Load Balancing for MPI-Based MapReduce

Bachelor's Thesis

Paulina Reichle

Date: 17.03.2025

Reviewer: Prof. Dr. Peter Sanders

Advisors: M.Sc. Matthias Schimek
M.Sc. Tim Niklas Uhl

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

MapReduce is a well-known programming model for processing large amounts of data in a parallel manner. The popularity of MapReduce frameworks is due to them handling the complications that come with parallel computations, such as communication between processes and ensuring load balancing. MapReduce works by mapping the input data to key-value pairs, grouping these by key, and then reducing all values of a key. MapReduce does not give guarantees concerning load balancing, but skewed data can lead to key-value pairs being unevenly distributed amongst processes during the execution of MapReduce, which is inefficient. Much real-world data is skewed, which makes load balancing an important issue. In this thesis, we develop, to the best of our knowledge, the first MPI-based MapReduce library with load balancing, thereby facilitating the interoperability of MPI and MapReduce. We implement a load-balancing algorithm for MapReduce proposed by Sanders.

Our load-balancing algorithm uses a combination of prefix sums and hashing to calculate a better distribution of data during the execution of MapReduce. The sizes of keys are determined by aggregating their local sizes after the map phase with hashing, and then a prefix sum is used to calculate which process will handle the reduction of which key. We tested our library using multiple MapReduce applications to gauge its usability under various circumstances.

Our load-balancing algorithm achieves up to 39% less deviation of the maximum number of key-value pairs allocated to a process after the shuffle phase from a completely even distribution. When the input data is skewed and the application spends a significant time of its execution in the reduce phase, this leads to a reduction in execution time of up to 22%.

Deutsche Zusammenfassung

MapReduce ist ein bekanntes Programmiermodel, das für das parallele Verarbeiten großer Datenmengen verwendet werden kann. Die Popularität von MapReduce Frameworks liegt darin begründet, dass sie die komplizierten Feinheiten die mit paralleler Programmierung einhergehen, wie z.B. die Kommunikation zwischen Maschinen oder das Sicherstellen von Lastbalancierung, dem Benutzer abnehmen. MapReduce nimmt die eingegebenen Daten und mappt sie zu Schlüssel-Wert-Paaren, gruppiert diese nach Schlüssel und führt schließlich für jeden Schlüssel eine Reduktion auf allen dazugehörigen Werten aus. MapReduce gibt keine Garantien bezüglich Lastbalancierung, aber unbalancierte Eingabedaten können dazu führen, dass die Schlüssel-Wert-Paare während der Ausführung von MapReduce ungleichmäßig zwischen den Prozessoren verteilt sind, was ineffizient ist. Da viele reale Datenmengen unbalanciert sind, ist Lastbalancierung ein wichtiges Thema. In dieser Bachelorarbeit, entwickeln wir eine MPI-basierte MapReduce Library mit Lastbalancierung und vereinfachen damit die Interoperabilität von MPI und MapReduce. Wir implementieren eine Lastbalancierungsalgorithmus für MapReduce eingeführt von Sanders.

Unser Lastbalancierungsalgorithmus benutzt eine Kombination von Präfixsummen und Hashing um eine bessere Verteilung der Daten während der Ausführung von MapReduce zu berechnen. Hierfür bestimmen wir die Größen der Schlüssel nach der Map-Phase und benutzen eine Präfixsum um zu berechnen welcher Prozesssor die Reduktion von welchem Schlüssel behandelt. Wir testen unsere MapReduce Library mit verschiedenen MapReduce Anwendungen, um die Verwendbarkeit unserer Library für verschiedene Parameter beurteilen zu können.

Unser Lastbalancierungsalgorithmus erreicht eine besser Lastverteilung von bis zu 39% weniger Abweichung von der optimalen Verteilung. Wenn die Eingangsdaten unbalanciert sind und die Anwendung einen nicht unerheblichen Teil ihrere Ausführung in der Reduce-Phase verbringt, führt dies zu einer Verringerung der Laufzeit um bis zu 22%.

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Karlsruhe, den 17.03.2025

Contents

1 Introduction

MapReduce is a programming model [6] developed to process large amounts of data in a parallel manner. It is based on the realisation that a lot of computations for processing large datasets can be expressed through a map function followed by a reduce function or, for more complex computations, multiple iterations of map and reduce. Those large computations are usually executed in parallel on multiple machines as the datasets can be too large for the memory of a single machine to handle. An implementation of MapReduce also handles all the extra work that comes with executing a program in parallel, e.g. inter-machine communication. Therefore, MapReduce makes it easier to implement computations for processing big amounts of data, as it hides the details of parallelisation. Big Data plays an important role in today's world, and the rise of electronic devices everywhere increases the amount of data at our disposal [3]. This makes it all the more important to have efficient programs to analyse and process these large datasets.

A range of MapReduce implementations, such as Hadoop, Spark, MR-MPI, and Mimir already exist. Each one implements the MapReduce programming model a little differently and refines the original model in different ways. These refinements include, amongst other things, fault tolerance or building a MapReduce library well-suited for running on high-performance computers often used in academic settings.

Load balancing is another possible refinement as the MapReduce programming model does not offer any guarantees concerning load balancing. Using load balancing for MapReduce is important, however, because a lot of real-world data is skewed, e.g. the graph topology of the internet fits in with the power law [8]. This means vertices have different degrees, and, in particular, there are some vertices with a very high degree. Using a graph like this as input for a MapReduce application will often lead to an inefficient execution because of an uneven distribution of data.

1.1 Contribution

The MapReduce library we develop, in contrast to the already existing ones, focuses on the novel load-balancing principle laid out in the paper 'Connecting MapReduce Computations to Realistic Machine Models' [25] by Sanders. Our library is, to the best of our knowledge, the first MPI-based MapReduce library with load balancing. Sanders proposes two main algorithms to achieve load balancing. We focus on one of the two, which uses hashing and prefix sums to achieve a better distribution of key-value pairs after the shuffle phase. We create a library which can process data according to the MapReduce programming model. We implement a standard hash-based shuffle and a shuffle using load balancing. Our load-balancing shuffle achieves a balanced distribution of data using a combination of hashing and prefix sum. It distributes the intermediate data produced by the map phase evenly in preparation for the reduce phase. We then compare our two implementations for three different applications: WordCount, PageRank and k-means clustering. Here, we evaluate our implementation regarding execution time and distribution of values after the shuffle phase. We also compare our library to the MR-MPI library.

1.2 Structure of Thesis

The following chapter 2 introduces the MapReduce programming model and two models of computation for MapReduce. Chapter 3 provides an overview of other MapReduce implementations and previous research into load balancing for MapReduce. Chapter 4 follows with explaining the theoretical algorithms behind our implementation and how we adapted them while implementing them in practice. Then, we explain the applications which we use to test our library in chapter 5. Chapter 6 lays out our experiments and our evaluation of our library based on these experiments. Finally, we conclude our research in chapter 7.

2 Preliminaries

2.1 MapReduce

Dean and Ghemawat [6] proposed the MapReduce programming model to handle processing big amounts of data. Without MapReduce, they implemented many distributed computations for processing data. As all these computations were for distributed systems, each time they had to handle issues, such as distributing the data, communication between machines, fault tolerance and load-balancing. Even when a computation is straightforward, this leads to a lot of work. This is where a MapReduce library is useful because it hides these details, and the user only needs to specify the computation.

MapReduce is based on the realisation that many computations can be split into a map and a reduce phase. With this abstraction in place, a library can be built, which takes as user input a map and a reduce function, as well as the input data. The framework handles all the scheduling of tasks and communication between machines. This means the user can quickly create applications for processing data without needing to reimplement common issues, such as communication, fault tolerance, etc.

Each MapReduce iteration consists of two computational phases: map and reduce. The user of a MapReduce library sets a map function and a reduce function, which will be applied to the data in their respective phases. During the map phase, each input element is passed to the user-provided map function, producing a set of intermediate key-value pairs. Then, those intermediate pairs are reordered and grouped together. All pairs with the same key are grouped together. Dean and Ghemawat [6] do not name this part of a MapReduce iteration, but it is commonly referred to as the shuffle phase and will be referred to as such in this thesis. During the reduce phase, the user-provided reduce function is applied to a key and all its corresponding values. Usually, the output of a reduce phase can be used as input for a map phase. This way multiple MapReduce iterations can be chained.

Figure 2.1 shows a simple WordCount MapReduce application. Each process receives a string as input and splits it into its individual words during the map phase. For each word, a key-value pair is created. Then, the key-value pairs are sent to the processes in such a way that all pairs with the same key are sent to the same process. Finally, during the reduce phase, for each key (here: word), the number of values (here: occurrences) are summed up.

2.2 *MRC*

The MapReduce class *MRC* is a theoretical model of computation. It was created by Karloff et al. [18] for developing efficient algorithms using the MapReduce paradigm. For an algorithm to be considered to be in *MRC*, each call to a map or reduce function must

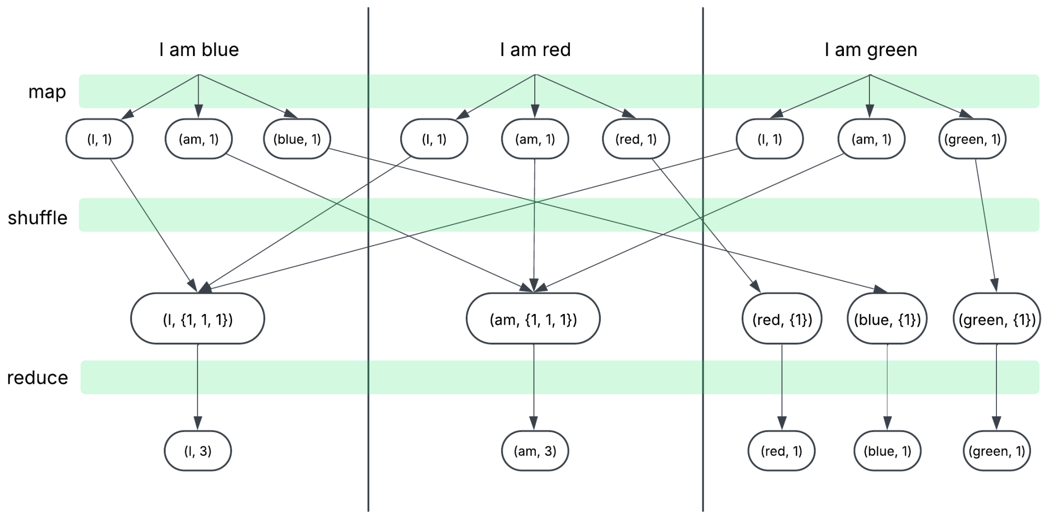

Figure 2.1: A simple MapReduce example depicting a WordCount application.

be sublinear in space ($O(n^{1-\epsilon})$) and polynomial in time. The total space needed must be subquadratic ($O(n^{2-2\epsilon})$), and the entire execution is only allowed to take a logarithmic amount of iterations, i.e. be within $O(\log^i n)$ number of iterations. The algorithm only needs to produce a correct solution with a probability of at least $\frac{3}{4}$. The number of available machines must be substantially sublinear in data size.

2.3 MRC^+

Algorithms, which are efficient in the MRC model, are not always efficient in practice if they take advantage of the full leeway of the model. Some loopholes of the MRC model are, e.g. not requiring algorithms to have speedup over sequential algorithms and allowing the algorithm to take up quadratic space. This is why Sanders proposes an adapted model MRC^+ [25]. MRC^+ is better at predicting efficiency and scalability. This is achieved by also taking into account work and communication volume. For this, the following parameters are needed:

- p: number of PEs
- μ: map function
- ρ: reduce function
- w : total time to evaluate μ and ρ on their input
- $\hat{w}$: maximum time needed by a call of μ or ρ
- m: total number of machine words throughout an iteration
- $\hat{m}$: maximum number of machine words passed as an input to one call of μ or ρ

With these parameters, the following theorem is formed

Theorem: If all PEs store $O(m/p + \hat{m})$ words of the input, then one MapReduce iteration can be implemented with expected local work and bottleneck communication volume [25]

$$\Theta\left(\frac{w}{p} + \hat{w} + \log p\right) \text{ and } \Theta\left(\frac{m}{p} + \hat{m} + \log p\right) \tag{2.1}$$

2.4 Prefix Sum

Let a be an array of numbers and a_i be the number at index i in a. The inclusive prefix sum $\hat{x}$ is defined as $\hat{x}_j = \sum_{i=0}^{j} a_i$ and the exclusive prefix sum x is defined as $x_j = \sum_{i=0}^{j-1} a_i$ with $x_0 = 0$. The prefix sum is, therefore, the sum of all previous values, either inclusive or exclusive of the current index.

Prefix sums are commonly used in algorithms and can be used for load balancing.

3 Related Work

Since the introduction of the MapReduce programming model, many different MapReduce frameworks and libraries have been created. The frameworks and libraries differ in their implementations of the MapReduce programming model, e.g. in how data is stored, how machines communicate, etc. They also extend the original MapReduce programming model. But they all simplify the execution of parallel algorithms by hiding the details of parallel processing from the user. The user only needs to provide the input data and specify which functions should be applied to the data, and the framework handles all details like scheduling and communication. The following sections will introduce some well-known existing MapReduce frameworks and libraries and some existing load-balancing algorithms for MapReduce.

3.1 Hadoop

Hadoop is a project by the Apache Software Foundation. It is arguably the most popular MapReduce framework. It contains two main modules: Hadoop Distributed Files System (HDFS) and Hadoop MapReduce. HDFS is a distributed file system and Hadoop MapReduce is the framework for processing data stored in HDFS according to the MapReduce programming model [9].

HDFS was designed for storing very large files in clusters. Each file is split into blocks. The blocks are then replicated and stored across nodes. This results in a fault-tolerant file system. To manage data access, each cluster has a master NameNode and multiple slave DataNodes. The NameNode handles file system namespace operations such as opening or closing a file. DataNodes are responsible for blocks within a node and handle operations such as read and write [10]. The NameNode is in charge of choosing on which DataNode to place a file block. When choosing where to place a new block, the NameNode tries to distribute the blocks evenly across the cluster. But as it also needs to consider other constraints such as where replicas of a block are stored, load imbalances can still arise. To ensure a balanced system in the long-term, HDFS employs a balancer for cluster-wide balancing. The balancer rebalances the data by moving blocks between DataNodes [9].

