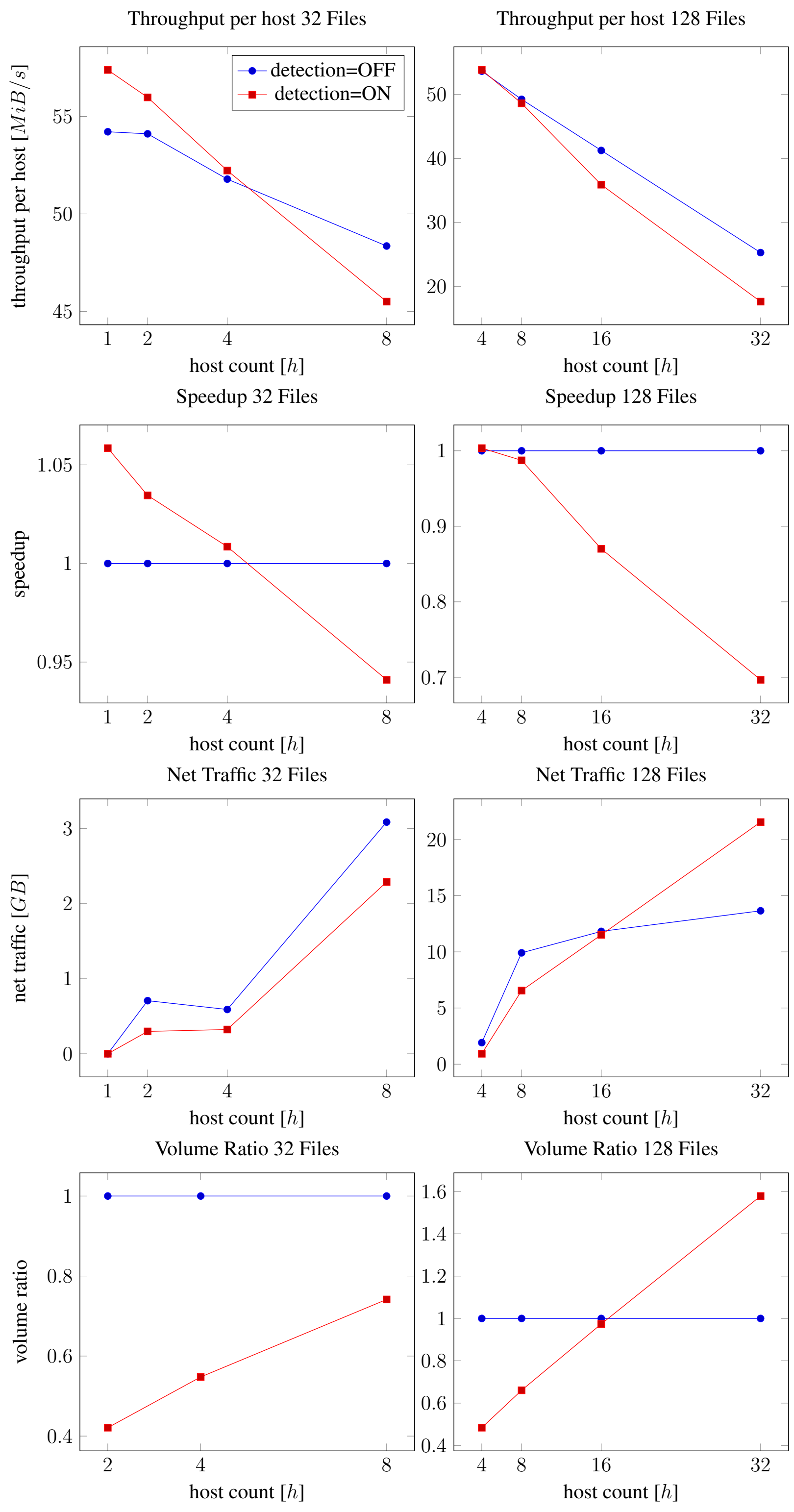

Figure 7.5: Strong Scaling of WordCount algorithm for 32 and 128 files

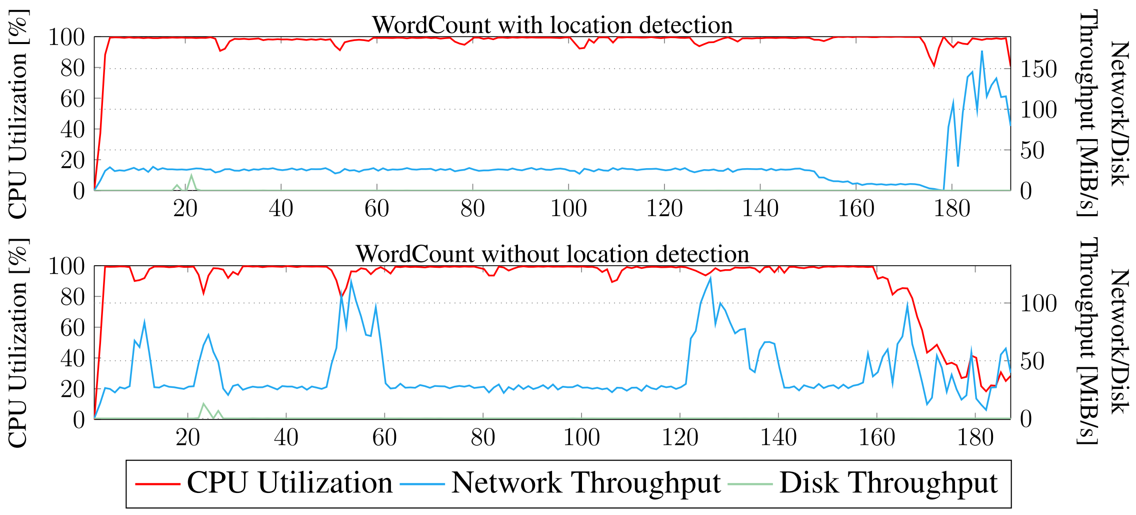

Figure 7.6: CPU, disk and network stats for WordCount algorithm, $h = 4$, files = 100

Figure 7.5 shows the same metrics for strong scaling in the *WordCount* algorithm. Again, we can see that scaling with duplicate detection is worse than without duplicate detection. Again the main reason is the Bloom Filter data structure growing large.
Figure 7.6 shows CPU, network and disk utilization stats for *WordCount* on 4 hosts with 100 files in total. It is notable that overlap of communication and reduction is not possible when duplicate detection is enabled, as the detection has to be performed before data can be shuffled. Therefore, all communication is performed only after the pre-reduction is finished. The total amount of communication is lower than without Duplicate Detection.

The solution for this problem is shrinking the Bloom filter, as the total number of unique hashes is usually way lower than the upper bound, which is the sum of all uniques for each processor.
Figure 7.7 shows throughput and total net traffic for 512 files on 32 hosts, which is a total of 128 workers. The total net traffic is already lower than without Duplicate Detection, when the dSBF size is halved.
The minimum amount of network traffic is reached with a shrinking factor of 8, which is 30% lower than without duplicate detection. Higher shrinking rates increase the FPR by a large margin. Therefore they have higher total net traffic. On shrink factors of 8 and higher, the throughput is less than 5% below the baseline, it does however not overtake the baseline.

