Master Thesis

Improving Distributed External Sorting for Big Data in Thrill

Jonas C. Dann

Submission Date: 02.09.2019

Supervisors: Prof. Dr. Peter Sanders
Dr. Timo Bingmann

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

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 02.09.2019

Abstract

Big Data analytics and scientific computing require processing ever growing amounts of data. Competition between companies and between academics applies pressure to analyze data ever faster. In recent years, Big Data processing frameworks like Apache SPARK, Apache FLINK and THRILL came into popularity to aid this development. Such frameworks simplify the algorithm development significantly, by hiding most of the complexities of working with distributed computer clusters and external disks. This thesis focuses on Thrill. One of the most important nontrivial algorithmic primitives in these frameworks is sorting. Sorting accounts for a significant percentage of computer use overall. As a first step to improving sorting in THRILL, this thesis will implement the state of the art distributed external sorting algorithm Canonical Merge Sort. However, Canonical Merge Sort shows weaknesses with certain data sets, since it only uses local knowledge about the data set distribution when redistributing data. Thus, an improvement of the algorithm is proposed and implemented in THRILL, called Online Sample Sort. Experiments along several scale factors and with several data sets are conducted and compared with the original Canonical Merge Sort algorithm. Results show improvements in overall performance.

Kurzzusammenfassung

Big Data Analytics und Scientific Computing verarbeiten immer mehr Daten. Der Wettbewerb unter Firmen und unter Akademikern baut Druck auf, Daten immer schneller zu analysieren. In den letzten Jahren haben deswegen Big Data Frameworks wie Apache SPARK, Apache FLINK und THRILL an Popularität gewonnen. Diese Frameworks vereinfachen das Entwickeln von Algorithmen zur Verarbeitung von großen Datenmengen signifikant, indem sie die Komplexität des verteilten Rechnens und des Arbeitens mit Festplatten verstecken. Diese Thesis wird sich auf THRILL konzentrieren. Eines der wichtigsten nicht trivialen algorithmischen Primitiven in diesen Frameworks ist Sortieren. Sortieren macht einen großen Teil der Computernutzung aus. Als ein erster Schritt, um Sortieren in THRILL zu verbessern, wird der moderne verteilte externe Sortieralgorithmus Canonical Merge Sort in THRILL implementiert. Jedoch zeigt Canonical Merge Sort Schwächen bei bestimmten Datensätzen, da es nur lokales Wissen über die Daten verwendet um sie Umzuverteilen. Deswegen wird eine Verbesserung, genannt Online Sample Sort, des Algorithmus vorgeschlagen und in THRILL implementiert. Darauffolgend werden Experimente mit verschiedenen Datensätzen und Skalierungsfaktoren durchgeführt, die Online Sample Sort mit Canonical Merge Sort vergleichen. Die Ergebnisse zeigen eine Verbesserung der Laufzeit.

Acknowledgements

I would like to thank my masters thesis advisor Dr. Timo Bingmann for the many interesting meetings and helpful input during the period of my masters thesis. I would also like to thank Daniel Ritter who inspired my interest in the scientific aspect of computer science, which drove me to pursue my masters degree and in consequence this masters thesis.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

1.1 Motivation

Big Data analytics and scientific computing require processing ever growing amounts of data. On the one hand, companies analyze business data in realtime in order to gain valuable insights into their operations and their growth potential. Utilizing analytics results faster than competitors is an important competitive advantage. This drives the transition away from slow over night batch processing to low latency and high throughput processing in analytics. On the other hand, scientific computations, especially in the natural sciences, nowadays rely heavily on complex simulations and large amounts of sensor data. This empathizes high-performance computing on very large computer clusters that also has to be accessible by scientists without a computer science degree.

Traditionally, processing large amount of data with low latency was only possible with hard to develop and maintain handcrafted distributed algorithms. In recent years however, multiple frameworks emerged that hide most complexities, while retaining very good performance. Examples include Google MapReduce [6], Apache Spark [30], Apache Flink [1] and Thrill [5]. These frameworks provide abstractions for interaction with external memory, i.e. disks and solid state drives (SSDs), and distributed computation on a cluster of networked computers.

This thesis will work with the high-performance framework Thrill. Compared to the Scala/Java-based frameworks, it provides an efficient `C++` implementation, which allows optimization down to bare metal. Similar to the other frameworks, Thrill provides a collection of algorithmic primitives, called *scalable algorithmic primitives*, that are based on well known operators in functional programming, e.g. `Map`, `ReduceByKey`, `Window` and `Sort`.

One of the most important nontrivial algorithmic primitive for processing large amounts of data, as in business analytics and scientific computing, is sorting. It has many applications in e.g. establishing invariants and building indices [3]. Thus, it is not surprising that sorting accounts for a significant percentage of computer use [17]. Sequential and even distributed sorting are well researched topics in computer science, however sorting in a Big Data distributed data processing framework, i.e. Thrill, has unique challenges that are not attributed to when looking at sorting in a vacuum.

It has two main challenges. First, a generic formulation of the sorting problem that does not permit many assumptions and has to be robust in different environments and for different inputs. This requires the algorithm, implemented in the framework, to perform well up to very large clusters and for all types of data sets. The use case of the algorithm cannot be anticipated beforehand, because of the universal nature of the framework. Second, an implementation in a distributed data processing framework that dictates interfaces and data structures, as well as interaction over the network and with external memory. This attributes to unique architecture choices of the framework, the algorithm is implemented in.

Thrill streams elements from one scalable algorithmic primitive to the next, because the result of a scalable algorithmic primitive cannot be materialized in memory, due to the size of the input data. Currently existing distributed external sorting algorithms often assume input data residing on disk and thus do not take advantage of this.

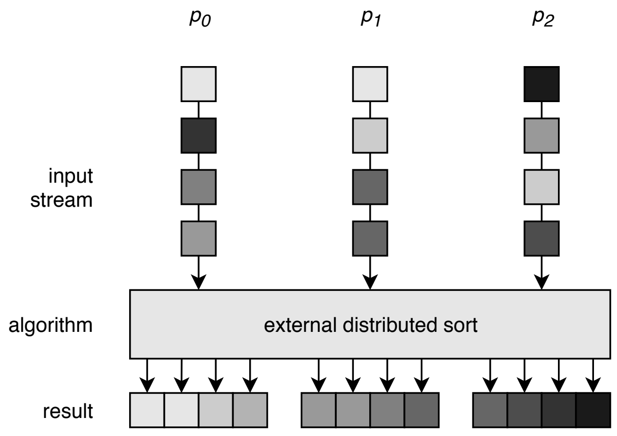


Figure 1: External Distributed Sorting Problem.

Figure 1 shows the distributed external sorting problem, for streaming input data, exemplarily for three processors (p_0 - p_2). Single elements of the input stream are streamed into the algorithm independently on each processor. The result has to be a sorted sequence of all elements that is distributed such that each processor holds the same amount of elements in the end and the largest element of p_i is smaller than the smallest element of p_{i+1}. The goal of this thesis is to design an algorithm that solves this problem better than any existing algorithms.

1.2 Contributions

The goal of this thesis will be developed pursued with the method of algorithm engineering. In contrast to purely theoretical computer science, which analyses algorithms on theoretical models of computers, algorithm engineering combines theoretical algorithm development and software engineering in an iterating process.

Figure 2 shows the algorithm engineering process. It has a falsifiable hypothesis at its core and starts with a theoretical algorithm *design*, which is motivated by real applications. The process continues with the *analysis* of the algorithm design, which yields performance guarantees through deduction. Thereafter, the algorithm design is *implemented* in an algorithm library and *experiments* with real inputs show the impact of the design decisions.

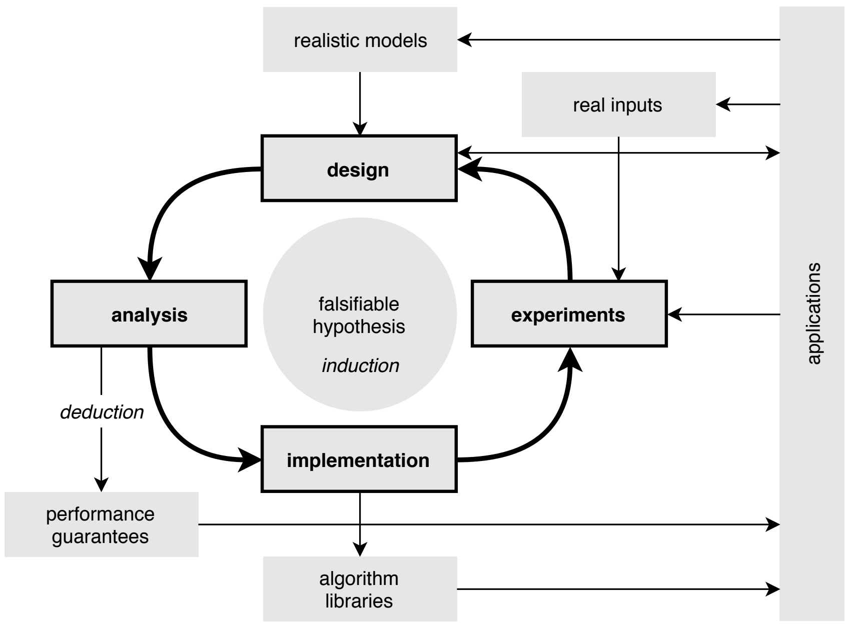


Figure 2: Algorithm Engineering.

The insights gained from the experiments are fed back into the design process and the cycle starts over.

In this thesis, the process of algorithm engineering is applied to distributed external sorting on stream input. The state of the art distributed external sorting algorithm Canonical Merge Sort (CMS) [23] will be used as the starting point of the process. However, CMS was not implemented specifically for stream input. A cycle of algorithm engineering will be applied, resulting in a new algorithm called Online Sample Sort (OSS). OSS relies on a different approach than CMS using samples. Part of the implementation of OSS thereby is finding the best stream sampling technique. This results in the following contributions of this thesis:

(C1) Implementation of CMS in Thrill.

(C2) Design, analysis and implementation of OSS.

(C3) Evaluation of the stream sampling techniques Reservoir Sampling [28] and Online Sampling [18].

(C4) Experiments with CMS and OSS in Thrill.

1.3 Outline

After the introduction (chapter 1), chapter 2 provides the theoretical foundations for this thesis. It starts with a description of key sorting concepts and an overview of the abstractions assumed in Thrill. Furthermore, it provides a detailed look into relevant parts of Thrill and a review of the sorting algorithm previously used in Thrill.

Chapter 3 introduces CMS as a baseline and reference for the proposed new algorithm. Canonical Merge Sort is introduced theoretically with an in-depth look into a very critical subalgorithm. This is followed by implementation details and an analysis of weaknesses of Canonical Merge Sort.

Chapter 4 proposes the new algorithm Online Sample Sort, with the idea and a detailed description of the algorithm. Sampling, a distinction of Online Sample Sort to Canonical Merge Sort, is discussed in more detail. The chapter is ended with a review of related work.

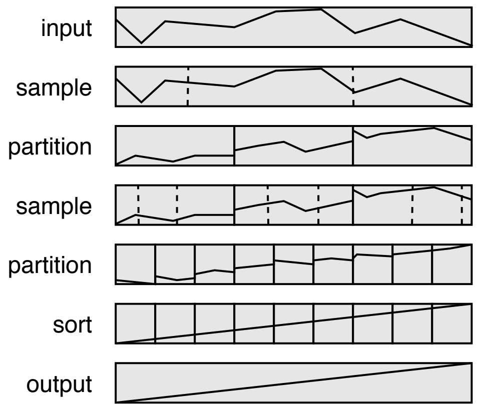
The proposal of Online Sample Sort is followed by a thorough review of sampling and Online Sample Sort compared to Canonical Merge Sort and the previously used sorting algorithm in chapter 5. Previous to the experiments themselves, the systems and input data generators are presented.

The thesis is concluded with a discussion of the results and an outlook on further development opportunities (chapter 6).

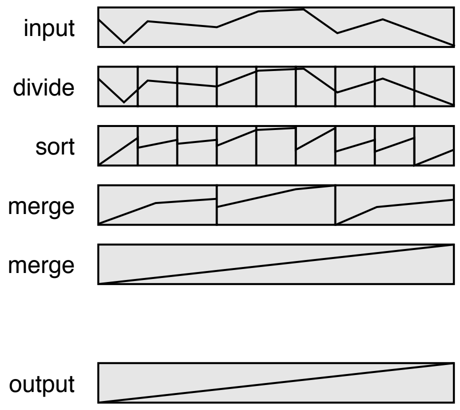
2 Preliminaries

2.1 Sorting

This section introduces fundamental ideas about sorting. Sorting is permuting an input sequence of elements $S_{\text{input}} = \langle e_0, e_1, ..., e_n \rangle$ into an output sequence S_{output}, such that $e_i < e_j, \forall e_i, e_j \in S_{\text{output}} : i < j$ under a total order $\leq$. Despite its simple definition, sorting is one of the most thoroughly researched field of algorithmics. It is integral to many higher level algorithms and applications. The proven lower bound of sorting is $O(n \log n)$. Over time, two main paradigms for sorting in $O(n \log n)$ time emerged, based on the principle of divide and conquer, Sample Sort and Merge Sort. The sections 2.1.1 and 2.1.2 explain the two paradigms.



(a) Sample Sort



(b) Merge Sort

Figure 3: Sorting Paradigms

2.1.1 Sample Sort

Sample Sort, also called Distribution Sort [17], is based around the idea of partitioning the unsorted input sequence into subsequences recursively, until the sequences are small enough to be sorted by a highly optimized sorting algorithm for small sequences. The partitioning is based on r samples drawn from each of the current sequences. The approach is depicted in figure 3a. A sampling step, where a random sample $S = \langle S_0, S_1, ..., S_r \rangle$ (depicted by dashed lines) is drawn from each of the current sequences (sequences are separated by solid lines), is followed by a partitioning step, which partitions each of the current sequences into $r + 1$ smaller sequences, such that the elements in sequence i are bigger then S_{i-1} and smaller or equal to S_i. One specialization of Sample Sort is the more well known Quicksort [15], which uses just one splitter, also called pivot.

Algorithm 1 shows high level pseudo code for the Sample Sort algorithm, which also introduces oversampling [24]. Oversampling improves the quality of the samples. Input

Algorithm 1: Sample Sort.

```
Input: Sequence $E$, Oversampling Factor $k$, Splitter Count $r$
1 SampleSort($E = \langle e_0, e_1, ..., e_{n-1}\rangle, k, r$) begin
2     if $n/k < t$ then return SmallSort($E$) ;    // Average bucket size below threshold.
3     let $S = \langle S_0, S_1, ..., S_{k(r+1)}\rangle$ denote a random sample of $E$
4     sort $S$
5     $\langle s_0, s_1, ..., s_r\rangle \leftarrow \langle -\infty, S_k, S_{2k}, ..., S_{rk}, \infty\rangle$    // Select splitters.
6     foreach $e \in S$ do
7         find $j \in [0..r] : s_j < e \leq s_{j+1}$
8         place $e$ in bucket $B_j$
9     return Concat(SampleSort($B_0, k, r$), SampleSort($B_1, k, r$), ...,
      SampleSort($B_r, k, r$))
Output: Sorted Sequence
```

parameters for the algorithm are the sequence itself, an oversampling factor and the splitter count. This advanced version of Sample Sort does not use the sample directly, but uses splitters drawn from the sample to partition sequences. The oversampling factor defines how many samples are taken from the sequence for each splitter. The splitter count defines how much the sequence is split up in every step. Both parameters can be tuned for specific use cases.

The algorithm starts in each step by pulling a random sample S of size $k(r+1)$ from the input sequence and sorting the sample S (lines 3 and 4). Thereafter, r evenly distributed spiltters are chosen from the sample (line 5). The goal of this first step is to choose splitters that will divide the sequence into evenly sized subsequences. The lower bound for recursion depth $\log_p n$ is only reached when the current sequences are partitioned evenly in each recursion step. Thus, oversampling is used to control the robustness of the algorithm against non representative samples that will inevitably happen with random samples. More oversampling results in splitters that divide the sequence more evenly, but also more work for sorting the sample.

In a second step, elements are partitioned into buckets $\langle B_0, B_1, ..., B_r\rangle$ (line 6-8). Each element is compared against the splitters, which denote the boundaries of the buckets and placed into the respective bucket. After partitioning is finished and each element was placed into its bucket, all elements in bucket B_i precede all elements in bucket B_{i+1}. Finding the buckets can be accelerated by using a binary search tree on the splitters. In a last step, sample sort is called recursively with the buckets as input sequences (line 9). The recursion terminates, when the average bucket size falls below a threshold t (line 2). In this case, another sorting algorithm (e.g. Quicksort), tuned for small inputs, is used. The concatenation of the subsequences is the sorted output sequence.

2.1.2 Merge Sort

Merge sort [17], depicted in figure 3b, first divides the input sequence into m small sequences (separated by solid lines) that can, as in the end of sample sort, be easily sorted by an algorithm optimized for small sequences. Thereafter, the sequences are k-way merged in multiple steps. Meaning that in each step every k sequences are merged into one, which results in a recursion depth of $\log_k m$. Figure 3 shows Sample Sort and Merge Sort next to each other, which illustrates that they are orthogonal approaches to the problem of sorting [29].

Algorithm 2: Merge Sort.

```
Input: Sequence $E$, Max Small Sort Size $t$, Merge Degree $k$
1 MergeSort($E = \langle e_0, e_1, ..., e_{n-1} \rangle$) begin
2     $m = n/t$
3     let $M_i = \langle e_{0+im}, e_{1+im}, ..., e_{(m-1)+im} \rangle : \forall i \in [0, m)$ denote $E$ split into $m$ equally sized
      parts
4     foreach $i \in [0, m)$ do
5         SmallSort($M_i$)
6     return Merge($k, M_0, M_1, ..., M_{m-1}$)
Output: Sorted Sequence
```

Algorithm 2 shows high level pseudo code of a possible implementation of the Merge Sort algorithm. Similar to Sample Sort, Merge Sort gets the sequence of elements E as input, as well as the maximum number of elements that `SmallSort` is optimal for t and the merge degree k. The size t may be limited by main memory. The merge degree k has to be chosen as big as possible, which may be limited by main memory too.

The algorithm starts by determining the number of small sequences m, the input sequence should be divided into (line 2). This depends on the input size n and t. Thereafter, the input sequence is divided into m equally sized partial sequences M_i (line 3). This is done by splitting the input sequence at positions $im : \forall i \in [1, m)$. The partial sequences are then each sorted with the `SmallSort` algorithm (line 4 and 5).

The algorithm finishes by merging all partial sequences recursively with merge degree k (line 6). In each round of merging, every k partial sequences are merged into one. Merging is done by moving the currently smallest element in the k partial sequences to the current end of the resulting sequence until all k sequences are empty. Merging has a lower bound of $O(n \log k)$. The currently smallest element of a sequence is always the first remaining element in the sequence and the resulting sequence is also sorted. The resulting sequences are used as input for the next round of merging until there is only one sorted sequence left. This last sequence is returned as the sorted output sequence.

Sample Sort, despite both paradigms having theoretically optimal runtime, typically has lower constant factors on real systems [29]. Analysis of Sample Sort and Merge Sort algorithms in hierarchical memory scenarios also showed that the Sample Sort approach makes better use of lower level caches [21].

2.2 Abstractions

The thesis is based on two abstractions that are assumed in Thrill. One is the bulk-synchronous parallel external memory (EM-BSP) model [7], a combination of a parallel computation model and an external memory model. Such a model eases the description and reasoning about distributed external algorithms implemented in Thrill. Message passing is introduced as the second abstraction for inter processor communication in the EM-BSP model. In message passing, processors communicate by sending messages to each other over a shared medium. Both abstractions also aid portability of the developed algorithms.