To execute a MapReduce iteration with Hadoop, the user needs to start a MapReduce job. To start a MapReduce job, the user only needs to provide the location of input and output data, as well as a map and reduce function, to be applied to the data. The input data is read from the HDFS file system at the beginning of a MapReduce job. After the execution of the job, the output pairs are written back to the file system. In Hadoop, a MapReduce job is only one MapReduce iteration, i.e. a map phase followed by a reduce phase. To apply multiple iterations of MapReduce to the data, multiple MapReduce jobs can be chained. Ideally, the computation happens on the same nodes where the data is stored. In addition to the original MapReduce programming model, it is possible to define a Combiner, which combines the local values for a key after the map phase before sending them to a Reducer. This cuts down communication costs [11].

To group key-value pairs with the same key during the shuffle phase, a Partitioner calculates for each key-value pair to which Reducer it will be allocated. The Partitioner calculates which Reducer a key-value pair is sent to based on the key and the number of Reducers. Hadoop offers multiple implementations of the Partitioner: BinaryPartitioner, HashPartitioner, KeyFieldBasedPartitioner, RehashPartitioner, and TotalOrderPartitioner.

The HashPartitioner is the default. It calculates the hash of a key $h(k)$. The key-value pair is then sent to the partition which corresponds to $h(k)$ modulo the number of Reducers. The RehashPartitioner behaves similarly to the HashPartitioner, except that it rehashes the hash and then calculates the partition. This can lead to a more even distribution in some cases. The KeyFieldBasedPartitioner partitions based on only part of the key. Usually, a hash function is used to hash part of the key and calculate the partition. The TotalOrderPartitioner partitions the keys based on the total order of the keys. For this, an external source needs to be provided, which contains the split points for each partition. The BinaryPartitioner partitions the keys based on the representative array of bytes of each key. These Partitioners should lead to an equal distribution of key-value pairs over the Reducers, if each key appears with roughly the same frequency. However, if some keys occur more often than others, load imbalances can occur. If none of the provided Partitioners are suitable for the user. The user can also implement their own Partitioner, which decides where to send a key-value pair based on the key, the value, and the number of partitions. If, before execution, it is known that specific keys might be allocated more values than others, this can be considered when implementing a custom Partitioner, but otherwise, load balancing depends on an even distribution of values amongst keys [9].

Other efforts to load balance include the aforementioned HDFS balancer. If file blocks are balanced across DataNodes, the input data for the map phase is balanced. Hadoop also contains a scheduler for scheduling multiple MapReduce jobs started by different users and considerations have been made for how to balance these. However, no further considerations for load balancing within a MapReduce job have been made.

3.2 Spark

Apache Spark [28] was originally a research project at UC Berkeley in 2009. Since 2013, it has been a part of the Apache Software Foundation [12]. Spark was developed with the goal of keeping the fault tolerance and scalability offered by other distributed data-parallel computing tools such as Hadoop while improving the performance of iterative computations. In Hadoop, results after one iteration are written back to disk and must be reloaded if the results are needed for another iteration. This is very inefficient. Spark improves the runtime for iterative jobs by keeping a working set of data in memory.

To use Spark, the user only needs to specify which operations should be performed on the data; Spark then handles all the details, like scheduling and ensuring load balancing and fault tolerance. With Spark, data can be processed according to the MapReduce model, but Spark also offers other operations to process data.

The main abstraction of Spark is the way it stores data in so-called resilient distributed datasets (RDD). An RDD is a read-only collection of data distributed over machines. RDDs are divided into partitions, and partitions are allocated to nodes. Partitions can be derived from one another. This way, lost partitions can be rebuilt. RDDs can be constructed in multiple ways, e.g. they can be constructed from an HDFS file or another RDD. RDDs are lazy and only evaluated when accessed.

To use Spark, a developer writes a driver program, which specifies the high-level operations to be executed on the data. Spark offers many parallel operations, which can be executed on RDD. To execute a MapReduce iteration on an RDD, Spark provides the operations *map()* and *reduceByKey()*. To be able to apply the *reduceByKey()* operation to an RDD, it needs to contain objects in the form of pairs. RDD containing pairs can be constructed from a standard RDD with a special map function such as *mapToPair()*. For a MapReduce iteration, *map()* can be called on an RDD containing pairs or *mapToPair()* can be called on an RDD containing any objects. Those two map operations take a user-defined map function as a parameter, which will be applied to all dataset elements. Then, *reduceByKey()* can be called. It takes as a parameter a user-defined reduce function, which takes two values of the same type and returns a value of the same type.

The *reduceByKey()* operation first executes a shuffle phase, which redistributes the data across partitions in such a way that all pairs with the same key are assigned to the same partition. Spark first sorts the mapped data by partition and then writes the sorted data to a file. Then the nodes responsible for reducing read the relevant blocks from the file.

Like Hadoop, Spark uses a Partitioner to decide to which partition to send a key. It has a HashPartitioner, which partitions keys based on their hash code like in Hadoop. It also offers a RangePartitioner, which samples the keys of the RDD and determines the bounds for each partition. No guarantees are given for the accuracy of the bounds. Keys must be sortable for this partitioner. The RangePartitioner is useful if the reduced results are meant to be sorted. It is also possible to implement a custom partitioner. Spark, like Hadoop, does not consider how many values are associated with a key and does not include any other mechanisms to ensure that the mapped data is evenly balanced for the reduce phase.

3.3 MR-MPI

MR-MPI [22] is a lightweight MapReduce library built on top of MPI. It was mainly developed with graph algorithms in mind, but it can be used for all kinds of MapReduce applications. In contrast to Hadoop and Spark, MR-MPI does not provide a specific distributed file system or contain a job scheduler. It also does not offer any fault tolerance.

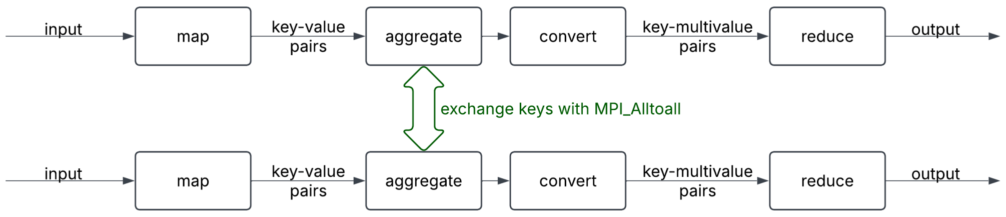

Figure 3.1: Depiction of phases in MR-MPI.

Instead, it is better suited for running on large supercomputing systems, which usually come with a scheduler and file system and make it difficult to integrate an all-in-one system like Hadoop.

Internally, data is stored in buffers, referred to as 'pages'. Each processor always keeps one page in memory. If the data does not fit into one page, the additional pages are kept on disk and pages are loaded as needed. During execution, data elements are stored as key-value pairs or key-multivalue pairs. A key-value pair consists of one key and one value, and a key-multivalue pair consists of one key and a list of values. Keys and values can be of any type or can be a combination of multiple types. A so-called MapReduce object manages all key-value and key-multivalue pairs, and the above-mentioned functions are always executed on one MapReduce object. For more complex applications, multiple MapReduce objects can be created. Each one contains its own set of data distributed over processors.

One MapReduce iteration takes at least three calls to the MR-MPI library: *map()*, *collate()* and *reduce()*. A *map()* call takes the location of the input data and a user-defined map function as input and produces key-value pairs. The input data can either be provided in the form of already existing key-value pairs or be read from a file.

The *collate()* call shuffles the data. MR-MPI splits the shuffle phase into two: *aggregate()* and *convert()*. Both these functions can be called separately. *Aggregate()* aggregates key-value pairs with the same keys on the same processor. This is achieved by hashing the keys and using the modulo operation to determine where to send them. *MPI_Alltoallv()* is used for communication as the communication pattern of *MPI_Alltoallv()* closely resembles the communication pattern of the shuffle phase. *Convert()* then forms key-multivalue pairs from pairs with the same key. A hash table is used for this. For each key-value pair, the key is hashed, and a lookup in the hash table is performed. If the key is already present, the value of the current pair is added, and if the key is not already present, a new key-multivalue pair is created and put in the hash table.

Then the reduce phase can be started with *reduce()*. All key-multivalue pairs are processed according to the user-provided reduce function. The user-provided reduce function can either produce new key-value pairs, which can be used as input for a new MapReduce iteration, or it can write the results back to an output file. Figure 3.1 shows the phases a MapReduce iteration goes through with the MR-MPI library.

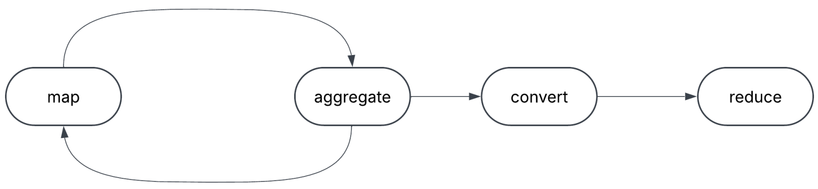

Figure 3.2: Schematic depiction of phases in Mimir.

MR-MPI does not consider skewed data or load balancing. It assumes that the input data is distributed fairly and that keys are mapped to a similar number of values, in which case the hash function will lead to a balanced data distribution.

3.4 Mimir

Mimir [15] was developed to be run on large supercomputers without persistent storage on nodes. Like MR-MPI, it is built on top of MPI, but it was developed to be more memory-efficient than MR-MPI. It is as fast as MR-MPI in cases where MR-MPI is executable in memory, but as Mimir is more memory-efficient than MR-MPI, it fits more data in memory. Therefore, there a cases where an algorithm is executable in memory for Mimir but only out of memory for MR-MPI. In these cases, Mimir vastly outperforms MR-MPI.

Mimir introduces the concept of key-value containers and key-multivalue containers to manage data during its execution. Like MR-MPI, Mirmir splits the shuffle phase into two, an aggregate phase and a convert phase, but it does not offer separate function calls. Instead, it integrates the shuffle phase into the *map()* and *reduce()* operations. The aggregate phase becomes part of the *map()* operation, and the convert phase is integrated into the *reduce()* operation. During the aggregate phase, key-value pairs with the same key are aggregated on the same process. To reduce memory usage, the aggregate phase is interwoven with the map phase. Each process has a send buffer with p partitions, one partition for each process. Processes create new key-value pairs with the map function and place them into the send buffer. If one partition is full, the map phase is paused, and the aggregate phase is started. Mimir performs a *MPI_Alltoall()* call, and the already created key-value pairs are exchanged. Then the exchanged key-value pairs are put into the local key-value containers, and the map phase continues. This is repeated until all input values have been mapped. By interweaving the two phases, it is avoided that each process needs to keep all of its locally mapped data in memory. Keys are sent to processes based on a hash function. Users can implement their own hash function if desirable. Figure 3.2 shows the phases of Mimir.

Then, the convert and reduce phases follow. During the convert phase, pairs with the same key are aggregated locally, turned into a key-multivalue and placed in the key-multivalue container. Then the reduce phase is executed by applying the user-defined reduce function to all key-multivalue pairs. These two phases can generally not be interwoven.

Like with MR-MPI, no specific attention is paid to load balancing. Instead, it is assumed that partitioning the keys based on a hash function will distribute the workload of the reduce phase evenly. The authors instead focus on other optional optimisations such as not sending the sizes of keys and values if all keys and/or all values are the same size or interweaving the convert and reduce phase in cases where the user-defined reduce function does not need all values present to be executed. This interweaving of the convert and reduce phase is called ‘partial reduction’ by the authors and works in the following manner: When hashing key-value pairs to buckets during convert, if a key-value pair is already present, a special user-defined function is called, and the two key-value pairs are reduced to a single key-value pair and placed back in the bucket. This is repeated for all key-value pairs, thereby reducing the amount of memory needed to store the key-multivalue pairs.

3.5 Load-Balancing Algorithms for MapReduce

While the MapReduce frameworks introduced above have not implemented load-balancing algorithms, there has been research into handling skewed data input for MapReduce. When data is skewed, i.e. some keys are paired with more values than others, ensuring that the number of keys per process is balanced is not enough to distribute key-value pairs evenly across processes. To calculate a balanced partitioning of keys for the reduce phase, the cost of applying the reduce function to a specific key and its values needs to be determined. This cost will be referred to as the size of a key. Usually, this is done by determining the number of values mapped to a key. The two main approaches for this are: calculating the sizes of keys after the completion of the map phase or estimating the size of each key before the end of the map phase and, therefore, enabling overlap between the map and shuffle phase.

For estimating the sizes of keys, a sampling algorithm can be used. The advantage of a sampling algorithm is that only part of the input data needs to be evaluated to determine a partitioning. The downside of a sampling algorithm is that insufficient sampling can lead to a still unbalanced partitioning. Zhuo Tang et al. [26] propose a load-balancing algorithm using sampling. They take a sample of the input data and apply the map function to it. The results of the map function are then scaled up to estimate the sizes of the keys. No guarantees for the accuracy of the sample are given, but the authors provide a formula to calculate a sample rate appropriate for the user’s needs based on cost and variance in the sample. Ramakrishnan, Swart and Urmanov [23] use a progressive sampling method, which checks after each iteration if the sample is representative or if more sampling data is needed. They use a hypothesis test to determine if the sample is big enough.

To then calculate the partitions, both algorithms sort the keys by size and partition them using a decreasing bin packing algorithm. If a key is bigger than the capacity of the bins (total amount of values divided by the amount of Reducers), it is split over multiple bins. This prevents a key, which is bigger than the mean amount of data ideally allocated to each Reducer, from becoming a bottleneck. During their experiments, both groups found that for skewed input data their load balancing algorithm performed better than the equivalent without load balancing.

Yanfang Le et al. [20] take a different approach to determining a balanced partitioning and propose an online algorithm. An online algorithm also enables the partitioning of the keys before the end of the map phase, as it does not need to know the final size of a key to determine the partitioning. For each key-value pair produced by a map function, it is checked if the key has already been assigned to a Reducer. If no assignment has happened, the unassigned key is assigned to the Reducer with the currently smallest load. This fully online algorithm can be optimised by introducing a small amount of offline sampling. By waiting until a few key-value pairs have been produced and then sorting them according to size, heavy keys can be found. The sorted keys are then assigned in descending order to the Reduce with the least current load. Then, the algorithm proceeds as the standard online algorithm. Both these algorithms have the advantage of not having much overhead over the standard hash-based shuffle phase, but an online algorithm does not guarantee an ideally balanced partitioning. The experiments performed by the authors showed an improvement in the maximum load assigned to a Reducer when using either of the two algorithms. The shuffle execution time, however, was slower than with the default hash-based algorithm. They did not compare execution time of an entire MapReduce iteration.

The LEEN [17] algorithm is an example of a load-balancing algorithm, which waits until the map phase is done before starting to partition the keys. It also considers data locality to reduce communication costs during the shuffle phase. It determines the frequency of each key on each node and tries to load balance the keys evenly amongst Reducers while also trying to assign each key to a Reducer, with a high number of key-value pairs with that key already present. Experiments performed by the authors showed an improvement in execution time in comparison to Hadoop for skewed input data.

