Karlsruhe Institute of Technology

Master's Thesis

Scalable Decentralized Fault-Tolerant MapReduce for Iterative Algorithms

Charel Mercatoris

Date: 01.09.2021

Reviewer: Prof. Dr. Peter Sanders
Advisor: M.Sc. Demian Hespe
M.Sc. Lukas Hübner

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

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

Karlsruhe, 30.08.2021

Zusammenfassung

Mit der Zunahme von parallelen Rechenknoten in Hochleistungsrechnersystemen steigt die Wahrscheinlichkeit von Hardwareausfällen für Anwendungen, welche auf Millionen von Knoten laufen. Aus diesem Grund müssen zukünftige Anwendungen Knotenausfälle erwarten, erkennen und behandeln.

Im Ramen dieser Arbeit entwickeln wir ein dezentrales, fehlertolerantes MapReduce Framework für iterative Algorithmen, das mit Knotenausfall umgehen kann. Unser Framework folgt dem Bulk Synchronous Parallel Modell (BSP) und verwendet eine zufällige statische Lastverteilung während der Shuffle Phase. Wir stellen einen Fehlertoleranzmechanismus vor, welcher während der Shuffle Phase zusätzliche *Selbstnachrichten* austauscht und im Speicher ablegt. Im Gegensatz zu bisherigen Ansätzen speichern wir keine Checkpoints auf einem fehlertoleranten Dateisystem.

Darüber hinaus stellen wir ein rein MPI basiertes MapReduce Framework, sowie eine hybride Parallelisierung mit mehreren OpenMP Threads pro Prozess vor. Wir beweisen die erwartete Laufzeit der verschiedenen MapReduce Phasen und Fehlertoleranzmechanismen.

Wir implementieren die "Word Count", "Page Rank", "Connected Component" und R-Mat Algorithmen in unserem Framework und führen Laufzeitexperimente durch. Wir beobachten einen superlineare Speedup für das Abspeichern und Senden der Selbstnachrichten. Die anderen Phasen, außer die Shuffle Phase, haben optimale Speedups, wenn die Eingabe im Vergleich zu der Anzahl an Prozessen und der Flaschenhals Last groß genug ist.

Abstract

The increase in parallel compute nodes in a high-performance computing (HPC) system is followed by an increase in hardware failures for applications running on millions of nodes. Therefore, future large-scale parallelized applications need to expect, detect and handle compute node failure.

We provide a decentralized fault-tolerant MapReduce framework for iterative algorithms designed to handle a HPC node failure. Our framework follows the bulk synchronous parallel model (BSP) by using random static load balancing during the shuffle phase. We propose a fault-tolerance mechanism, which exchanges additional *self-messages* during the shuffle phase and saves them in memory. In contrast to previous approaches, we do not save checkpoints on a fault-tolerant file system.

Furthermore, we provide a purely MPI based MapReduce framework, as well as a hybrid parallelization using multiple OpenMP threads per process. Moreover, we prove the expected runtime of the different MapReduce phases and the fault-tolerance mechanism.

Finally, we implement the word count, page rank, connected component, and R-Mat algorithms in our framework and perform runtime experiments. We can observe super linear speedups for the save self-message phase. The other phases of our framework, except the shuffle phase have optimal phases if the input is large enough compared to the number of processes and the bottleneck workload.

Contents

List of Figures

List of Tables

Algorithmenverzeichnis

1. Introduction

1.1. Motivation

High-Performance Computing (HPC) systems are designed to solve computationally intensive tasks by large-scale parallelization. In the future, the number of compute nodes in a HPC system is predicted to further increase. The exascale computing project suggests for instance to employ up to 10^6 nodes with 1000 cores each to achieve more than 10^{18} floating point operations per second [37, 67].
The probability of a failure during an execution on a HPC system increases with the number of used nodes and cores [38, 92]. The mean time between failures (MTBF) is a metric indicating the average time between two successive failures. Cappello et al. [92] predict that a conservative MTBF of 100 years for a compute node implies a failure every 53 minutes on average for an execution on 10^6 nodes. Furthermore, hardware failures are expected to become more frequent [91, 92]. The transistors and wires get smaller, which are more likely to fail. Moreover, they age more rapidly, which further increases their failure rate. Therefore, future large-scale parallelized applications need to expect, detect and handle compute node failure.
Processing large amounts of data requires a distribution of the computation between multiple machines [35]. Hence, software engineers need to take data distribution, parallelization, fault-tolerance, and load balancing into account. This led to the introduction of MapReduce libraries, which provide these functionalities [35]. Due to its simplicity and scalability, MapReduce is popular for 'big data' applications [63, 86].
Many MapReduce frameworks follow the master/worker design pattern and are designed to run on commodity hardware. Moreover, they do not employ the message passing interface (MPI) mostly used on high-performance computing (HPC) systems [16, 34, 35, 39]. While most MPI based implementations [46, 77] do not support fault-tolerance, the FT-MRMPI library [52] employs ULFM [23] to handle failures. The user-level failure mitigation (ULFM) MPI implementation provides tools to detect and recover from process failures. FT-MRMPI uses checkpoints saved on a fault-tolerant file system to continue after failure.
To the best of our knowledge, no in-memory decentralized fault-tolerant MPI based MapReduce framework for iterative algorithms without checkpoints to a fault-tolerant file system exists.