2.2.1 Bulk-Synchronous Parallel External Memory Model

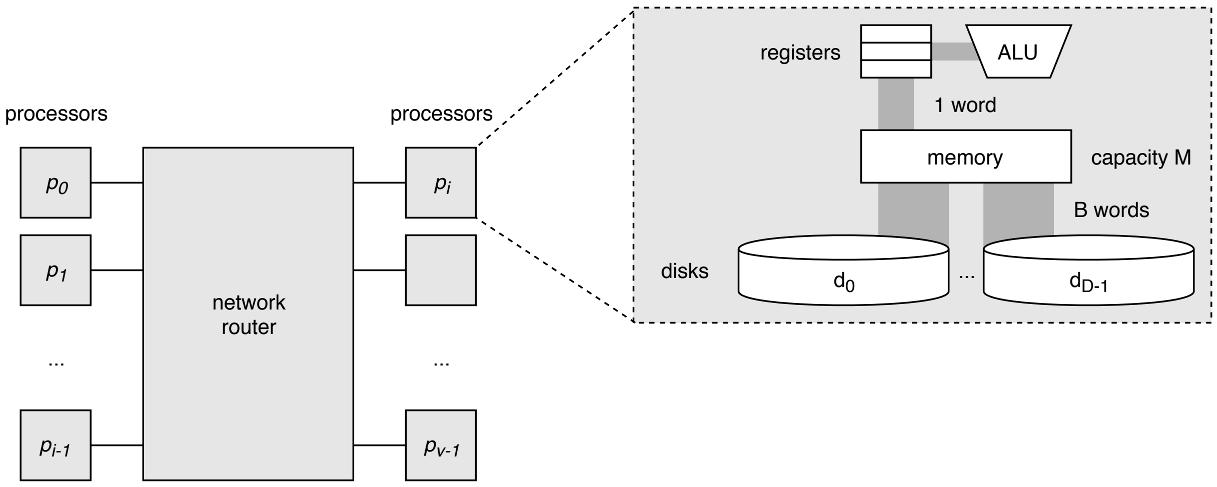


Figure 4: Bulk-Synchronous Parallel External Memory Model (adapted from [8]).

The EM-BSP model extends the bulk-synchronous parallel (BSP) model [25] with hierarchical memory similar to the parallel disk model (PDM) [27]. The BSP model assumes an architecture of multiple processors which are somehow interconnected. It describes parallel computations as iterations of supersteps of parallel, but independent, *computations of subproblems* on all processors followed by *communication* and *synchronization*. The BSP model uses the parameters problem size N, number of processors v, ratio between computation and bandwidth for communication g and time synchronizations take L. The BSP model measures the cost of an algorithm as the sum of time for computation, communication and synchronization. Thus, the BSP eases the description of and reasoning about parallel algorithms, by abstracting complex underlying systems into four parameters and a structure of three supersteps.

The PDM describes interaction of a processor with a set of parallel disks via internal main memory. It uses the parameters problem size N, size of internal main memory M, block size that is transferred between main memory and one disk B and number of disks D. The PDM measures the cost of an algorithm in I/O operations. Both models are combined into the EM-BSP model.

Figure 4 shows the architecture underlying the EM-BSP model. There are v processors $p_0, p_1, ..., p_{v-1}$ connected by a network router, which allows one-to-one communication between all processor pairs. This abstracts details about the network and thus hides more complex topologies. Each processor is a central processing unit (CPU), connected to a main memory and multiple disks. Neither memory nor disks are shared between processors. All communication occurs via network. As in the PDM, internal main memory of every processor is of size M, the bandwidth between main memory and each disk is B and the number of disks is D.

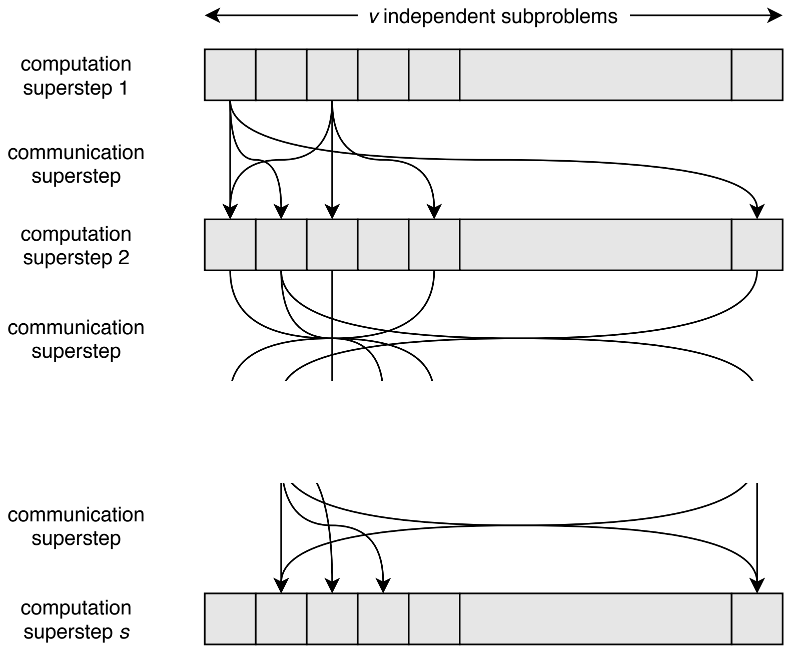


Figure 5: Processing in the EM-BSP Model (adapted from [8]).

Figure 5 shows a combination of supersteps that make up an algorithm implemented in the EM-BSP model. As with the original BSP model, the processors process independent subproblems of the algorithm. Between computation supersteps, data is exchanged between the processors in communication supersteps. Synchronization takes place between computation and communication supersteps.

The extension to hierarchical memory results in a new set of parameters for the model. Besides the combined parameters of the BSP model and the PDM, a parameter G is introduced, which is the ratio between computation and disk bandwidth. This parameter corresponds with the ratio between computation and network bandwidth g. A list of the EM-BSP parameters follows:

- N is the problem size
- v is the number of processors
- g is the ratio between computation and bandwidth of the router
- L is the time that synchronizations take
- M is the size of local main memory
- D is the number of disks
- B is the block size as bandwidth between main memory and a disk
- G is the ratio of local computation and local I/O bandwidth

The cost analysis for the EM-BSP model adds the component of time for I/O to the BSP models cost analysis. This results in the formula $t_{\text{comp}} + t_{\text{comm}} + t_{\text{I/O}} + L$ as the cost of an algorithm implemented in the EM-BSP model, where t_{comp} is the time for computation, t_{comm} is the time for communication, $t_{\text{I/O}}$ is the time for I/O operations and L is the time for synchronizations. Time is measured in computation cycles rather than seconds. The optimal runtimes are thus only achievable when all partial times are as low as possible, which requires optimal utilization of the CPU, network and parallel disks.

2.2.2 Message Passing

Message passing models communication between processors exclusively by sending data encapsulated in messages between processors. It is a defining characteristic of message passing that sender and receiver both have to perform operations for communication to happen. Message passing stands in contrast to function calling, where a function is directly addresses by name. Non the less, message passing is used to invoke certain behavior in the receiver processor. Message passing is universally applicable, expressive and performs well, because of low overhead and portability.

Message passing in Thrill is implemented with the message passing interface (MPI), a portable and efficient message passing library and specification. MPI provides a lot of functionality, but can be broken down to six functions that form the core capabilities. Other functions add flexibility, robustness and convenience, but message passing can be implemented with these six functions alone. Examples of other powerful and important functions are `MPI_Bcast` for broadcasting to all processors and `MPI_Reduce` for a reduce operation on all processors [14].

Table 1 shows the names and short descriptions of the six core functions. `MPI_Init` has to be the first function call to MPI in every program that uses MPI. The function has

Function Name	Short Description
`MPI_Init`	Initialize MPI.
`MPI_Comm_size`	Find out how many processes there are.
`MPI_Comm_rank`	Find out which process I am.
`MPI_Send`	Send a message.
`MPI_Recv`	Receive a message.
`MPI_Finalize`	Terminate MPI.

Table 1: Six Core Message Passing Interface Functions.

to be exactly called once on every processor involved in the execution of the program. `MPI_Init` sets up the MPI environment. Thereafter, all the other MPI functions can be used. `MPI_Comm_size` and `MPI_Comm_rank` return the number of processors in the current MPI environment and the rank of the current processor respectively. These are very important parameters in every parallel program, because they control the execution and distribution of work to the processors. The number of processors gives the amount of resources that can be used to execute the program and the rank can be used to determine how to split up work in the program.

`MPI_Send` is used to send messages to another processor. The receiving processor has to call `MPI_Recv` in order to receive the message. `MPI_Recv` is a blocking call that only returns when the message was received. An MPI message is made up of an address to the data in memory, a count as the number of elements at the memory position that are part of the message and a datatype that describes the the data exchanged between the two processors. Upon receiving, a memory position, count and data type have to be specified, where the message data is stored upon reception.

Finally in every MPI program, `MPI_Finalize` has to be called to terminate the MPI environment. `MPI_Init` cannot be called after finalization. How to run an MPI program on a set of processors is not part of the MPI specification and is left open to the user.

2.3 Thrill

This section introduces Thrill [5], the general purpose big data processing framework developed by Bingmann et al. Thrill is a dataflow framework similar to Google MapReduce [6], Apache Spark [30] or Apache Flink [1]. A dataflow framework provides an application programming interface (API) for processing massive amounts of data, by applying scalable algorithmic primitives to the data one after another. It gets its name from the data flowing from one operation to the next. The dataflow paradigm is similar to a lot of functional programming languages [20]. The framework hides such things as work parallelization, memory management, network communication, disk usage and scheduling. Thus, it provides big data processing capabilities for quick and easy application development.

Thrill sets itself apart from the other dataflow frameworks by two major differences. First, Thrill is written in a natively compiled language, i.e. C++, which allows for explicit control over memory and better performance overall. Second, Thrill conceptualizes data as an array instead of as a multiset. The order property of arrays is missing in multisets and allows operations important in application development, like sorting, prefix sums, window scans and zipping [5].

The algorithms developed in this thesis are integrated and tested in Thrill. Thrill programs are compiled and run on h hosts in parallel. Each host runs the same program simultaneously. The hosts have to share a network where communication is possible between each pair of hosts. Each host has a number of threads called workers and external memory such as disks. The number of threads corresponds to the number of cores c minus one for network communication. Consequently, there are $P = h(c - 1)$ worker threads in total. The number of processors v in the EM-BSP is equal to the total number of threads P in Thrill.

This section will introduce the necessary components and concepts needed to understand the decisions made in the algorithm engineering process for this thesis. The section will start with introducing the basic conceptual data structure used throughout Thrill, namely distributed immutable datasets (DIAs). Followed by an explanation of the dataflow graph, which models how data in DIAs flow through the processing nodes. DIAs and the dataflow graph build the conceptional foundation for a deeper technical look into files, network streams and the memory layout which concludes the section.

Data in Thrill is conceptually represented as a DIA, an array of individual elements. An array was chosen as the underlying data structure, which is unique compared to Apache Spark and Apache Flink, to facilitate the order property of arrays. This introduces the concept of locality, which can be very useful when designing applications in Thrill, and can e.g. be used to implement invariants. Furthermore, DIAs are distributed and immutable. The data is distributed over all nodes involved in the execution of the application in some way. However, the distribution of data is not visible to or controllable by the API user. Individual elements cannot be accessed by the API user directly. Data in DIAs can only be manipulated by DIA operations, which process a DIA in whole and return a result DIA. In Thrill, DIAs are immutable, which facilitates reasoning over applications implemented in Thrill.

DIA operations are partitioned into four categories, namely *sources*, *local operations*, *distributed operations* and *actions*. Most DIAs operations are implemented as template classes with a parameter for a user defined function (UDF), which can be used to customize the DIA operations behavior. For brevity, only a few examples of operations are named here. A complete list of operations can be found in [5].

Sources. Source operations newly create a DIA without a previously existing DIA. This include multiple ways of generating a DIA from a generator function (`Generate`) and reading a DIA from a file (e.g. `ReadBinary`).

Local Operations. Local operations (LOps) take a DIA as input and perform a computation that does not require communication between workers. Thus, the computation has to be on a per element basis. The result of a LOp is another DIA. Examples of those operations are the well known functions `Map`, `FlatMap` and `Filter`.

Distributed Operations. Distributed operations (DOps) take a DIA as input and perform a computation that does require communication between workers. Examples of those operations are `GroupByKey`, `Merge`, `Window` and `Sort`.

Actions. Actions are DIA operations that take a DIA and return a result that can be further processed by the user, unlike DIAs. Thus, actions are the only way to extract results from the API. Examples of those operations are `Size`, `AllGather`, `WriteBinary` and `Sum`.

Internally, transparent to the API user, DIA operations are lazily constructed into a dataflow graph, where nodes in the graph represent DIA operations and directed edges model data dependencies between the DIA operations. The resulting graph is directed and acyclic. With the lazy construction, DIA operations are only executed, when an action operation is encountered, since actions are the only operations returning a result the API user can work with. When an action is encountered, all operations on the path to this action are executed in order.

Figure 6 depicts a dataflow graph and the corresponding data representation as a series of DIAs. DIA A is sorted with the `Sort` operation and a compare function c to DIA B. The compare function c has to be a total order on the input data. DIA B in return is mapped with the `Map` operation and a custom function f. The result is stored in DIA C. However, this would only be executed when an action operation is encountered. Next to the dataflow graph is the series of DIAs $A-C$ simplified to 12 elements processed by three workers $p_0 - p_2$. The `Map` operation locally processes each element and stores it in the same position in the resulting DIA. The `Sort` operation however requires communication between the workers and stores the elements in sorted order. This simple example shows the interactions between nodes and workers on a high level.

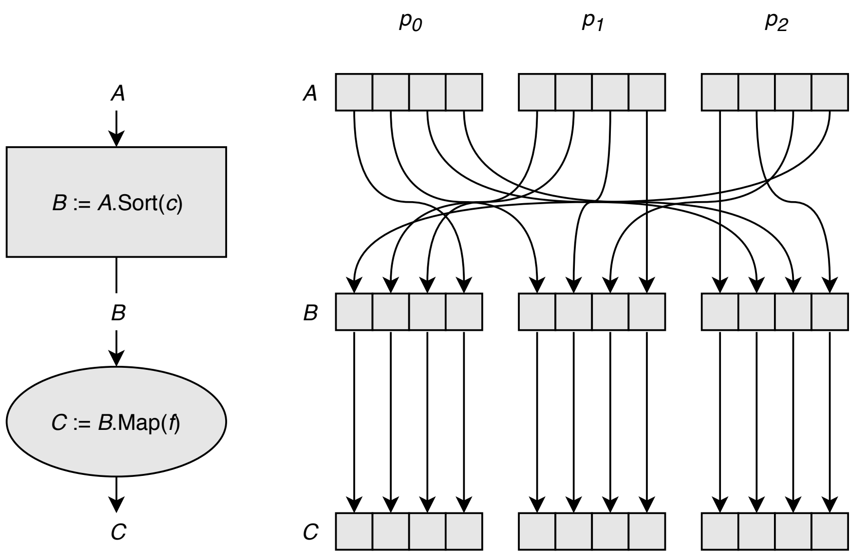


Figure 6: Dataflow Graph and DIA Representation (adapted from [5]).

Even though DIAs are the concept the API user sees, DIAs are not normally materialized internally in Thrill. DIA operations use intermediate representations that are suited to the specific operation. This allows optimizations of the dataflow on the dataflow graph. One optimization is pipelining. Pipelining is used to save overhead and collapse nodes in the dataflow graph. Since pipelining is not applicable to all possible DIA operations, it is applied only to LOps that process single elements of the DIA without dependencies to other elements and the first local computation step in DOps. The pipelining eliminates intermediate storage arrays and possibly reads from and writes to the disks. This optimization can also be interpreted as combining local computations into one computation superstep, instead of doing multiple computation supersteps, in the EM-BSP model and saving the overhead of the synchronizations otherwise needed [13].

Figure 7 shows the same example used in figure 6 implemented into a staged framework for pipeline implementation in Thrill. It shows sorting, followed by mapping, on two processors (p_0 and p_1). DOps are implemented in the three stages *PreOp*, *MainOp* and *PostOp*. Sources have MainOp and PostOp and Actions have PreOp and MainOp. This concept allows easy implementation of pipelining. MainOps cannot be pipelined. Thus, pipelining is applied between MainOps. After a MainOp, the PostOp opens a new pipeline and emits elements of the conceptual DIA, resulting from the MainOp, one by one. Thereafter, LOps are chained as long a no PreOp is encountered. The PreOp closes the pipeline and collects the data to pass on to the next MainOp. In figure 7, pipelined operations are enclosed in striped darker boxes. The first pipeline is placed between the MainOp of the `ReadLines` operation and the MainOp of the `Sort` operation. The second pipeline is placed between the MainOp of the `Sort` operation and the MainOp of the `WriteLines` operation. Since the `Map` operation is a LOp, it is incorporated as one step

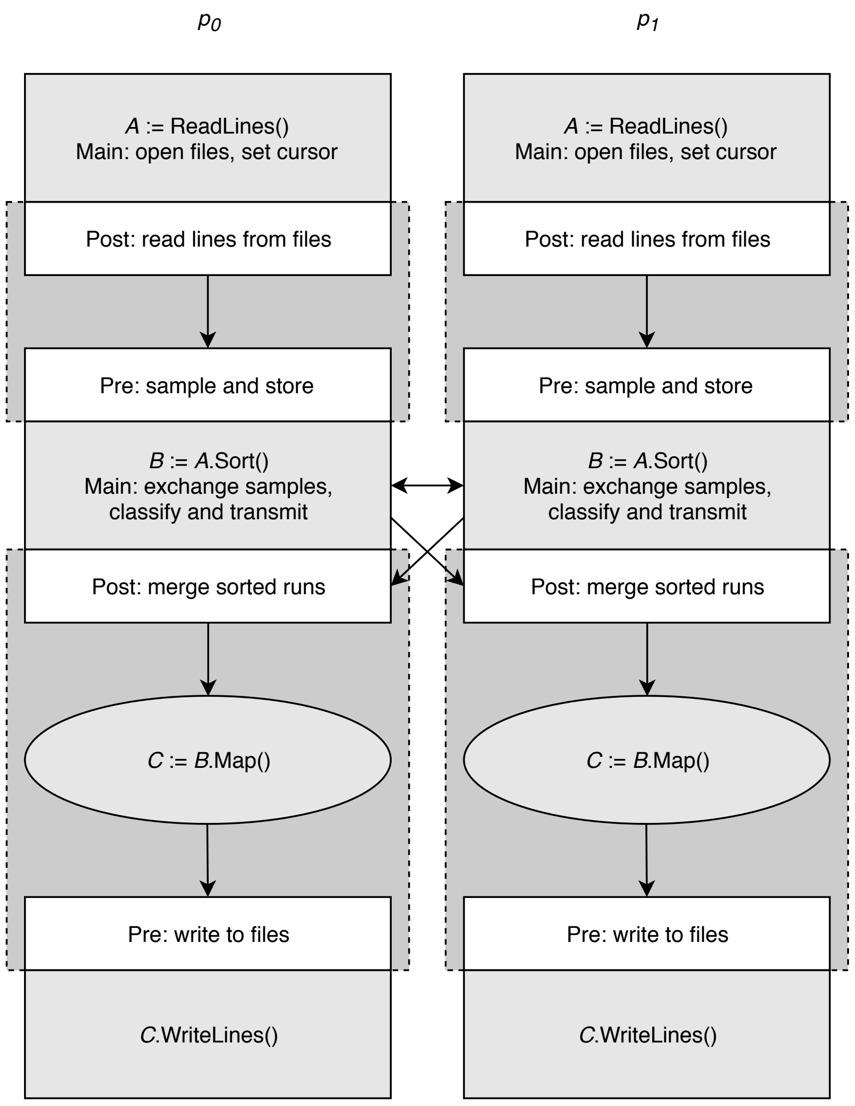


Figure 7: DIA Operations Stages (adapted from [5]).

in the second pipeline. The implementation details of PreOp, MainOp and PostOp are individual for each operation, but in general as much as possible should be done in the PreOp and PostOp to aid pipelining.