7.3.2 Page Rank

In the *PageRank* benchmark, we performed the PageRank algorithm on hyperbolic graphs generated by the NetworKIT [45] suite. Each of these graphs is generated using an average degree of 40, a gamma of 3 and a temperature of 0. The PageRank algorithm used 10

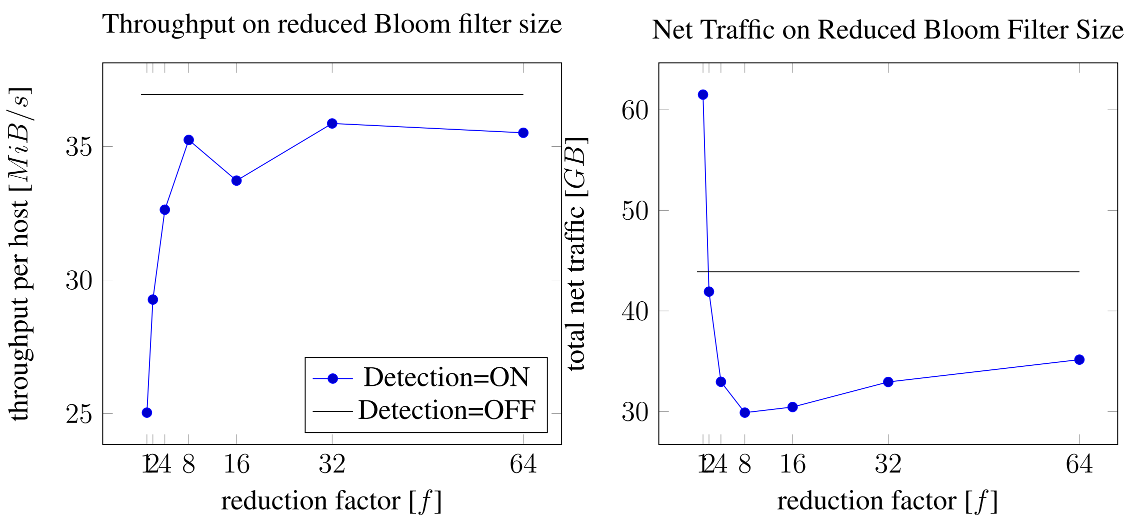

Figure 7.7: Data throughput and network traffic on shrinked Bloom filter

iterations to compute the PageRank of the graph network. Additionally we performed the *PageRank* algorithm on a real world web graph, *uk2002*, which is a part of the 10. DIMACS graph partitioning challenge [5].
Figure 7.8 shows the CPU, network and disk utilization stats for *PageRank* on 4 hosts with 2^{21} pages, both using the *PageRank* implementation with *InnerJoin*. In this graph we can see that in both variants the average CPU utilization is very high. Communication is therefore presumably not the bottleneck of the algorithm.

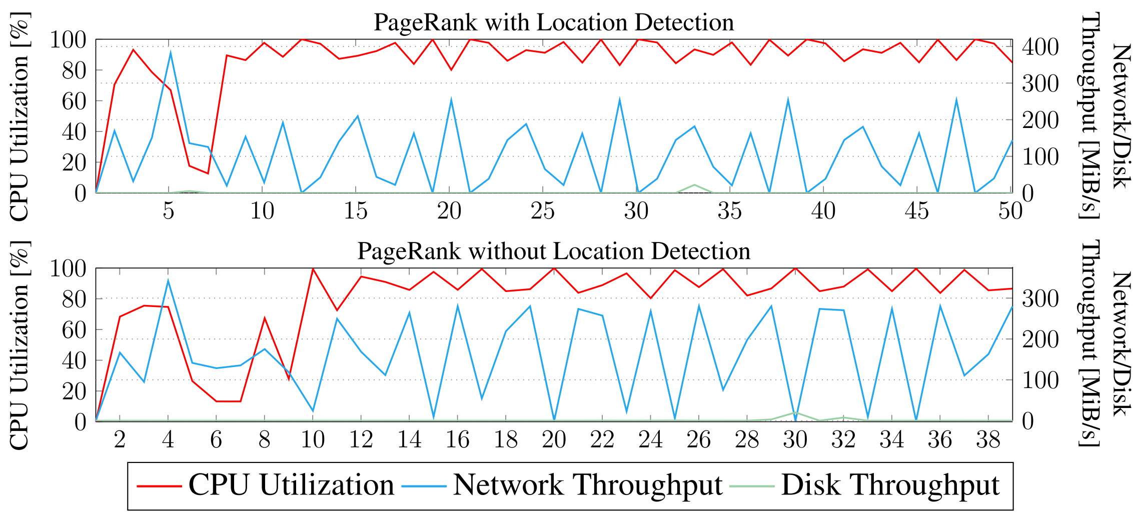

Figure 7.8: CPU, Disk and Network Stats for PageRank Algorithm, $h = 4$, 2^{21} Pages

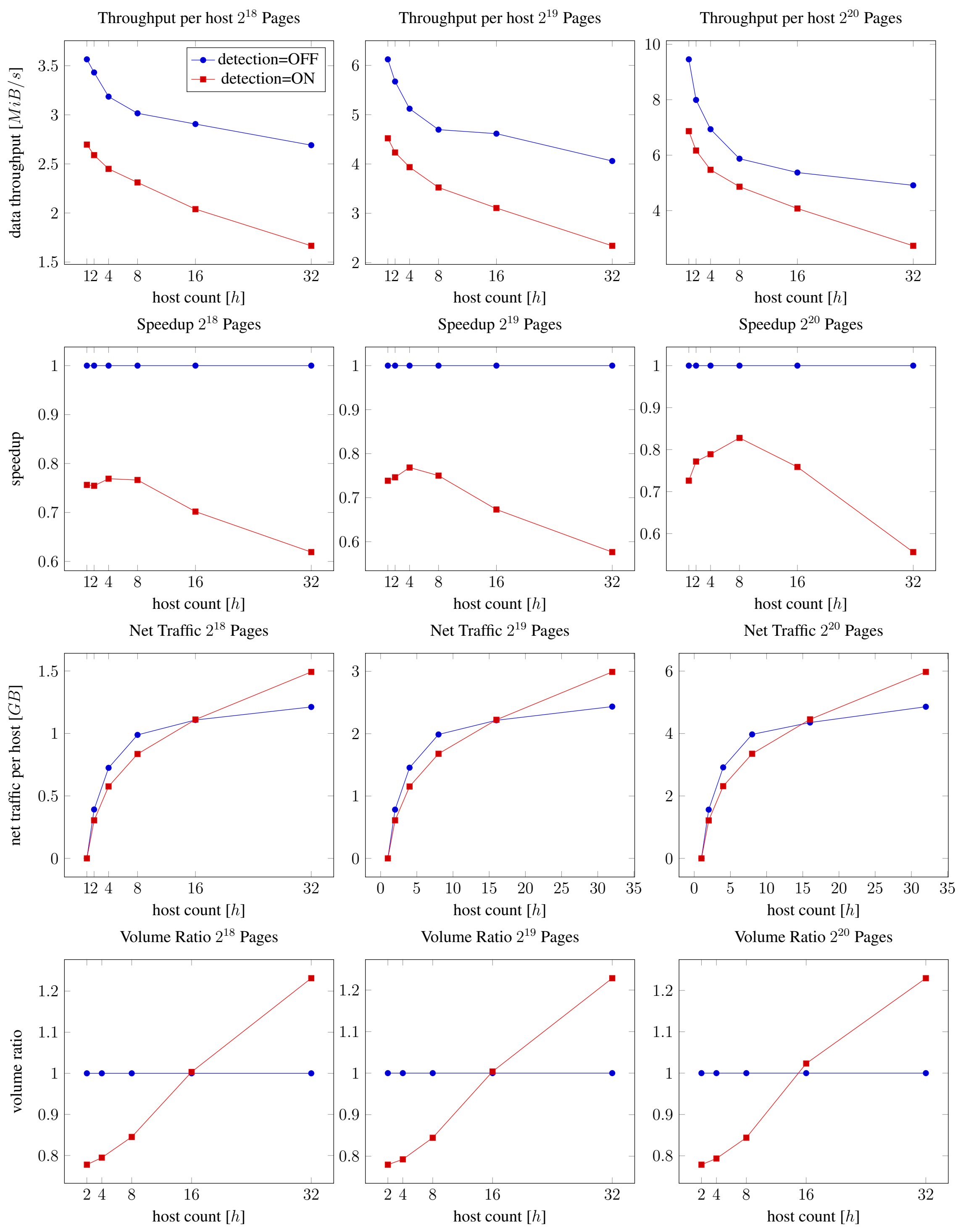

Figure 7.9: Weak Scaling of PageRank algorithm for graphs with 2^{18}, 2^{19} and 2^{20} nodes per host