1.2. Contribution

In the following thesis we engineer a decentralized, in-memory, and fault-tolerant MapReduce framework for iterative algorithms running on high-performance computing (HPC) systems. We introduce a fault-tolerance mechanism for single process failures, which we generalize for compute node failures. Contrary to previous approaches, we do not save periodic checkpoints on a fault-tolerant file system. The messages send between processes on the same node (*self-messages*) are saved in-memory on a different node. Our MapReduce framework uses these self-messages and the data sent during the previous shuffle phase to recover the data lost by a process failure.
We implement an MPI based MapReduce framework using the bulk synchronous parallel model, with random static load balancing, and supporting single node failures for iterative algorithms. Furthermore, we prove their expected runtime of $\mathcal{O}\left(\frac{m+w}{p}\right)$, where m is the number of machine words, w the total work load, and m as well as w are large enough (Section 6). Moreover, we implement a hybrid MPI version of our fault-tolerant MapReduce framework using multiple OpenMP threads on each process and prove their expected runtime.
Additionally, we implement a Word Count, a PageRank, a connected component, and a R-MAT

MapReduce algorithm in our framework. We use these algorithms as a benchmark to perform strong and week scale experiments on real world data sets. We conclude that saving self-messages scales well with increasing number of processes and is faster than the other phases.

1.3. Structure of Thesis

In Section 2, we introduce a theoretical model for high-performance computing and MapReduce operation as well as some general definitions. Furthermore, we describe the Word Count, PageRank, connected component, and R-MAT MapReduce algorithms (Section 4), which we use as benchmark during our experiments. Section 5 contains the general overview of our MapReduce framework as well as a single process and node fault-tolerance mechanism. We describe the different phases of our purely MPI based algorithm (Section 6) and prove their runtime. Moreover, Section 7 contains our hybrid MapReduce algorithm and its expected execution times. We explain our experimental setup in Section 8, which we use during our runtime experiments in Section 9. We conclude our thesis in Section 10. Finally, Sections A to C contain additional information and figures.

2. Fundamentals

We design a parallel library for execution on high-performance computing systems. We describe a theoretical parallel model in Section 2.1. Our algorithm is based on the MapReduce programming paradigm introduced in Section 2.2. Section 2.3 contains some general graph definitions used in different benchmark algorithms in Section 4. Section 2.4 describes the balls in bins model.

2.1. High-Performance Computing (HPC) Systems

High-Performance Computing (HPC) Systems are designed to solve computationally intensive tasks by large-scale parallelization. As a large network of interconnected machines, HPCs are able to process large amount of data in parallel. Figure 1 illustrates their general structure. A HPC consists of *compute nodes* linked by a *communication network*. For simplicity, we will use the term *node* instead of *compute node* in this thesis. Each *node* has its own local memory, therefore we can model HPC systems by using a distributed memory model [86].
The compute nodes in the HPC we use follow the Non-Uniform Memory Access (NUMA) principle [64]. A compute node can be divided into multiple *NUMA nodes*, which possess their own local memory and a fixed number of *cores*. *Cores* can access the local memory on another NUMA node on the same *compute node*. To access the local memory of a different NUMA node a core needs to request and transfer the data over a network. This is slower than the memory access its own NUMA node, which we have to take into consideration during the implementation of an algorithm.
To execute a parallel program and communicate between processes on different cores we use the Message Passing Interface MPI [29]. MPI starts p *MPI processes* on different cores and provides operations to send messages between them. In this thesis we use the term processes for MPI processes. Each process has its own unique rank between 1 and p, where call n its size. To execute an MPI program on a HPC we have to specify the number of nodes n and tasks t per node. Then the batch system starts $p = n \cdot t$ processes, which can be bound to a specific core. This prevents the process to switch its execution to another core, which could lead to performance loss.

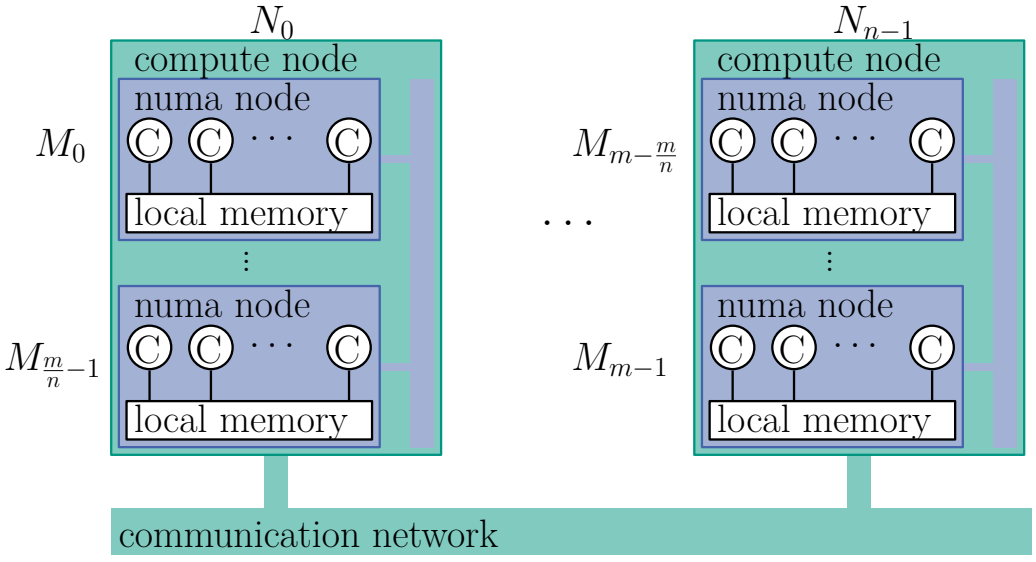


Figure 1: General simplified structure of a high-performance computing systems and illustration of the thoeretical HPC model.