For processing of big data, network communication in large clusters and temporary storage of intermediate results on disks is very important. Thrill handles these transparently to the user with `Stream` and `File` respectively. `Stream` represents an asynchronous connection to every other worker on the cluster which transports arbitrary data. `File` represents external disk memory and hides disk parallelism.

Both `Stream` and `File` work on binary blocks of a predefined byte size (2 MB by default). The blocks themselves have no additional information but the serialized data and some necessary meta data. Serialization takes place with almost zero overhead. This is possible, because fixed length data types are directly stored in binary form and variable length strings and vectors are prepended with their length. Due to the strong typing in the framework, no additional type information or separators have to be stored. Blocks are read and written with `BlockReader` and `BlockWriter`. `BlockReader` reads from a `BlockSource`, which can be either a `StreamBlockSource` or a `FileBlockSource`. `BlockWriter` writes to a `BlockSink`, which can be either a `StreamBlockSink` or a `FileBlockSink`.

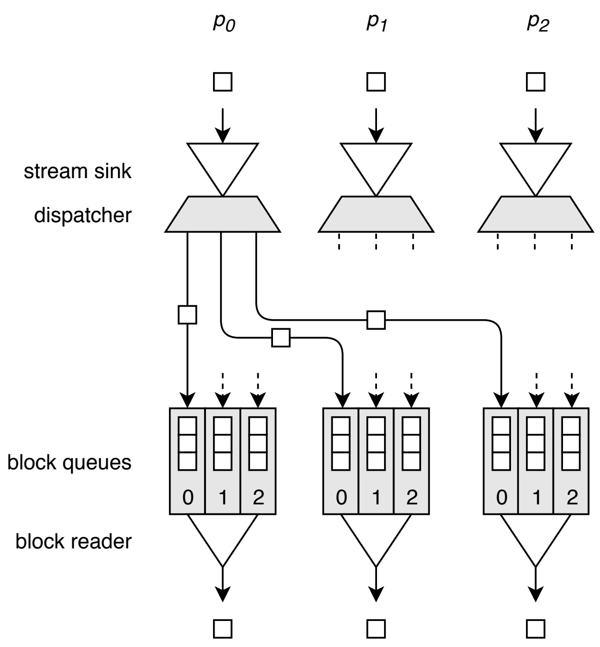


Figure 8: Block Flow through CatStream.

There are two types of streams in Thrill, `CatStream` and `MixStream`. Blocks in a `CatStream` are delivered in worker order. On every worker, all blocks from worker 0 are read first. Only when worker 0 is finished, blocks from worker 1 can be read and so on. A `MixStream` provides received blocks in order of arrival, which depending on the network is arbitrary. Figure 8 depicts how blocks flow through a `CatStream` when data is communicated between workers. For clarity, only the connection arrows of p_0 are shown. Blocks are inserted into the `StreamBlockSink` on each worker ($p_0 - p_2$) with a `BlockWriter`. The `StreamBlockSink` passes the block to a dispatcher, which depending on the mode of communication sends the blocks over the network to the receiving worker. At the receiving worker, the block is enqueued into a block queue for each sending worker. Thus, on every worker there are P block queues, where P is the amount of workers. A `BlockReader` can then read the blocks in order. In a `MixStream`, the blocks are not inserted into multiple block queues, but just one. Whenever blocks in worker order are not required, a `MixStream` should be chosen, because it does not block the entire stream when the currently lowest non finished worker is blocking.

Since communication and file handling both work on the same block abstraction, no additional copying has to take place, when data is read from a file and transmitted over the network or received from the network and written to a file.

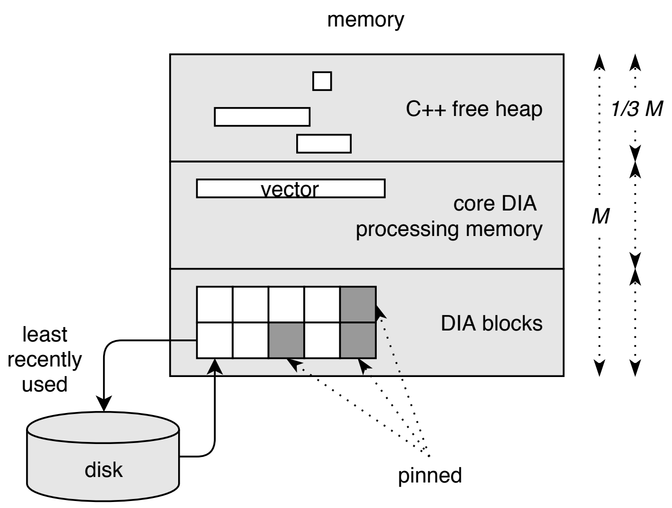


Figure 9: Thrill Memory Layout.

Figure 9 shows the three partitions of main memory in Thrill. Memory is partitioned into free floating heap memory, DIA operations memory and block memory. The free floating heap memory can be used by the API user arbitrarily. DIA operations memory holds data during DIA operations, e.g. intermediate results and DIA elements. DIA operations memory is negotiated between operations, where every operation registers the relative amount of memory it needs, which is divided fairly on execution. Lastly, block memory hold blocks that currently reside in memory. Thus, blocks that are neither on the network nor stored on disk. All blocks are managed by the `BlockPool`. It reference counts the blocks and tracks the memory usage. When there are too many blocks currently in memory, the least recently used block is swapped out to the disk. Thus, `File` at first only exist in memory, but with time get swapped out to disk. Currently used blocks may be *pinned* to avoid swapping, resulting in a `PinnedBlock`. Pinning can also be used for asynchronous prefetching, which can largely speed up disk interactions.

2.4 External Distributed Sorting in Thrill

Before this thesis, Thrill already supported distributed external sorting. The framework implemented a version of distributed external Sample Sort. However, the implementation lacks in performance and is thus replaced by the work of this thesis. Since algorithm engineering is always in part motivated and informed by previous work, this section will provide an in-depth look into this version of Sample Sort and deduce weaknesses that will be improved upon.

```
1  auto sorted = api::Generate(
2      ctx, n,
3      [&ctx, &n, &generator_type, &rng](size_t i) -> uint64_t {
4          auto p = ctx.my_rank();
5          auto P = ctx.num_workers();
6          // Calculate local index that is in range [0, n / P)
7          auto local_i = i - n * p / P;
8          // Calculate global index such that elements are striped over processors
9          auto global_i = real_i * P + p;
10         return generator(global_i, n, generator_type, rng);
11     })
12     .Sort().AllGather();
```

Listing 1: Sort Usage.

Listing 1 shows a usage example of the `Sort` DIA operation. Namespace `api` in line 1 refers to the Thrill API. `Generate` is a source operation that generates with a Thrill context `ctx`, a number of elements `n` and a generator function (a lambda in this case). The Thrill context is an object unique on every worker that holds information about the underlying Thrill environment, e.g. networking and statistics. The generator function maps an index of `size_t` to an element of the DIA element type (`uint64_t`). In this example `rng` is a `std::std::mt19937_64`. The `Generate` function returns a DIA that is then sorted with the default order `std::less` and the `Sort` operation. `AllGather` is an action operation that executes `Generate` and `Sort` and gathers the result as a vector on each worker. This should only be performed with small amounts of data in test environments, since `AllGather` only works when the complete result fits into memory. The `Sort` function is overloaded and can also be passed a compare function that maps two values of the DIA element type to `bool`. One requirement to a new sorting algorithm in Thrill is the same interface and ease of use as `Sort`.

Figure 10 shows a conceptual view of the algorithm. The elements are streamed into the DIA operations PreOp on every worker, where elements are passed into a sampler and at the same time written into a file. When the file blocks fill the available memory, the elements are swapped to disk. After the element stream finished, the MainOp starts.

The MainOp begins by drawing samples from the sampler locally and sending the samples to worker 0. The number of samples s depends on the total number of elements currently in the file. The number of samples is calculated by $s = \log_2(n)\frac{1}{\text{imbalance}^2}$, where imbalance is a constant number between 0.01 and 0.5. Worker p_0 concatenates all samples into one vector, sorts them and draws $p-1$ equidistant splitters from the vector. One splitter between every two adjacent workers. The splitters are then sent back to each worker. The splitters are built into a splitter tree. All local elements are passed through the splitter tree which returns the index of the worker this element has to be sent to. Items are passed trough a stream to their respective worker and stored into a vector at the receiver. When

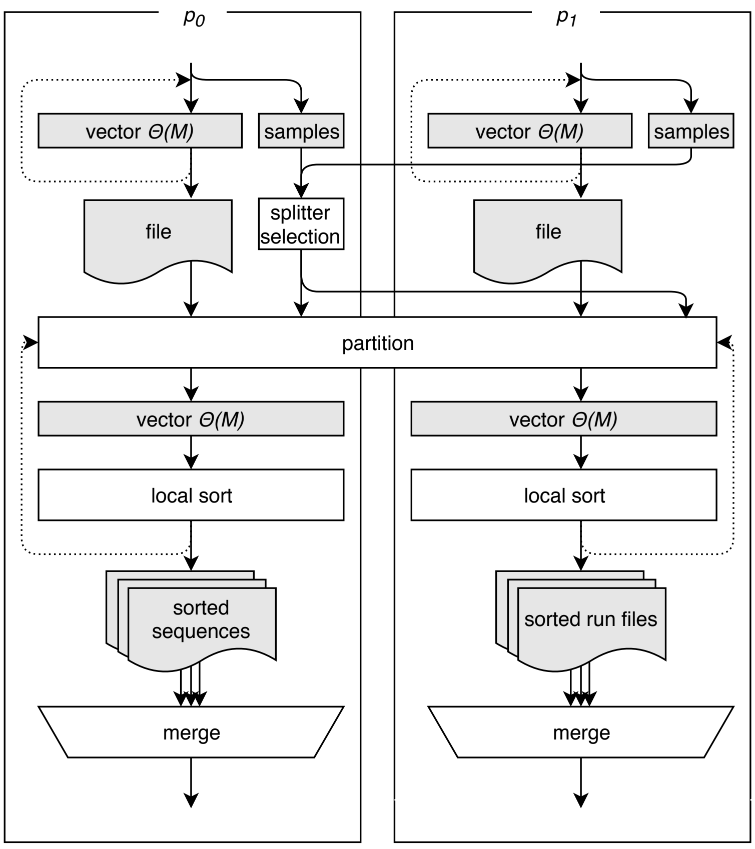


Figure 10: Thrill Old Sample Sort.

the vector fills the available memory, it is sorted and stored as a sorted run on disk. This concludes the MainOp. Hereafter, all elements reside on the correct worker, but are not sorted in total locally.

The PostOp takes all sorted runs, reads them from disk and merges them. In each step, the lowest element of all sorted runs is streamed into the next pipeline. This output is sorted, because the runs are sorted and always the smallest element is taken first.

This approach of sample sort however, has weaknesses. Two factors are very important for an external distributed sorting algorithm. Reads from and writes to disk and network communication. This algorithm has the minimum amount of network communication, but possibly read from and writes to disk more often than necessary. In the PreOp, knowledge extracted from the elements currently in memory can be used to already communicate them before writing to disk, which in the best case eliminates one roundtrip to the disk.

3 Canonical Merge Sort

3.1 Algorithm

Rahn, Sanders and Singler established an external distributed sorting algorithm in 2010 called CMS [23]. The algorithm is based around the idea of redistributing elements approximately before writing them to disk for the first time and correcting possible mistakes in distribution later on. They also presented a randomization technique for disk residing data, where the distribution mistakes in the first pass over the data were negligible, however in Thrill data does not reside on disk but is streamed into the algorithm. In comparison to the currently implemented Sample Sort algorithm, this algorithm possibly saves one round of disk writing and reading, but therefore sacrifices some amount of network communication for correction. Under the assumption of mostly randomly distributed input data, this algorithm requires only minimal additional network communication and saves almost a whole roundtrip to disk.

For this thesis, CMS is implemented in Thrill as a baseline state of the art external distributed sorting algorithm. The new algorithm OSS proposed by this thesis will be inspired by and measured against CMS. Thus, this section will introduce the concepts of CMS and is followed by three sections about implementation details. The implementation details will inform decisions made in the new algorithm and will show how the algorithm is optimized for best performance.

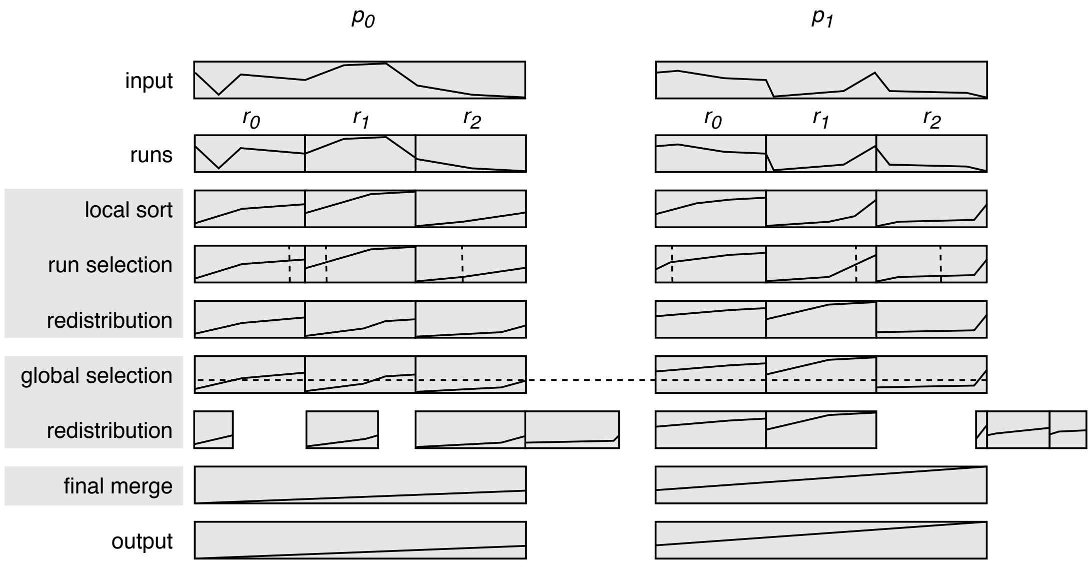


Figure 11: Canonical Merge Sort Algorithm.

CMS can be split up into three phases *run formation*, *correction* and *final merge*. Figure 11 shows how data is processed by the algorithm on $P = 2$ processors (p_0 and p_1) in the three stages. The algorithm starts with random input data on each processor. The input data is divided into runs ($r_0 - r_2$) that fit into memory. The last run does not have to be the same size as the other ones. For clarity, all three runs are the same size in this example. The steps local sort, run selection and redistribution constitute the run formation phase. The runs are processed one after another in the run formation phase. The current run is sorted on all processors in parallel and divided into P equal parts, where elements in part i are smaller than elements in part $i+1$. The vertical dashed line marks the splitting position on each processor for the parts. The parts are then redistributed. In the example, p_0 has the 1/2 smallest elements of each run and p_1 has the 1/2 largest elements of each run after the run formation. Since only data from the current run is used to decide about communication of elements, there are now still elements on the wrong processor.

After the run formation, the whole sequence of all runs is split up into P parts (marked by the horizontal dashed line) and elements still residing on the wrong processor are again redistributed in the correction phase. As already mentioned, this redistribution is negligible for mostly randomly distributed input data, but produces some communication overhead. The sorted runs are merged in the final merge phase and returned as one correctly sorted output sequence.

3.2 Implementation

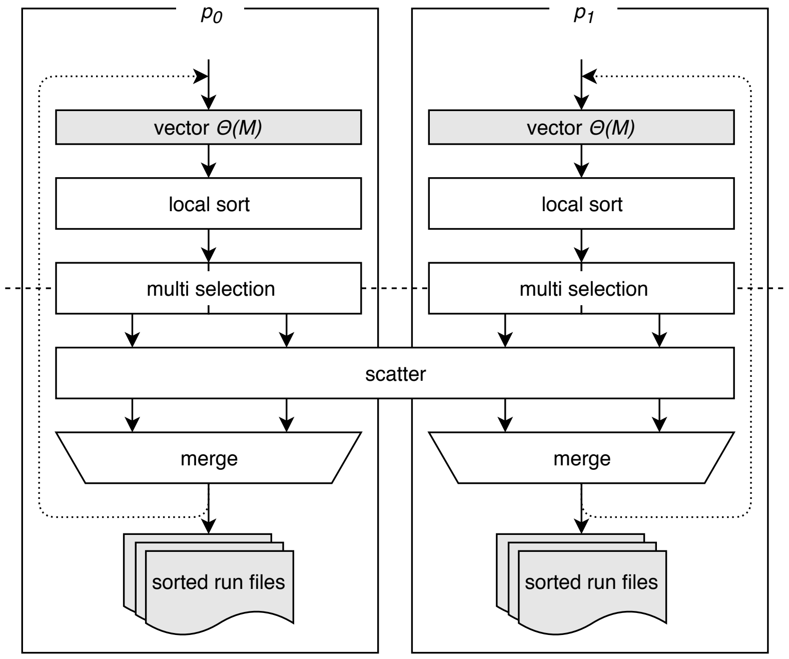


Figure 12: Canonical Merge Sort Run Formation Phase.

The three phases of CMS coincidentally map directly to PreOp (run formation), MainOp (correction) and PostOp (merge) in Thrill. Figure 12 shows implementation details of the first of the three phases. Elements are streamed into a vector buffer of size

$\Theta(M)$. When the buffer is filled with elements, the buffer is locally sorted, which takes $\mathcal{O}(M \log M)$, and waits for the other processors. After all processors have locally sorted their data, a Multi Sequence Selection (MSS) algorithm selects $P-1$ splitters (only s_0 in this example) that are equidistant over all sorted vectors. MSS will be explained in more detail in section 3.3. The splitters are communicated to all processors and used to scatter the buffers with a `CatStream`.

Scatter is a communication pattern, where each processor splits up a sequence into P parts, according to $P-1$ splitters and sends each part to the processor p_i matching the parts index in the sequence. For the example CMS in figure 12 that means all elements smaller than s_0 are communicated to p_0, all elements larger and equal to s_0 are communicated to p_1. The streams of incoming data from each processor are merged with a loser tree and written to a run file.

A loser tree (or tournament tree) is a runtime optimal k-way merge algorithm. It merges k sorted input sequences into one sorted output sequence. It gets its name from an underlying binary tree data structure. The tree has k leaves, which are initialized with the first value of each corresponding input sequence. To finish initialization, each inner node compares the value of its children and remembers which one is the smaller, such that the root node knows the smallest of the current values. To gain the sorted output sequence, the element from the root node is popped until no elements are left. Each time the root node element is popped, the path to the input sequence of this element is backtracked. The next element from this input sequence is inserted into the leaf and the comparisons along the backtracked path are redone. This only takes $\mathcal{O}(n \log k)$, because the tree is of height $\log k$. In the context of the algorithm this takes $\mathcal{O}(M \log P)$ for number of processors P.