SkewTune [19] introduces a dynamic load-balancing algorithm. A dynamic load-balancing algorithm does not need to calculate the sizes of keys, as it only takes action after the imbalance has occurred. If SkewTune detects a task with a very high remaining process time, it splits up the task over two or more machines to reduce its remaining process time. The advantage of the dynamic approach is that no overhead is added if the data is already evenly distributed. It does, however, add more data transfers. The experiments of the authors showed an improvement in execution time in comparison to standard Hadoop.

4 Load Balancing Map Reduce

The following sections describe the shuffle algorithms of our MapReduce library. Section 4.1 describes a straightforward hash-based shuffle similar to Mimir or MR-MPI [15, 22]. Section 4.2 describes a static load-balancing shuffle algorithm [25]. Section 4.3 gives some insight into how the theoretical algorithms were implemented.

4.1 Simple MapReduce Algorithm

As described in section 2.1, after the application of the map function μ to the input data, we have a distributed set of key-value pairs on which we want to apply the reduce function ρ. Each key-value pair (k, v) is composed of a key k and a value v. The key and value do not need to have the same data type and can also have a different data type from the initial data stored in memory. Before the reduce phase can begin, key-value pairs with the same key must be collected on the same PE. During the shuffle phase, each PE iterates over its key-value pairs and calculates the hash $h(k)$ of each key for a hash function h with range $0..s$. Then, each key-value pair is sent to the PE

$$\lfloor p \cdot h(k)/s \rfloor. \tag{4.1}$$

Therefore, collecting keys with the same hash on the same PE. To apply the reduce function, key-value pairs with the same key must be locally aggregated. This can be achieved with a hash map. The keys of the pairs are used as keys in the hash map, and for each key, a list of values is stored. For each key-value pair, the key is hashed, and a lookup is performed on the hash map. If the key is already in the hash map, the value of the key-value pair is added to the list of values for that key. If the key is not in the hash map, it is added by creating a pair with that key and a list containing the value. This concludes the shuffle phase, and the reduce phase can begin.

4.2 Load-Balancing Shuffle Phase

In the above-described algorithm, the keys are expected to be uniformly distributed amongst the PEs after the shuffle phase for a good hash function. However, if multiple keys with a disproportionally large amount of corresponding values are mapped to the same PE, the number of values per PE will not be even. To overcome this problem, Sanders [25] proposes a static load-balancing algorithm for the shuffle phase, which is efficient in the MRC$^+$ model. A combination of hashing and prefix sums should ensure that the data is evenly distributed amongst the PEs before the reduce phase. This is achieved by taking into account how many values are mapped to a key as opposed to assuming all keys are mapped to roughly the same number of values.

To distribute the key-value pairs evenly, the number of values per key needs to be calculated

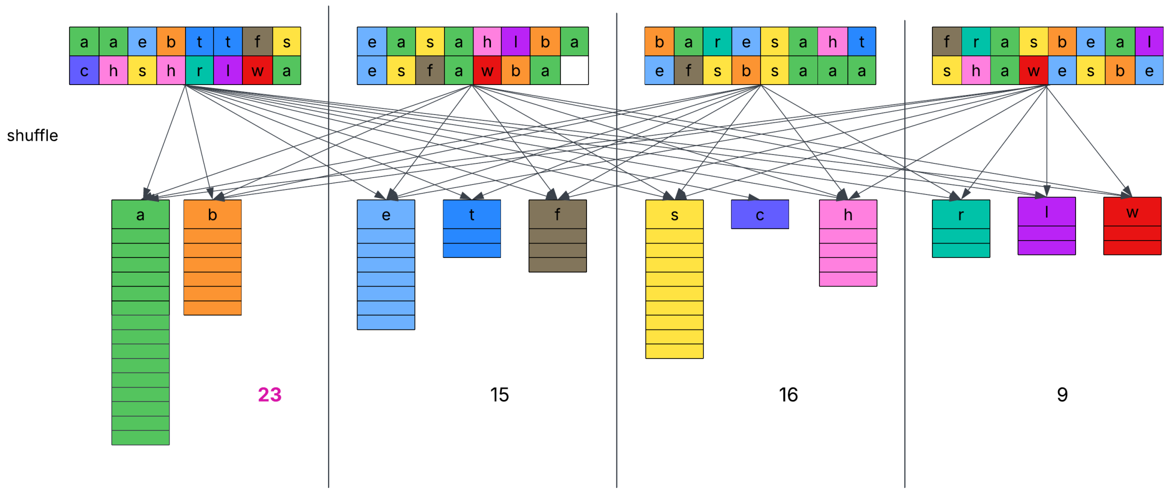

Figure 4.1: Example of hash based shuffle phase. The highest workload for the following reduce phase for an individual PE is 23. Adapted from [25]

first. To avoid a high communication overhead created by sending entire key-value pairs, only constant-sized pairs are sent in this step. For this each key-value pair $b = (k, v)$ is mapped to a pair $b' = (h(k), |b|)$. This will be referred to as a key-size pair. h is a hash function with range m^d for a $d > 2$. Hash functions with such a range have a low probability of collisions. $|b|$ describes the size of b. Therefore, some values might carry a bigger load than others. Then b' is send to PE $\lfloor p \cdot h(k)/m^d \rfloor$. Exactly as in the shuffle phase described above, this results in all pairs with the same key being sent to the same PE. Now, the number of values per key can be calculated. For this, the key-size pairs are sorted locally, and the sizes are aggregated, which results in a distributed array of pairs $(k' = h(k), v')$. v' indicates how many values are associated with key k. Then, a prefix sum n can be calculated over the number of values. Based on the prefix sum for each key $h(k)$, a target PE is calculated with the following formula:

$$t(k') = \lfloor \frac{p \cdot n(k')}{m'} \rfloor \tag{4.2}$$

m' denotes the total amount of values distributed over all PEs. Then, the PEs must be informed about where to send their key-value pairs. A pair $(h(k), t(k'))$ is sent from the PE, which calculated the prefix sum for the key to each PE which stores at least one key-value pair with that key. Then, the key-value pairs are sent to the final target PE, and the reduce phase can begin. If each PE after the map phase stores $O(m/p + \hat{m})$ data, then the shuffle needs local work and bottleneck communication volume $O(m/p + \hat{m} + logp)$ and each PE is expected to receive $O(m/p + \hat{m})$ amount of data.

Figure 4.2 shows the prefix sum based shuffle phase applied to a WordCount problem. First, key-size pairs are sent to PEs based on their hash value. Comparing this figure with figure 4.1 shows that each PE receives the keys it would receive during a hash-based shuffle. Then, for each key, the number of values is calculated. Based on that, the prefix sum is calculated. And then with equation 4.2 target PEs are calculated for each key. The load-balancing approach leads to, e.g. key b being sent to the PE with rank 1 instead of

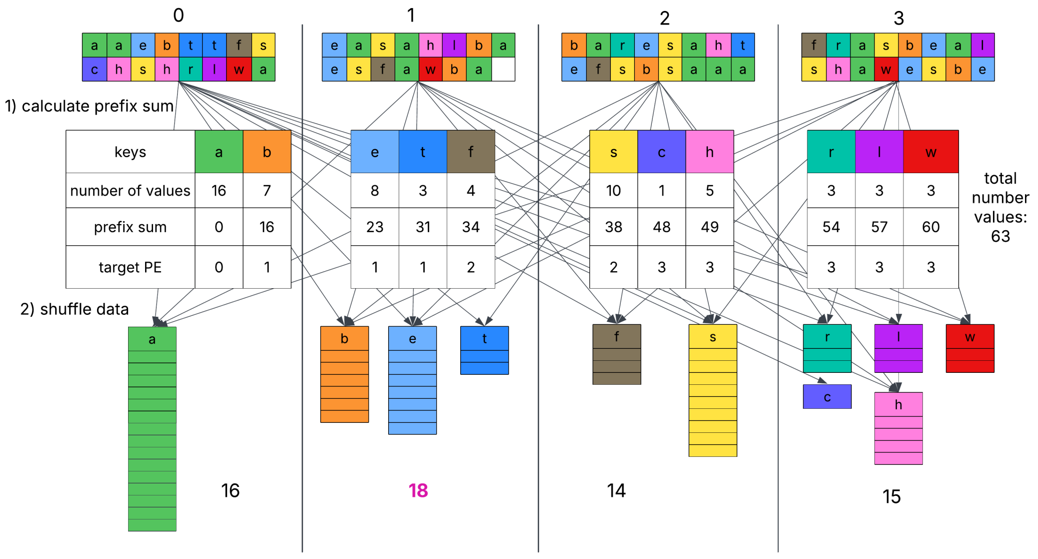

Figure 4.2: Example of prefix sum based shuffle algorithm. The highest workload for the following reduce phase for an individual PE is 18. Adapted from [25]

the PE with rank 0. The number of keys per PE is no longer even, but the number of values and, therefore, work load is more evenly distributed. The highest workload is 18 with load-balancing as opposed to 23 with the hash-based shuffle.

4.3 Implementation

Algorithm 4.1: map

```
Function map(vector input_data, function mapping):
    vector key_value_pairs
    foreach element in input_data do
        pair ← mapping (element)
        key_value_pairs.push_back(pair)
    end
    return key_value_pairs
```

We implemented the above-described algorithms in the following manner. Our MapReduce library was written in C++20 and designed to work on distributed systems where each PE has its data in its own memory during execution. The library is executed in memory. It is assumed that each PE terminates successfully, i.e. no fault-tolerance mechanisms are implemented.

To handle communication between PEs, we use the Message Passing Interface (MPI). We chose the combination of MPI and C++ as this lends itself to being easy to use for scientific computing using supercomputers. MPI can be called from C++, but as it was designed for C and Fortran, using it in C++ code is unintuitive. That is why we use the C++ MPI binding library KaMPIng (Karlsruhe MPI next generation)[1] [27]. KaMPIng facilitates

[1] `https://github.com/kamping-site/kamping`

Algorithm 4.2: reduce

```
Function reduce(vector key_value_pairs, function reduction):
    //shuffle phase:
    shuffled_pairs ← distributeByKey(key_value_pairs)
    hashmap map  //use map to group values by key
    foreach pair = (k,v) in shuffled_pairs do
        if map.contains(k) then
            map.at(k).push_back(v)
        else
            map.insert({k, {v}})
        end
    end
    //reduce phase:
    foreach pair = (k, values) in map do
        reduced_data.push_back(reduction (k, values))
    end
    return reduced_data
```

MPI calls by, e.g. allowing std::vectors to be passed in as send or receive buffers instead of just raw pointers and by automatically deriving buffer sizes or data types. KaMPIng also helps to avoid boilerplate code by handling send and receive count exchanges for MPI calls like *MPI_Alltoallv* and *MPI_Allgatherv*. These features lead to a simplified code, which is important for quick prototyping.

Our MapReduce library offers two operations: map and reduce. Map is a local operation and is executed simultaneously on all PEs. It takes a vector containing the input data and the user-defined map function as parameters and returns a vector containing the key-value pairs produced by applying the user-defined map function to the input data. Algorithm 4.1 shows pseudocode for the map operation.

The reduce operation is also responsible for the shuffle phase. Therefore, this operation is not just executed locally and contains some communication. How much communication depends on whether the user chooses to execute the reduce operation with or without explicit load balancing. The reduce operation takes a vector of key-value pairs as input, usually, the vector of key-value pairs which was returned by the map operation and a user-defined reduce function. It returns a vector of output data.

Algorithm 4.2 shows pseudocode for the reduce operation. The difference between the load-balancing case and the non-load-balancing case is how the *distributeByKey* part of the shuffle phase is handled. Here, *distributeByKey* means the part of the shuffle phase which redistributes the key-value pairs after the map phase in preparation for the reduce phase. Algorithms 4.3 and 4.4 show the pseudocode for the *distributeByKey* part of the shuffle phase in the non-load-balancing and the load-balancing case, respectively. Clearly, in the load-balancing case, a lot of overhead is added through the additional computation and communication in the form of MPI calls needed to compute a better distribution of key-value pairs.

The hash-based shuffle is a straightforward implementation of the algorithm laid out in section 4.1. For our load-balancing shuffle, our main adaption is that we do not send a key-size pair for every key-value pair present locally during the first step. Instead, we use a hash map to determine the local size of a key, i.e. how many key-value pairs are stored locally after the map phase for each key. Then, for each key, only a single key-size pair is sent on to determine the global size of the key. This cuts down on communication costs and computational costs when calculating the prefix sum. We only aggregate the size of a key and not also the values because not for all MapReduce applications is it possible to do a local reduction and still produce a correct result.

Algorithm 4.3: distributeByKey without load balancing (hash-based)

```
Function distributeByKey(vector key_value_pairs):
    vector send_buffer_key_size
    foreach pair = $\{k, v\}$ in key_value_pairs do
        target_PE ← $p \cdot \frac{h(k)}{maxHashValue}$
        put key-value pair into send buffer according to target
    end
    MPI_Alltoallv(recv_buffer)
```

The map and reduce operations are template functions, i.e. they can handle a high variability of data types. The user-defined functions are taken as input in the form of functors. This means the user-defined function can contain state. The library takes a simple approach to sending data and can only send constant-sized types, which means the data types of the key and value of the key-value pairs passed to the reduce operation as input both need to be of constant size. We did this because of time constraints. This does reduce the communication cost as no length for a key or value needs to be communicated because each key has the same length and each value has the same length. Finally, keys need to be hashable and comparable. The user can set their own hash function for a key by implementing std::hash for their key data type.