An algorithm using a *hybrid parallelization* starts an MPI processes per node and parallelizes locally using a shared memory library like OpenMP. Then each node can use t parallel *threads*, which have random access to their local shared memory. This approach can reduce the MPI communication and achieve a speedup compared to pure MPI parallelization. Since the shared memory access time of a thread at another NUMA node is slow, we can start an MPI processes per NUMA node.
We model a HPC sytsem (Figure 1) as tuple $S = (C, M, N)$, where C is a set of cores, M is a set of m NUMA nodes and N a set of n compute nodes. Each NUMA node $M_i \in M$ is a set of t cores. The set M partitions C into equally sized and pairwise disjoint sets. The compute nodes N partition the NUMA nodes M into equally sized and pairwise disjoint sets.
We represent an MPI execution on a HPC system $S = (C, M, N)$ as tuple (S, P, γ), where $P = \{1, ..., p\}$ is the set of p MPI processes. The function $\gamma : P \to 2^C$ maps each process to a set of unique cores. Each core corresponds to at most one process and each process has the same nonzero number of cores. We call γ the process binding of an MPI execution. This process binding allows us to allocate physical resources to MPI processes and address the non-uniform memory access problem of HPC compute nodes.

2.2. MapReduce

A *MapReduce* [35, 86] *framework* or *library* is used to apply user-defined functions to a list or multi-set $A \subseteq I$ of *input* elements. We are interested in sequences of MapReduce operations, especially iterative algorithms, where input and output of two successive operations are the same.
A *MapReduce algorithm* is a sequence $< \mathrm{MR}_1, ..., \mathrm{MR}_j, ..., \mathrm{MR}_l >$ of l *MapReduce operations* MR_i. Let $A_i \subseteq I_i$ be a multi-set, where A_i are the *input elements* of MR_i and I_i represents the input type. We define K_i as the set of keys and V_i the set of values. Let $D_i \subseteq O_i$ be the multi-set of output elements, where O_i is the output type. Each MapReduce operation MR_i is a tuple $(\mu_i, \rho_i, s_i, h_i)$, consisting of the following functions:

- A *user-defined map* function μ_i to produce multiple key-value pairs from an input element:
$$\mu_i : I_i \to 2^{K_i \times V_i}$$
- A *user-defined reduce* function ρ_i to map the key-values pairs to output elements:
$$\rho_i : (K_i, \mathrm{list} < V_i >) \to O_i$$
- A bijective serialization function s_i to transform keys and values to and from a byte sequence:
$$s_i : K_i \cup V_i \to \mathrm{list} < bytes >$$
- A function to hash keys with a seed in $\mathbb{N}$:
$$h_i : K_i \times \mathbb{N} \to \mathbb{N}$$

Let MR_i and MR_{i+1} be two successive MapReduce operations with $i \in \{1, ..., l-1\}$. To ensure fault-tolerance we require that the output of MR_i is equal to the input of MR_{i+1}:
$$D_i = A_{i+1}.$$

We omit the indices for readability if we consider only a single MapReduce operation MR. Algorithm 1 and Figure 2 illustrate the application of a user-defined map and reduce function. We can divide a MapReduce operation into three different phases:

1) *map phase*: First, the *user-defined map* function μ produces for each *input* element $e \in A$ zero or multiple *key-value* pairs as *intermediate* data

$$B = \bigcup_{e \in A} \mu(e) \subseteq K \times V,$$

where K is a set of keys, V a set of values and B a list or multi-set.

2) *shuffle phase*: During the *shuffle phase* we group the *intermediate key-value* pairs by their keys and produce *key-values* pairs $(k, X) \in C$, where

$$C = \{(k, X) : k \in K \wedge X = \{x : (k, x) \in B\} \neq \emptyset\}.$$

Note that the *shuffle phase* in a parallel *MapReduce framework* requires communication to gather all *pairs* with the same key on the same process.

3) *reduce phase*: The *reduce phase* applies the *user-defined reduce* function ρ to all *key-values* pairs and produces the *output* elements

$$D = \bigcup_{(k,X) \in C} \rho(k, X) \subseteq O.$$

The set O represents the possible *output* elements, while D is the multi-set or list of *output* elements.

All in all, Lines 2 and 5 in Algorithm 1 represent the *map* and *reduce* phase. Line 3 corresponds to the *shuffle phase*. In the following sections, we will discuss different *MapReduce frameworks* with slightly different *user-defined map* and *reduce* functions, but the general phases and ideas remain the same.
During our runtime analyses, we use the following parameters of the $\mathcal{MRC}^+$ complexity class (Section 3.2,[86]). Let MR=(μ, ρ, s, h) be a MapReduce operation. Let w be the total time needed to apply the user-defined map μ or reduce ρ functions on all input elements A sequentially. This includes serializing and deserializing all key-value pairs as well as hashing their keys. The term $\bar{w}$ indicates the maximum time to apply a user-defined map μ or reduce ρ function. Let m bet the number of machine words needed to save the input A, output D and intermediate elements B. The user-defined map or reduce functions produce or consume at most $\bar{m}$ machine words.

Algorithm 1: High level map reduce application

`Input`: $A, \mu : I \to \text{list}(K, V)$,
$\rho : (K, \text{list}(V)) \to \text{list}(O)$

1 $B : \text{list}(K, V)$
2 `foreach` $e \in A$ `do` $B = B \cup \mu(e)$
3 $C = \text{shuffle}(B)$
4 $D : \text{list}(O)$
5 `foreach` $(k, X) \in C$ `do` $D = D \cup \rho(k, X)$
6 `return` D

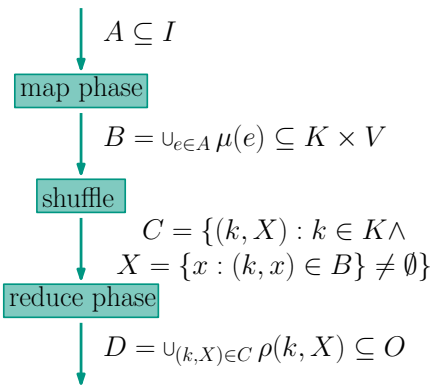


Figure 2: Illustration of the phases in a MapReduce operation [86].