After the data has been written to disk, the iteration starts over. Each iteration takes $\mathcal{O}(M \log M + P \log(M) + Mg + M \log P + \frac{MG}{DB})$ for time to communicate one element over the network g, time to write to disk G, number of parallel disks D and block size B. $\mathcal{O}(P \log(M))$ stands for the MSS runtime, which is explained in section 3.3. $\frac{n}{PM}$ iterations of the run formation phase are performed.

After the whole input data has been received, the next phase starts. Figure 13 depicts the correction phase. All data in the run files is considered at once and the same MSS as in the run formation phase is used to find the globally correct splitters. This should amount to small chunks of data from every run that have to be communicated to the other processors. Communication is again done in a scatter step that redistributes the data over a stream and writes it back to disk in run files. The correctly distributed data remains untouched in the run files. In most cases the runtime of this phase should be negligible, but this depends on the input data set. Thus, no runtime is assigned to this phase in the runtime analysis.

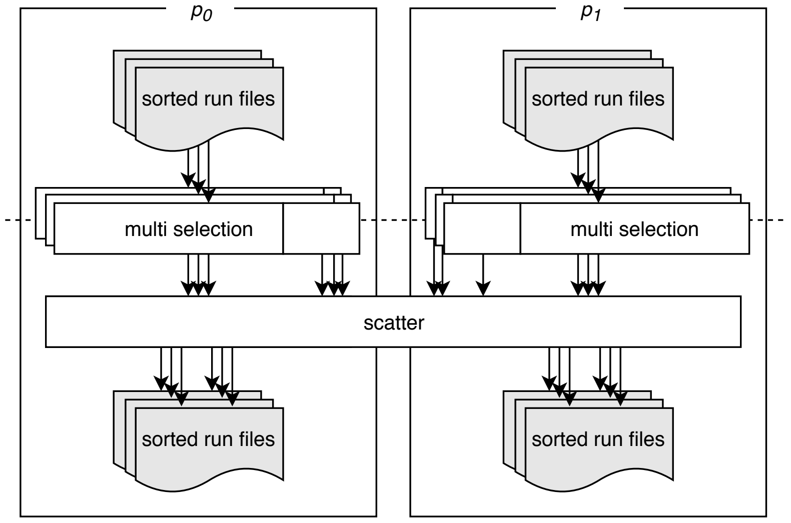


Figure 13: Canonical Merge Sort Correction Phase.

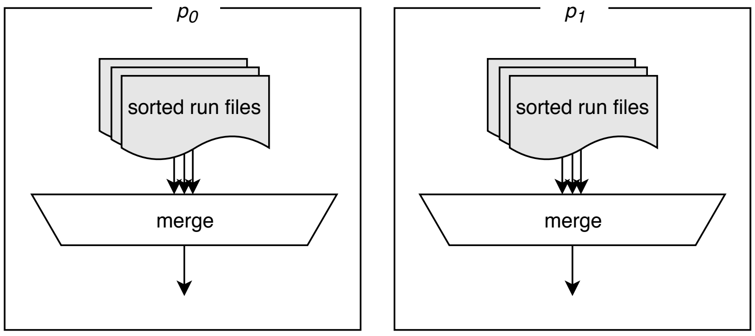


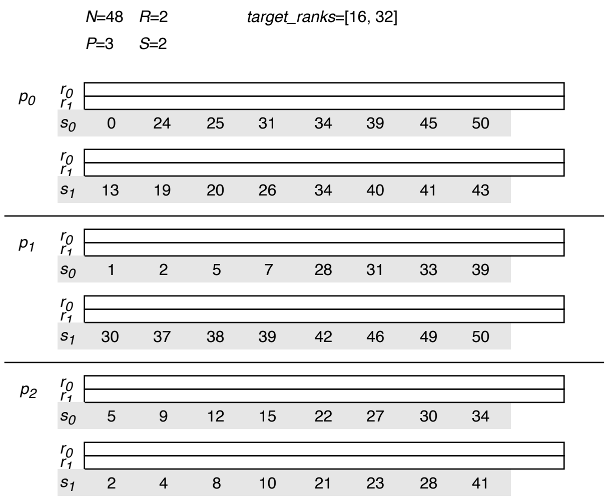
Figure 14: Canonical Merge Sort Final Merge Phase.

In the merge phase, the elements in the run files are merged with a loser tree and passed on to the next DIA operation. Figure 14 shows the merge phase. No communication takes place in this last phase. This takes runtime $\mathcal{O}(\frac{nG}{PDB} + (n/P)\log(n/PM))$ if the data is directly processed by the next DIA operation. The CMS algorithm implementation in Thrill has the same interface as the original `Sort` algorithm. The API functions name is `CanonicalMergeSort`.

3.3 Multi Sequence Selection

MSS is one of the key subalgorithms in CMS. It takes S input sequences $\langle s_0, s_1, ..., s_{S-1}\rangle$ on each of the P processors and calculates R splitters for each sequence. For simplicity of this explanation, we assume that S is the same on each processor. The output splitters denote the partial sequences that when merged result in non overlapping partitions. Thus, RSP splitters are calculated in total. In the context of CMS, $R = P - 1$ and all elements before the first splitter in each sequence have to be communicated to the first processor, all elements between the first splitter and the second splitter in each sequence have to be communicated to the second processor and so on.

Thrill already used an implementation of MSS that had to be fixed and optimized to work in CMS. This section will explain in detail how the resulting algorithm works. The idea of the algorithm is orthogonal to the partitioning algorithm in [26], which grows partitions exponentially until no data is left. MSS performs RSP binary searches in parallel, shrinking ranges that the correct splitters have to be in. It is organized into steps that perform one iteration of each binary search, until no iterations are possible.

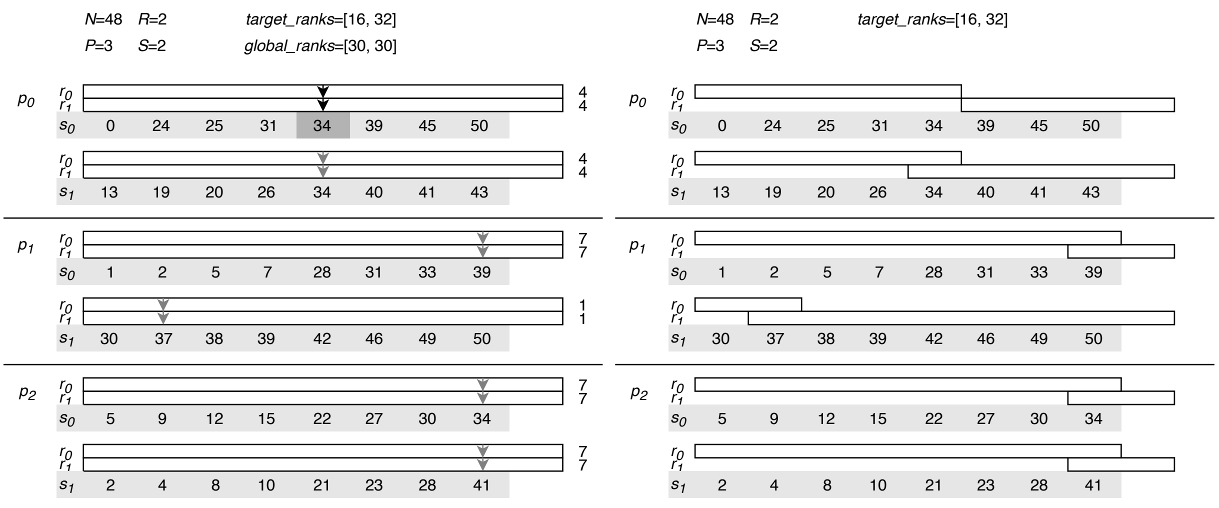


N=48 R=2 target_ranks=[16, 32]
P=3 S=2
p0 r0 r1 s0 0 24 25 31 34 39 45 50
r0 r1 s1 13 19 20 26 34 40 41 43
p1 r0 r1 s0 1 2 5 7 28 31 33 39
r0 r1 s1 30 37 38 39 42 46 49 50
p2 r0 r1 s0 5 9 12 15 22 27 30 34
r0 r1 s1 2 4 8 10 21 23 28 41

Figure 15: Multi Sequence Selection Initialization.

To initialize the algorithm, the target ranks are calculated as $T = \langle t, (2t), ..., (Rt)\rangle$, where $t = n/P$. Each binary search is based on a range that is initialized with $L_{ij} = 0$ as the left boundary and $W_{ij} = S_i.size()$ as the width for $i \in [0..S], j \in [0..R]$ on each processor. Figure 15 shows example sequences and the initial parameters of the algorithm. The MSS is performed on 48 elements, distributed over 3 processors $(p_0 - p_2)$ with 2 sequences (s_0, s_1) each. This results in 2 splitters (r_0, r_1) for each sequence and target ranks of 16 and 32. The ranges of the binary searches are depicted as white boxes with black borders above each grey sequence. After initialization, L_ij is 0 and W_ij is 8 for $i \in [0..2], j \in [0..2]$ on each processor.

Each step of the algorithm is divided into the three substeps *pivot selection*, *global rank calculation* and *search*. Pivot selection halves each range and takes as pivot the element at that position. The pivots are then exchanged with all processors in an all-to-all communication and reduced by splitter to the pivot coming from the biggest range. Thus, resulting in one pivot per splitter that is the same on each processor. If there are multiple biggest ranges, the pivot coming from the lowest ranking processor and lowest sequence index is chosen.



N=48 R=2 target_ranks=[16, 32]
P=3 S=2 global_ranks=[30, 30]
p0
s0 0 24 25 31 34 39 45 50
s1 13 19 20 26 34 40 41 43
p1
s0 1 2 5 7 28 31 33 39
s1 30 37 38 39 42 46 49 50
p2
s0 5 9 12 15 22 27 30 34
s1 2 4 8 10 21 23 28 41
r0
r1
4
1
7

(a) Global (b) Search

Figure 16: Multi Sequence Selection Step 1

Figure 16 shows the global rank calculation and search substeps of the first MSS step. The pivot selection resulted in 34 from p_0 s_0 as the pivot for both splitters, which is marked darker and with a black arrow in figure 16a. In global rank calculation, a binary search for each range is performed, resulting in an index of the pivot for each range, and stored in l. The indices are marked with a grey arrow and written at the back of the range in figure 16a. To determine the global rank of the pivot, the resulting l values are summed up by splitter and saved into global ranks G.

Figure 16b shows the result of the search substep. The search substep shrinks the ranges depending on the global ranks G in comparison to the target ranks T for all sequences and splitters. For sequence i and splitter j the range is shrunk as follows:

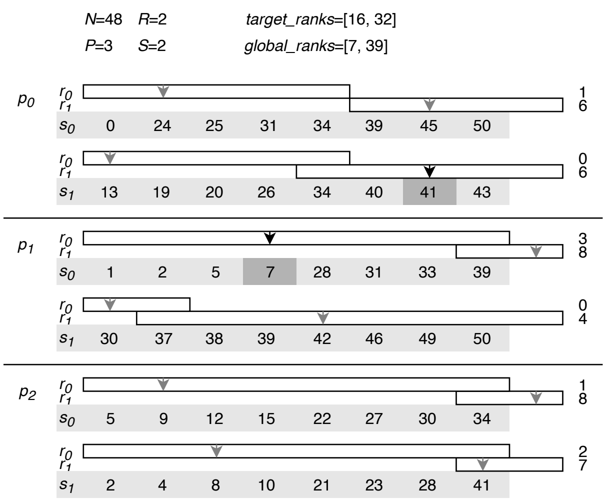
(i) If $T_j = G_j$, then $L_{ij} = l$ and $W_{ij} = 0$

(ii) If $T_j < G_j$, then $W_{ij} = l - L_{ij}$

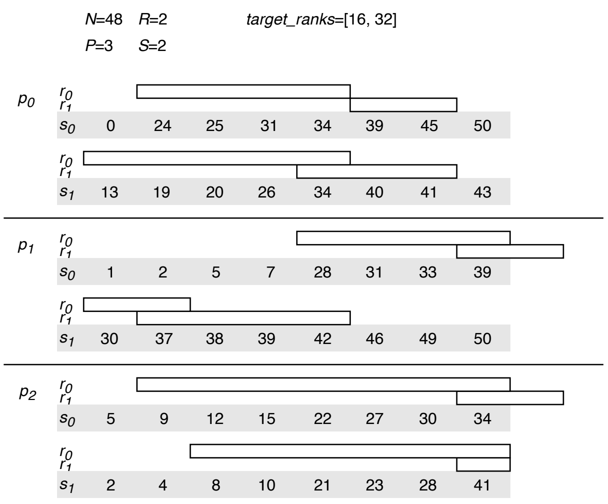
(iii) If $T_j > G_j$ and pivot is from sequence i on this processor, then $L_{ij} = l$ and $W_{ij} = W_{ij} - (l - (L_{ij} + 1))$

(iv) Else if $T_j > G_j$, then $L_{ij} = l$ and $W_{ij} = W_{ij} - (l - L_{ij})$

If the target rank is equal to the global rank, the binary search is done and the range is closed. If the target rank is smaller than the global rank, the right boundary of the range is moved to the local rank and the left boundary stays. If the target rank is bigger than the global rank, left boundary is moved to the local rank and the right boundary stays. The left boundary is moved one additional element to the right, if the pivot came from the current sequence on the current processor to exclude the current pivot. Without this, there is no guarantee that the ranges ever close completely. In each step, at least one range is halved. There is no guarantee that any of the other ranges are shrunk.



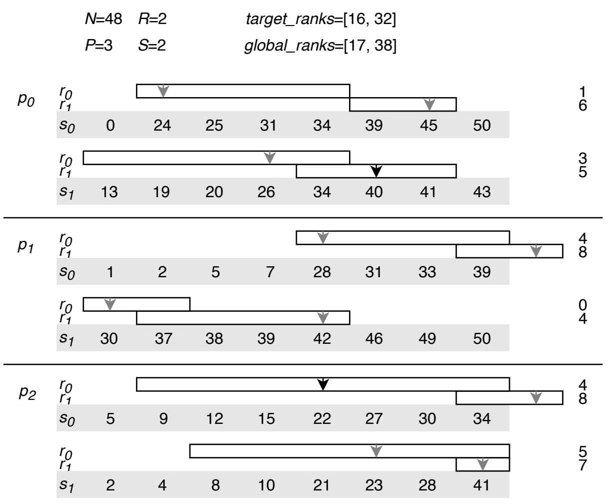
(a) Global



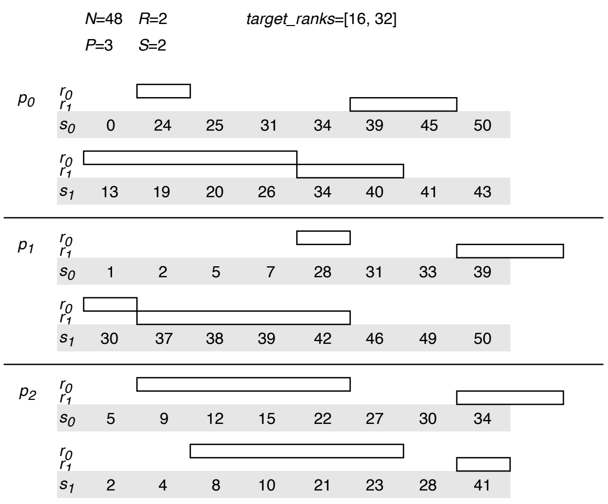
(b) Search

Figure 17: Multi Sequence Selection Step 2

Figure 17 shows the global rank calculation and search substeps of the second MSS step. Since s_0 on p_1 has the largest range for r_0 and s_1 on p_0 has the largest range for r_1, 7 and 41 are selected as pivots. The range of splitter 1 on s_1 on p_2 is already closed. The resulting splitter has to be index 7 with value 41 for s_1 on p_2.



(a) Global



(b) Search

Figure 18: Multi Sequence Selection Step 3

Figure 18 shows the global rank calculation and search substeps of the third MSS step. For splitter 0, the pivot is out of range for sequence s_1 on p_1 and since the range is closed already, no elements of s_1 on p_1 can be in front of splitter 0.

MSS terminates after all ranges are closed to width 0. Figure 19 shows the result for the example sequences. The dashed lines denote the splitting positions. Thus, everything less or equal to 20 is in front of splitter 1 and everything less or equal to 34 is in front of

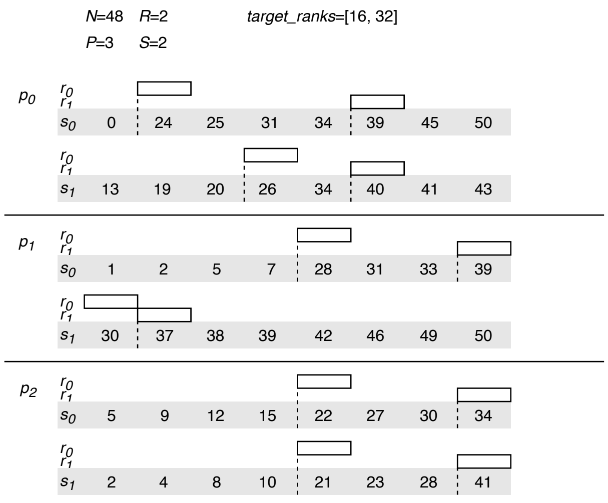


Figure 19: Multi Sequence Selection Result.

splitter 1. The resulting partitions are exactly of equal size. MSS returns the values of L to CMS.

The described MSS does not necessarily converge with duplicate elements in the data set. Thus, in the Thrill implementation, the processor index, the sequence index and the index in the sequence are used as a tie breakers during comparison of two elements of the same value in this order.

Furthermore, step 1 in the above example showed optimization potential of a naive implementation of this algorithm. A binary search is performed for each splitter on each sequence on each processor. However, the pivots of the splitters are the same. Thus, a binary search can be saved by storing the result of the last binary search for each sequence and using it if the pivot of the previous splitter for this sequence was the same, instead of performing another binary search with the same result.

The runtime of this MSS algorithm is $\mathcal{O}(RSP \log(n/(SP)))$. The sequences have length $n/(SP)$. A binary search ($\mathcal{O}(\log n)$) is done for each of the RSP ranges. This runtime is optimal [12].

3.4 Optimizations

After the naive implementation of the CMS algorithm, several optimizations were implemented in order to improve the implementations runtime performance on real systems. Both scatter operations that are performed in the run formation and correction phases respectively at first used `CatStream`. `CatStream` provides an API to read from each sender on each receiver independently, which enables easy merging of the arriving already sorted sequences. However, for sorting the order preserving property of `CatStream`, which produces overhead, is not necessary. The first optimization thus was to implement reading independently from the senders in a `MixStream`.

Another problem that was obvious with the first preliminary experiments was the runtime of the MSS in the correction phase. In the correction phase, most of the data resides in files on disk, but MSS requires repeated binary searches on this data. The way Thrill files are structured, blocks have to be read into memory in their entirety instead of just reading one element from a block. Furthermore blocks are swapped out fast this way and have to be read repeatedly instead of caching single elements. As an optimization, `SampledFile`, as an alternative to regular File, is introduced.