Algorithm 4.4: distributeByKey with load balancing (prefix sum based)

```
Function distributeByKey(vector key_value_pairs()):
    vector send_buffer_key_size
    hashmap map
    // determine local sizes of keys
    foreach pair = (k, v) in key_value_pairs do
        if map.contains($h(k)$) then
            map.at($h(k)$) ← map.at($h(k)$) + 1
        else
            map.insert({$h(k)$, 1})
        end
    end
    foreach pair = (k, size) in map do
        target_PE ← $p \cdot \frac{h(k)}{maxHashValue}$
        insert tuple {h(k), size, rank} into send_buffer_key_size to send
         it to target PE
    end
    recv_buffer_key_size ← MPI_Alltoallv(send_buffer_key_size)
    sort recv_buffer_key_size based on $h(k)$
    vector key_sizes, vector distinct_keys
    loop through buffer, collect distinct keys and sum sizes for each
     key

    vector prefix_sum
    prefix_sum ← exclusive_scan(key_sizes)
    int previous_sum ← MPI_Exscan(prefix_sum.back())
    add previous_sum to local prefix_sum to create global prefix_sum

    number_pairs
    if rank = p - 1 then
        number_pairs ← prefix_sum.back() + key_sizes.back()
    end
    MPI_Bcast(number_pairs, root = p - 1)

    vector target_PEs
    for $i$ ← 0 to distinct_keys.size() do
        target_PEs[$i$] ← $\frac{p \cdot \text{prefix\_sum}[i]}{\text{number\_pairs}}$
    end
    for every tuple in recv_buffer_key_size send pair with hash of key and
     target back to sender
    recv_buffer_key_target ← MPI_Alltoallv(send_buffer_key_target)
    hashmap map
    foreach pair = (h(k), t) in recv_buffer_key_target do
        map.at(h(k)) ← t
    end
    foreach pair = (k, v) in key_value_pairs do
        target ← map.at(h(k))
        put key-value pair into send buffer according to target
    end
    recv_buffer_key_value ← MPI_Alltoallv(send_buffer_key_value)
    return recv_buffer_key_value
```

5 Applications (Theory)

To test our MapReduce library in different scenarios, we chose multiple applications. The applications we used are: WordCount, PageRank and k-Means Clustering. The following sections describe the algorithms of those.

5.1 WordCount

WordCount is a simple MapReduce application, often used as an easy example to explain MapReduce. WordCount determines for each word how often it occurs in a text. It is a straightforward algorithm which is easy to implement. Therefore, it is useful as an application for testing a MapReduce library.

For a WordCount application, the input is provided in the form of a text. The text is split up into parts; either by the application or the input text is already divided into smaller parts. Each text part is an input element. The map function takes a part as input and splits it into its individual words. For each word detected by the map function, a pair (word, 1) is produced as an intermediate key-value pair. The reduce function then takes its input in the form of a word as a key and a list containing an entry for each time the word occurred. Consequently, the reduce function can sum up the values and thereby calculate the number of occurrences for the key. Algorithm 5.1 shows the map and reduce function in pseudocode.

WordCount is particularly useful for testing a load-balancing MapReduce because real-world texts usually contain some words many times and other words only a few times. Words like ‘a’ or ‘the’ probably occur multiple times for every 100 words in a text, but words like ‘arcane’ or ‘conundrum’ are more likely to only occur once or twice in an entire text. Because the words are used as keys in WordCount, this leads to an unequal distribution of values amongst the keys.

Algorithm 5.1: WordCount

```
Function map(String text):
    foreach word in text do
        emit (word, 1)
    end
Function reduce(String word, List values):
    count ← 0
    foreach value in values do
        count ← count + value
    end
    return count
```

5.2 PageRank

PageRank is an algorithm created by the co-founders of Google Sergey Brin and Lawrence Page [4]. PageRank was used to determine in which order Google will present webpages in a search result. For this purpose, each webpage is assigned a rank. The rank of a page represents the value of each page, i.e. the likelihood it will contain useful information for the author of the search query. The rank of a page is based on how many pages link to that page. The more pages link to a page, the higher the rank of that page. This is based on the assumption that many pages linking to a page means that the page contains valuable information. PageRank also takes into account which pages link to a page. If a 'high value' page, e.g. Wikipedia, links to another page, that carries more weight than if a random small page links to another page.

PageRank is a graph algorithm because webpages can be represented by a directed graph in the following manner: Each vertex in the graph is a webpage, and the graph contains edge (v, u) if and only if page v links to page u. $N(v)$ denotes the set of neighbours of v, i.e. all pages which v links to and $|N(v)|$ denotes, therefore, the number of outgoing edges from v. $B(u)$ denotes all vertices which have u as a neighbour, i.e. all pages which link to u.

PageRank is calculated iteratively. Each page is assigned an initial rank. Then, in each iteration, each page v distributes its rank equally over its neighbours. This is done by sending each neighbour a proportional part of its rank i.e. each neighbour receives $PR(v)/|N(v)|$. Then, each vertex can calculate a new rank based on the values it receives.

This calculation is expressed by equation 5.1 [7]. After a certain number of iterations, the ranks will converge, and the algorithm is finished.

$$PR(u) = (1-d) + d \sum_{v \in B(u)} \frac{PR(v)}{|N(v)|} \tag{5.1}$$

d is a dampening factor used to express that a user might not follow the links of a page and instead go to a random page.
This algorithm is easily adaptable for MapReduce: the map function takes as input a vertex v and produces for each neighbour u of v a key-value pair: $(u, PR(v)/|N(v)|)$. The reduce function then receives as input a vertex u and all partial rank values u has received from pages which link to u. Consequently, the reduce function calculates the new rank for u with equation 5.1. Algorithm 5.2 shows the map and reduce function in pseudocode. Then, after each MapReduce iteration, it is checked if the ranks have converged. For each vertex, its new and its previous ranks are compared by calculating the absolute difference between the two. The differences are summed up, and if the total difference between two iterations is smaller than a certain value (e.g. 0.0001) the algorithm is finished.

Algorithm 5.2: PageRank

```
Function map(Vertex $v$):
    foreach $u \in N(v)$ do
        emit $\left(u, \frac{PR(v)}{|N(v)|}\right)$
    end
Function reduce(index, List values):
    sum $\leftarrow 0$
    foreach value in values do
        sum $\leftarrow$ sum $+$ value
    end
    return $(1-d) + d \cdot$ sum
```

5.3 K-Means Clustering

Clustering algorithms take a data set and group its elements into clusters to put similar elements into the same cluster and different elements into different clusters based on an element's properties [21]. We chose k-means clustering as one of our applications to evaluate the performance of our library for user-defined reduce function with a higher complexity and longer reduce execution time, in this case, quadratic complexity. K-means clustering groups the data into k clusters. Each cluster has a centre, and a data element is put into the cluster to whose centre it is the closest in distance. To determine the distance between two data elements, we calculate the Euclidean Distance between the two. Suppose the data set contains n-dimensional points, the distance between two point $P = (p_1, ..., p_n)$ and $Q = (q_1, ..., q_n)$ is then given by following formula [2]:

$$d(P, Q) = \sqrt{\left(\sum_{i=1}^{n} (p_i - q_i)^2\right)} \tag{5.2}$$

Because the k-means algorithm usually takes a large data set as input, which is likely to be distributed amongst multiple machines, it is useful to adapt the k-means algorithm for MapReduce. K-means is an iterative algorithm. In each iteration, new centres are calculated.

Anchalia et al. [2] propose an algorithm for calculating k-means with MapReduce. The map function takes a data element as input, chooses the nearest centre, and emits those as a pair. Then, during shuffle, points mapped to the same centre are grouped together. This is the new cluster. The reduce function takes a cluster as input and recalculates its centre. For the first round, the clusters are chosen at random.

We calculate the new centres in a naive way by pairwise comparing all points in a cluster and choosing the point with the smallest average distance to all other points in the cluster as the new centre. The reduce function has the complexity of $O(n^2)$. We chose to recalculate the centres in this way to be able to see how our MapReduce library performs when the application of the reduce function is more complex and, therefore, takes up more time.

Algorithm 5.3: K-Means Clustering

```
Function map(Vertex $v$):
    smallest_distance ← $\infty$
    nearest_centre
    //take advantage of features of functors by passing in current
     centres as parameter when creating this user-defined map
     function foreach centre in centres do
        distance ← $d(v, c)$
        if distance < smallest_distance then
            smallest_distance ← distance
            nearest_centre ← centre
        end
    end
    return (nearest_centre, $v$)
Function reduce(centre, List values):
    smallest_distance ← $\infty$
    new_centre
    foreach vertex in values do
        distance ← $0$
        foreach $u$ in values do
            distance ← distance + $d(u,$ vertex$)$
        end
        if distance < smallest_distance then
            smallest_distance ← distance
            new_centre ← vertex
        end
    end
    return (new_centre, centre)
```

6 Experimental Evaluation

We evaluate our load-balancing MapReduce library by comparing the load-balancing shuffle with the hash-based shuffle. For this, we execute both variations for all three applications introduced in the previous section. This chapter begins by explaining our experiment setup, then we evaluate our library for the three applications: PageRank, k-means clustering and WordCount. Finally, we explore a variation of MapReduce using local reduction and then we compare our library to the MR-MPI library.

6.1 Setup

We use the supercomputer SuperMUC-NG at the Leibniz-Rechenzentrum to test our MapReduce library. The SuperMUC-NG consists of 6336 thin compute nodes with 48 cores and 96 GB memory and 144 fat compute nodes with 48 cores and 768 GB memory. We use the thin nodes for our experiments. Our library is written in C++20 and compiled with gcc version 11.2.0. using the optimisation flag -O3. We use the KaMPIng library on top of Intel MPI 2019.

To evaluate our load-balancing algorithm for the shuffle phase, we measure the execution time of our MapReduce library for the different applications introduced in the previous section. To measure the execution time, we take advantage of the functionality offered by the KaMPIng library. The KaMPIng library provides a timer, which makes it easy to measure the execution time, even split into phases and over multiple iterations. The KaMPIng timer is built on top of *MPI_Wtime*. When measuring the entire execution time, we measure from the beginning of the map phase to the end of the reduce phase for WordCount. For PageRank and k-means clustering, we measure the execution time over multiple iterations. The other benchmark we look at is the distribution of key-value pairs after the shuffle phase. For this, we calculate the deviation from the mean of key-value pairs per processor. A deviation of 0% on all processes signifies that every processor receives the same number of key-value pairs.

For alltoall communication, the standard *MPI_Alltoallv* offered by KaMPIng is used. KaMPIng also offers other communicators like the grid communicator, which enables *MPI_Alltoallv* with a latency of $\sqrt{p}$. This did not improve the execution time of our experiments except on 4096 processes in some cases, which is why we used the standard *MPI_Alltoallv*. The grid communicator should, however, be considered when executing our MapReduce library on even more processes.

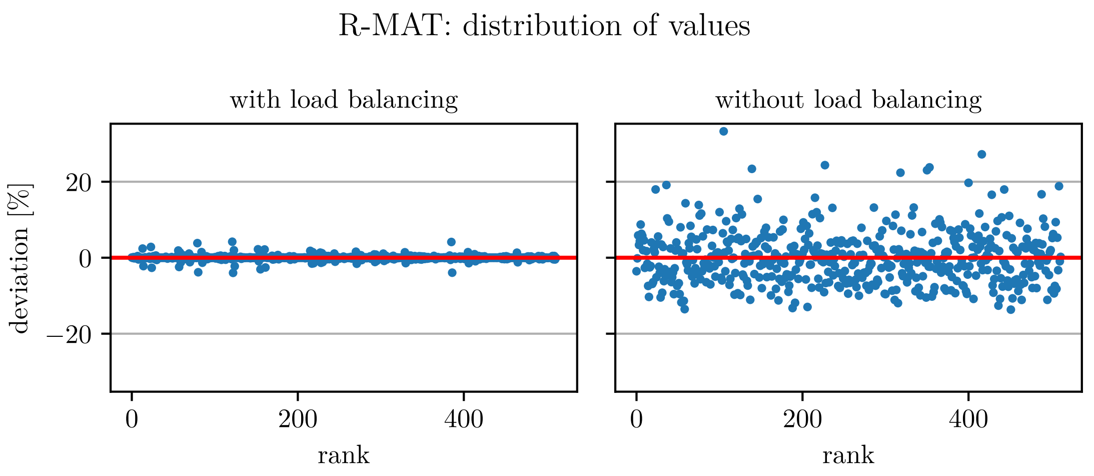

Figure 6.1: Deviation of the number of key-value pairs after the shuffle phase from the mean for the PageRank algorithm executed on an R-MAT graph with 2^{25} vertices and 2^{29} edges.

6.2 PageRank

We start our evaluation of MapReduce with the PageRank algorithm. As PageRank is a graph algorithm, a graph is needed as input. We use the graph generator KaGen[1][14] for generating input. KaGen is able to generate a variety of different types of graphs without communication. We execute PageRank on an R-MAT graph and an Erdös-Rényi graph. The R-MAT graph is an irregular graph and the Erdös-Rényi graph is regular. For each type, we analyse the execution time with and without using our load-balancing algorithm over 10 iterations.

6.2.1 R-MAT

We used the R-MAT graph generator [16] provided by KaGen. R-MAT graphs closely resemble real-world graphs [5] and contain some vertices with very high degrees, which leads to imbalances during PageRank. This makes an R-MAT graph well-suited for testing our load-balancing algorithm.

For generating an R-MAT, four parameters a, b, c and d with $a + b + c + d = 1$ are used. The adjacency matrix of the graph is divided into four quadrants. An edge is placed in the upper left quadrant with probability a, in the upper right with probability b, in the lower right with probability c and in the lower left with probability d. After an edge is placed in a quadrant, the quadrant is divided into quadrants again, and the process repeats until one entry in the adjacency matrix is left. This is repeated until all edges are placed [16].

If the parameters a, b, c and d are not all the same, a graph with a non-uniform distribution of edges amongst its vertices is created. Therefore, some vertices will have a much higher

[1]`https://github.com/KarlsruheGraphGeneration/KaGen/`

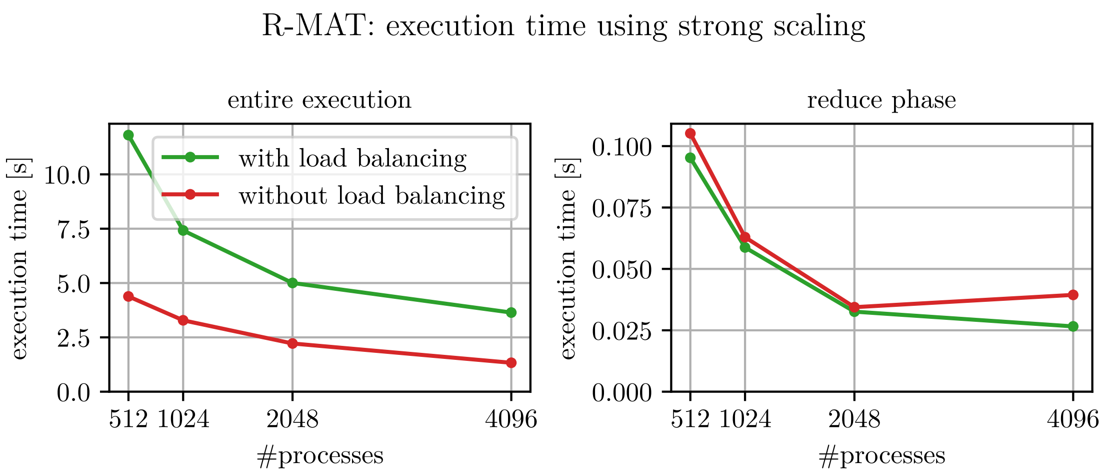

Figure 6.2: Execution time of PageRank on an R-MAT graph with 2^{25} vertices and 2^{29} edges.

degree than others, and consequently, during the MapReduce iteration, some keys will be paired with more values than others.