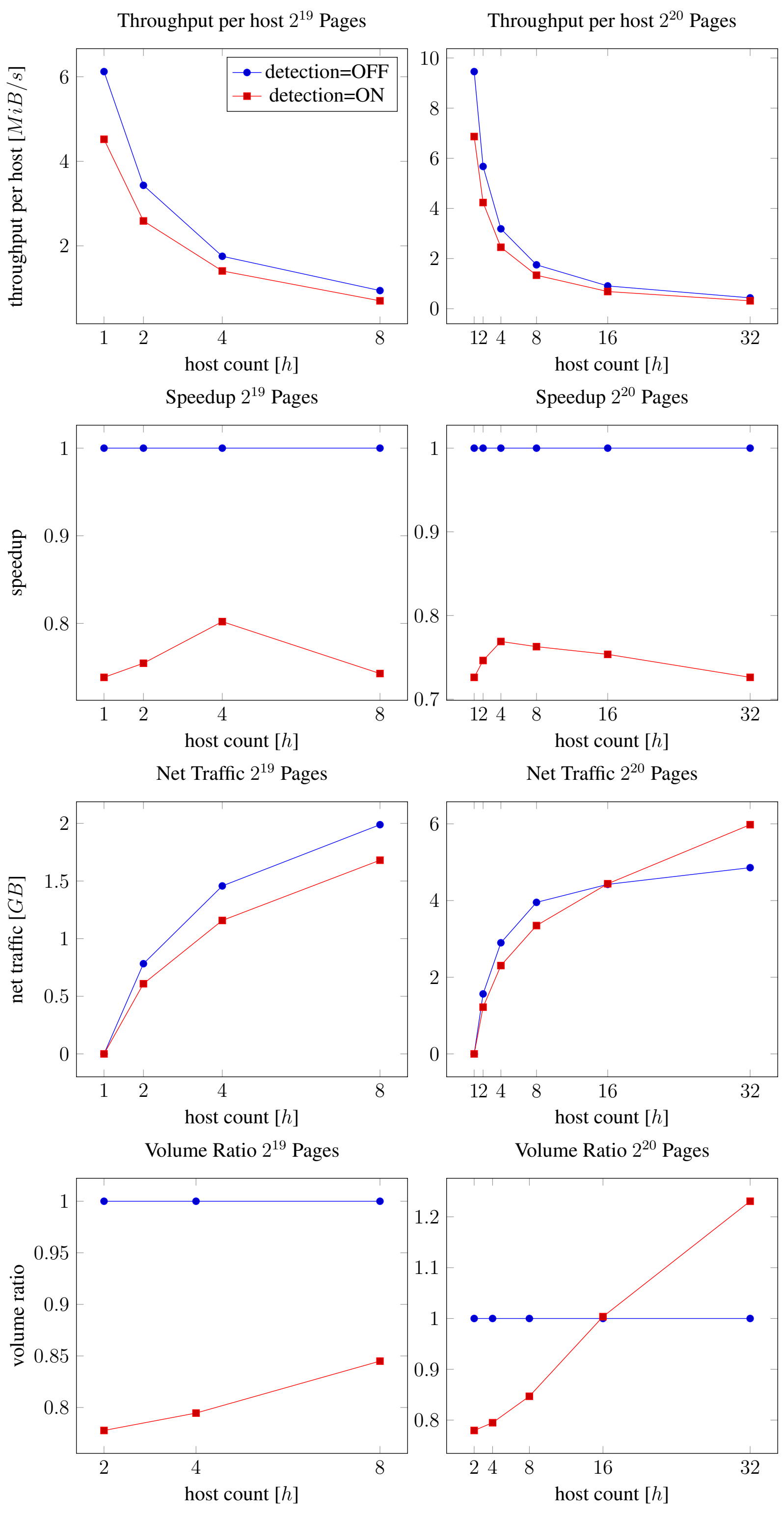

Figure 7.10: Strong Scaling of PageRank algorithm for 2^{19} and 2^{20} nodes

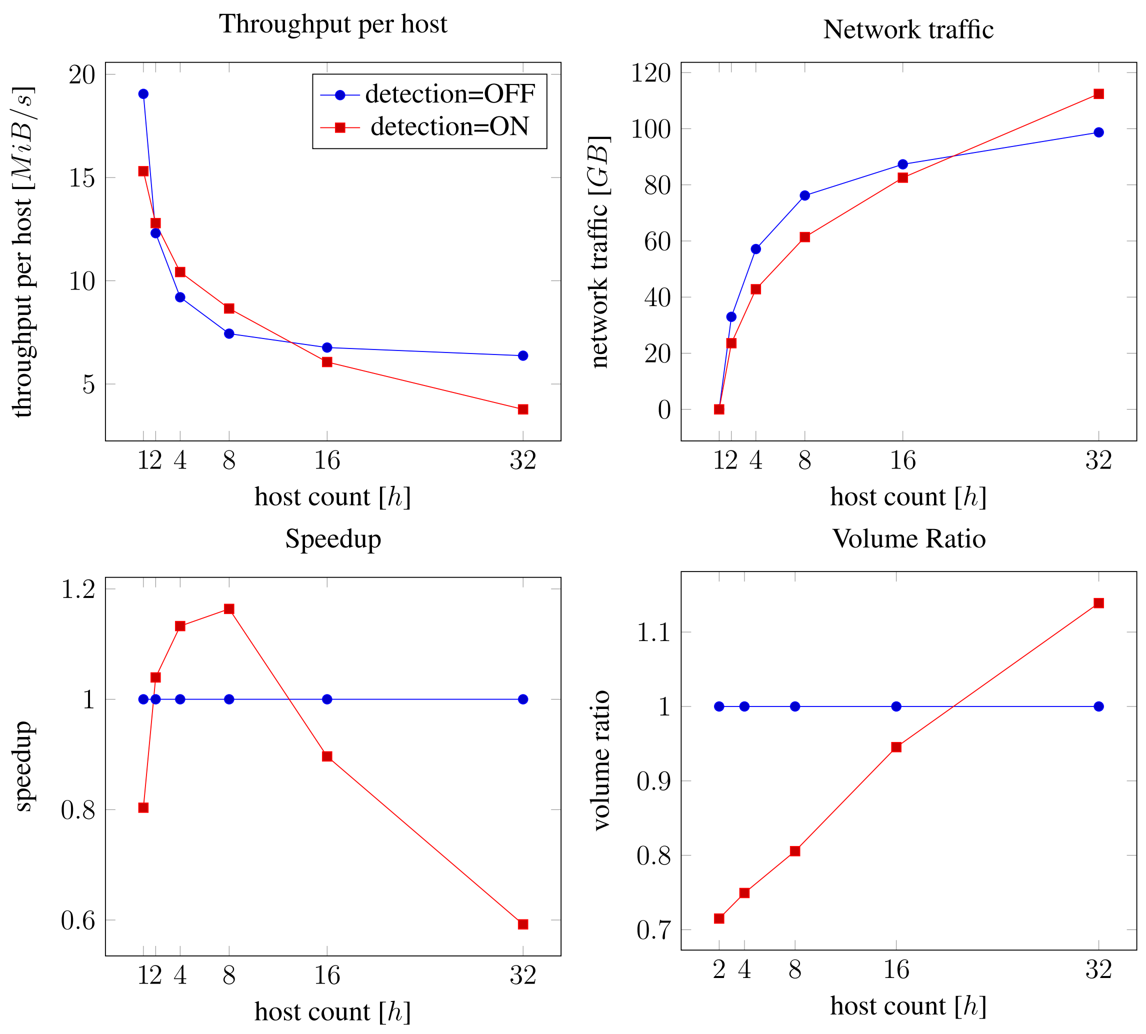

Figure 7.11: Results of PageRank algorithm for *uk2002* web graph

Figure 7.9 shows the weak scaling of PageRank with and without location detection. In every experiment, location detection has no speedup but slows the program down. This can be explained due to the increased computation necessary. As communication does not seem to be the bottleneck, the reduced communication volume does not reduce the running time.
With less than 16 hosts in total, the total communication volume is reduced when using location detection. With 32 hosts, the communication volume ratio rises to 1.25. This happens due to the large upper bound of elements, which is way larger than the actual number of individual keys. Figure 7.10 shows the according strong scaling results for PageRank with 2^{19} and 2^{20} pages in total. The results are similar to the results from the weak scaling experiments.