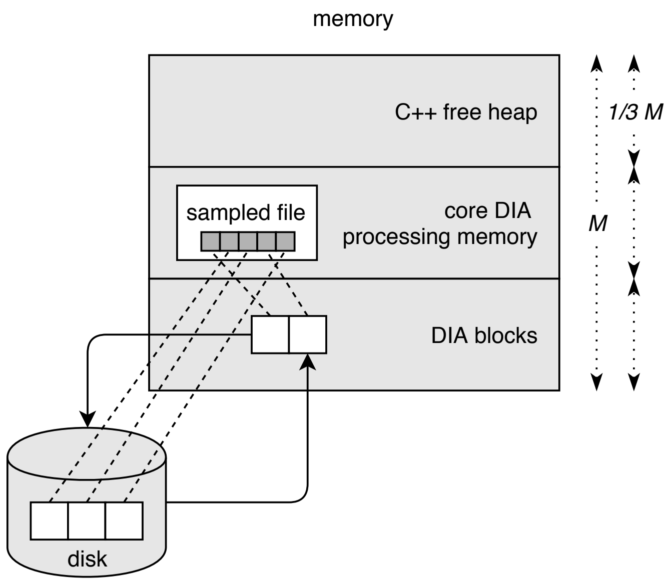


Figure 20: Sampled File Memory Layout.

Figure 20 depicts the memory layout of a `SampledFile`. The first element of each block is drawn as a sample and stored in the `SampledFile` object in the core DIA memory. Thus, at least one element of each block is always in memory. The rest of the block is stored in DIA blocks memory and on disk as normal. This optimization only works if $t << b$ and $n/b << M$ for file block size b and size of element type t. Otherwise the samples in memory take up too much space and a more sophisticated approach has to be chosen.

A special version of binary search is implemented in `SampledFile` that first only searches on the samples in core DIA processing memory and therewith finds the block where the element that is searched for resides in. This block is subsequently pinned and searched in. `SampledFile` reduced the overall runtime of the CMS implementation significantly.

Despite `SampledFile`, the MSS was still very slow on larger clusters due to the number of ranges growing polynomial with the number of processors. Thus, an optimization inspired by appendix B of [23] for initialization of binary search from samples was implemented. Appendix B outlined an initialization step, which shrinks all ranges to one block. Thus, the selection could cache the one block of each sequence on each processor that the MSS is finalized in. This in theory saves $\log n - \log b$ steps of MSS for file block size b and a lot of disk interaction. However, the resulting splitter positions may be incorrect, because the correct splitter positions are not necessarily in the initialized ranges for all sequences. This thesis proposes a similar initialization approach that yields correct splitter positions.

The initialization step first sends all samples from all processors to processor 0. Each sample is sent in a `struct` that additionally contains the size of the corresponding block and the processor and sequence it came from. The samples are concatenated into one sequence and sorted by value of the sample. Then a prefix sum on the block sizes is performed. Prefix sum maps each value to the sum of all previous values. The resulting sizes are then shifted to the left by one. Thus, each sample is stored with an approximate rank in the whole sequence. For each splitter, a binary search for rank $r = (i+1)n/(R+1)$ for splitter index i is performed and a search from this position backwards sends the first sample found for each sequence to each processor. Each processor initializes the ranges for the sequences with the samples block start position as left and b as the width. If the backwards search does not return a sample for a sequence, the first block is taken as the range. This results in b wide ranges for each splitter on each sequence. Additionally, the ranges have to be expanded, if a pivot search returns an index outside the range.

This implementation of approximate initialization of binary search is bound by the available memory M on processor 0. The input size in bytes nt has to be smaller than $(M/t)b$ for size of element type t. The number of samples m is nt/b. The runtime of the approximate initialization step is $\mathcal{O}(mg + m + m \log m + rm + rsg)$.

This concludes contribution C1.

3.5 Analysis

The runtimes of the phases (excluding the correction phase) add up to a total runtime of CMS of $\mathcal{O}(\frac{n}{PM}(M \log M + P \log(M) + Mg + M \log P + \frac{MG}{DB}) + \frac{nG}{PDB} + (n/P)\log(n/PM))$. Thus, the run formation and final merge phases have similar runtimes independent of the data set. The runtime of the correction phase is mostly influenced by the amount of data that has to be communicated, which differs for different data sets. Communication in the correction phase results from the run formation phase placing elements on the wrong processor. Elements are placed on the correct processor in the run formation phase, the more correct the MSS result is in the context of the whole data set.

One data set from [4] that forces a lot of communication in the correction phase is sorted (*Sorted*) input. Sorted will be implemented as $S[i] = iP + p\%P$ where i is the index in the sequence, p is the index of the processor and P is the amount of processors. Another data set that forces a lot of communication in the correction phase is sliding window sorted input (*Window*). Sliding window sorted input is generated with $S[i] = iP + p\%P + \text{rng}()\%w$ with sliding window size w. The larger w gets, the more random the input is.

Both data sets force communication in the correction phase, because the run formation phase places a lot of elements on the wrong processors. This is due to the local knowledge of a run being a bad predictor of the overall data set. The newly developed algorithm in chapter 4 will improve upon this weak spot of CMS.

4 Online Sample Sort

4.1 Algorithm

As already discussed in chapter 3, the weakness of CMS is in distribution of the data in the run formation phase. The run formation phase does not use all known information about the data set, but only what can be deduced from the current in memory run. Let R_i be the data in a run for $i \in [0..f)$ and number of runs $f = n/M$. Let $D_i = R_0 \cup R_1 \cup ... \cup R_i$ be the union of all runs until i for $i \in [0..f)$. CMS uses only the knowledge in R_i to redistribute data in R_i.

The idea of the new algorithm proposed by this thesis OSS is to keep the three phase structure of the CMS algorithm and improvements of CMS over the old Sample Sort algorithm implemented in Thrill, but introduce an improved version of the run formation phase that incorporates information about all processed data into redistribution. OSS would thus use all knowledge in D_i to redistribute data in R_i. Since not all data in D_i can be kept in memory, a representative sample of the already seen data D_i should be used to decide about data redistribution at the end of each run. Furthermore, OSS will use the Sample Sort paradigm.

The hypothesis is that OSS will reduce the amount of data residing on the wrong processor after the run formation phase and thus will reduce the runtime of the correction phase for real data inputs, due to less overall communication and disk interaction. The critical performance indicator, which allows confirmation of the hypothesis, is the total amount of communication in comparison to CMS, since total amount of communication in contrast to runtime is not implementation dependent.

4.2 Implementation

The newly developed algorithm that is presented in this section is called OSS. Its goal in comparison to the old Thrill Sample Sort implementation is to sample, partition and redistribute the input data in one run over the data. This saves a pass over the data to generate a sample at the cost of a correction step. It is divided into the same three phases as CMS *run formation*, *correction* and *final merge*. The Sample Sort paradigm is only used in the run formation, but is introduced because Sample Sort implementations seem to have lower constant factors than Merge Sort implementations.

Figure 21 illustrates the run formation phase of OSS. Input is streamed into a vector buffer of size $\Theta(M)$ and at the same time passed into a sampler. The sampler keeps a representative sample of the already processed input data at any time at a reasonable memory usage. Details about sampling are given in section 4.3. When the vector is full, the run is finalized. All P processors send their sample to p_0, the samples are concatenated

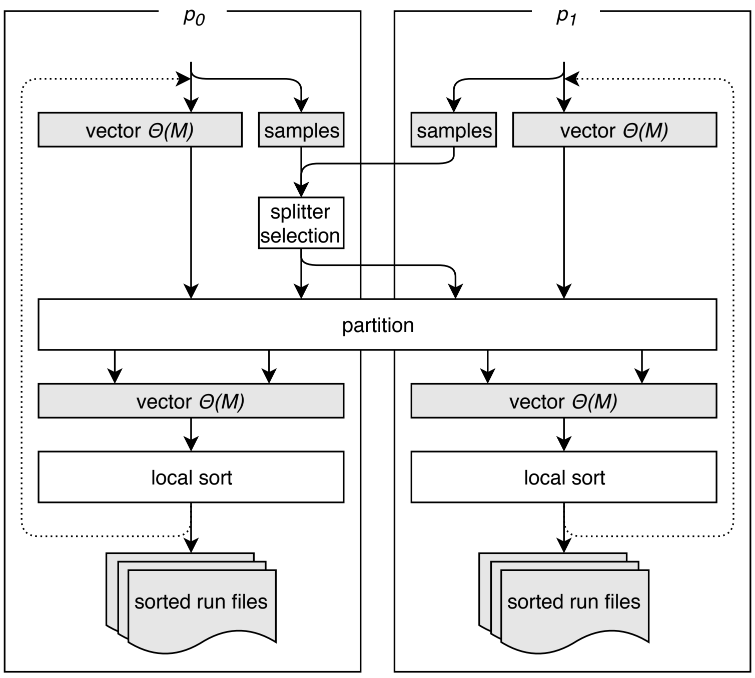


Figure 21: Online Sample Sort Run Formation Phase.

into one sequence and sorted and $p-1$ equidistant splitters are extracted. The splitters are used to partition the buffer into P partitions ($\mathcal{O}(M \log P)$), one for each processor. The partitioning is done with a splitter tree, the same approach as the old Thrill Sample Sort implementation. After an element is assigned a partition, it is directly communicated to the respective processor. The received elements are put into a vector of size $\Theta(M)$ which is sorted locally ($\mathcal{O}(M \log M)$) and written to a file ($\mathcal{O}(\frac{MG}{DB})$). The vector at the start of the run formation iteration can be reused here. If more elements are received than fit in the vector, the vector is sorted locally, written to a file an cleared. This results in multiple sorted run files for one run. Since the splitters do not result only from knowledge about the current in memory run, it can happen that data is distributed unevenly over the processors. Each iteration takes $\mathcal{O}(M + M \log P + Mg + M \log M + \frac{MG}{DB})$. The first M stands for the runtime of the sampler. $\frac{n}{PM}$ iterations of the run formation phase are performed.

Figure 22 shows the correction phase of OSS. It works similar to the correction phase in CMS. The only difference is that in OSS, the splitters are not calculated with a MSS, but directly taken from the sampler. Since the sampler by this time has seen all the data, the sample is representative of the whole data set. This takes no time, since the samples were already used for the last run. Again the splitters are used to scatter the file to the correct processors and stored in the final run files that are passed to the next phase. In most cases the runtime of this phase should be negligible, but this depends on the input data set. Thus, no runtime is assigned to this phase in the runtime analysis.

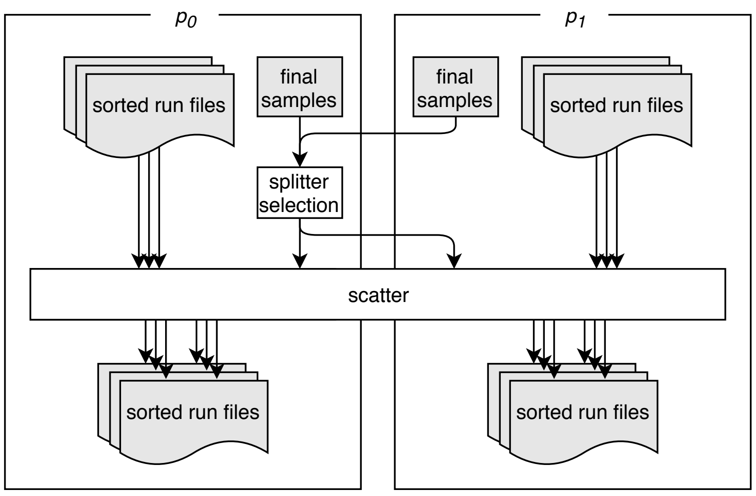


Figure 22: Online Sample Sort Correction Phase.

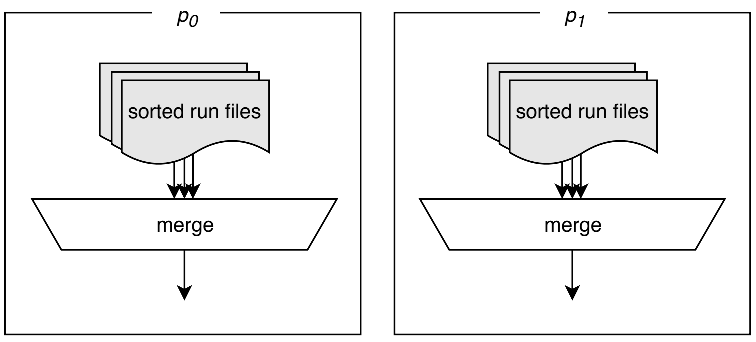


Figure 23: Online Sample Sort Final Merge Phase.

Figure 23 shows the merge phase of OSS. It works exactly the same as the merge phase in CMS. It takes the finished run files and merges them into one output stream. This again takes runtime $\mathcal{O}(\frac{nG}{PDB} + (n/P)\log(n/PM))$. The OSS algorithm implementation in Thrill has the same interface as the original `Sort` algorithm. The API functions name is `OnlineSampleSort`.

4.3 Sampling

Different sampling techniques can be used to implement sampling. A well known and widely used one, established by Vitter, is *Reservoir Sampling* [28]. Another technique for sampling is described in [18] by Manku, Rajagopalan and Lindsay, in this thesis it will be called *Online Sampling*. Furthermore, [19] introduces randomization into the Online Sampling approach in that could improve the runtime and protect the sampling technique against adversary data sets. The Online Sampling papers promise lower memory requirements than Reservoir Sampling, but miss evaluation against Reservoir Sampling in the context of different data sets and scale factors in practice. This section will introduce both sampling techniques and also explain how they run distributed on P processors. A detailed comparisons of both techniques in practice will be presented in chapter 5. Reservoir Sampling was already implemented in Thrill and Online Sampling was implemented for this thesis.

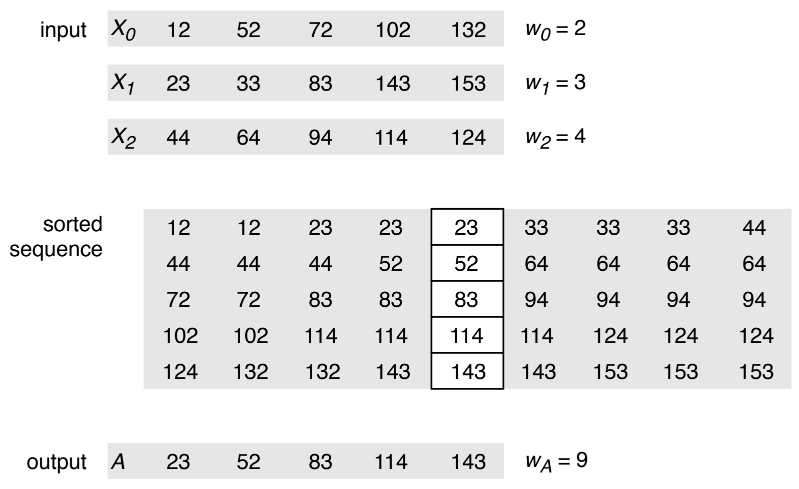
For the use in this thesis, both sampling techniques share an interface. The total size of the data set is not known beforehand and thus not known to the samplers. A `Put` method is called whenever a new element is streamed into the sort algorithm. Upon completion of a run, a `GetSamples` method is called to return the current representative sample.

Reservoir Sampling maintains an underlying reservoir of S candidates that represent the data set. The reservoir can be drawn as a sample of the already seen data at any time with the `GetSamples` method, which communicates the P reservoirs in an all-to-all communication step so every processor has all reservoirs and returns the concatenated reservoirs as the sample. The `Put` method inserts values until the reservoir is full and thereafter randomly replaces candidates in the reservoir with new elements. Lets denote the processed elements as d_i, a sequence of size D. Whenever the `Put` method is called, the processed data set is d_{i+1} of size $D+1$. Thus, the candidates in the reservoir of data set d_{i+1} have probability $S/(D+1)$ of being in a truly random sample. This is implemented by replacing a random candidate of the reservoir with probability $S/(D+1)$ each time `Put` is called.

Online Sampling is based on a framework of three operations `New`, `Collapse` and `Output`. These three operations are performed on a sequence of underlying buffers $\langle B_0, B_1, ..., B_{b-1}\rangle$, resulting in two parameters for Online Sampling, b for number of buffers and k for size of buffers. Each buffer X also has a level l_X and a weight w_X.

`New` takes an empty buffer E and initializes it. The weight is always initialized as $w_E = 1$. If there are still multiple empty buffers its level is initialized with $l_E = 0$ otherwise its level is initialized with the lowest overall level $l_E = x : x <= l_i)\forall i \in [0..b)$, where l_i is the level of buffer B_i. The `Put` operation of Online Sampling then inserts elements into the empty buffer until full. When an element should be inserted into a full buffer, `New` is called and the element is inserted into a new empty buffer. If all b buffers are full, `Collapse` is called.

`Collapse` first sorts all buffers $X_0, X_1, ..., X_c$ with number of buffers passed to `Collapse` c. It initializes an auxiliary buffer A with the sum of the input buffers weights $\sum_{i=0}^{c-1} w(X_i)$ as its weight w_A. The elements in the input buffers should now be merged and k elements at equidistant positions inserted into auxiliary buffer A. To account for the input buffers weights, each element in X_i is considered as taking up w_i spaces in the merged sequence. Figure 24 shows an example of the merged sequence and the resulting auxiliary buffer. Three input buffers with $k = 5$ and weight 2, 3 and 4 are sorted and merged into one sequence and and k equidistant elements are moved into the auxiliary buffer. The equidistant target rank of element A_i is $tr(i, w_A)$. If w_A is even $tr(i) = iw_A + w_A/2$ and otherwise



input						
X_0	12	52	72	102	132	$w_0 = 2$
X_1	23	33	83	143	153	$w_1 = 3$
X_2	44	64	94	114	124	$w_2 = 4$

sorted sequence								
12	12	23	23	23	33	33	33	44
44	44	44	52	52	64	64	64	64
72	72	83	83	83	94	94	94	94
102	102	114	114	114	114	124	124	124
124	132	132	143	143	143	153	153	153

output						
A	23	52	83	114	143	$w_A = 9$

Figure 24: Online Sampling Buffer Collapse (adapted from [18]).

$tr(i) = iw_A + (w_A + 1)/2$. To conclude the `Collapse` operation, the auxiliary buffer A replaces one of the input buffers and the remaining input buffers are emptied.

The selection of elements for the output buffer can be implemented without materializing all copies of elements. Therefore, a loser tree is initialized with the sorted input buffers as input sequences. Each time an element is popped from the loser tree, a total index counter is increases by the weight of the buffer w_i, the element is coming from. If the total index counter exceeds the current target rank for the output buffer element, the current popped value is appended to the output buffer A and the next target rank is calculated.

`Output` collapses all full buffers into an auxiliary buffer locally without changing any of the underlying buffers and communicates the auxiliary buffer and if present a partially filled buffer to processor p_0. Processor p_0 concatenates all partially filled buffers into full and possibly one remaining partially filled buffer. If two partially filled buffers L and H are of different weight, with $w_L < w_H$, the buffer with less weight L is resampled. Only elements in steps of $w(H)/w(L)$ are taken from L and concatenated to the other buffer. Thereafter, all auxiliary buffers and buffers resulting from the partial buffer merging are collapsed into one buffer and sent back to all processors, which return the buffer as the result of the `Output` operation. For Online Sampling, `GetSamples` thus calls `Output`.

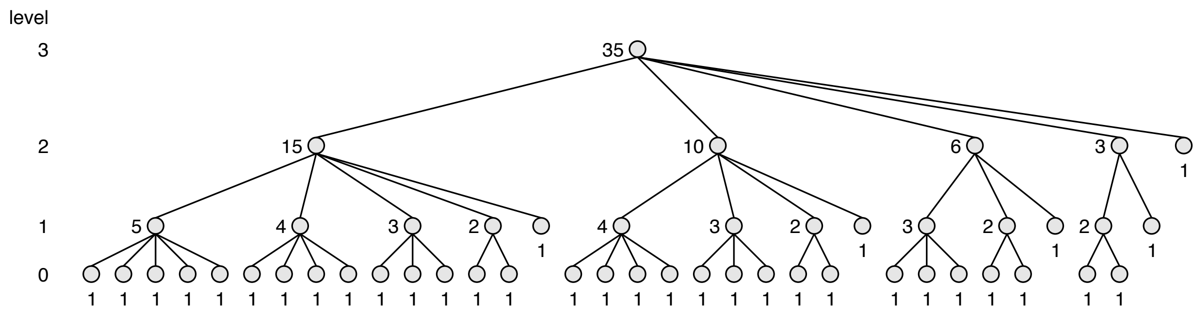


Figure 25: Online Sampling Buffer Collapse Tree (adapted from [18]).