We generate an R-MAT graph with the parameters $a = 0.57, b = 0.19, c = 0.19$ and $d = 0.05$. This corresponds to the graph 500 benchmark [1]. We generate 2^{25} vertices and 2^{29} edges and have an edge factor of 16, which is suggested by the graph 500 benchmark. These parameters lead to an irregular graph. Because the degree of vertices is varied, we distribute the vertices between processors based on their edges, i.e. each processor doesn't necessarily store the same number of vertices but roughly the same number of edges. This ensures a balanced input for the map function.

Figure 6.1 shows the distribution of key-value pairs amongst the processors after the shuffle phase with and without load-balancing, respectively. The red line visualizes the ideal distribution of key-value pairs, i.e. no deviation from the mean of key-value pairs per processor. If all processors were to receive the same number of key-value pairs after the shuffle phase, all dots of the plot would be on the red line. Our load-balancing algorithm leads to a more balanced distribution of key-value pairs amongst processors than in the case of no explicit load balancing. With load balancing, the maximum deviation is 4.2% and without load balancing, it is 33%.

Figure 6.2 compares the run time of PageRank on the R-MAT graph in both cases. Even though with load-balancing, the data is distributed evenly, the execution of PageRank takes longer. Figure 6.2 also shows the execution time of just the reduce phase. The execution of the reduce phase is quicker when the data is load-balanced during the shuffle phase, which is to be expected. However, the y-scale shows that the reduce phase only makes up a fraction of the entire execution time. Therefore, this speedup in the reduce phase is not reflected in the entire execution time. On the contrary, the execution time with the load balancing is significantly slower. This is because the load balancing adds overhead in the form of additional computation to calculate the more even distribution of

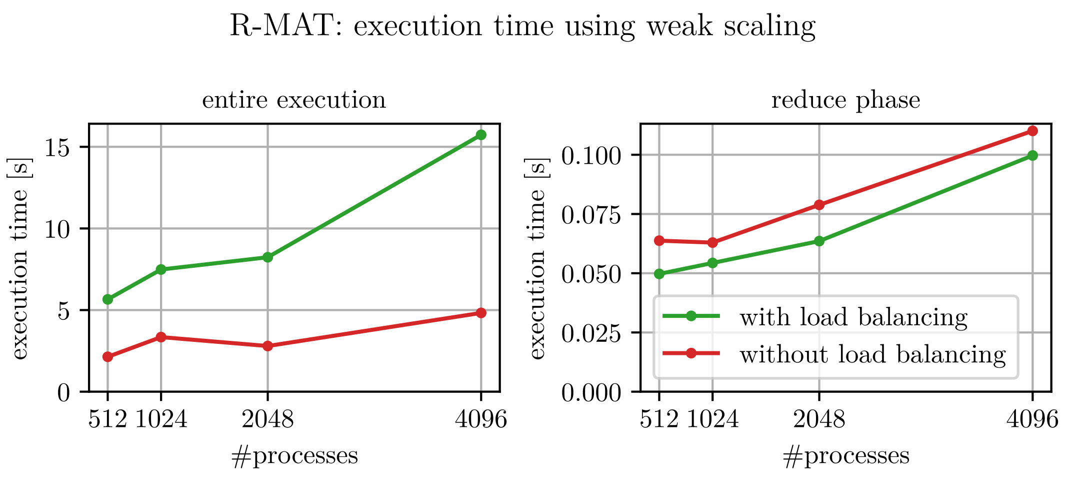

Figure 6.3: Execution time of PageRank on an R-MAT graph using weak scaling with 2^{15} vertices and 2^{19} edges per process.

key-value pairs and in the form of additional communication to communicate the calculated distribution.

Figure 6.3 shows the execution time when using weak scaling. We generate 2^{15} vertices and 2^{19} edges per processor. To get a better overview of where time is spent during the execution, Figure 6.4 shows the execution time of PageRank on the R-MAT graph split into four phases: map, distributeByKey, convert and reduce. DistributeByKey and convert together make up the shuffle phase. DistributeByKey in the non-load-balancing case contains the calculation of the target processes for keys based on their hash value and the *MPI_Alltoallv* call to redistribute the key-value pairs. In the load-balancing case, distributeByKey contains all steps to calculate the prefix sum and the target processes for all keys, which uses two additional *MPI_Alltoallv* calls as well as the *MPI_Alltoallv* call to redistribute the key-value pairs. Section 4.3 provides a more detailed description of which computational steps happen during each phase.

Convert performs the same steps in both cases; it groups the key-value pairs based on their keys. As expected, the shuffle phase takes longer in the case of load balancing. In both cases, load balancing and non load balancing, most of the time is spent during the distributeByKey phase. The execution time of the rest of the phases in comparison does not increase by much. The distributeByKey phase, i.e. the computation of the prefix sum and the communication of it adds the additional time visible when looking at the entire execution time.

As distributeByKey takes up the most time, we want to take a closer look at it. Figure 6.5 shows the breakdown of the distributeByKey phase. For the load-balancing algorithm, distributeByKey performs the following steps. First, the local sizes of keys are calculated with the use of a hash map and pairs consisting of the hash of a key and its size are created. Then, these pairs are exchanged with an *MPI_Alltoallv*, and all key-size pairs for the same key are sent to the same process. Then the process calculates the prefix

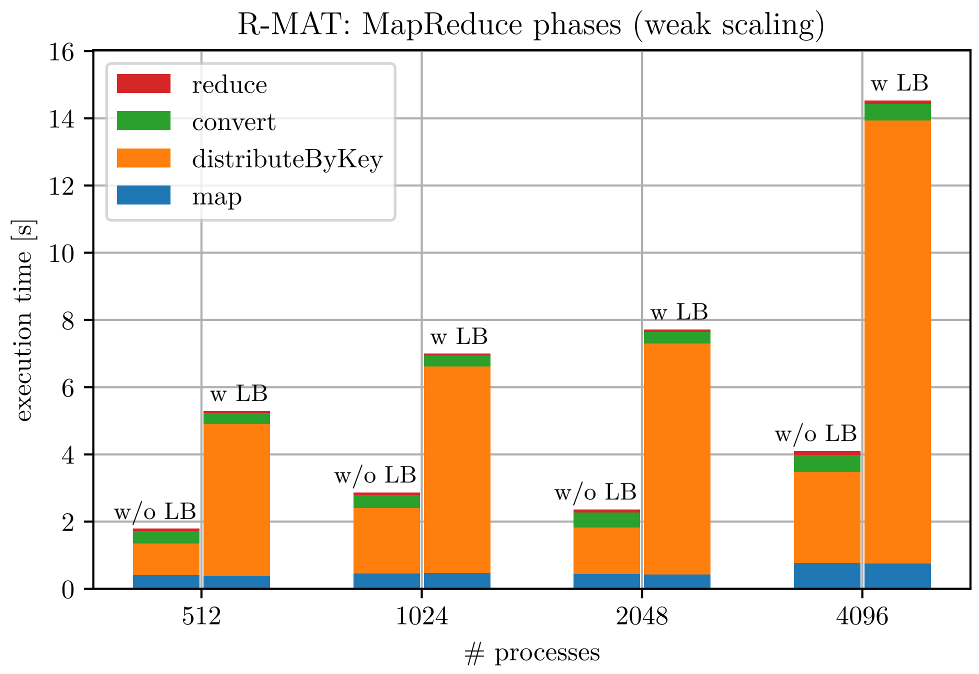

Figure 6.4: Time spent in each phase during execution of PageRank on R-MAT graph using weak scaling with 2^{15} vertices and 2^{19} edges per processor.

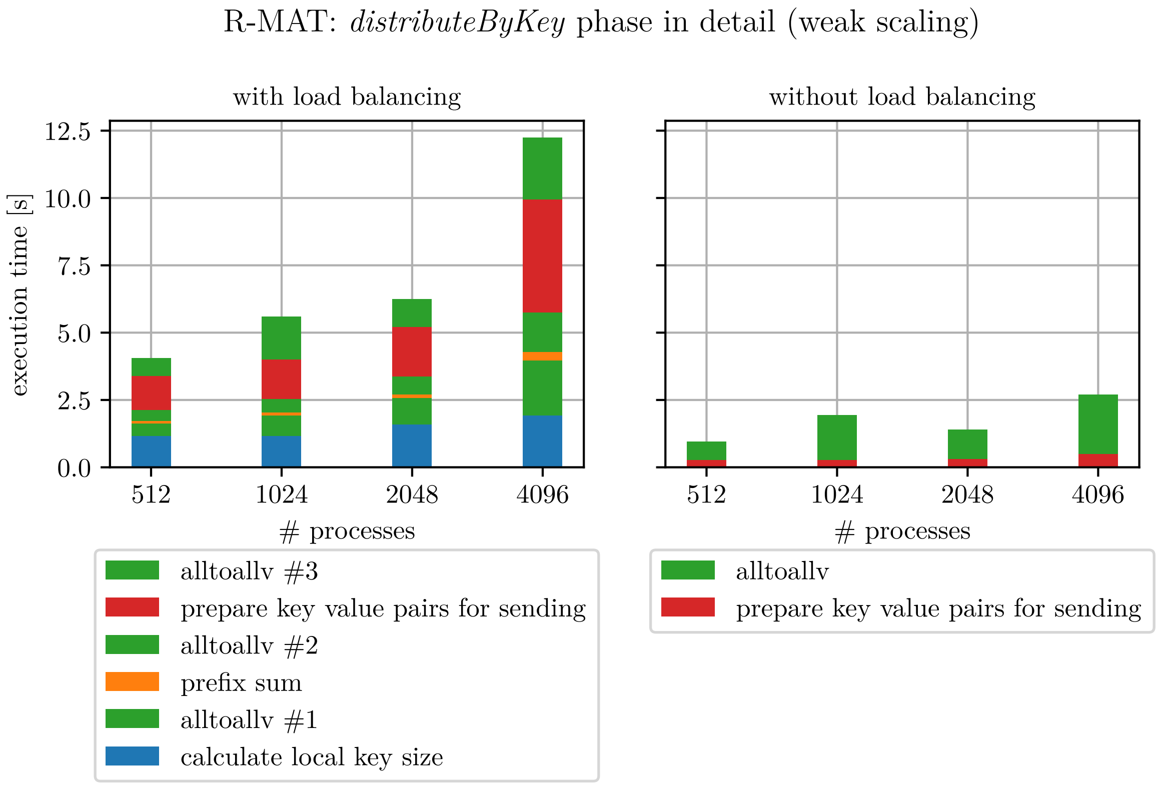

Figure 6.5: Time spent in each phase of the distributeByKey phase during execution of PageRank on R-MAT graph using weak scaling with 2^{15} vertices and 2^{19} edges per processor.

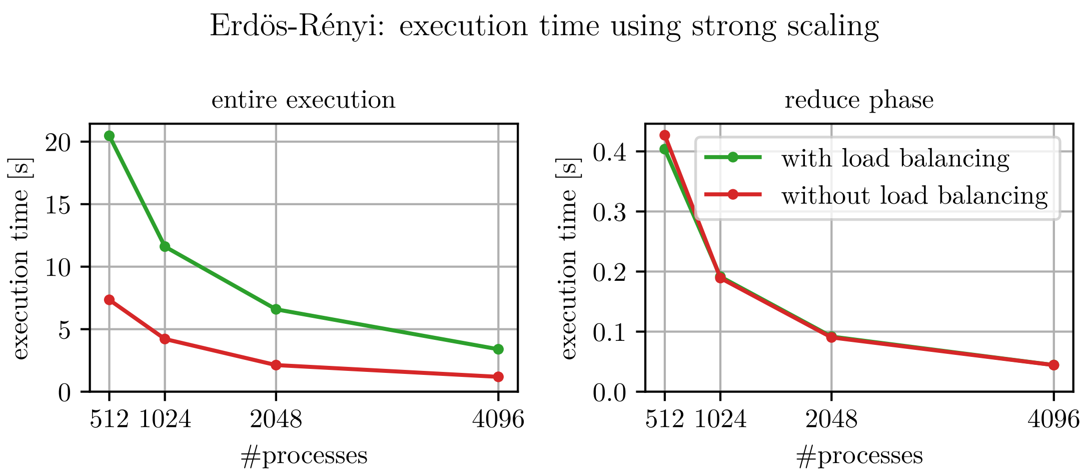

Figure 6.6: Execution time of PageRank on an Erdös-Rényi graph with 2^{27} vertices and 2^{31} edges.

sum based on the key sizes and, based on that, the target process for each key. Each process creates pairs of the hash of the keys and their target process. These are then exchanged with *MPI_Alltoallv* #2. Based on the key-target pairs, the processes place the key-value pairs in a send buffer. Then, finally, the key-value pairs are exchanged with the final *MPI_Alltoallv*. The last two steps are also necessary for the distributeByKey phase without load balancing. The step of preparing key-value pairs by putting them in the send buffer is faster without load balancing because the only steps needed are to iterate over the key-value pairs and place them in the send buffer based on the hash of the key. With load balancing, the target process is not hash-based. Therefore, to determine the target process, the correct key-target pair produced by the previous calculations needs to be found; for this purpose, the key-target pairs are put in a hash map. Inserting in a hash map and then looking up the correct target process in the hash map adds extra time in comparison to being able to do a simple calculation to determine the target process. The time needed for a look up or insert also increases for a higher variety in keys, even if the number of key-value pairs overall stays the same. This is why the time needed to prepare the key-value pairs increases for more processes with load balancing.

6.2.2 Erdös-Rényi-Graph

We just looked at PageRank on skewed input, now we also want to consider PageRank on a balanced input and use an Erdös-Rényi graph as input. The Erdös-Rényi model is another way of generating random graphs. In this model, a graph with n vertices and m edges is created by randomly choosing it from all possible graphs with n vertices and m edges [24]. This leads to an expected equal distribution of edges between vertices, and therefore, each vertex has roughly the same indegree.