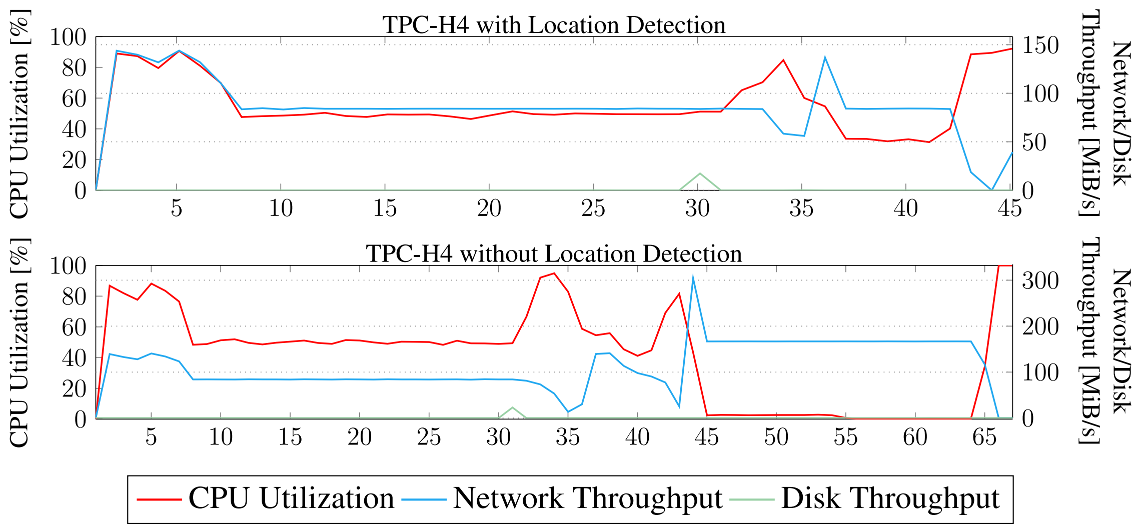

Figure 7.12: CPU, Disk and Network Stats for TPC-H4, $h = 4$, 16 GiB Input Size

Figure 7.11 shows *PageRank* results for the *uk2002* web graph with 1 to 32 hosts. With 2 to 8 hosts, duplicate detection lowers the total running time of the algorithm with a speedup factor of up to 1.16.

7.3.3 TPC-H4 Benchmark

We performed the *TPC-H4* Benchmark on the database generated from the official generator script. In the micro-benchmark, we only measure the running time of the actual join operation. Per GiB of table size, the tables used for the *TPC-H4* benchmark take up 880 MiB.

In Figure 7.12 detailing CPU, network and disk utilization stats, we can see that without location detection we have a large step in which only communication is performed and all CPU cores are idle. This does not happen when this communication volume is reduced.

In Figure 7.13 we can see that the throughput with location detection on is signifcantly higher for each number of hosts benchmarked.
The peak speedup factor is reached at 4 hosts and it is above 10.5 for each workload size. At 32 hosts, the speedup is at 2.2 - 2.7 depending on the workload size.

The main reason for this speedup is that the table elements are large and nearly all elements can be kept locally. Without location detection, table elements are shuffled by hash. Due to the network traffic being the bottleneck for *TPC-H4*, this results in significantly higher throughput for all workload sizes. Strong scaling for the *TPC-H4* micro-benchmark is shown in Figure 7.14. Like in the weak scaling experiment, we see large speedups at 2-8 hosts, which grow smaller as the number of hosts increases. This is due to the increasing size of the dSBF.