2.3. Graph Theory

In this section we list our graph theory related definitions and notations, which we use in our benchmark algorithms in Section 4. Let $G = (V, E)$ be a *directed graph*, with a *vertex set* V and a *edge set* $E \subseteq V \times V$. We call $v \in V$ a *vertex* and $(x, y) \in E$ an *edge*. The vertices x and y are called endpoints of edge (x, y). In this thesis we do not allow the synonym *node* for a vertex, since this term is reserved for compute nodes in a HPC system. We define the set of *out-neighbors* of a vertex $v \in V$ as $O(v) = \{u|(v, u) \in E\}$ and the set of *in-neighbors* as $I(v) = \{u|(u, v) \in E\}$. We define $O_+(v) = O(v) \cup \{v\}$ as the outgoing neighbors of $v \in V$ including v. Similar we set $I_+(v) = I(v) \cup \{v\}$ for $v \in V$. Let $S \subseteq V$ be a subset of vertices of the graph G. The *induced subgraph* $G[S] = (S, E')$ of G is the graph with vertices S and edges $E' = \{(u, v) \in S \times S|(u, v) \in E\}$ containing the edges of $e \in E$ with endpoints in S. A *path* in G is a sequence $(v_1, v_2, ..., v_n)$ of pairwise distinct vertices $v_i \in V$, with $(v_i, v_{i+1}) \in E$. The *length* of a path is the number of edges connecting its vertices. For a vertex set S and graph G, the induced subgraph $G[S] = (S, E')$ is a *connected component* if each distinct vertices $u, v \in S$ are connected by a path from u to v and v to u.

2.4. Balls in Bins

The balls in bins model analyzes the action of sequentially assigning m balls into n bins. We chose each bin interdependently and uniformly at random. We are interested in the expected maximum number $b(m, p)$ of balls in a single bin, where m indicates the number of balls and p the number of bins. In the context of random static load balancing $b(m, p)$ indicates the maximum number of subproblems allocated to a process. We use m for the number of subproblems and p for the number of processes.
The analysis of algorithms online load balancing use the balls in bins model [78]. While using hash tables, we assign elements to locations according to a hash function. This function should distribute elements with an equal probability. We can use the balls in bins model to analyze hash conflicts, which occur if multiple elements are mapped to the same location. During random online load balancing, a network of n servers handles m requests. If we assign requests randomly between the servers, we can analyze the problem using the balls in bins model.
Our MapReduce library combines hashing and random static load balancing. We use the balls in bins model to analyze the time complexity (Section 7). In particular, we require Theorem 2.1 to get a bound for the expected maximum number $b(m, p)$. Raab et al. [78] prove Theorem 2.1, the for us relevant case of their Theorem 1.

Theorem 2.1. *Let M be the random variable counting the maximum number of balls in a bin. We consider the balls in bins model with m balls and p bins. Then $P(M > k_\alpha) = o(1)$ if $\alpha > 1$ and*

$$k_\alpha = \begin{cases} \frac{m}{p} + \alpha\sqrt{\frac{2m}{p}\log(p)} & \text{if } p\log(p) \ll m \leq p\,\text{polylog}(p) \\ \frac{m}{p} + \alpha\sqrt{\frac{2m\log(p)}{p}\left(1 - \frac{1}{\alpha}\frac{\log\log(p)}{2\log(p)}\right)} & \text{if } p\log^3(p) \ll m \end{cases}.$$

Proof. Raab et al.[78] prove this theorem. □

During our run time analyzes in Sections 6 and 7 we require Lemma 2.2.

Lemma 2.2. *Given the balls in bins problem with m balls and p bins with expected maximum occupancy $b(m, p)$, $m \in \mathbb{N}$ and $p \in \mathbb{N} \setminus \{1, 2\}$. If $m \in \Omega(p\log(p))$, then $b(m, p) \in \mathcal{O}\left(\frac{m}{p}\right)$.*

Proof. This Lemma follows directly from Theorem 2.1. We perform a more detailed proof in Section A. □

3. Related Work

In Section 3.1 we discuss how MPI applications can be made fault-tolerant using different mechanisms. In particular, this section introduces the User-Level Failure Mitigation library (ULFM). We design our framework with ULFM in mind. We present a theoretical analysis of MapReduce computations using realistic machine models in Section 3.2. Section 3.3 introduces six MapReduce frameworks designed for multiple platforms and different optimization goals. Section 3.6 provides us with a worst-case analysis for randomized static load balancing used in our MapReduce library.