Figure 6.6 compares the execution time. As is the case for the R-MAT graph, the execution takes longer in the case of load balancing. The difference is that, here, the execution time of the reduce phase takes roughly the same time with and without load balancing. This

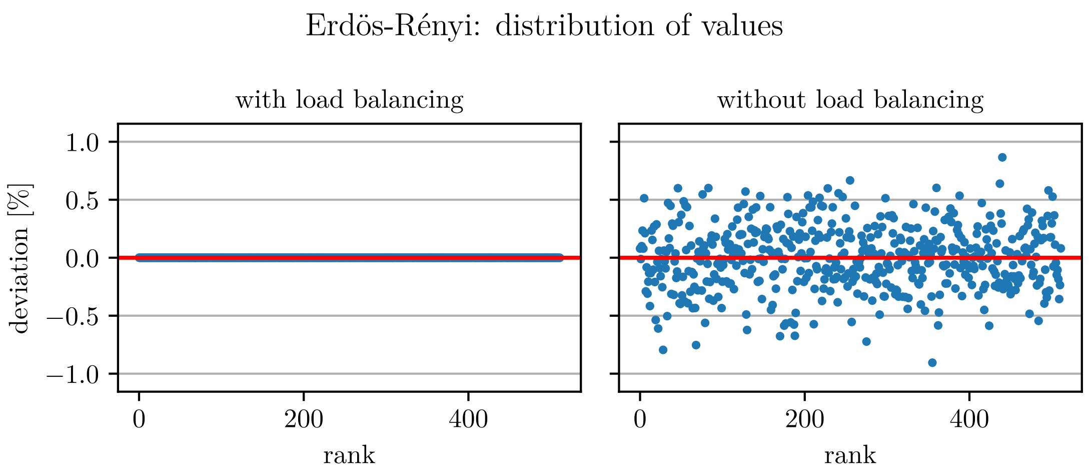

Figure 6.7: Deviation of the number of key-value pairs after the shuffle phase from the mean for the PageRank algorithm executed on an Erdös-Rényi graph with 2^{27} vertices and 2^{31} edges.

is not surprising because the Erdös-Rényi model creates a graph with an expected equal distribution of edges amongst vertices. This leads to vertices having similar degrees, notably, there are no high-degree vertices in the graph. This is corroborated by Figure 6.7, which shows the distribution of key-value pairs after the shuffle phase. The key-value pairs are more evenly distributed in the load-balancing case, but even without load-balancing, the deviation from the mean is below 1%. For our load-balancing algorithm, we even achieve a maximum deviation of 0.0011%. This is because the Erdös-Rényi graph does not contain any high-degree vertices. Therefore, the PageRank algorithm does not produce any especially big keys which disturb the calculation of the prefix sum. Hence, load balancing has an impact on the execution time of the reduce phase for the R-MAT graph but not the Erdös-Rényi graph.

6.3 K-Means Clustering

We test our MapReduce library with k-Means clustering. To create a data set as input, we use the standard random number generator provided by the C++ standard library. We generate 2^{26} 3-dimensional points and chose $k = 2^{15}$ points as initial centres. Then the map and reduce function described above are applied to the input data. Figure 6.8 shows the execution time of five iterations of the k-means clustering algorithm.

K-means clustering performs better with explicit load-balancing than without. Figure 6.8 shows that the reduction of values, when the values are more evenly distributed with load-balancing, is faster than without load balancing. On 4096 processes, the execution time decreases from 50 seconds to 39 seconds with load balancing. This is a 22% decrease in execution time.

Figure 6.9 shows the deviation of key-value pairs from the mean with and without load balancing. Load balancing achieves significantly less deviation from the mean than without

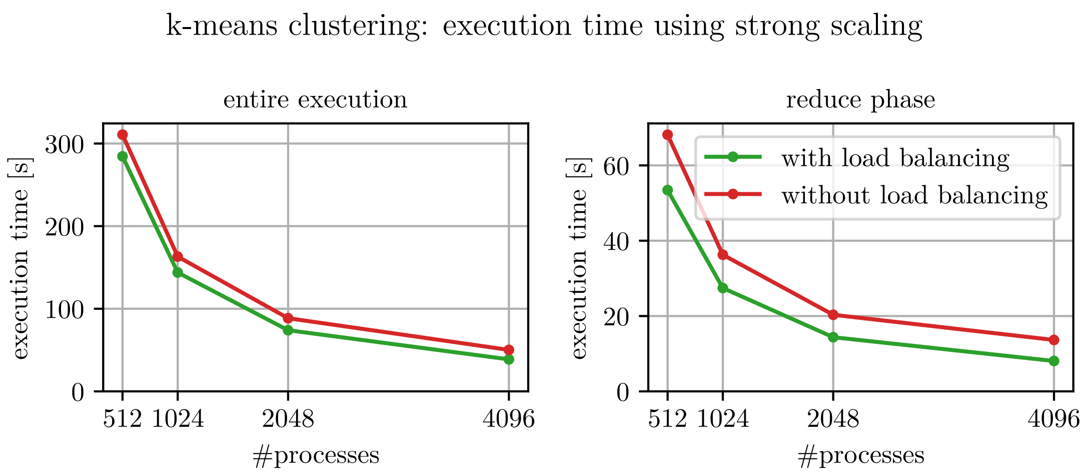

Figure 6.8: Execution time of the k-means clustering algorithm with 2^{26} 3-dimensional points and $k = 2^{15}$ centres.

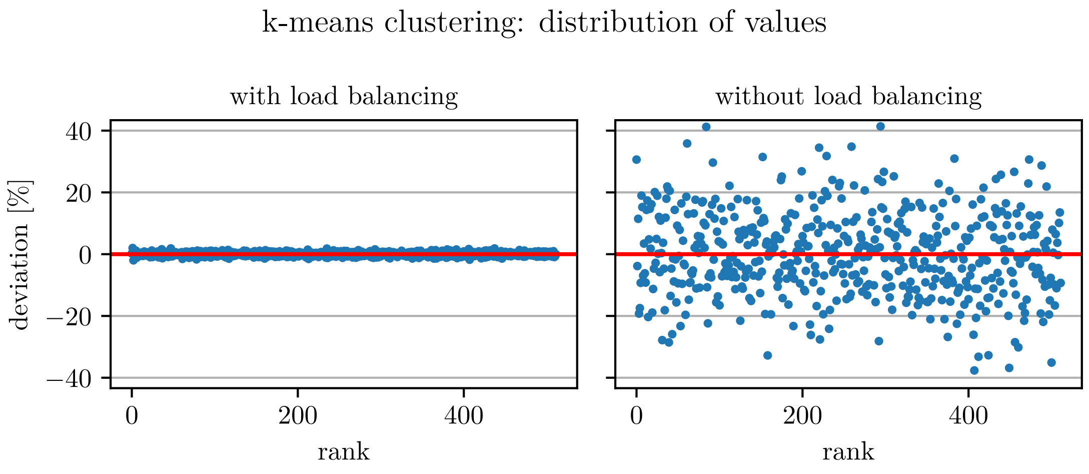

Figure 6.9: Deviation of the number of key-value pairs after the shuffle phase from the mean for the k-means clustering algorithm with 2^{26} 3-dimensional points and $k = 2^{15}$ centres on 512 processes.

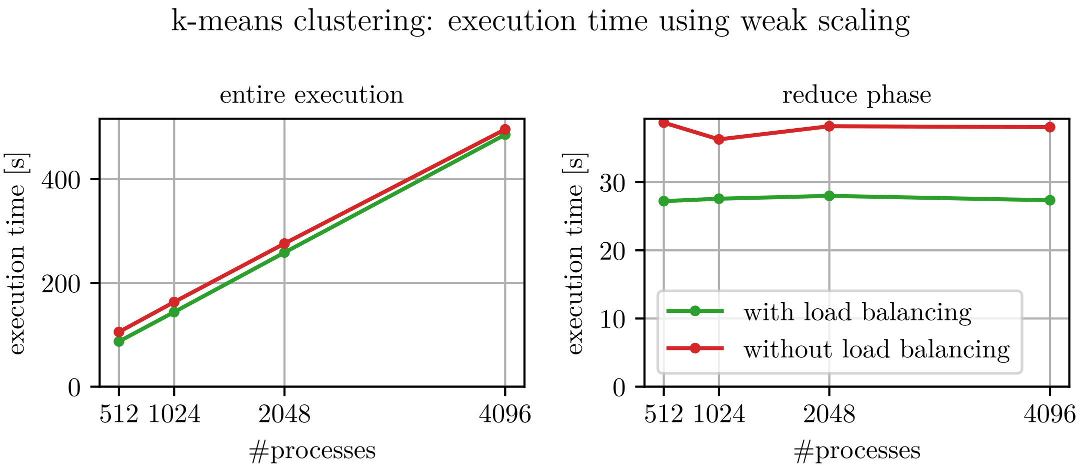

Figure 6.10: Execution time of the k-means clustering algorithm using weak scaling with 2^{18} 3-dimensional points and $k = 2^5$ centres per process.

load balancing. We reduce the maximum deviation from 41.4% to 2%. This is why the reduction in the load-balancing case is faster than without it. The reduction function for k-means clustering also has quadratic complexity, which means additional values have a bigger impact on the execution time of the reduce phase.

Figure 6.10 shows the execution time with weak scaling. The entire execution time increases significantly with more processes, but the execution time of only the reduce phase barely increases. Figure 6.11 shows the breakdown of the entire execution time into the phases. The execution time is dominated by the map and the reduce phase. The map phase, in this case, is responsible for the increase in execution time. Because k is increased proportionally to the number of processes used, for k-means clustering on 4096 processes during the map phase, each point needs to be compared to eight times the number of centres as compared to on 512 processes. This means that even though the number of points and centres per process is the same for 512 and 4096, the workload for a point during the map phase is bigger on 4096 processes. This leads to a drastic increase in execution time using weak scaling.

It is clear that for k-means clustering, load balancing provides an advantage. Further, this leads to the conclusion that load balancing is especially useful in cases where the application of the reduce function takes a while.

6.4 WordCount

We use the WordCount application to test our MapReduce library. As described above, WordCount takes a text as input and computes how often each word occurs in the text. As input, we use ebooks provided by Project Gutenberg [13]. We create a txt file containing the 100 books listed under ‘Top100 books last 7 days’. To increase the input size, the input file is read multiple times during the map phase.

Figure 6.12 shows the entire execution time and the time spent during the reduce phase.

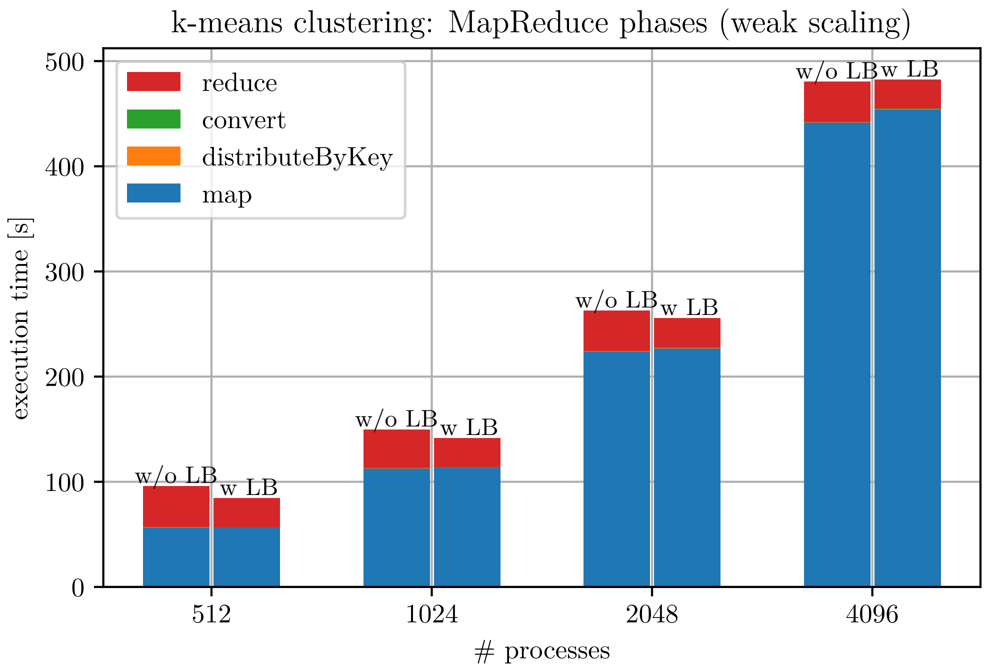

Figure 6.11: Time spent in each phase during execution of the k-means algorithm using weak scaling with 2^{18} 3-dimensional points and $k = 2^5$ centres per process.

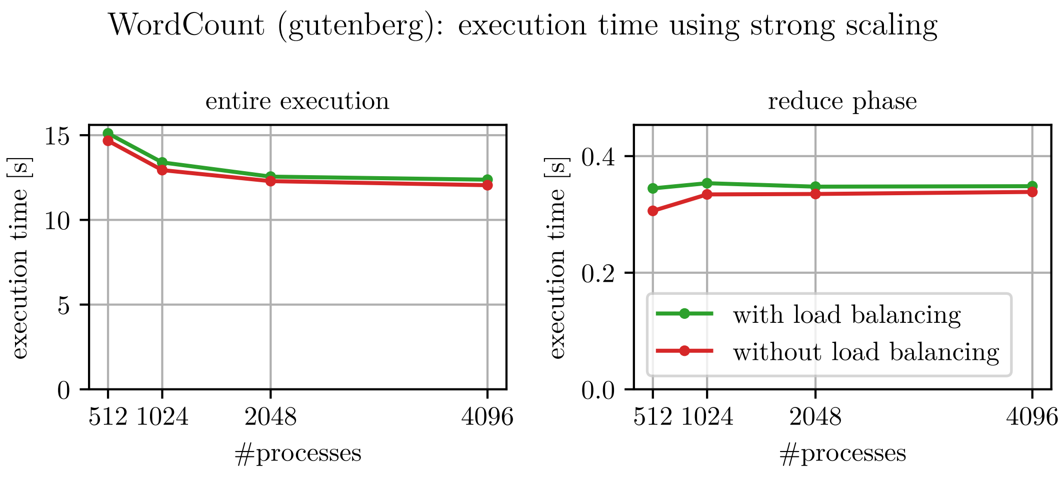

Figure 6.12: Execution time of WordCount with 100 different books as input.