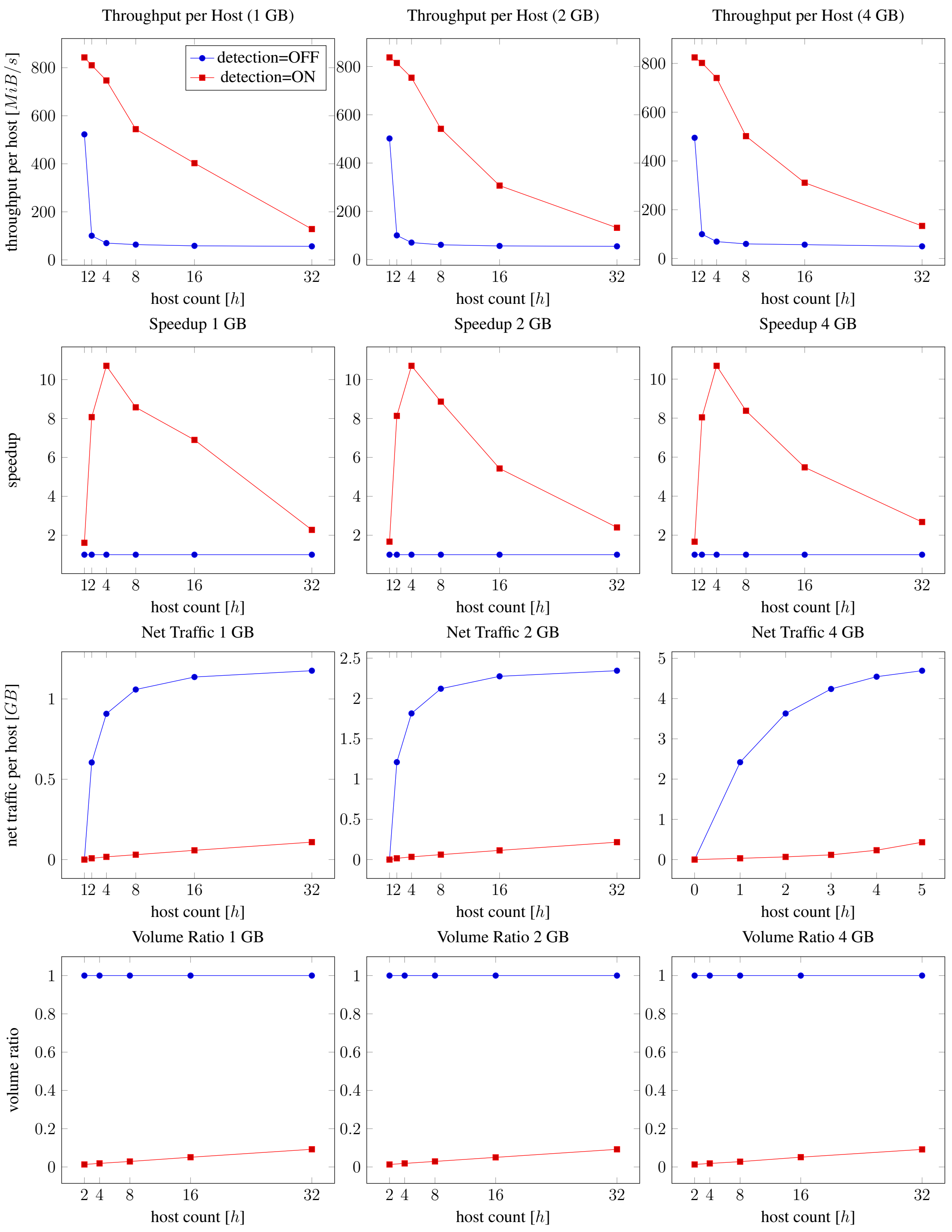

Figure 7.13: Weak Scaling of TPC-H4 algorithm for 1,2 and 4 GiB per host

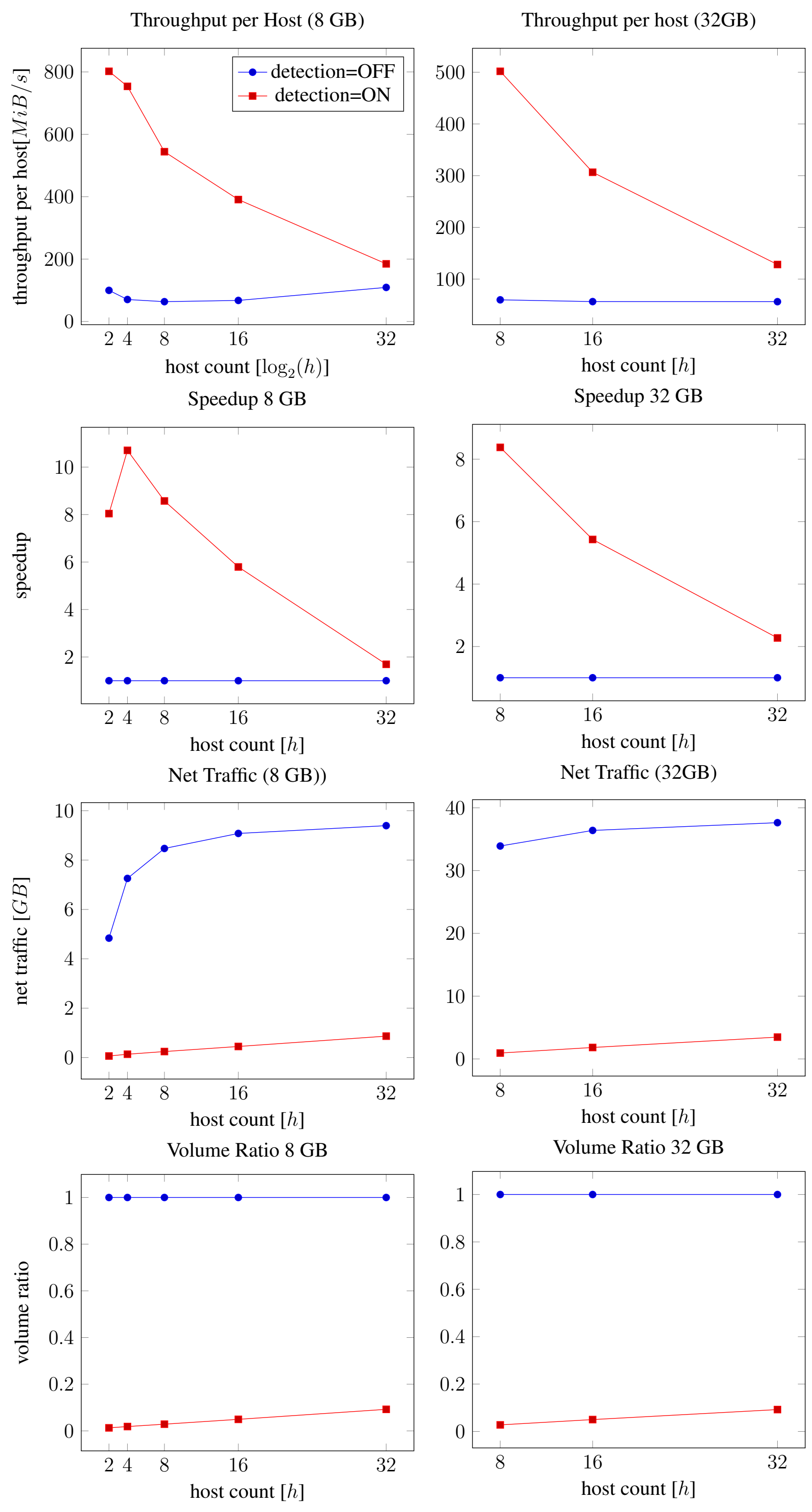

Figure 7.14: Strong Scaling of TPC-H4 micro-benchmark with 8 and 32 GiB

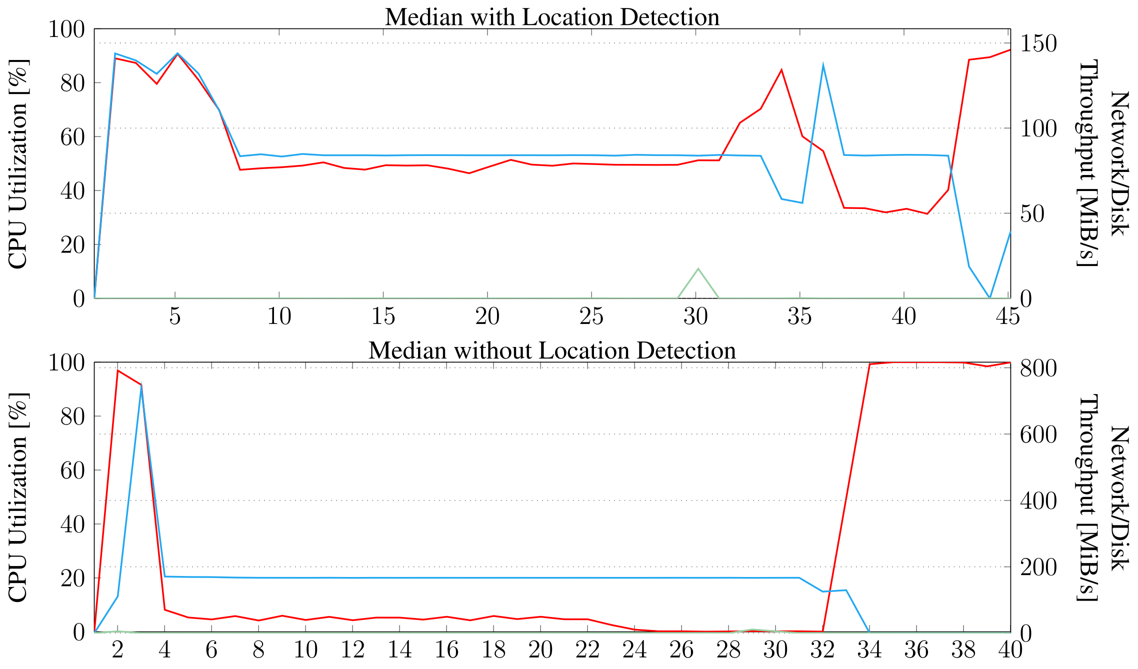

Figure 7.15: CPU, Disk and Network Stats for Median, $h = 4$, 10^9 Elements

7.3.4 Median

In the *Median* micro-benchmark, we group 128 consecutive elements in a DIA and apply a median function on them. We performed weak scaling experiments with a total of 2^{26}, 2^{27} and 2^{28} elements per host as well as strong scaling experiments with 2^{29} and 2^{31} elements in total.
In Figure 7.15 we can see the long communication step in *GroupBy* due to all elements being shuffled over the network. The CPU utilization is only at 100% when we perform the actual grouping in the push phase of the *GroupBy* operation.

The weak scaling results are shown in Figure 7.16, the strong scaling results are shown in Figure 7.17.
The results of the benchmarks are similar to the results of the *TPC-H4* micro-benchmark. On a single host, the speedup factor is at 0.46 - 0.61 depending on input size. The peak speedup factor is 2.15 for 2^{31} elements on 8 hosts. As seen in other benchmarks, the speedup is bad when the amount of elements per host is small and improves on larger amounts of data per host.
The total speedup is lower than in the *TPC-H4* benchmark, as individual elements in the *Median* benchmark are smaller than elements in the *TPC-H4* benchmark. The amount of reduced communication is therefore much lower.