3.1. Fault-Tolerant MPI

The Message Passing Interface (MPI) is a library used to efficiently parallelize applications running on high-performance computing (HPC) systems [29]. An MPI program starts multiple processes in parallel on multiple cores, which can be situated on different compute nodes. This library provides operations to communicate between these MPI processes. MPI can detect if a communication between processes was unsuccessful, but does not provide any functions to recover in case one of the parallel processes fails and crashes. Standard MPI programs need to abort the execution in the presence of a failure. The current state of an MPI implementation is saved in a communicator, which is damaged in case of a failure.
One approach to recover MPI applications from failures is to rely on a rollback recovery strategy [23]. In case of a process failure the application aborts and restarts. By saving checkpoints during execution, the restarted MPI program can resume from a later state and does not need to recompute from the beginning. An example of this approach is the *detect/restart* fault-tolerance mechanism of FT-MRMPI described in Section 3.3.4.
The CoCheck MPI implementation [93] provides fault-tolerance by performing periodical synchronous checkpoints. During an execution of an application using CoCheck MPI, the library makes and saves the state of the entire application automatically. In case a failure occurs, CoCheck MPI can roll back to the last saved state and resume its execution. CoCheck MPI performs the checkpoints synchronously to save a consistent state of all processes at once. A major drawback of this approach is the synchronization overhead and saving the entire application state can be expensive depending on the application [93]. Libraries like Starfish MPI [14] and MPICH-V [26] have a similar approach and perform checkpoints or log the sent messages in order to recover from a global saved state in case of a failure.
Reinit [28, 47] is a fault-tolerant MapReduce framework following the bulk synchronous parallel (BSP) paradigm. The user has to periodically save synchronous checkpoints. In case a failure occurs, the application restarts from the last checkpoint. The major difference between this framework and CoCheck MPI is that the user has to manage the checkpoints. Depending on the use case, this does not require to save the entire application state. This can be more time and space efficient depending on the application [28]. Note that Reinit does not abort the current MPI execution, but respawns new processes to replace the lost ones.
fault-tolerant MPI libraries like FT-MPI [41, 42] and ULFM [23] on the other hand provide functions to detect the failed processes and recover the communicator of the MPI library. This means we do not have to create expensive checkpoints. FT-MPI and ULFM provide the tools for the library user to detect and resolve failures on its own. Therefore, the user can choose the fault-tolerance mechanism, which he wants to implement. This can increase the efficiency of fault-tolerant applications. This allows each algorithm to define its own fault-tolerance strategy. For instance, a program which reads a large data set into memory from a file has a large execution state. The algorithm does not need to include this data set into its checkpoints if it

is not modified. In case of a failure the data can be reread from the file.
The fault-tolerant MPI library FT-MPI [41, 42] provides three different operations to handle and recover from process failures. These procedures produce a new functioning communicator, similar to the standard `MPI_Comm_{create, split, or dup}` functions. A call to *shrink* removes the failed processes from the communicator and renames the processes consecutively. This requires to use `MPI_Comm_rank` to get the new rank of the current process. The *blank* operation keeps the rank of the remaining processes, but a communication with a failed process is invalid and results in an error. The *rebuild* operation spawns new MPI processes equal to the number of failed processes.
Since we use ULFM [23] in our implementation, we are going to explain this framework in more detail. ULFM stands for user-level failure mitigation, which implies that the fault-tolerance and failure handling are left to the library user. Similar to FT-MPI, ULFM provides operations to detect failures and receive a functioning communicator. We use a *shrink* procedure, which removes the failed processes. ULFM restores MPI objects and enables communication after a failure.
ULFM notifies a failure to the user on a per operation bases. Only processes which participate in a communication with a failed process get notified. Other processes do not notice the failure. Non-communicating processes can continue their execution. Furthermore, it is possible that during a failed collective operation only part of the processes receive an error message. This can happen for instance during an `MPI_Broadcast` operation. A part of the processes can receive their message successfully, while the rest could not receive their data.
ULFM's error handling is called a non-uniform error reporting, since failures are known only locally. An advantage of this approach is that failures do not have to be propagated to all MPI processes. This has performance advantages compared to notifying each process [23]. If we want to propagate the error to all processes, then we need to send heartbeat messages between each process to test if a process is still alive. Unfortunately, this results in a large amount of system noise by small messages, which is inefficient in MPI.
Note that this non-uniform error reporting is not viable for all algorithms, therefore ULFM provides the collective non-synchronizing `MPI_Comm_revoke` operation. It has a similar behavior than `MPI_Abort` and does not require a symmetric call on all processes. This means at least one process has to call `MPI_Comm_revoke` and the remaining processes get notified. After detects an error, a process can use *revoke* if failure requires all processes to participate during the recovery. The following MPI communication functions will return a revoke error message. Note that a process has to perform a communication to determine whether another process called `MPI_Comm_revoke`.

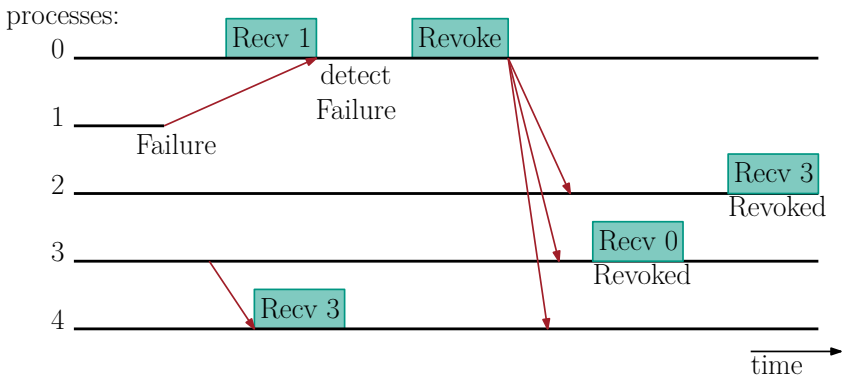


Figure 3: Illustration of the ULFM workflow to detect and notify failures [23]. The horizontal lines represent the execution of a process. The operations on each process are executed from left to right.