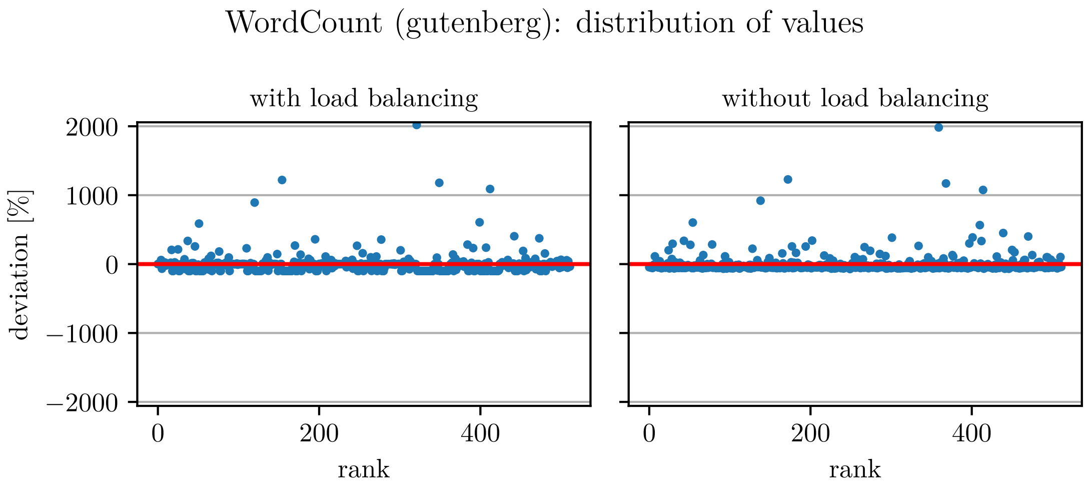

Figure 6.13: Deviation of the number of key-value pairs after the shuffle phase from mean for WordCount with the gutenberg books input.

The entire execution time decreases a little for more processes, but the execution time is again faster without load balancing. The execution time of the reduce phase does not decrease as expected.

Figure 6.13 provides some indication to explain the behaviour during the reduce phase. Load balancing does not lead to a significantly better distribution of key-value pairs after the shuffle phase. Notably, in both cases, there is one processor which receives 2000% of the ideal amount of key-value pairs. The 100 books used as input collectively contain about 13 Mio. words. ‘The’ is the most common word and occurs 511,371 times in the 100 books. This means ‘the’ makes up nearly 4% of all words in the input file. Ideally, if we use 512 processors to process the input file, each processor would receive about 0.2% of all words in the input file. Because one processor receives the key ‘the’, it receives about 4% of all words in the input file, which is 2000% more key-value pairs than the ideal number of key-value pairs. This deviation from the ideal distribution increases with the increase in processors. For an execution of WordCount on 1024 processors, each processor would ideally only process 0.01% of all words. In this case, the processor receiving the key ‘the’ receives 4000% of the ideal number of key-value pairs. This leads to the execution time of the reduce phase not decreasing with an increase in processors used.

Figure 6.14 shows the execution time of WordCount with weak scaling using ebooks from Project Gutenberg. The execution time is split up into phases to provide insight into where the execution time is spent and how a disproportionally big key impacts the execution time. The unequal distribution of key-value pairs, even with load balancing, leads to the convert phase dominating the execution time. Some keys being very big does not have as much of an effect on the distributeByKey phase.

At the beginning of the map phase, the key-value pairs are distributed evenly because the input text is distributed evenly. For the computation of the prefix sum, each processor first locally computes the size of a key, and these local sizes are then used to compute

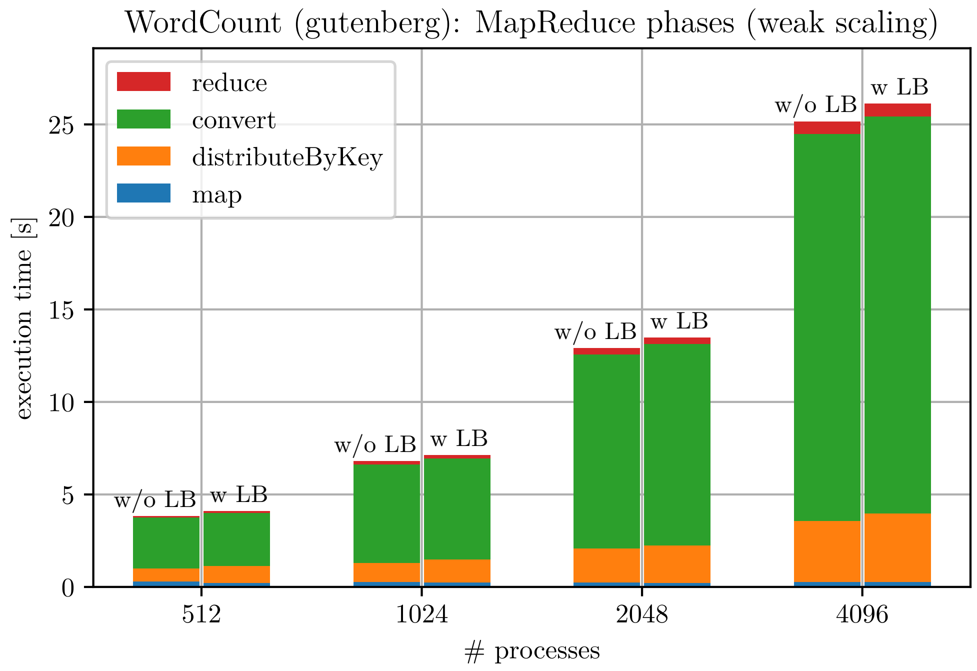

Figure 6.14: Execution time of WordCount using weak scaling with gutenberg books as input.

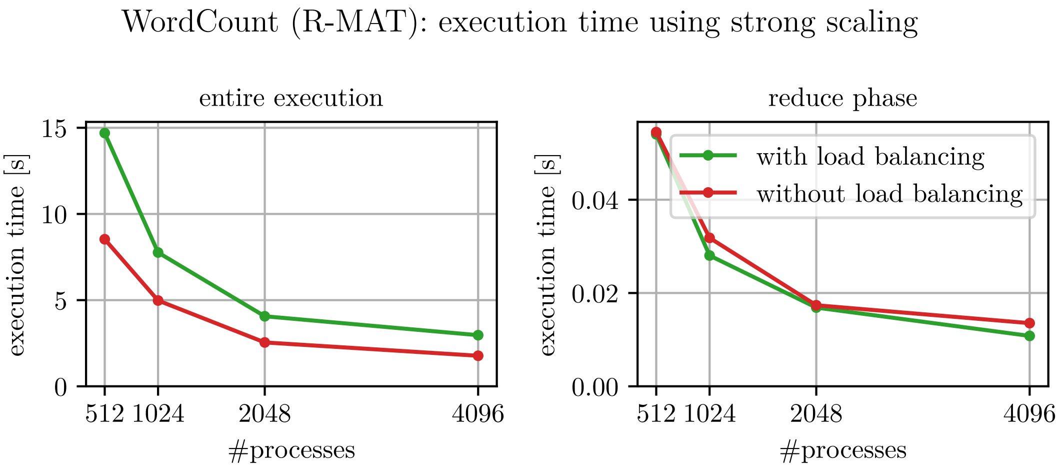

Figure 6.15: Execution time of WordCount with R-MAT edge list as input.

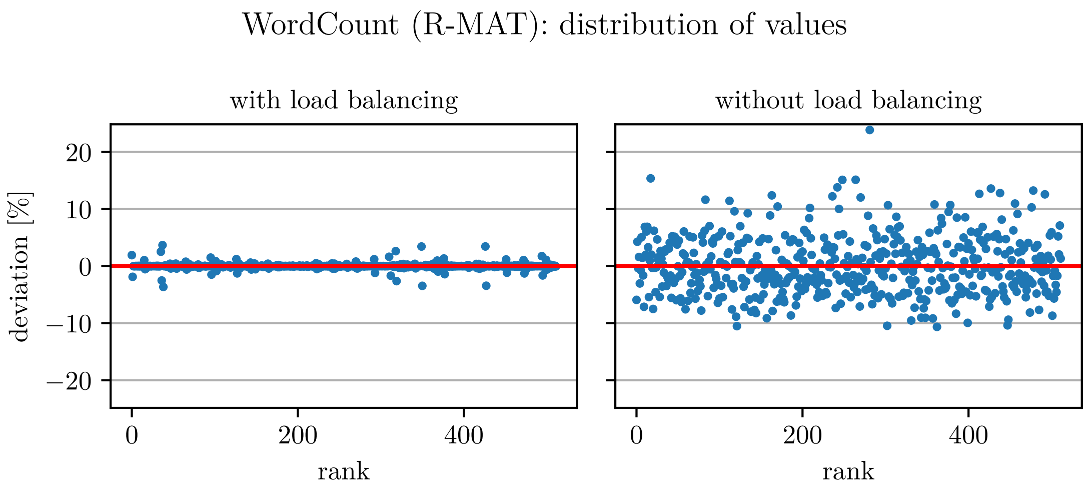

Figure 6.16: Deviation of the number of key-value pairs after the shuffle phase from the mean for WordCount with R-MAT edge list as input.

the total size of the key. The local computation of sizes is not influenced by the big keys, because the key-value pairs are still distributed evenly. Computing the total size of keys and the prefix sum is not influenced by the big keys because the preceding computation of the local sizes means that, even for the big keys at maximum, p key-size pairs are needed to calculate the total size. Finally, preparing the key-value pairs to send them to the right processor is not impacted by the big keys because the key-value pairs are distributed evenly after the map phase. The convert phase is, however, impacted by the big keys because at the end of the distributeByKey phase, the key-value pairs are exchanged, and all pairs with the same key are sent to the same processor. Therefore, one processor receives all key-value pairs for a big key and collecting all values for a key for a very big number of values takes time.

To evaluate the performance of our MapReduce library for the WordCount application in a less unbalanced environment, we use the edge list of an R-MAT graph as input. Each line in the edge list contains two numbers representing an edge between the two vertices with those numbers as indices. WordCount, therefore, calculates the degree of each vertex. We use the same parameters for generating the edge list as during PageRank, which leads to some vertices having a much higher degree than others and therefore creating imbalances in the key sizes.

Figure 6.15 shows the execution time of WordCount with the R-MAT input. The execution time is again faster without load balancing, but the execution time of the reduce phase is a little bit better with load balancing. Figure 6.16 shows that the load-balancing algorithm leads to a more even distribution of key-value pairs, as expected. WordCount performs as expected without disproportionally big keys.

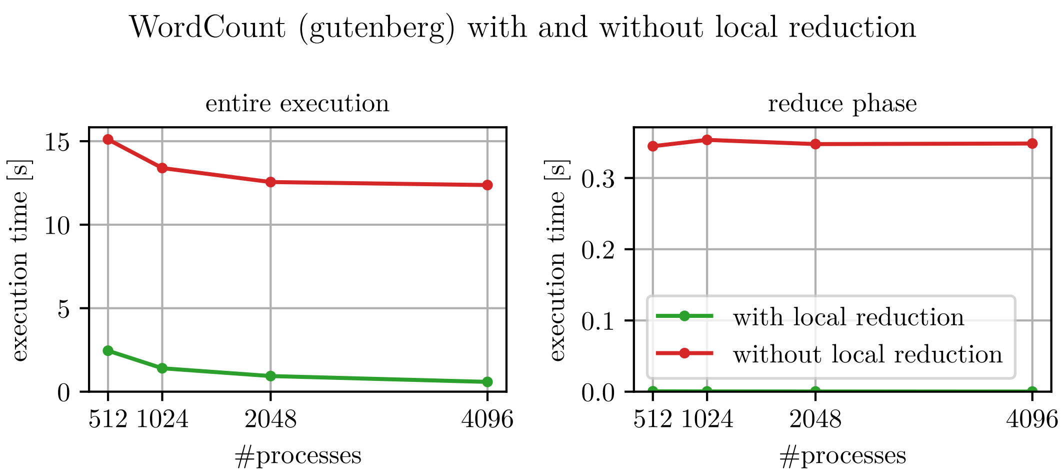

Figure 6.17: Execution time of WordCount with load balancing with gutenberg books as input with and without using local reduction.

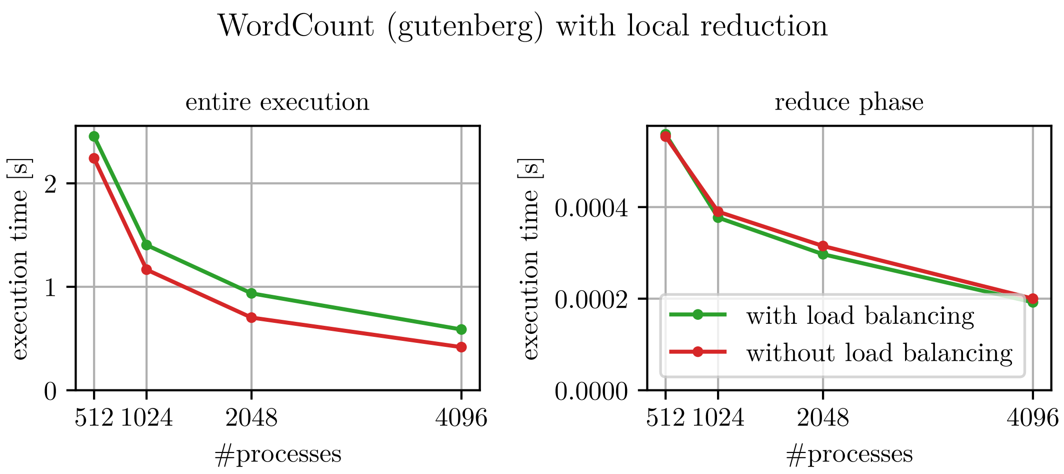

Figure 6.18: Execution time of WordCount with and without load balancing with gutenberg books as input using local reduction.

6.5 Local Reduction

We saw with WordCount that disproportionately large keys lead to our load-balancing algorithm not working as desired, but they are also a problem without load balancing. One way to prevent these very large keys is to perform a local reduction after the map phase and before the shuffle phase. This means collecting all values for a key that are local to the process after the map phase and performing a local reduction on these. For WordCount, this local reduction takes the form of building the sum of the values. This local reduction is possible for WordCount because the WordCount reduction is just building a sum and is, therefore, commutative and associative and can be split up into steps. This local reduction is also offered by other existing frameworks like Hadoop, but it is not applicable for all MapReduce applications. For example, it is not possible for our version of k-means clustering, as for our k-means clustering, it is necessary to compare all values with each other.