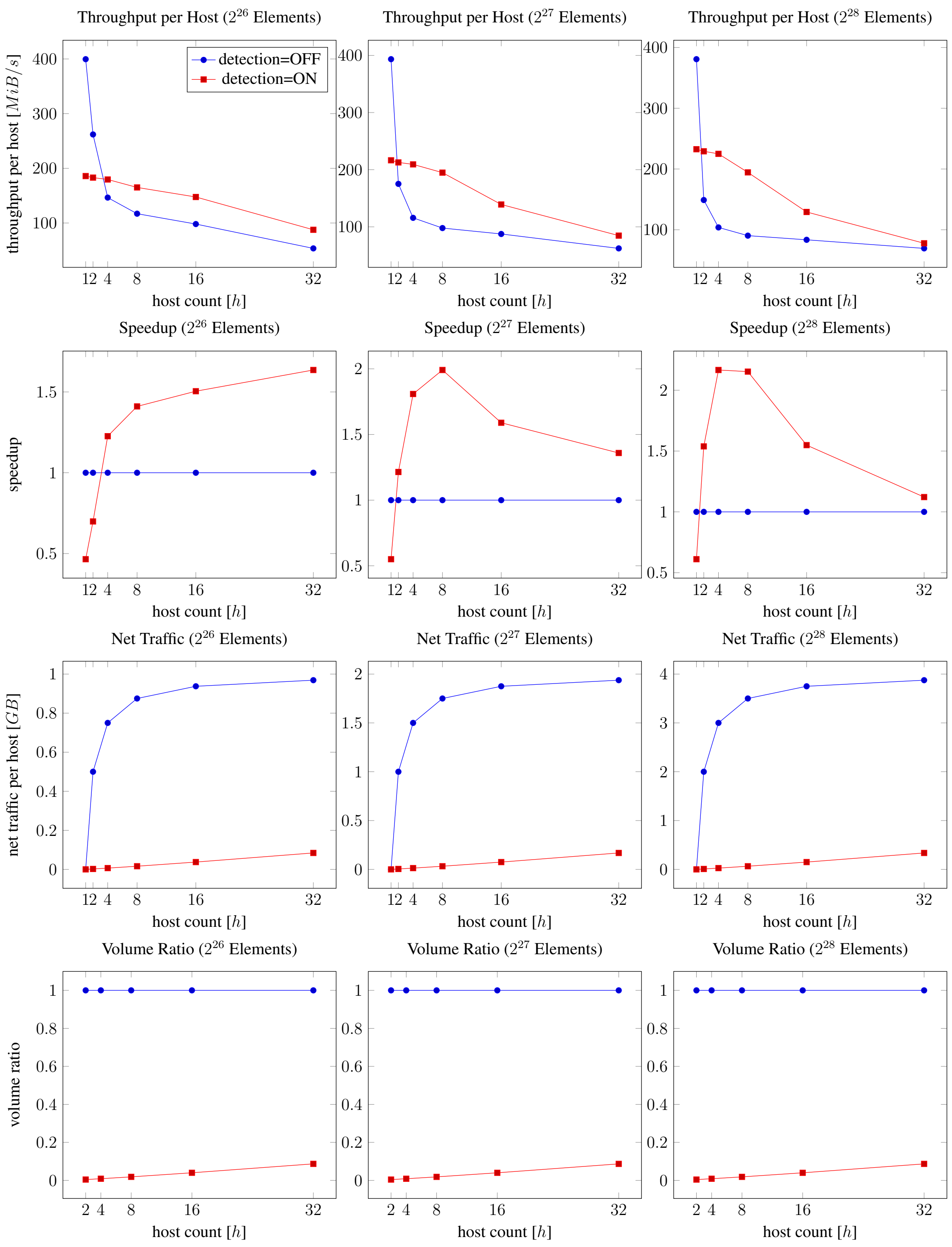

Figure 7.16: Weak Scaling of Median micro-benchmark for 2^{26}, 2^{27} and 2^{28} elements per host