Figure 3 illustrates the failure detection workflow at the user side. Process 1 crashes and cannot participate in the communication with process 0, whose receive operation returns an error. Meanwhile, processes 3 and 4 can communicate even though the process already failed. After receiving the error message process 0 performs the revoke operations, which is noticed by processes 2 and 3 during their next communication. Note that process 2 gets a revoke error although it's a communication with process 3 and not 0 or 1. Furthermore, process 4 continues its execution without noticing that process 1 failed.
After a process has called the `MPI_Comm_revoke` operation, no successful communication is possible. Therefore, the user has to call the `MPI_Comm_shrink` command, which produces a new functioning communicator by removing the failed processes. This command can be compared to the `MPI_Comm_dup` or `MPI_Comm_split` operations, which produce a new communicator. During a call of `MPI_Comm_shrink`, the processes are renamed and get ranks from 0 to p_{new}, where p_{new} is the number of remaining processes. If a failure occurs during shrinking the communicator, the additionally failed processes are removed too.
Revoking and shrinking are time expensive operations and not all algorithms require to synchronize for recovery. Thus, ULFM provides point-to-point communication to test if a process has failed or not. These are the only communication operations, which can be used if a communicator has been revoked.
The `MPI_Comm_agree` operation computes the logical and of all boolean values provided by each process. This operation is costly and should be used sparsely to get a consistent view of the status of a communicator. If a process has failed and is contained in the communicator, then ULFM uses *false* as input for this process. This operation competes successfully even if the communicator contains failed processes.
To sum up, ULFM provides operations to detect failures and restore a functioning MPI communicator. The user has to implement the error handling and needs to resort lost data. ULFM allows to send messages between functioning processes by using a communicator containing failed processes. The user can use a *revoke* operation to notify all processes that a failure occurred.
MATCH [51] is a benchmark suit to compare fault-tolerant MPI implementations, especially ULFM and Reinit. MATCH implements six different proxy applications from the ECP [81] and LLNL ASC [71] application suits using both frameworks. The benchmark algorithms are often used in high-performance computing, for instance iterative solvers, multi-grid and molecular dynamics. Proxy applications are simplified benchmark algorithms used to quickly test key features of different frameworks. MATCH performs the checkpoints by using the fault-tolerant interface (FTI) [21]. FTI is a framework allowing the user to make and manage checkpoints efficiently. The MATCH [51] tests conclude that Reinit outperforms ULFM for their benchmark algorithms.
Unfortunately, our application cannot use the Reinit interface easily. Our fault-tolerant MapReduce implementation does not save checkpoints on disk, but exchanges additional messages. Therefore, ULFM provides a good interface for our use case.

3.2. Theoretical Models

Karloff et al.[55] propose two complexity classes to analyze theoretical MapReduce algorithms, the MapReduce Class ($\mathcal{MRC}$) and the Deterministic MapReduce Class ($\mathcal{DMRC}$). $\mathcal{DMRC}$ contains the MapReduce algorithms consisting of a polylogarithmic amount of map m_j and reduce r_j operations executed as sequence $< m_1, r_1, ..., m_k, r_k >$. Let n be the input size and $\epsilon > 0$ be a constant. The map m_j and reduce r_j operations have a polynomial time complexity and require sub-linear space: $\mathcal{O}(n^{1-\epsilon})$. The memory requirement of the key-value pairs pro-

duced by a map operation m_i must be sub-quadratic: $\mathcal{O}(n^{2-2\epsilon})$. $\mathcal{MRC}$ is a $\mathcal{DMRC}$, which allows randomization. The final output of a MapReduce algorithm in $\mathcal{MRC}$ has a probability of $\frac{3}{4}$ to be correct.
The complexity class $\mathcal{NC}$ contains parallel algorithms, which require $\mathcal{O}(\log^i n)$ time by using $\mathcal{O}(n^j)$ parallel processors, where $i, j > 0$. $\mathcal{P}$ is the class containing algorithms that can be solved in polynomial time. Karloff et al.[55] prove that $\mathcal{DMRC} \subseteq \mathcal{P}$ and if $\mathcal{P} \neq \mathcal{NC}$, then $\mathcal{DMRC} \subsetneq \mathcal{NC}$. The authors suspect but could not prove that $\mathcal{P} \subsetneq \mathcal{DMRC}$.
These theoretical MapReduce classes have disadvantages arising from a gap between theory and practice [86]. First, $\mathcal{MRC}$ algorithm do not have to attain a speedup compared to the best sequential algorithm. Furthermore, they can use $\mathcal{O}(n^{2-2\epsilon})$ space, which can lead to memory inefficient algorithms or even algorithms that exceed the available storage space. The complexity class $\mathcal{MRC}^+$ solves these problems by applying a more realistic machine model [86]. $\mathcal{MRC}^+$ introduces four new parameters. The term w indicates the total time needed to apply the user-defined map μ and reduce ρ function on the input data sequentially. The maximum time over all user-defined map or reduce functions applied on all input elements is $\bar{w}$. $\mathcal{MRC}^+$ uses m to express the total memory used to save the input, output and intermediate key-value pairs. The term $\bar{m}$ describes the maximum number of machine words produced or consumed during a user-defined map or reduce function call.
Let's execute a MapReduce algorithm on a distributed memory computing system with p processes and the input data being distributed so that each process stores $\mathcal{O}(\frac{m}{p} + \bar{m})$ words. Then the MapReduce algorithm can be implemented so that each expected local work is $\Theta(\frac{w}{p} + \bar{w} + \log p)$ and has a communication bottleneck volume of $\Theta(\frac{m}{p} + \bar{m} + \log p)$. The local work is the number of clock cycles and waiting rime needed by a process and bottleneck volume is the maximum number of machine words communicated by a process.
Figure 4 gives an overview of a bulk synchronous parallel (BSP) based algorithm to achieve the theoretical complexity. The MapReduce algorithm starts with the input data distributed evenly between the different processes. Then each process applies the user-defined map function. The static load balancing by distributing the input evenly can lead to imbalanced execution. Therefore, we can use distributed work stealing to redistribute work during the map and reduce phase if the initial partitioning was imbalanced.