Different policies for collapsing buffers are possible with this framework for Online Sampling. In this thesis a policy named *New Algorithm* in [18] is used, due to its low memory requirements compared to other policies. The policy is based on the buffer levels introduced earlier. The `Collapse` operation collapses all buffers of the lowest overall buffer level $x : x <= l_i \forall i \in [0..b)$. Due to the level initialization in the `New` operation, there are always at least two buffers of the lowest level. The resulting buffer is assigned the level $x + 1$. Figure 25 shows how buffers are collapsed in this level based policy. Buffers are marked by black dots with the weight under them. The level in the tree marks the buffer level from 0 at the bottom to 3 at the root. Buffers are created from left to right with weight 1 and if there exist five full buffers, the lowest level is collapsed, which is marked by the edges. The first five buffers are thus merged into one with weight 5 (leftmost in level 1). This is repeated with decreasing numbers of buffers in level 0, until one empty buffer is assigned level 1 and collapsed with the other buffers into the first buffer on level 2 width weight 15.

Online sampling in this form is prone to adversary data sets, due to its deterministic nature. Furthermore, collapsing buffers is an expensive operation due to sorting and merging runtimes. Thus, [19] introduces randomization with a randomization factor r. A step preliminary to inserting elements into the buffers is prepended, where first a randomization buffer of size r is filled with elements and upon a full randomization buffer, one random element is moved into the current buffer. All other elements in the randomization buffer are discarded.

4.4 Analysis

The complete runtime of the algorithm is $\mathcal{O}(\frac{n}{PM}(M + M \log P + Mg + M \log M + \frac{MG}{DB}) + \frac{nG}{PDB} + (n/P) \log(n/PM))$. This is almost the same theoretical runtime as CMS, only different in the MSS and sampler. The runtime improvements lie in constant factors and less communication and disk interaction for real data inputs. This concludes contribution C2 The magnitude of the improvement will be explored in chapter 5.

4.5 Related Work

In recent years, much progress has been made in the field of distributed sorting. Axtmann et al. worked on scaling distributed sorting to the largest available systems with [2] and [3]. The algorithm used to scale is Super Scalar Sample Sort introduced by Sanders et al. in [24]. Recent improvements of the algorithm made it possible to run in-place [4]. This was possible because Sample Sort lends itself to parallelization, but is also competitive on single processors. However, this recent work assumes data residing on disk and not provided by a stream.

The idea of Sample Sort was already worked on in 1991 by DeWitt et al. in [10] and in 1993 by Nodine and Vitter in [21] and later followed up with [22]. However, this work focuses on utilizing memory hierarchies and especially parallel disks. An algorithm approaching lower bounds for parallel disk usage was later presented by Dementiev and Sanders in [9]. This is not discussed in this thesis, because Thrill abstracts disk interaction away.

Sampling in this thesis was influenced by stream selection and approximate sorting approaches. In 2017, Karp presented streaming selection and one pass approximate sorting algorithms in [16]. Another inspiration was approximate sorting of streams by Farnoud et al. in [11].

5 Results

5.1 Systems

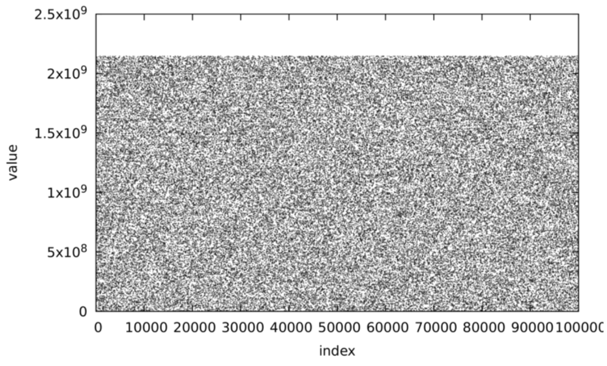
The experiments are performed on three different systems *ForHLR I*, *ForHLR II* and *Laptop*. ForHLR stands for research high performance computer (Forschungs Hochleistungs Rechner). Both ForHLR I and ForHLR II are large clusters of computers that are made accessible by the Steinbruch Centre for Computing (SCC) at Karlsruhe Institute of Technlogy (KIT) for research purposes. Both ForHLR clusters run the batch job system Slurm.

ForHLR I provides two login nodes where jobs can be submitted to Slurm. Additionally, it provides 512 "thin" nodes and 16 "fat" nodes which are connected with a InfiniBand 4X FDR Interconnect network, with a theoretical network bandwidth of 50 Gbit/s. This thesis uses the "thin" nodes for computation, since there are not enough "fat" nodes. Each "thin" node has two Deca-Core Intel Xeon E5-2670 v2 processors (Ivy Bridge) with a clock speed of 2.5 GHz (max. turbo 3.3 GHz), 64 GB of random access memory (RAM) and two 1 TB disks. Thrill is configured to use the local disks of each node. Each processor has 25 MB L3-Cache and 64 KB L1-Cache Each and 256 KB L2-Cache one each core. Each "thin" node has a theoretical top performance of 400 GFLOPS. The whole cluster runs Red Hat Enterprise Linux (RHEL) 6.x as the operating system. For compilation, the login node uses GCC 8.3.0 and OpenMPI 3.1.4 is used. A test writing 10 GiB of random `size_t` elements to disk and reading them back again yielded a maximum write speed of 192.812 MiB/s and a maximum read speed of 254.957 MiB/s. A test of the network bandwidth revealed a network speed of 1893.03 MiB/s. It was tested with an all-to-all communication of 1GiB on 16 "thin" nodes, with random `size_t` input.

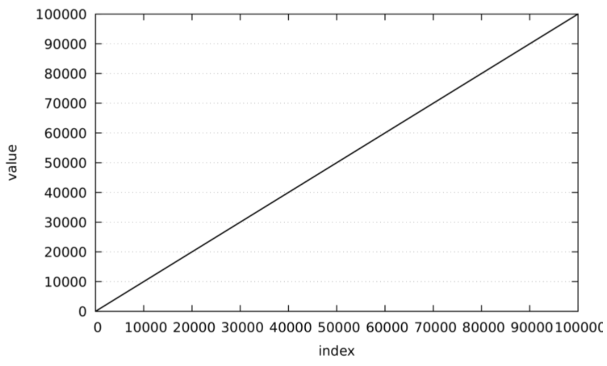
ForHLR II provides five login nodes where jobs can be submitted to Slurm. Additionally, it provides 1152 "thin" nodes and 21 "fat" nodes which are connected with a InfiniBand 4X EDR Interconnect network, which is capable of . This thesis uses the "thin" nodes for computation, since there are not enough "fat" nodes. Each "thin" node has two Deca-Core Intel Xeon E5-2660 v3 Prozessoren (Haswell) with a clock speed of 2.6 GHz (max. turbo 3.3 GHz), 64 GB of RAM and a 480 GB SSD. Thrill is configured to use the local disks of each node. Each processor has 25 MB L3-Cache and 64 KB L1-Cache Each and 256 KB L2-Cache one each core. Each "thin" node has a theoretical top performance of 832 GFLOPS. The whole cluster runs RHEL 7.x as the operating system. For compilation, the login node uses GCC 8.3.0 and Open MPI 3.1.4 is used. A test writing 10 GiB of random `size_t` elements to disk and reading them back again yielded a maximum write speed of 459.304 MiB/s and a maximum read speed of 559.238 MiB/s. A test of the network bandwidth revealed a network speed of 8926.75 MiB/s. It was tested with an all-to-all communication of 1GiB on 16 "thin" nodes, with random `size_t` input.

The Laptop system runs a Quad-Core Intel Core i7-5600U processor with a clock speed of 2.6 GHz (max. turbo 3.2 GHz), 16 GB of RAM and a 250 GB SSD. The processor has 4 MB of cache and runs Arch Linux. For compilation, GCC 8.2.1 and Open MPI 4.0.1 is used. However, this system is only used for non performance critical experiments.

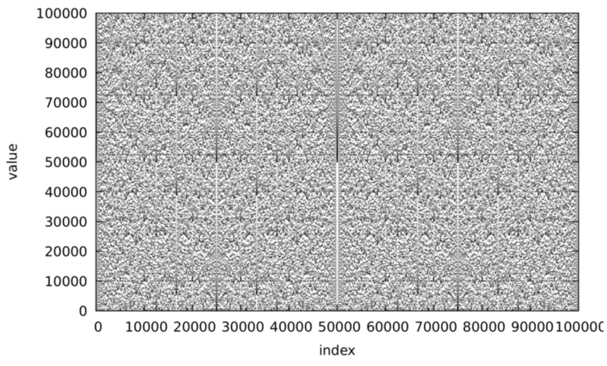
5.2 Data Generators



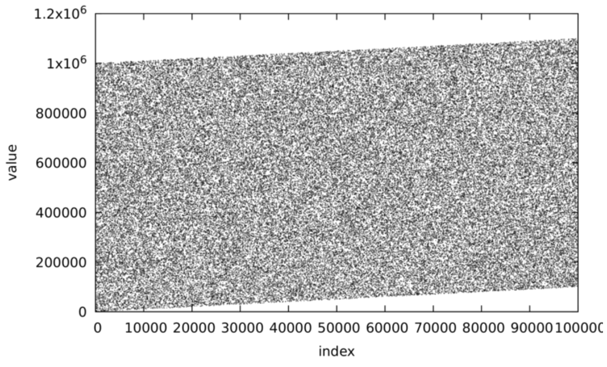
(a) Uniform.



(b) Sorted.



(c) TwoDup.



(d) Window.

Figure 26: Generator Output

For the following experiments, several different data generators will be used that cover a wide range of possible inputs. In these benchmarks, data is generated as a function of the current index and directly passed to the benchmark as single data points. The current index $i = IP + p\%P$ is dependent on the rank p of the processor that the data is generated on, where I is the local index and P is the number of processors. This index calculation ensures a striping of the input over the processors. The data set is not materialized beforehand. The generators have an index `i` as input parameter and return a `uint64_t` value. The following range of generators is inspired by [4]:

Uniform $generator(i) = random()$ where $random()$ returns a random value from a uniform distribution.

Sorted $generator(i) = i$.

TwoDup $generator(i) = (i^2 + n/2)\%n$ where n is the size of the input.

Window $generator(i) = random(0, s) + i$ where $random(0, s)$ returns a random value in the range $[0, s)$.

Figure 26 shows plots of the generator input for 100,000 elements. For the Window generator, $w = 1,000,000$. The x-axis plots the value and the y-axis plots the index.

5.3 Sampling

In section 4.3, two sampling techniques were discussed that are possible candidates for OSS. This section will compare the two approaches along for the inputs described in section 5.2 and along several scale factors. The following experiments were performed and are discussed in this section:

(Exp1) Convergence of sample to perfect sample for all data generators.

(Exp2) Error for input size to sample size ratio.

(Exp3) Histogram of ranks of complete sample for Reservoir and Online Sampling.

(Exp4) Error for parameters b and k for Online Sampling.

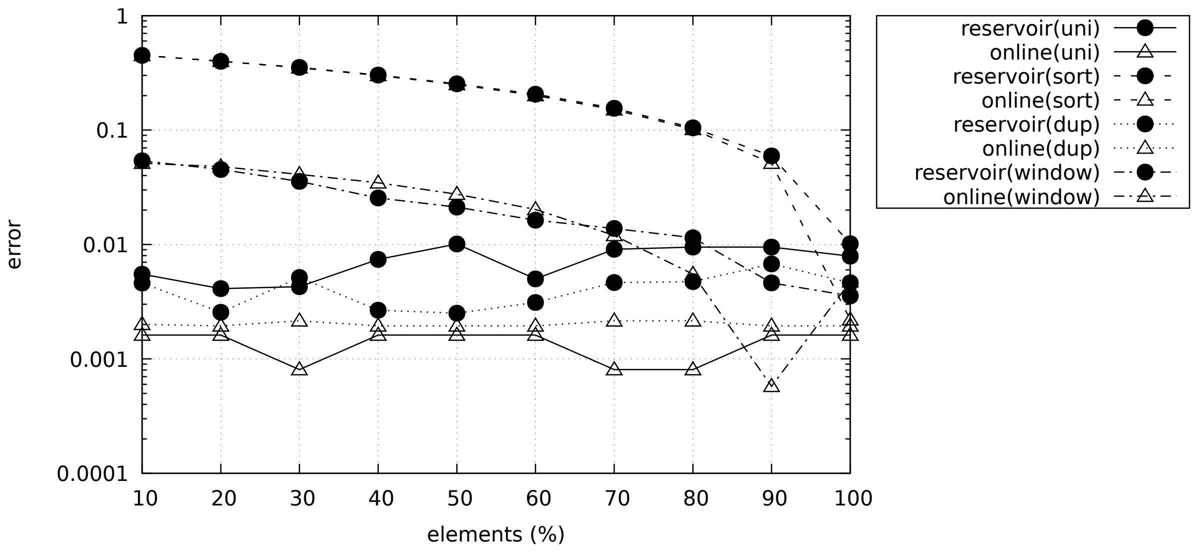


Figure 27: Sampling Convergence.

Figure 27 shows the sample convergence experiment Exp0 that was executed on the Laptop system. For this experiment $n = 64,000,000$ elements of the different data inputs were inserted into a Reservoir Sampler and an Online Sampler. The memory usage in elements of both samplers was $s = n/1000$. The Reservoir Sampler had a fixed reservoir of size 1600 on each processor and the parameter of the Online Sampler were $b = 4$ and $k = 400$ on each core. After every $n/10$th element, a sample was drawn. The generated sequence was stored and sorted afterwards. A window size w of $n * 10$ was chosen.

Every element of the 10 drawn samples was then ranked in the complete sequence and an error to the target rank was calculated. The target ranks are $i * n/s + n/2s$ for $i \in [0, s)$ The average of the sample rank to target rank error was then plotted on the y-axis for the amount of already processed elements on the x-axis. The y-axis is on a logarithmic scale so small differences in error on the low end of the scale are visible.

For the Sorted and Window generator, the error is very high at first and decreases linearly the more elements are processed. This is expected and cannot be improve much due to the nature of the input. The Online Sampler does not improve the Reservoir Sampler result by much. The Online Sampler however performs better on the Uniform and TwoDup data inputs. The Online Sampler does not show much deviation due to its deterministic nature. The Reservoir Sampler however shows deviation for both data inputs. Both samplers perform slightly better on the TwoDup data input. Both overall error and deviation are important factors for OSS since the redistribution of data in the run formation relies on it. Thus, Online Sampling wins this first experiment.

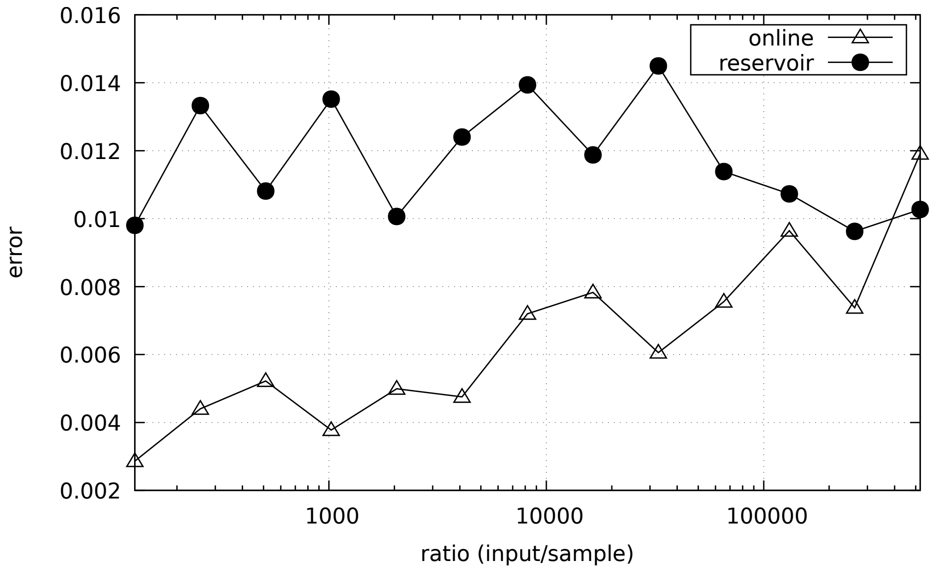


Figure 28: Sample Size Ratio and Error.

Another important scale factor of sampling is the ratio n/s of input size n to sample size s (Exp1). Figure 28 shows the result of sampling with different ratios. The experiment was done with ratios r of 128 to 2^{19} and five iterations per ratio. In each iteration, a Reservoir Sampler and a Online Sampler were created with parameters $b = 4$ and $k = 100$ and reservoir size 400 on each of the 4 processors. A sequence of $b * k * P * r$ elements was generated with the Uniform generator and inserted into the samplers. The average error of the samples was calculated the same way as in Exp0. The average error of the five iterations for each ratio r is plotted as the error on the y-axis. The ratio is plotted on the x-axis logarithmically. The experiment was conducted on the Laptop system.

The line with the dots in 28 plots the Reservoir Sampling error and the line with the triangles the Online Sampling error. The Online Sampling error is lower than the Reservoir Sampling error as expected for all but the last data point. The Reservoir Sampling error in contrast to the Reservoir Sampling error however shows no upward trend and thus being lower at a ratio of 2^{19}. To illustrate however how small the sample would be we assume a input size of 1 TB. The sample on all processors combined would be 2 MB for a ratio of 2^{19}, which is very small and can be much bigger for most real world applications. In conclusion, Reservoir Sampling is only better for large input sizes with very small sample sizes.

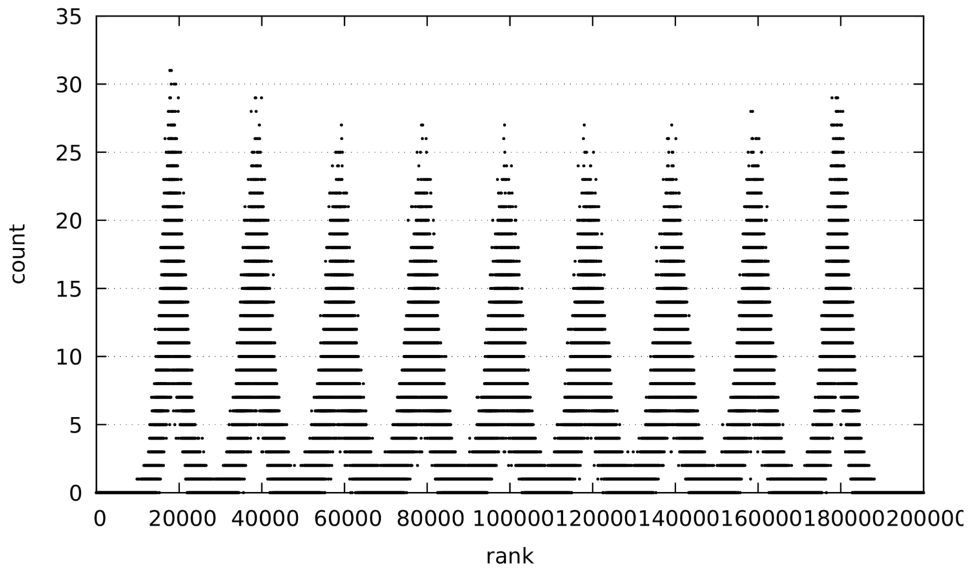


Figure 29: Online Sampling Histogram.