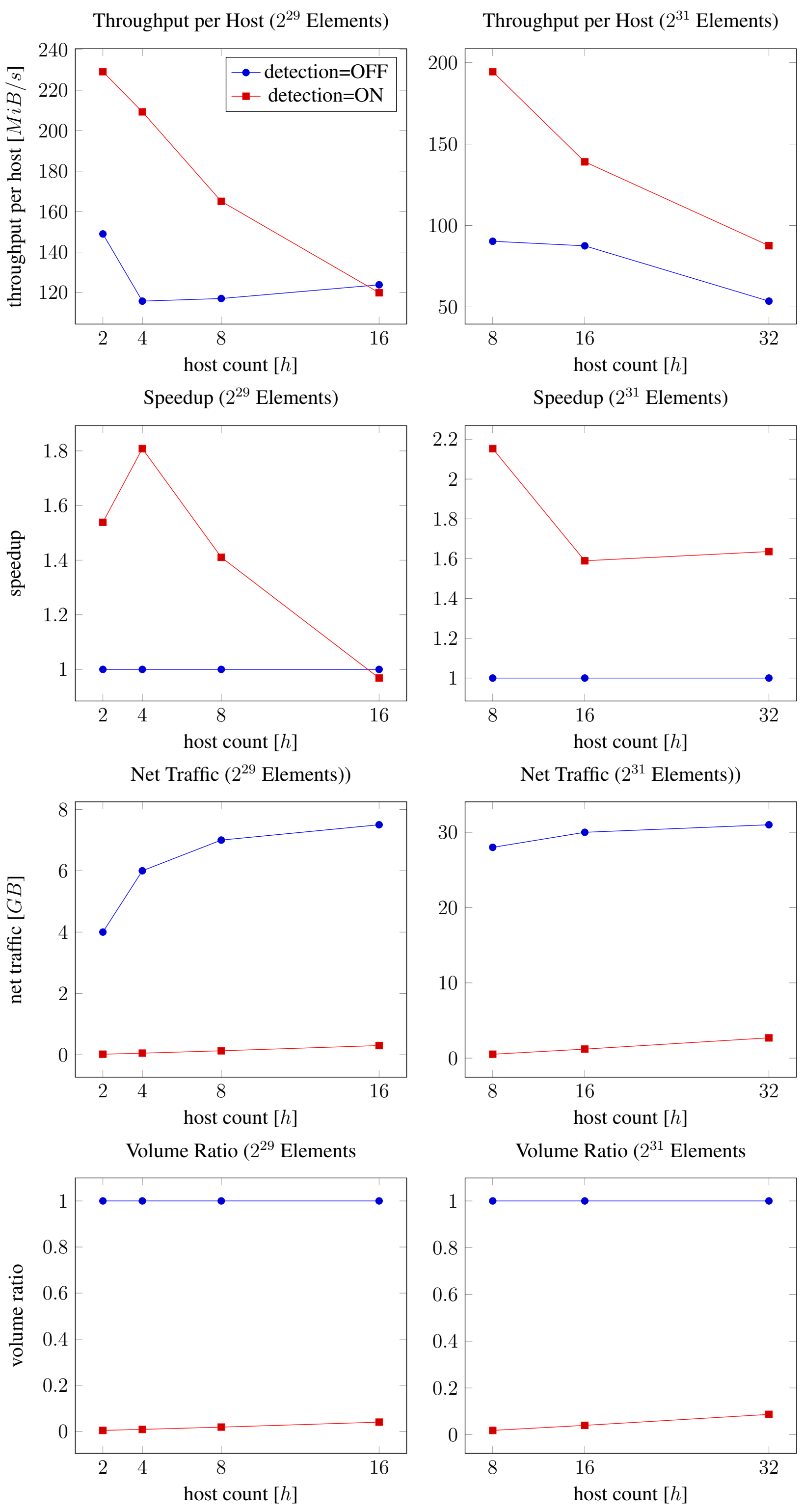

Figure 7.17: Strong Scaling of Median computation for 2^{29} and 2^{31} elements

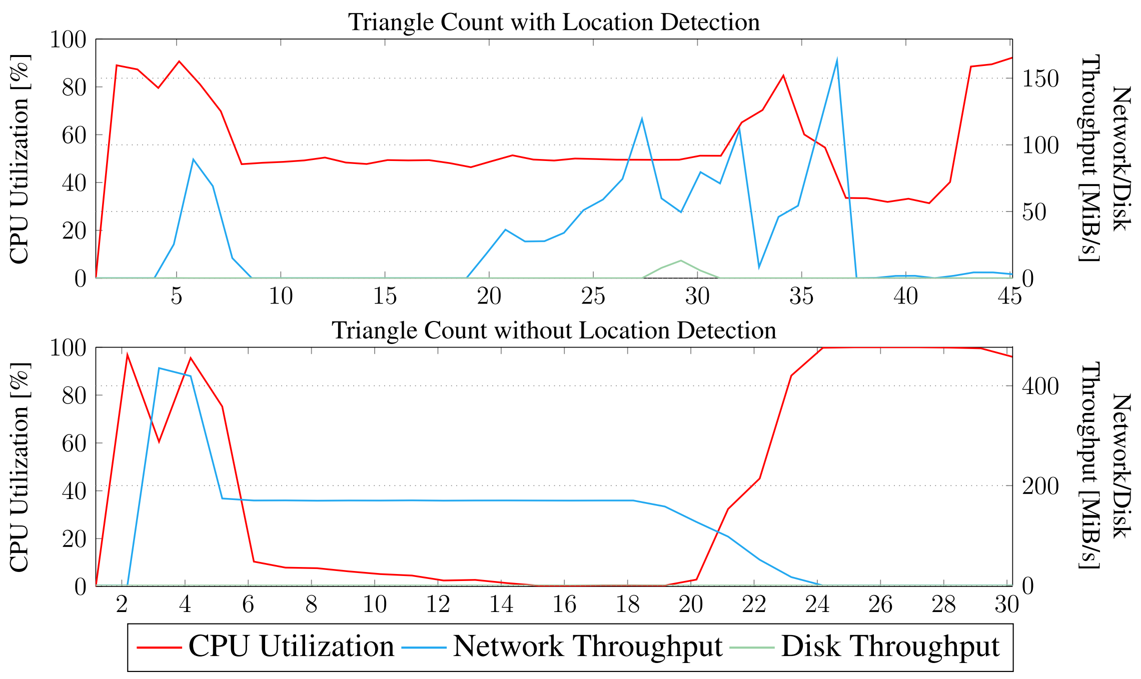

Figure 7.18: CPU, Disk and Network Stats for Triangle Count, $h = 4$, 2^{20} Vertices

7.3.5 Triangle Counting

Triangle Counting is performed on random graphs with an average vertex degree of 40. These graphs were generated using the graph generator in the Thrill framework. We performed weak scaling experiments with 2^{15}, 2^{16} and 2^{17} vertices per host and strong scaling experiments with 2^{19} and 2^{20} vertices in total.
As the number of intermediate edges is very large and every edge usually appears only once, the performance of location detection is very poor in the *Triangle Counting* experiment.
The *Triangle Counting* experiment was mainly conducted to measure the performance of *InnerJoin*, which is tested with a large amount of elements in the DIA storing all edges with length 2.

Figure 7.18 shows CPU, Network and Disk stats for Triangle Counting in a graph with 2^{20} vertices. Without location detection, the shuffle step for the intermediate edges takes up a large portion of the total running time.
In Figures 7.19 and 7.20 we can see that location detection does not work well within the triangle count algorithm.
The algorithm using *InnerJoin* without location detection has good strong scaling results and has reasonable weak scaling of ≈ 0.5 between 1 and 32 hosts.
The total running time with location detection is largely dominated by the large amount of accesses in the table which stores the best targets for elements.

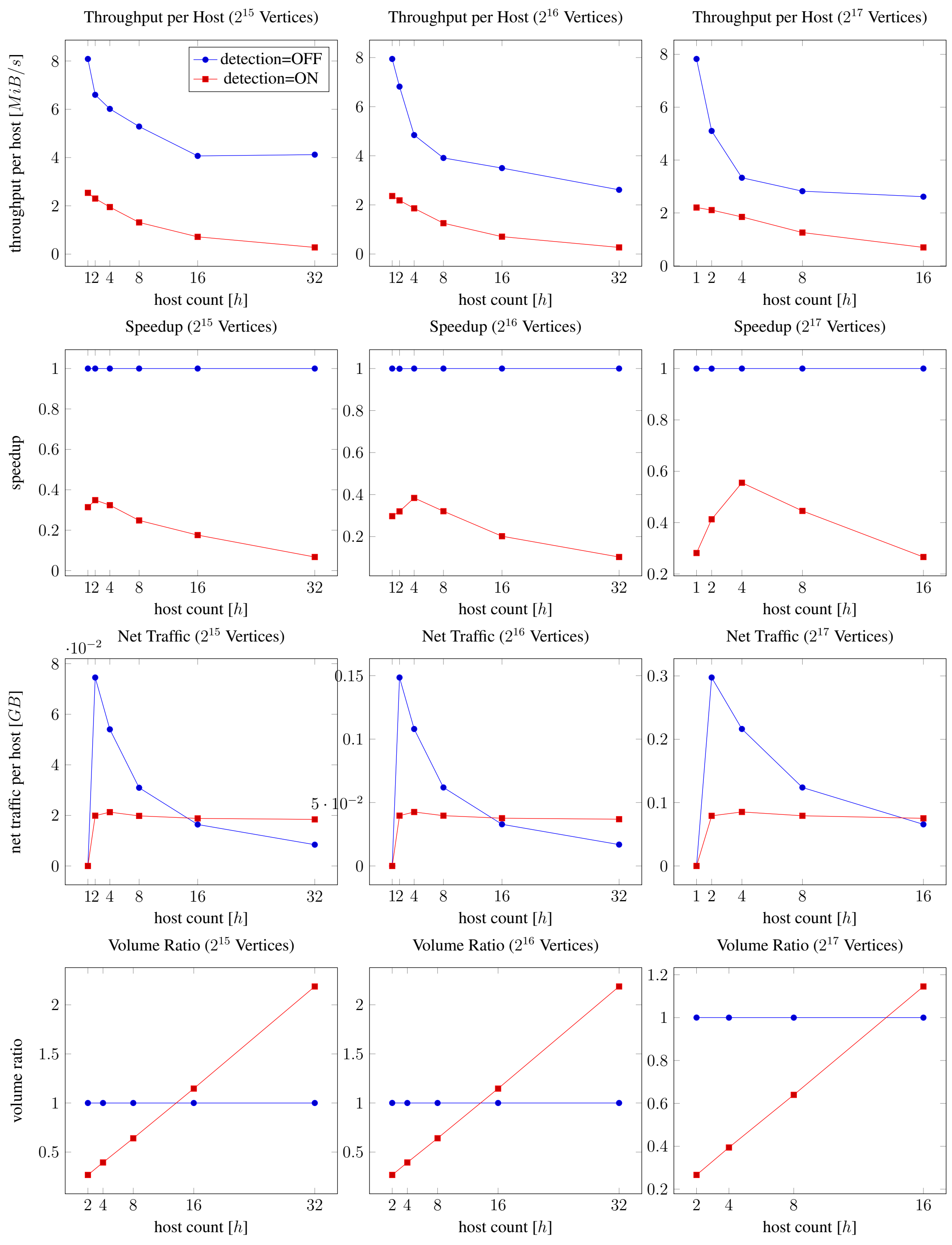

Figure 7.19: Weak Scaling of Triangles micro-benchmark for 2^{15}, 2^{16} and 2^{17} vertices per host