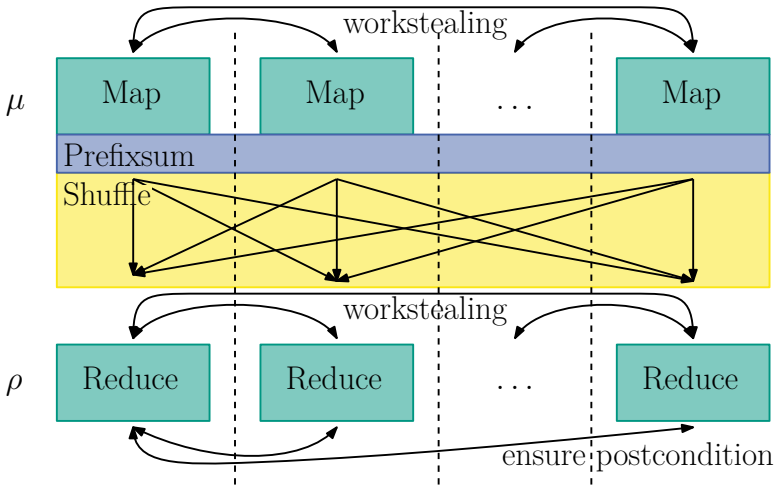


Figure 4: Illustration of the steps in the BSP MapReduce algorithm [86]. It consists of a map phase, followed by a prefix sum to determine the destination of key-value pairs and then a shuffle phase. Furthermore, the BSP MapReduce algorithm uses work stealing during the reduce phase followed by a random redistribution of output elements.

The main goal during the shuffle phase is to get the key-value pairs produced by the map phase with the same keys to the same process. Furthermore, the algorithm should distribute the data evenly so that each process receives $\mathcal{O}(\frac{m}{p} + \bar{m})$ words. Mapping the hashed keys to processes as done in previous sections can produce an imbalanced distribution. By computing a prefix sum over the hashed keys and determining the send data per hash value, the key-value pairs can be better distributed. Finally, during the reduce phase one call to a user-defined reduce function can produce multiple elements. From the second produced element onward, the elements are sent to a random process. This procedure ensures that the output data is distributed evenly and a potential new MapReduce operation can be performed.

3.3. MapReduce Libraries

In the following sections, we introduce multiple parallel MapReduce libraries for big data processing. In Sections 3.3.1 and 3.3.2 we introduce the first MapReduce framework developed by Google, as well as Hadoop, its open source implementation. Section 3.3.3 describes MR-MPI and Mimir, two MPI based MapReduce libraries designed to run on HPC systems. Section 3.3.4 contains a fault-tolerant MapReduce library using MPI and ULFM for communication and parallelization. In Section 3.3.5 we explain Twister, a fault-tolerant framework for iterative MapReduce algorithms. We introduce Thrill and Spark in Section 3.3.6. Moreover, we describe the interfaces of these frameworks as well as their map, reduce and shuffle phase algorithms. If provided, we explain how their fault-tolerance mechanisms work.

3.3.1. Google MapReduce

Dean et al. [34, 35] present a MapReduce framework in order to facilitate parallel computing, while ensuring fault-tolerance. Google introduces this framework to analyze large data collections such as crawled documents and web request logs. These algorithms and data manipulations are usually straightforward and requires the application of same simple functions. To finish the computation in a reasonable time, Google requires parallelization on thousands of machines. Therefore, programmers need to ensure load balancing, fault-tolerance and data distribution. For instance, the PageRank algorithm introduced in Section 4.2 is conceptually simple, but the introduction of fault-tolerance and parallelization results in longer design time. The Google MapReduce library facilitates the implementation of large-scale parallel algorithms, by abstracting from load balancing, fault-tolerance and data distribution.
The user of the Google MapReduce framework defines a *map* function μ and a *reduce* function ρ. In contrast to the interface introduced in Section 2.2, the input and output are key-value pairs.
Dean et al. [34] designed the Google MapReduce framework to run on the computing hardware used at Google. Especially MapReduce is optimized for a large cluster consisting of thousands of dual-processor x86 machines running Linux and connected by a 1 GB/s Ethernet connection network. A MapReduce algorithm accesses the input and output date from a distributed fault-tolerant file system (GFS) [48]. This file system saves the data redundantly to prevent a data loss in case of a failure.
Figure 5 illustrates the Google MapReduce implementation. The user program defines the *map* and *reduce* functions and invokes the MapReduce framework. This spawns one master process and multiple worker threads on different machines of the cluster. The master assigns work and ensures load balancing, while the worker apply the user-defined map and reduce functions. The input files are saved on the GFS file system, which MapReduce divides into M *splits* with a size of $16 - 64$ MB. These splits correspond to M different map tasks. The master assigns a map