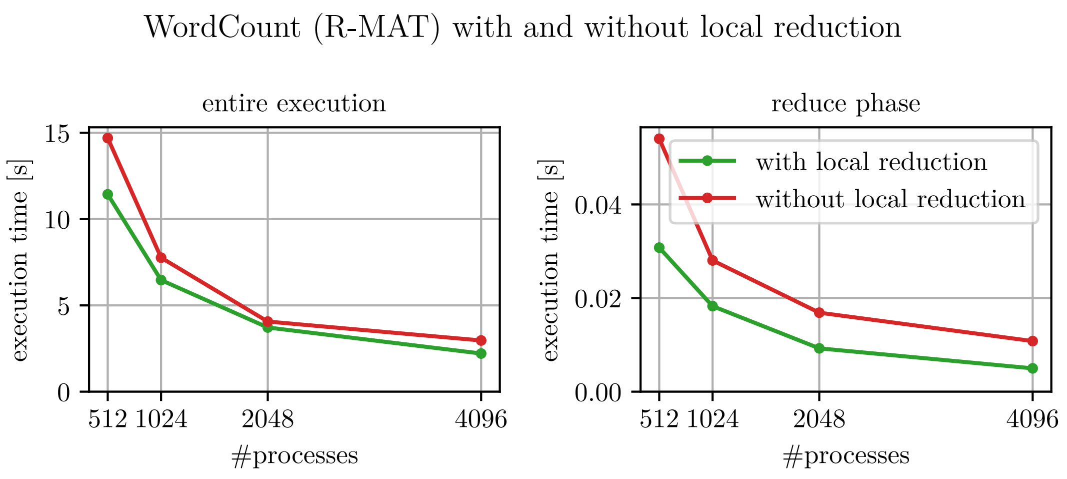

Figure 6.19: Execution time of WordCount with load balancing with R-MAT edge list as input with and without using local reduction.

We test this for WordCount by building the sum of all values for a key after the map phase. Figure 6.17 shows the execution time of WordCount using load balancing with and without using local reduction with the gutenberg ebooks as input. Using the local reduction results in a drastically improved execution time. The execution time of the reduce phase with local reduction is very small. This is to be expected because the bottlenecks during convert and reduce created by high frequency words like ‘the’ do not exist anymore. After the local reduction, the maximum number of key-value pairs per key is p because each process contains at most one key-value pair for each key.

Figure 6.18 shows the local reduction with and without load balancing. Load balancing leads to an increase in the entire execution time and has no significant impact on the reduce execution time. Again, this is to be expected as the execution time of the reduce phase is very fast.

Then, we evaluate the local reduction on input that does not contain disproportionately large keys. Figure 6.19 shows the execution time of the local reduction on WordCount with load balancing with an R-MAT edge list as input. The execution time of the reduce phase is significantly reduced with local reduction. This is to be explained by the reduced number of key-value pairs due to the local reduction. The entire execution time is also a little reduced, but local reduction has less of an impact on the execution time when no disproportionately large keys are present.

6.6 MR-MPI

Finally, we compare our MapReduce library to an already established one. We compare our library to the MR-MPI library as it is also MPI-based. We compare the execution times for the three applications we chose. As our implementation is in memory, we set up MR-MPI to also only execute in memory to not cause any additional overhead with IO-operations. In the case of WordCount, this meant increasing the ‘page’ size of MR-

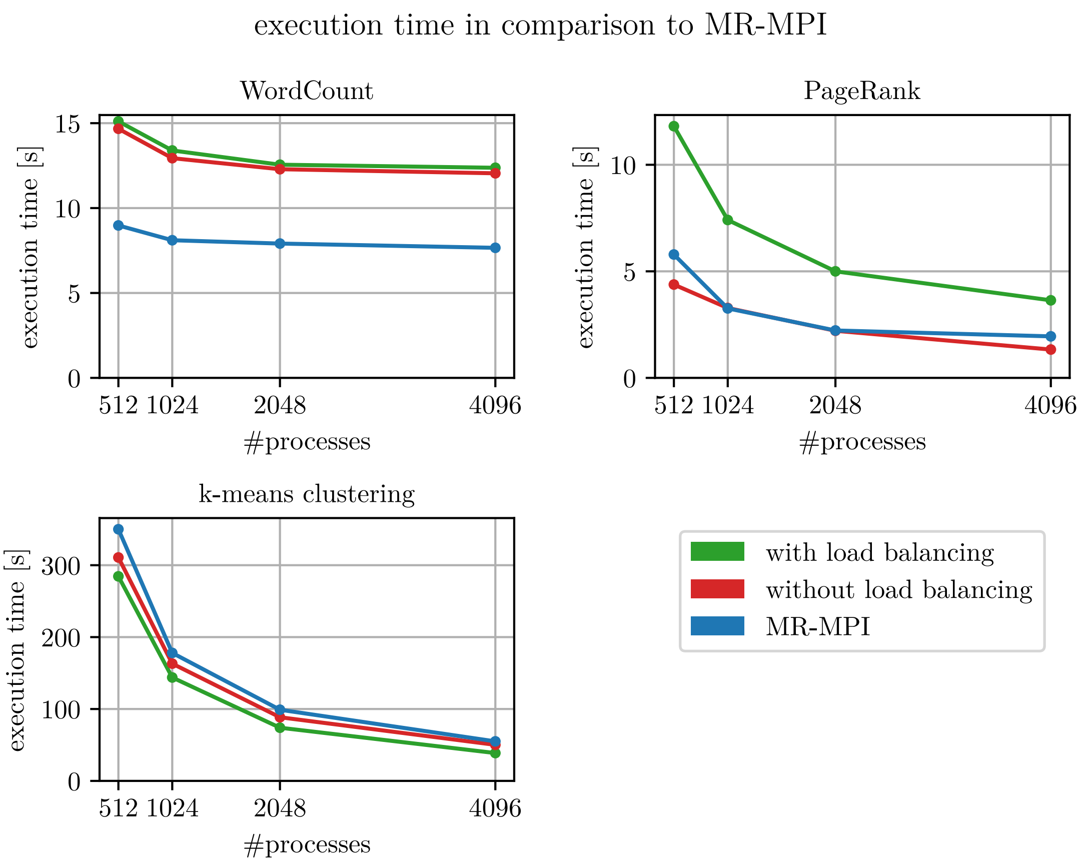

Figure 6.20: Execution time of the three applications: WordCount, PageRank and k-means clustering using our MapReduce library with and without load balancing and using the MR-MPI library.

MPI.

Figure 6.20 shows the execution time for all three applications using MR-MPI and our MapReduce library with and without load balancing. For PageRank, our library without load balancing performs roughly the same as MR-MPI. For k-means cluster, our library without load balancing performs better than MR-MPI. However, this is caused by the hash functions to determine where to send a key being different. In this case, MR-MPI allocated more keys to a process than our library. Because k-means clustering has quadratic complexity in the reduce phase, this leads to a comparatively large increase in execution time during the reduce phase and causes the longer execution time overall. Finally, WordCount performs better for MR-MPI. This is because MR-MPI uses variably sized keys. As our MapReduce library does not have variably sized keys, our WordCount keys all have a size of 20 characters to accommodate for longer words. However, because many words, and most importantly the most frequent words, are shorter than 20 characters, this leads to additional overhead, e.g. when sending key-value pairs or copying keys into send buffers etc. As MR-MPI does not have this limitation, it has a shorter execution time for WordCount.

7 Conclusion

In this thesis, we developed a MapReduce library built on top of the KaMPIng library. We implemented the load-balancing algorithm for the shuffle phase proposed by Sanders. With this algorithm, the key-value pairs are distributed more evenly amongst the PEs using a combination of hashing and a prefix sum. We also implemented MapReduce without explicit load balancing and using just a standard hash-based approach during the shuffle phase. We adjusted the algorithm to practice and evaluated its performance in comparison to the hash-based shuffle. For this, we used three different applications: PageRank, k-means clustering and WordCount.

Our experiments show that our load-balancing algorithm achieves a more even distribution of key-value pairs after the shuffle phase. This is most notable for skewed input data. For skewed input, our load-balancing algorithm achieves up to 39% less deviation from a completely even distribution of key-value pairs. This resulted in a reduction of the execution time of up to 22% in the cases where a substantial part of the execution time is spent in the reduce phase. If the reduce phase does not make up a substantial part of the entire execution time, the reduced time of the reduce phase achieved by the load balancing does not offset the additional time accrued by the calculation of the balanced distribution of key-value pairs during the shuffle phase.

Future Work We have shown that load balancing can be useful, but there is room to improve our MapReduce library. Some smaller improvements to reduce the overhead of calculating the load balance could include, e.g. keeping track of the balanced partitioning of keys to machines if the sizes of keys stay the same over multiple iterations to avoid having to recalculate the partitioning of keys to machines every iteration. Other larger improvements could include the handling of disproportionately large keys because, as seen during our evaluation, disproportionately large keys cause problems. Apart from the local reduction we explored, large keys could be split up during the shuffle phase, and their reduction could happen in parallel on multiple machines. This would even be possible for applications where local reduction is not a possibility.

Further, other load-balancing techniques should be explored, most importantly the other load-balancing algorithm proposed by Sanders, the work stealing algorithm, which is a dynamic load-balancing algorithm. Exploring this load-balancing algorithm in practice could be very interesting as it does not contain the same overhead of our static load-balancing algorithm of calculating sizes of keys, instead, machines signal if they have no more work left, and other machines can send them work.

Apart from exploring further load-balancing techniques, our MapReduce library could be expanded in other ways to make it more usable. Those improvements could include making it possible to use flexible-sized keys and values, which we did not include due to time constraints. Another possibility would be to introduce fault tolerance as currently the entire execution needs to be started from scratch, should a process fail.

Bibliography

[1] Graph 500. *Graph 500.* URL: https://graph500.org/ (visited on 22/02/2025).

[2] Prajesh P. Anchalia, Anjan K. Koundinya and Srinath N. K. ‘MapReduce Design of K-Means Clustering Algorithm’. In: *2013 International Conference on Information Science and Applications (ICISA).* 2013, pp. 1–5. DOI: 10.1109/ICISA.2013.6579448.

[3] Loris Belcastro et al. ‘Programming big data analysis: principles and solutions’. In: *Journal of Big Data* 9.4 (2022). URL: https://doi.org/10.1186/s40537-021-00555-2.

[4] Sergey Brin and Lawrence Page. ‘The anatomy of a large-scale hypertextual Web search engine’. In: *Computer Networks and ISDN Systems* 30.1 (1998). Proceedings of the Seventh International World Wide Web Conference, pp. 107–117. ISSN: 0169-7552. DOI: https://doi.org/10.1016/S0169-7552(98)00110-X. URL: https://www.sciencedirect.com/science/article/pii/S016975529800110X.

[5] Deepayan Chakrabarti, Yiping Zhan and Christos Faloutsos. ‘R-MAT: A Recursive Model for Graph Mining’. In: *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*, pp. 442–446. DOI: 10.1137/1.9781611972740.43. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43.

[6] Jeffrey Dean and Sanjay Ghemawat. ‘MapReduce: simplified data processing on large clusters’. In: *Commun. ACM* 51.1 (2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[7] Neelam Duhan, A. K. Sharma and Komal Kumar Bhatia. ‘Page Ranking Algorithms: A Survey’. In: *2009 IEEE International Advance Computing Conference.* 2009, pp. 1530–1537. DOI: 10.1109/IADCC.2009.4809246.

[8] Michalis Faloutsos, Petros Faloutsos and Christos Faloutsos. ‘On power-law relationships of the Internet topology’. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.* SIGCOMM ’99. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1999, pp. 251–262. ISBN: 1581131356. DOI: 10.1145/316188.316229. URL: https://doi.org/10.1145/316188.316229.

[9] Apache Software Foundation. *Apache Hadoop.* 2006. URL: https://hadoop.apache.org/ (visited on 14/01/2025).

[10] Apache Software Foundation. *Apache Hadoop - HDFS Architecture.* 2006. URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (visited on 14/01/2025).

[11] Apache Software Foundation. *Apache Hadoop - MapReduce Tutorial.* 2006. URL: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html (visited on 14/01/2025).

[12] Apache Software Foundation. *Apache Spark.* 2013. URL: https://spark.apache.org/ (visited on 27/01/2025).

[13] Project Gutenberg Literary Archive Foundation. *Project Gutenberg.* URL: https://www.gutenberg.org/ (visited on 05/01/2025).

[14] Daniel Funke et al. 'Communication-free massively distributed graph generation'. In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 200–217. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2019.03.011`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731518304684`.

[15] Tao Gao et al. 'Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems'. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 1098–1108. DOI: `10.1109/IPDPS.2017.31`.

[16] Lorenz Hübschle-Schneider and Peter Sanders. 'Linear Work Generation of R-MAT Graphs'. In: *Network Science* 8.4 (2020), pp. 543–550.

[17] Shadi Ibrahim et al. 'LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud'. In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. 2010, pp. 17–24. DOI: `10.1109/CloudCom.2010.25`.

[18] Howard Karloff, Siddharth Suri and Sergei Vassilvitskii. 'A Model of Computation for MapReduce'. In: *Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 938–948. DOI: `10.1137/1.9781611973075.76`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9781611973075.76`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9781611973075.76`.

[19] YongChul Kwon et al. 'SkewTune: mitigating skew in mapreduce applications'. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 25–36. ISBN: 9781450312479. DOI: `10.1145/2213836.2213840`. URL: `https://doi.org/10.1145/2213836.2213840`.

[20] Yanfang Le et al. 'Online load balancing for MapReduce with skewed data input'. In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2014, pp. 2004–2012. DOI: `10.1109/INFOCOM.2014.6848141`.

[21] Shi Na, Liu Xumin and Guan Yong. 'Research on k-means Clustering Algorithm: An Improved k-means Clustering Algorithm'. In: *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*. 2010, pp. 63–67. DOI: `10.1109/IITSI.2010.74`.

[22] Steven J. Plimpton and Karen D. Devine. 'MapReduce in MPI for Large-scale graph algorithms'. In: *Parallel Computing* 37.9 (2011). Emerging Programming Paradigms for Large-Scale Scientific Computing, pp. 610–632. ISSN: 0167-8191. DOI: `https://doi.org/10.1016/j.parco.2011.02.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0167819111000172`.

[23] Smriti R. Ramakrishnan, Garret Swart and Aleksey Urmanov. 'Balancing reducer skew in MapReduce workloads using progressive sampling'. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: `10.1145/2391229.2391245`. URL: `https://doi.org/10.1145/2391229.2391245`.

[24] Erdos Renyi. 'On random graph'. In: *Publicationes Mathematicate* 6 (1959), pp. 290–297.

[25] Peter Sanders. 'Connecting MapReduce Computations to Realistic Machine Models'. In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 84–93. DOI: `10.1109/BigData50022.2020.9378039`.

[26] Zhuo Tang et al. ‘An intermediate data placement algorithm for load balancing in Spark computing environment’. In: *Future Generation Computer Systems* 78 (2018), pp. 287–301. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2016.06.027`. URL: `https://www.sciencedirect.com/science/article/pii/S0167739X16302126`.

[27] Tim Niklas Uhl et al. ‘KaMPIng: Flexible and (Near) Zero-Overhead C++ Bindings for MPI’. In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2024, pp. 1–21. DOI: `10.1109/SC41406.2024.00050`.

[28] Matei Zaharia et al. ‘Spark: cluster computing with working sets’. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, p. 10.