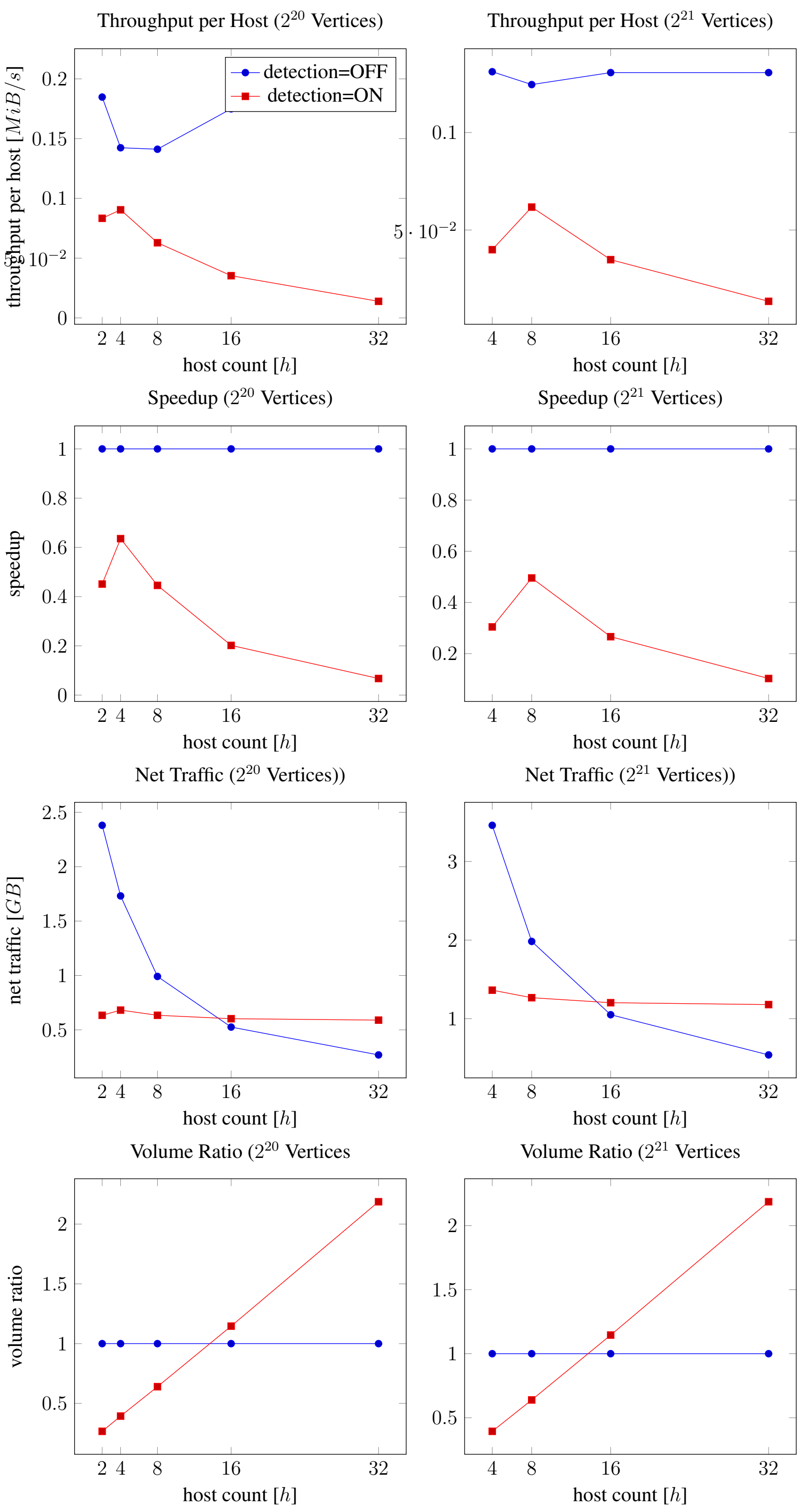

Figure 7.20: Strong Scaling of triangle count algorithm for 2^{20} and 2^{21} vertices

8 Discussion

8.1 Conclusion

In this thesis we implemented the *InnerJoin* operation and duplicate detection using distributed single-shot Bloom filters in the Thrill framework. We measured the performance of these parts in multiple algorithmic micro-benchmarks within the Thrill framework.
We can see that using dSBF can reduce the total network traffic by a large margin in some algorithms while it has no traffic benefits in others. In algorithms which are network bound - such as the TPC-H4 database query benchmark, this can also decrease the total running time of the program.
In algorithms, which are not network bound or do not have a large amount of removable network traffic, the addititonal computation time does not yield a reduced total running time but slows the program down.
It is however nearly always possible to reduce the amount of total network traffic, at least when the upper bound of unique hashes is close to the real amount of unique elements.

8.2 Future Work

On a large number of hosts, the network traffic in the detection step becomes too large. This happens due to the high upper bound of unique hashes in the whole program, which is the sum of all uniques from each worker.
In the evaluation of *WordCount* in Section 7.3.1 we could see that reducing the upper bound and consequently the dSBF size lowers network traffic and running time of the algorithm. The correct shrinkage factor depends on the algorithm and the underlying dataset. It could be beneficial to apply heuristics or sampling to see how many elements appear on multiple workers. As the sweet spot in *WordCount* is relativiely wide, even rough heuristics could be beneficial.
It can also be beneficial to introduce the concept of hosts to the duplicate and location detection, as inner-host communication is significantly faster than intra-host communication. This could additionally lower intra-host network traffic.
The *Triangle Counting* benchmark that the `std::unordered_map` used for the mapping of keys to workers can become the bottleneck. A custom map could be able mitigate this bottleneck. In addition to *InnerJoin*, other join operations such as left outer join, right

outer join or outer join could be implemented similarly. These operations could be useful for several applications.

Bibliography

[1] libs3. `https://github.com/bji/libs3`. Accessed: 2016-11-16.

[2] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 1, 2015.

[3] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

[6] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. *arXiv preprint arXiv:1608.05634*, 2016.

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

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

[9] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[10] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 275:314–347, 2014.

[11] Transaction Processing Performance Council. Transaction processing performance council. *Web Site,* `http://www.tpc.org`, 2005.

[12] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[13] Chris J Date and Hugh Darwen. The sql standard. *SQL/92 mit den Erweiterungen CLI und PSM*, 1993.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *High performance interconnects, 2003. proceedings. 11th symposium on*, pages 44–51. IEEE, 2003.

[16] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[18] Solomon W Golomb. Run-length encoding. *IEEE Trans Info Theory*, pages 399–401, 1966.

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

[20] Shane Grant and Randolph Voorhies. cereal – a c++11 library for serialization. `http://uscilab.github.io/cereal/`. Accessed: 2016-10-24.

[21] Lee L Gremillion. Designing a bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, 1982.

[22] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[23] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[24] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[25] Shahan Khatchadourian, Mariano P Consens, and Jérôme Siméon. Having a chuql at xml on the cloud. In *AMW*. Citeseer, 2011.

[26] Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pages 100–105. ACM, 2012.

[27] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[28] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[29] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S Pérez. Spark versus flink: Understanding performance in big data analytics frameworks. In *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*, 2016.

[30] Robert E Martin. Filter for checking for duplicate entries in database, October 12 2004. US Patent 6,804,667.

[31] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[32] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[33] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

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

[35] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[36] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[38] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[39] Peter Sanders. Fast priority queues for cached memory. In *Workshop on Algorithm Engineering and Experimentation*, pages 316–321. Springer, 1999.

[40] Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *Big Data, 2013 IEEE International Conference on*, pages 15–23. IEEE, 2013.

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

[42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[43] Johannes Singler, Peter Sanders, and Felix Putze. Mcstl: The multi-core standard template library. In *European Conference on Parallel Processing*, pages 682–694. Springer, 2007.

[44] Haoyu Song, Fang Hao, Murali Kodialam, and TV Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *INFOCOM 2009, IEEE*, pages 2518–2526. IEEE, 2009.

[45] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005*, 2014.

[46] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[47] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[48] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[50] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.