Figure 29 shows a histogram of 10 splitters that result from Online Sampling on $n = 200,000$ elements. Memory requirements for the Online Sampler are $n/1000$ with parameters $b = 2$ and $k = 25$ on each processor. This experiment also ran on the Laptop system. The Online Sampling process is executed $100,000$ times with a Uniform generator. After each iteration, a sample is drawn and ranked in the complete sequence. The rank counts of all iterations are summed up, such that ranks that appear in more iterations are higher on the x-axis. The x-axis plots the number of iterations a rank appeared in and the y-axis plots the rank. Higher peaks and closer clustering of the data points around each target splitter are better.

Figure 30 shows a histogram of 10 splitters that result from Reservoir Sampling on $n = 200,000$ elements. Memory requirements for the samplers are $n/1000$, with a reservoir of size 50 on each core. The experiment was executed on the Laptop system. The Reservoir Sampling process was executed $100,000$ times with a Uniform generator. The histogram is created and plotted the same way as figure 29.

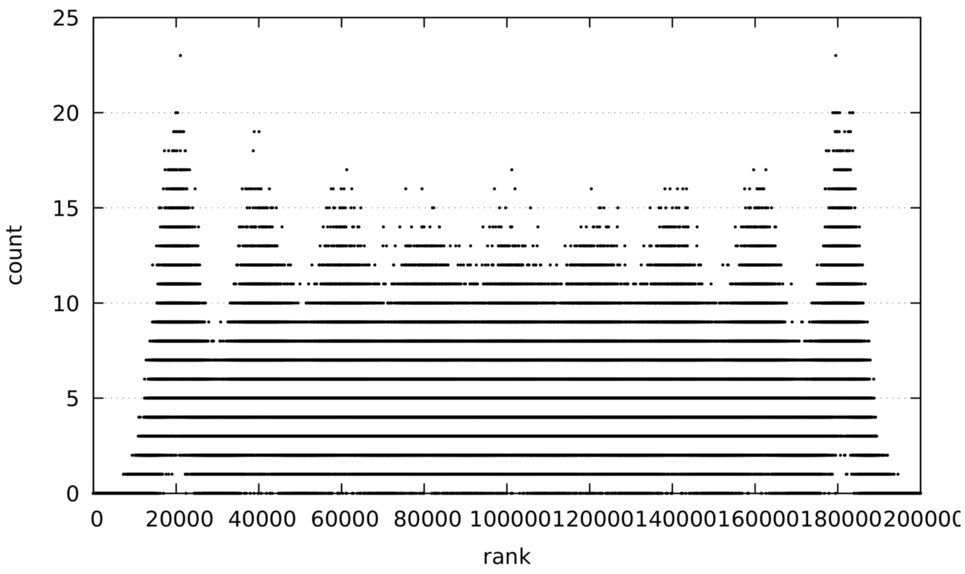


Figure 30: Reservoir Sampling Histogram.

As already observed in figure 27, the error and deviation visible in the two histograms is much lower for Online Sampling. However, the histograms show the magnitude of the difference well. Online sampling has ranks that appear as much as 35 times in 100, 000 iterations. The maximum for Online Sampling is 20 appearances. Furthermore, the deviations of adjacent splitters in the Online Sampling histogram overlap. Even though the mean error for Reservoir Sampling is low, the deviation will force much additional communication in OSS. Online sampling wins experiment Exp2 too.

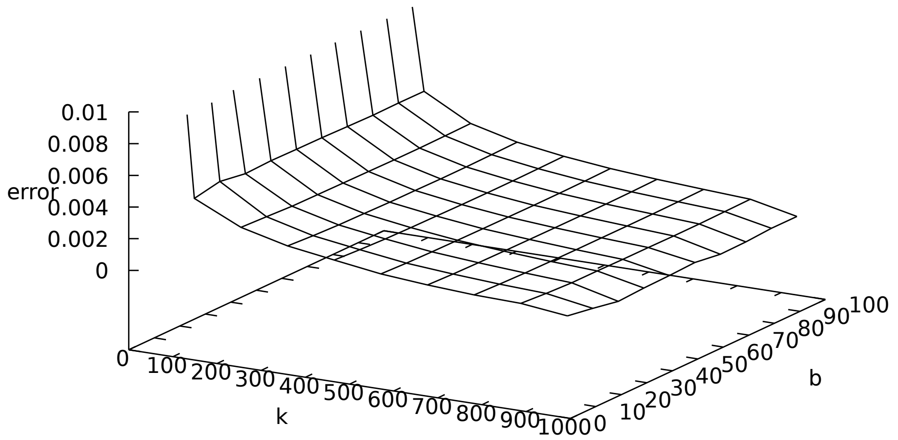


Figure 31: Online Sampling Parameter.

Exp0 to Exp2 prompt using Online Sampling for OSS. However, the best combination of parameters for Online Sampling is not clear. Figure 31 shows how the error of the resulting sample is influenced by the Online Sampling parameters b and k (Exp3). 200 combinations of parameters are tried out starting at 2 buffers for b and increasing up to 50 buffers in steps of 5. Buffer size k starts at 20 and is increased up to 1000 in steps of 50. For every combination, a Online Sampler is created and $b * k * p * 1000$ elements are inserted split up on $P = 4$ cores of the Laptop system. Thus, the size of the input is constant relative to the parameters b and k. After the data is inserted, a sample is drawn and the error is calculated as in Exp0. Buffer count b is plotted on the z-axis, buffer size k is plotted on the x-axis and the resulting sample error is plotted on the y-axis.

The experiment Exp3 shows how little impact a high number of buffers has on the total error. There should be at least 5 to 10 buffers on each processor, but more does not yield reduction in error. Buffer size however impacts the sample error much. Even very large buffers still yield error reduction. Buffer size should thus be prioritized when deciding upon parameters for the Online Sampler. This concludes contribution C3.

5.4 Sorting

After thoroughly comparing both sampling techniques, this section will compare the two sorting algorithms CMS and OSS from chapter 3 and chapter 4. CMS will act as a baseline algorithm that OSS is compared against. The two algorithms will be compared in the context of two scale factors, the different data generators, introduced in 5.2, and three performance measures. The scale factors are number of hosts and data per host. The performance measures are overall runtime, network communication and disk usage. This results in the following experiments:

(Exp5) Runtime for Uniform generator with scale factor hosts.

(Exp6) Amount of elements distributed to wrong processor.

(Exp7) Runtime for different generators.

(Exp8) Runtime for Uniform generator with scale factor amount of data per host.

(Exp9) Runtime of OSS with randomization.

In [23], where CMS was presented, experiments were done with 100 GB per host and a struct of key and value ($uint64_t key, uint64_t value$) as the element type of the input sequence. Element type will not be considered as a scale factor in this thesis and thus this struct will be used for all following experiments. 100 GB per host will be used as the baseline for all experiments.

Figure 32 shows experiment Exp5 which considers the scale factor number of hosts. The x-axis plots the different instances of the experiment, which are named after the abbreviation of the algorithm followed by the number of hosts in brackets. The y-axis plots the runtime in minutes. In a range of 4 to 64 hosts in steps of power of 2, runtimes for both CMS

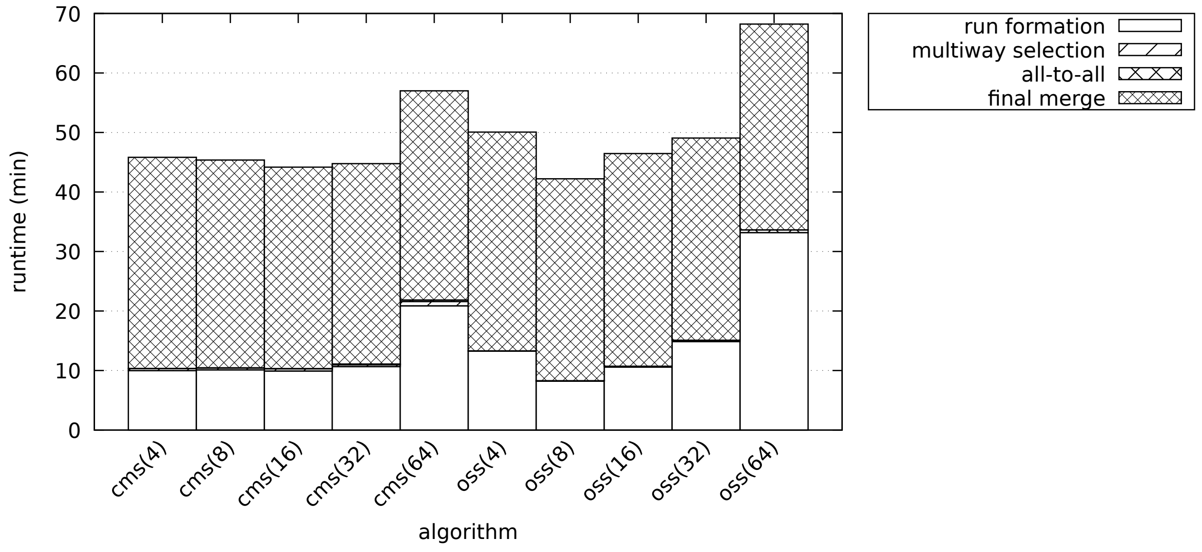


Figure 32: Runtimes.

and OSS are plotted. 100 GB of data per host and Uniform generator input data are fixed for all instances of the experiment. The experiment was done on the ForHLR I system. Each instance is split up into the phases of the algorithms. All-to-all refers to the communication in the correction phase, because OSS has no MSS step in the correction phase and [23] split up the runtime the same way.

The runtime is constant from 4 to 32 hosts and has a jump in run formation runtime for 64 hosts. The correction phase contributes a negligible runtime to the overall runtime. The final merge phase is constant in the number of hosts, which is expected, since it does not need any network communication and with a fixed amount of input data per host has to merge the same amount of data each time.

CMS is slightly faster than OSS on all combinations but 8 hosts by a very small constant factor, since using local knowledge of the runs for redistribution yields sufficiently good results for Uniform data input. The correction phase takes slightly longer for all of the CMS experiments due to the MSS and more redistribution. The final merge phase takes the same runtime for both CMS and OSS, which is expected, since the implementations are the same. The runtime advantage comes from a shorter run formation phase.

Figure 33 shows a detailed breakdown of the run formation phase. The numbers cannot be considered absolutely accurate, because the steps in each run formation overlap and are hard to separate, but it shows possible improvements and weaknesses of OSS. The x-axis again plots the instance of the experiment and the y-axis plots the runtime in seconds. The underlying data is coming from experiment Exp5. The local sort is a constant factor in all instances of the experiment, since the amount of data on each host is fixed. The sampling in OSS scales worse than the MSS in CMS. The sampling takes up approximately 10% of the total runtime. Ways of speeding up the sampling will be discussed later in Exp10. For

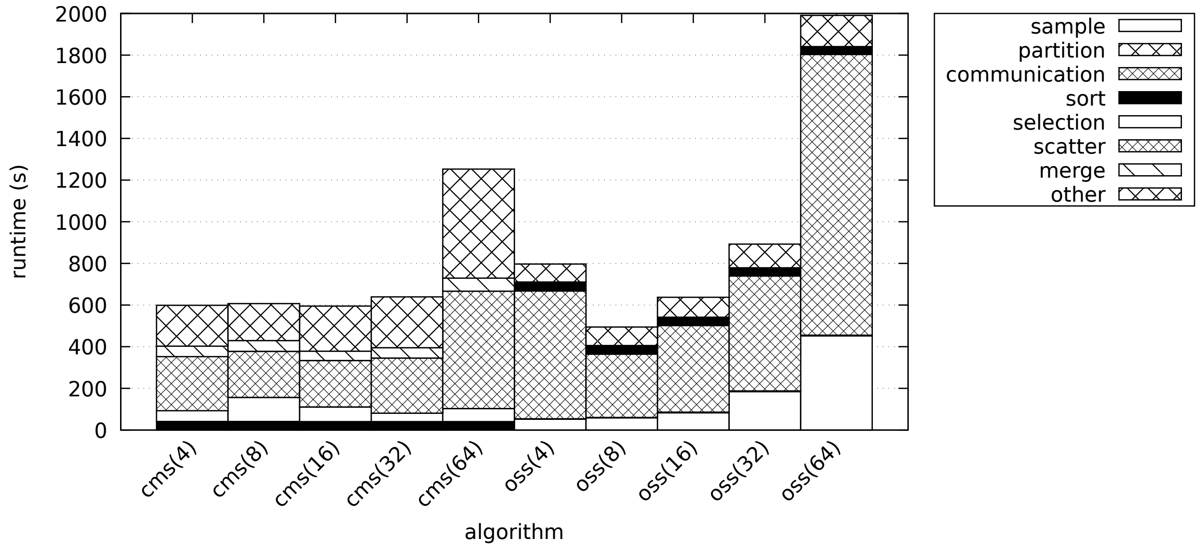


Figure 33: Run Formation Runtime.

OSS, the plot shows that partitioning and communication are overlapping heavily, such that some of the communication runtime should be considered as partitioning runtime. The rest of the CMS and OSS runtime can be considered as disk usage.

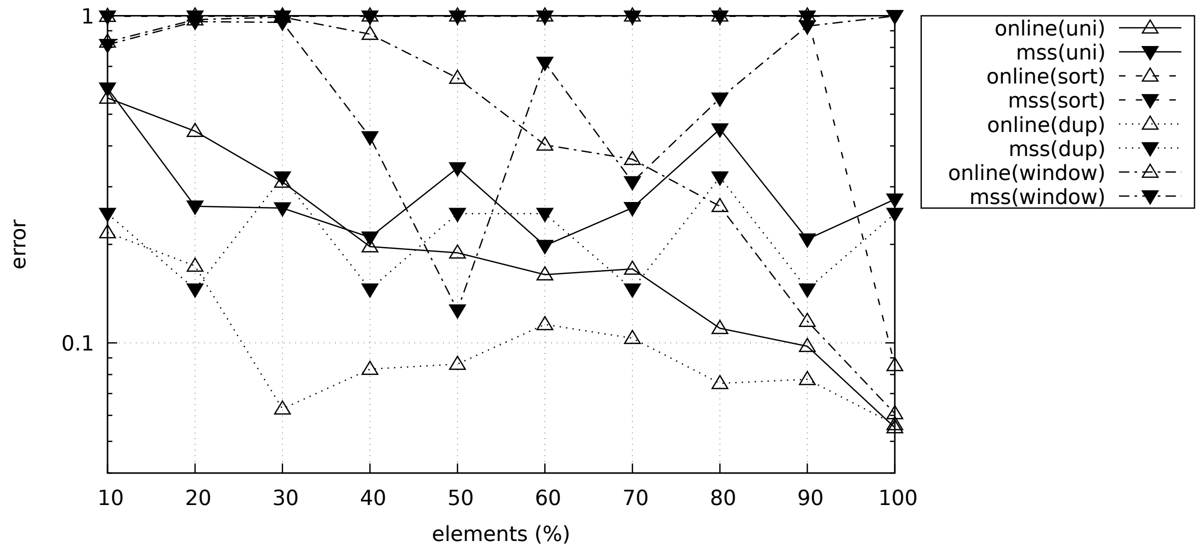


Figure 34: Amount of Elements Distributed to Wrong Processor.

Figure 34 (Exp6) shows what percentage of elements are redistributed to the wrong processor for different generator data sets, comparing MSS and Online Sampling results. This was simulated on the Laptop system with $n = 32,000,000$ elements and memory requirements for the sampler $n/1000$. Parameters of the Online Sampler are $b = 4$ and $k = 2000$. The window size for the Window generator is $w = 10n$. The number of splitters was 300, which matches a system with 16 hosts on one of the ForHLR systems. 10 runs were simulated.

The error was calculated as the share of elements of each run that reside on the wrong processor at the end of the run formation phase. 1 indicates that all elements of a run have been redistributed to the wrong processor and 0 indicates that all elements of a run have been redistributed to the correct processor. The x-axis plots the 10 runs with the amount of elements already processed up until the end of each run. The y-axis plots the error on a logarithmic scale.

All generators start at the same error in the first run for both MSS and Online Sampling. Uniform input for MSS shows deviation around error of 0.3 and is beaten by the error of the Online Sampling, which shows a downward trend and finishes at around error of 0.06. Errors for the TwoDup input are lower, but the relation of MSS and Online Sampling are similar. MSS shows deviation and Online Sampling shows a trend.

The error for the Window generator with Online Sampling is decreasing linearly, while the MSS shows a dip in the middle and goes up at the end again. The overall error for the sampling is much lower however. The error for the Sorted generator input data is almost the same for all but the last run, where the sampling has almost optimal redistribution as expected. This however is a minor improvement, especially as amount of runs increase with amount of data per host.

The error for the Sorted generator input data is much higher than would be initially suspected with the results from figure 27 in mind. The splitter error decreases linearly, while the redistribution error stays very high. Splitter error and redistribution error are thus not necessarily linked.

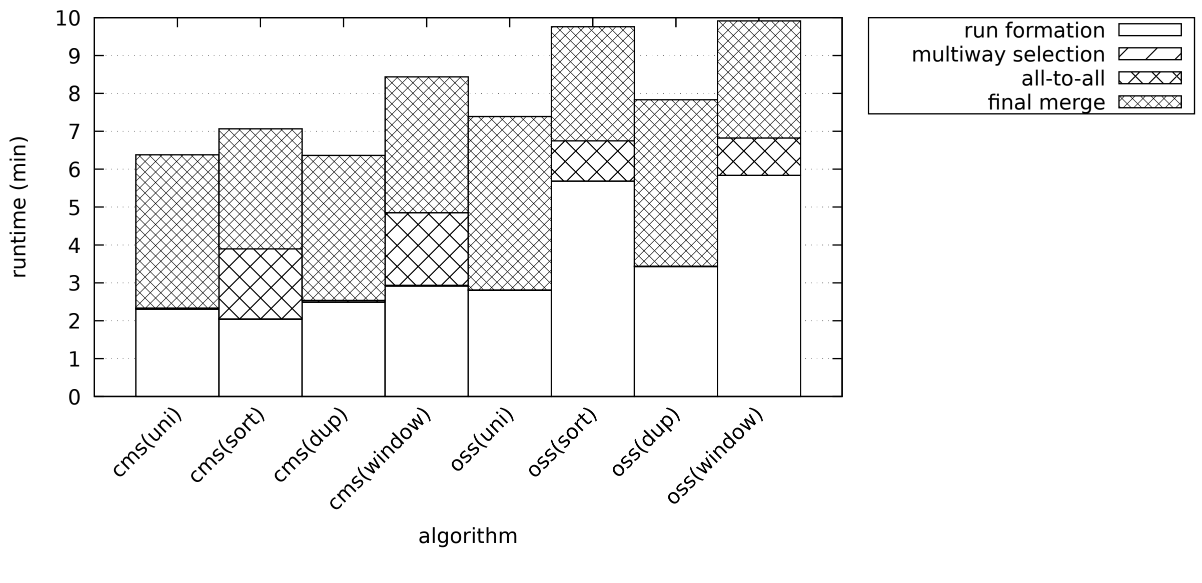


Figure 35: Runtimes for Different Generators.

Figure 35 shows the runtime of CMS and OSS for the different generators (Exp7). The x-axis plots the sorting algorithms with generator type in brackets. The y-axis plots the runtime in minutes. The experiment was performed on ForHLR II, with 4 hosts and 50

GiB per host. The Uniform data generator and the TwoDup data generator show similar results for each sorting algorithm. OSS again is slower, resulting from increased runtime in the run formation phases. The correction phase again takes negligible time. Runtime for Sort and Window generators however show promising results. The run formation phase takes considerably longer, however the correction phase for OSS is only half of the correction phase for CMS, which means that less data is transferred.

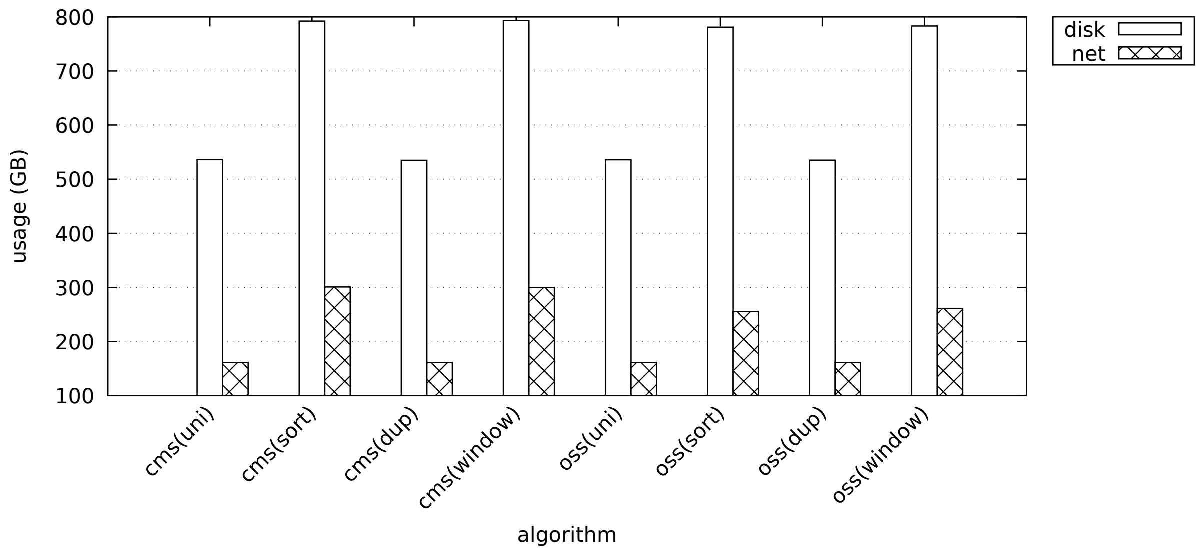


Figure 36: Total Disk and Network Usage.

Figure 36 shows the disk and network usage of experiment Exp6. The x-axis shows the sorting algorithms with data generator in brackets. The y-axis plots the usage in GB. Disk and network usage are the same for Uniform and TwoDup data generators for both sorting algorithms, this suggest no improvement in redistribution quality. Usage for the Sort and Window generator however are lower for OSS than for CMS. This suggests an slight improvement of redistribution quality.

Figure 37 shows the runtimes of both sorting algorithms under consideration of the scale factor amount of data per host (Exp8). The x-axis plots the sorting algorithms with amount of data per host in brackets. The y-axis plots the runtime in minutes. The experiment was performed on the ForHLR II system with 16 hosts and the Uniform generator data. The input size per host was scaled from 50 GiB to 200 GiB. The runtimes scale linearly in this experiment. OSS outperforms CMS on 200 GiB per host, which suggests that OSS better distribution shows impact on larger data sets.

Figure 38 shows the impact of randomization on the overall runtime and run formation run times (Exp9). The experiment was performed on the ForHLR II system with 16 hosts and 100 GiB of Uniform data generator. The x-axes plot the sorting algorithms with randomization factor r in brackets. The y-axes plot the runtime in minutes and seconds respectively. Randomization factor $r = 1$ means no randomization. The experiment shows

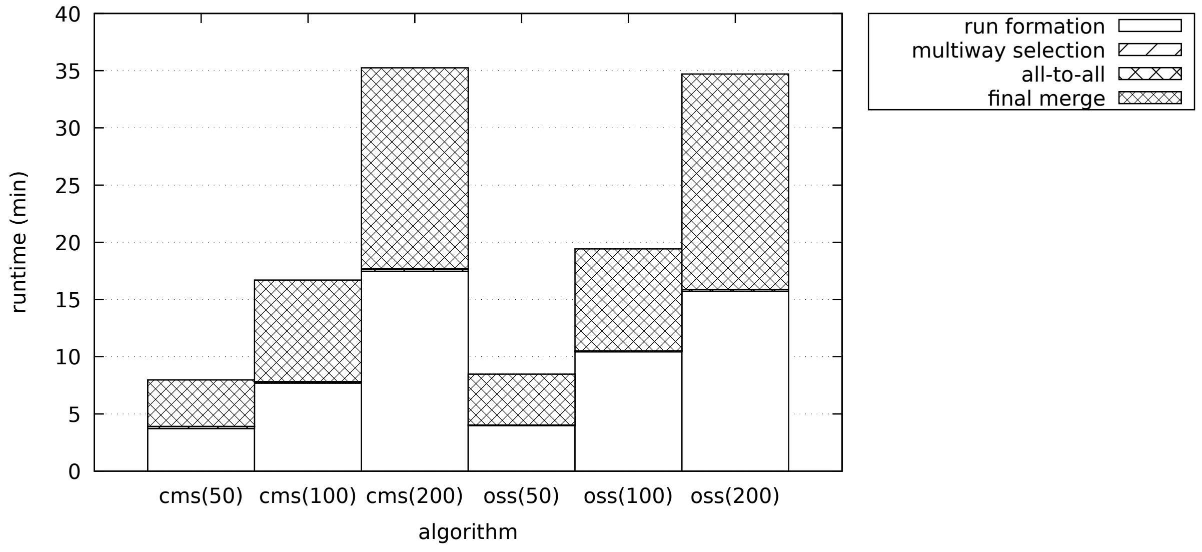


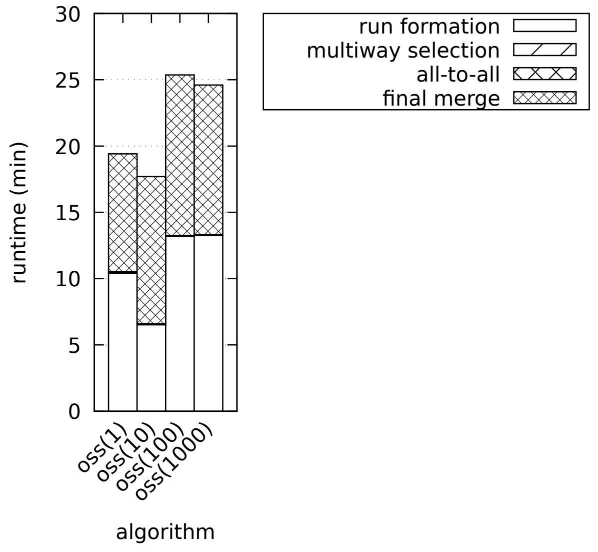
Figure 37: Runtimes for Amount of Data per Host.

that too high randomization factors make the run formation runtime longer, however moderate randomization factors can improve the run formation runtime. A randomization factor of $r = 10$ improves the runtime of the run formation by 33%. In the context of the total runtime, this is still 10%. This improves the OSS runtime over the CMS runtime significantly.

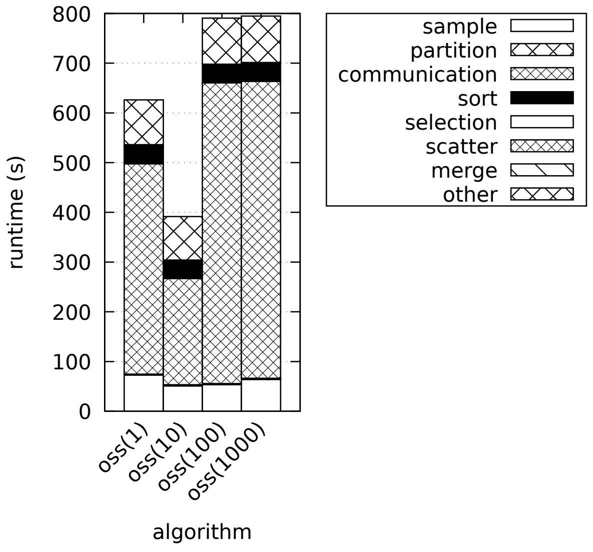
To conclude the results, profiles for CMS and OSS are shown to illustrate the different phases of the algorithms and when they are communication and disk bound. Figure 39 shows the system profile for CMS. The plot is from a run of CMS on ForHLR II, with 16 hosts (304 processors), the Uniform data generator and 100 GB of data per host.

The data set is processed in 10 runs in the run formation, which can be clearly seen from the 10 CPU spikes in the first half of the plot. Each spike corresponds to the local sort and MSS in each run formation iteration. Thereafter, the network traffic spikes to its maximum capacity, bounding the CPU. Shortly after the networks starts spiking, the disk writing spikes to the disk maximum speed, because elements from the previous run are written to disk, making space for the new elements that are merged into a file.

At roughly 470 seconds, the CPU spikes again for the MSS on all run files and a very short redistribution corrects the last elements residing on the wrong processor. The correction phase on Uniform generator input however only takes a few seconds, before the final merge phase starts. The last phase starts with a disk read spike, because, at the beginning of the merge, no elements are written to disk, but still cached in blocks in memory. When the first blocks are written to disk, read and write speed of the disks match at a combined maximum speed for the disks. At the end of the merge, when all remaining elements fit into memory, the CPU usage goes up and finishes the merge. There is no network communication in the final phase of the algorithm.



(a) Runtime.



(b) Run Formation.

Figure 38: Online Sample Sort Randomization.

Figure 40 shows the corresponding profile for OSS. The plot is from a run of OSS on ForHLR II, with 16 hosts (304 processors), the Uniform data generator and 100 GB of data per host. A detailed breakdown of this run can again be seen in figure 32 and figure 33.

Since both sorting algorithms are configured to use the same run size, the profile also shows 10 CPU spikes for 10 runs. However, network traffic, disk write and CPU usage stand in different relations to each other. OSS runs start with a small CPU and network spike for the sampling process and proceed with more network communication, since the partitioning and redistribution is network bound. When all data is redistributed, the CPU spikes for the local sort. The disk write of each run and the next sampling step overlap with each other.

The correction phase starts at around 450 seconds, but is even shorter than with CMS, because the samples are already known. The final merge again starts with a spike of disk read and is bound by disk speed until all remaining elements fit into memory. There is again no network communication in the final phase of the algorithm. This concludes contribution C4.

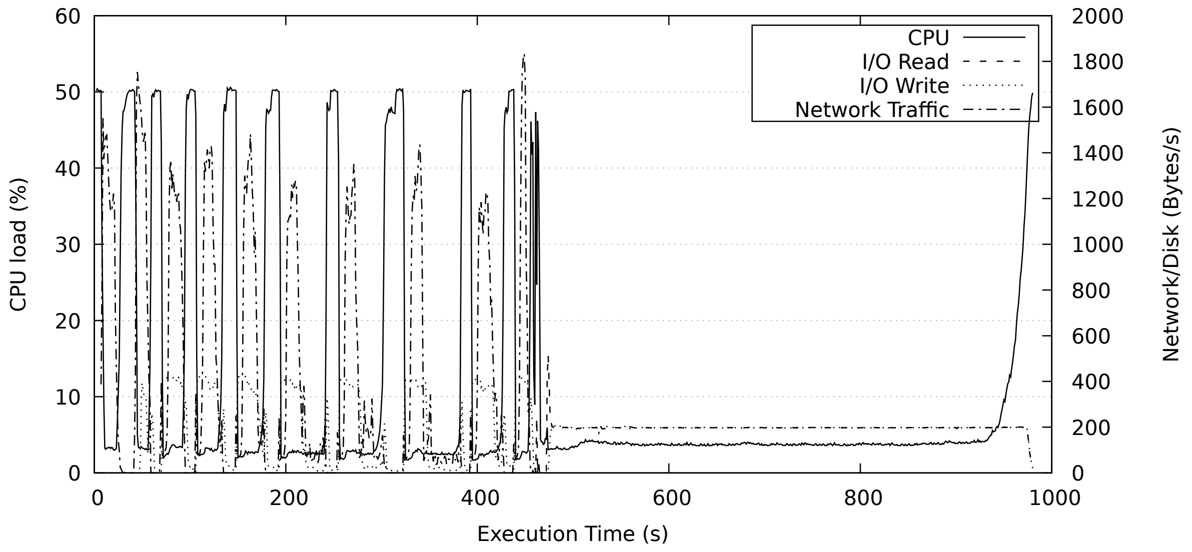


Figure 39: Canonical Merge Sort Profile.

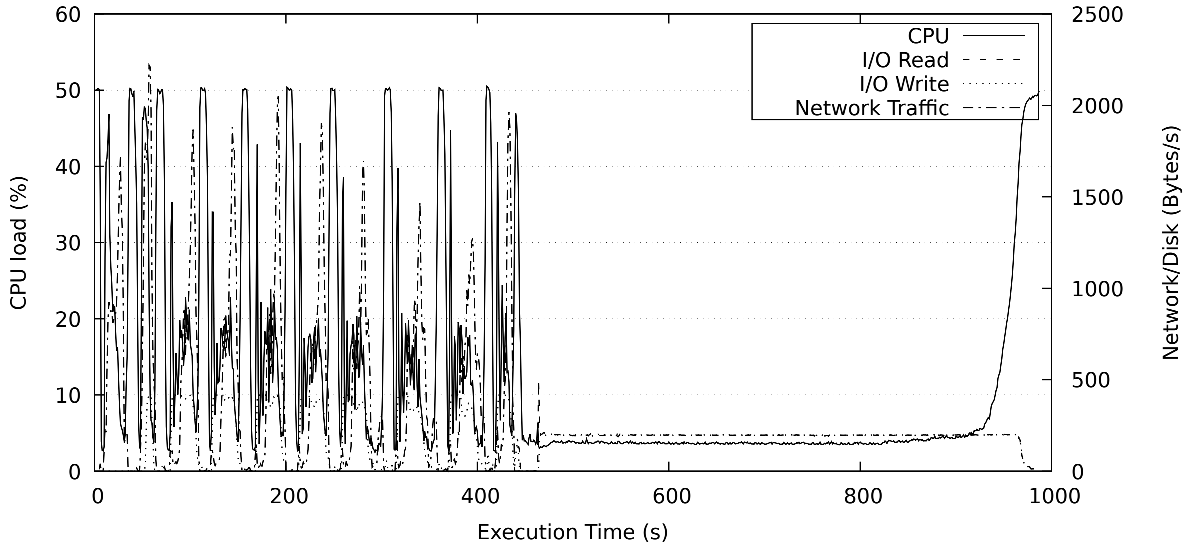


Figure 40: Online Sample Sort Profile.

6 Conclusion

6.1 Discussion

This thesis proposed a new distributed external sorting algorithm OSS (contribution C2). The algorithm is inspired by CMS, which was implemented and optimized in Thrill as a baseline state of the art distributed external sorting algorithm (contribution C1). Reservoir Sampling and Online Sampling were presented as sampling techniques for the implementation of OSS (contribution C3). The thesis was concluded with an evaluation of the sampling techniques and experiments with the sorting algorithms OSS and CMS (contribution C4).

The evaluation results highlighted Online Sampling as the superior sampling technique. It improves Reservoir Sampling in sample error and especially robustness and deviation of samples. Reservoir Sampling however remains unbeaten for very large input size to sample size ratios. Furthermore, it is not clear how number of buffers b for Online Sampling improves the error of the sample. It is unnecessary to use more than a few buffers.

The sorting experiments presented OSS as a distributed external sorting algorithm with similar runtime to CMS. OSS is faster for the Sorted and Window generators and with randomization. With randomization and a resulting low runtime of sampling, OSS shows lower constant factors than CMS, as expected. However, the fast improvement in data redistribution quality did not materialize. *This is due to splitter error and redistribution error not necessarily being linked.* The splitter error can improve without the redistribution error improving much, which was not expected. Overall, OSS is the faster algorithm when initialized with the correct parameters.

The code is available in the Thrill GitHub repository at `https://github.com/thrill/thrill`.

6.2 Outlook

The contributions in this thesis have potential to be further investigated. Possible advancements will be presented in the following. CMS and OSS could be tested and optimized for massively parallel systems as in [2]. Both CMS and OSS are bound by memory in some part of the algorithm and would not scale to massively parallel systems. The initialization step of MSS in CMS is bound by memory, where the samples from all processors have to fit into the memory of processor p_0. Online Sampling is bound by memory on processor p_0 too, since buffers and partial buffers from all processors have to fit into memory. Lastly, neither algorithm will scale to very large amounts of data per host. The `SampledFile` samples (one for each block) have to fit into memory. Another advancement is possible in sampling techniques. Reservoir Sampling and Online Sampling were two choices for sampling because of their popularity. It may however be possible to draw samples that are even better.

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical Massively Parallel Sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 13–23, New York, NY, USA, 2015. ACM.

[3] Michael Axtmann and Peter Sanders. Robust Massively Parallel Sorting. *arXiv:1606.08766 [cs]*, 2016.

[4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Timo Bingmann, Michael Axtmann, E. Jöbstl, Sebastian Lamm, H. C. Nguyen, A. Noe, Sebastian Schlag, M. Stumpp, T. Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 172–183, 2016.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[7] Dehne, Dittrich, and Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36(2):97–122, 2003.

[8] Dehne, Dittrich, Hutchinson, and Maheshwari. Bulk synchronous parallel algorithms for the external memory model. *Theory of Computing Systems*, 35(6):567–597, 2002.

[9] Roman Dementiev and Peter Sanders. Asynchronous Parallel Disk Sorting. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 138–148, New York, NY, USA, 2003. ACM.

[10] David. J. DeWitt, Jeffrey F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291, 1991.

[11] Farzad Farnoud (Hassanzadeh), Eitan Yaakobi, and Jehoshua Bruck. Approximate Sorting of Data Streams with Limited Storage. In Zhipeng Cai, Alex Zelikovsky, and

Anu Bourgeois, editors, *Computing and Combinatorics*, Lecture Notes in Computer Science, pages 465–476. Springer International Publishing, 2014.

[12] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in x + y and matrices with sorted columns. 24(2):197–208, 1982.

[13] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. 22(2):251–267, 1994.

[14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, 1999.

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

[16] Richard M. Karp. Streaming Algorithms for Selection and Approximate Sorting. In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, pages 9–20. Springer Berlin Heidelberg, 2007.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[18] Gurmeet S. Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 426–435. ACM, 1998.

[19] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 251–262. ACM, 1999.

[20] Claudia Misale, Maurizio Drocco, Marco Aldinucci, and Guy Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(1):1740003, 2017.

[21] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 120–129, New York, NY, USA, 1993. ACM.

[22] Mark H. Nodine and Jeffrey Scott Vitter. Greed Sort: Optimal Deterministic Sorting on Parallel Disks. *J. ACM*, 42(4):919–933, 1995.

[23] Mirko Rahn, Peter Sanders, and Johannes Singler. Scalable distributed-memory external sorting. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 685–688, 2010.

[24] Peter Sanders and Sebastian Winkel. Super Scalar Sample Sort. In Susanne Albers and Tomasz Radzik, editors, *Algorithms – ESA 2004*, Lecture Notes in Computer Science, pages 784–796. Springer Berlin Heidelberg, 2004.

[25] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[26] Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging Multiple Lists on Hierarchical-memory Multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):171–177, 1991.

[27] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, i: Two-level memories. *Algorithmica*, 12(2):110–147, 1994.

[28] Jeffrey S. Vitter. Random sampling with a reservoir. 11(1):37–57, 1985.

[29] Jeffrey S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.