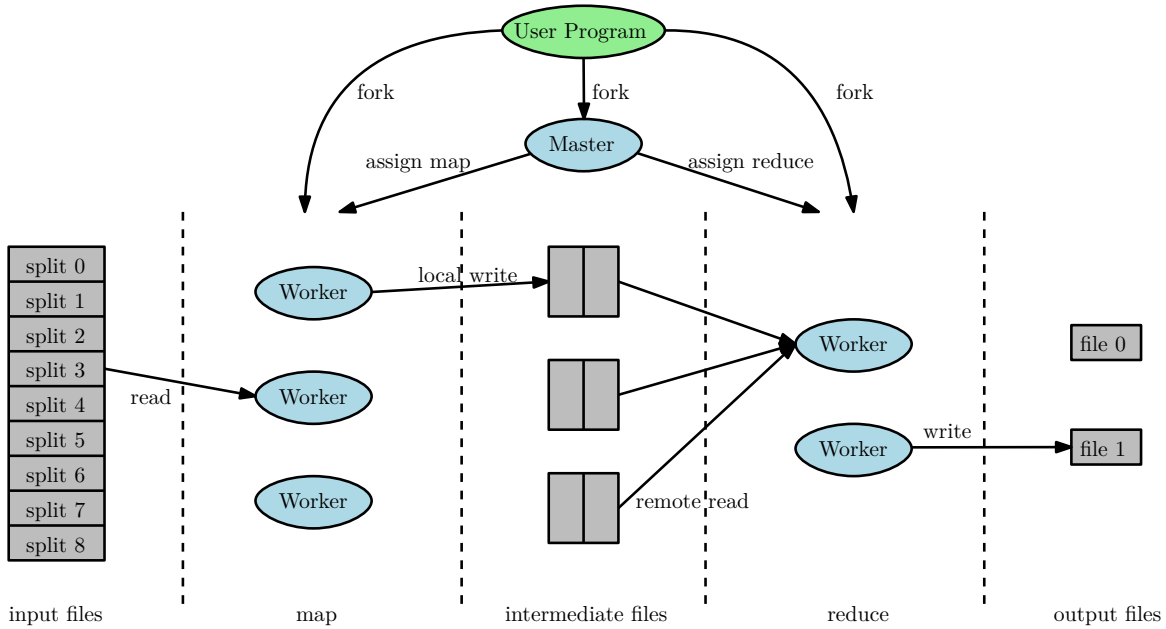


Figure 5: Illustration of the Google MapReduce master worker implementation [35].

task to a worker, which loads the corresponding split from the distributed file system. Then the mapper applies the map function μ to each element and saves the intermediate key-value pairs. The local disk of a map worker is divided into R partitions, where each partition corresponds to a reduce task. The mapper saves the intermediate pairs (k, v) periodically into partition $(h(k) \mod R)$, where h is a hash function. The map workers pass the partition locations back to the master, who waits until the map phase has processed all pairs of a partition p and assigns them to a reduce worker. This worker first acquires the corresponding intermediate key-value pairs and sorts them according to their key. MapReduce uses an external sorting algorithm if the key-value pairs do not fit into memory.

After sorting, the reduce worker iterates over the intermediate pairs and determines for each unique key the list of corresponding values. Then the reducer applies the user-defined *reduce* ρ function and saves the outputs to a file on the GFS file system. Finally, if no work remains the master terminates and the execution of the user program resumes.

Google MapReduce does not support in-memory iterative MapReduce algorithms. This is because the input and output have to be read from and saved to files on the distributed file system GFS. Nevertheless, the user can chain multiple MapReduce executions. In Section 3.3.2 we discuss how the Google MapReduce algorithm can be extended to in-memory MapReduce.

Google MapReduce provides fault-tolerance for worker failures. The master process pings the workers periodically. If the master receives no response after a certain time, it considers the worker to have failed. The intermediate key-value pairs are saved on the local disk of the map worker. Therefore, if the mapper fails, its produced data is lost. The master marks the tasks executed by the failed worker as uncompleted and redistributes it to a free worker. Similarly, the uncompleted map and reduce tasks have to be rescheduled. Finally, the master updates the data location for the reduce worker requiring data from unavailable local disks. The completed reduce tasks do not have to be rescheduled, because corresponding results are not saved locally but on a distributed file system.

Currently Google MapReduce does not support the failure of the master process. In case of its failure, we have to restart the MapReduce execution. We can resolve the failure of the master process by using periodic checkpoints [34]. The authors consider master failures unlikely and rescheduling the entire MapReduce computation more efficient.

3.3.2. Iterative Hadoop

Apache Hadoop [16] is an open source MapReduce implementation deriving the principles of Google MapReduce [75, 98]. While the Google library has a C++ interface, Hadoop provides a Java API and uses the Hadoop Distributed File System (HDFS) [87] instead of the GFS file system. Similarly to GFS, HDFS provides fault-tolerance. Hadoop requires the same definition of the user-defined map function μ and reduce function ρ, as given by Dean et al. [34, 35] and described in Section 3.3.2.
The basic Hadoop implementation follows the same logic as described in Section 3.3.1. This includes storing the intermediate data on local disks and all its implications for fault-tolerance. The input and output data have to be saved on the HDFS file system between MapReduce operations. Condie et al. [32] propose a pipeline system to exchange data between workers immediately, as well as between different MapReduce operations.
The general idea of the pipeline system is that a map worker sends its produced key-value pairs directly to the corresponding reducer. This does not require the storage of the intermediate data on the local disk. Therefore, each map thread opens a TCP connection to each worker who needs to receive the key-value pairs. This approach has two major disadvantages [32]. First, the mapper may need to send data to a reduce task which the master has not yet scheduled. Secondly, even if we could schedule each reduce task, we would need to maintain a TCP connection between nearly every map and reduce workers. This would come with serious scalability and performance issues.
To solve these problems, the mapper saves part of the intermediate key-value pairs on their local disk. Therefore, the mapper saves the pairs locally for unscheduled reducers. If the master schedules a reduce task, it pulls the necessary data as described in Section 3.3.1. To limit the number of TCP connections, each map worker has only a connection to a limited number of reduce workers. The mapper saves key-value pairs destined to not connected workers to the local disk. Furthermore, if a map worker cannot send data immediately over a TCP connection Hadoop buffers it locally. This prevents map workers to wait for reduce workers.
Iterative Hadoop has to modify the fault-tolerance mechanism of Google MapReduce to address the intermediate data coming through a pipeline. Each reduce worker receiving data through a pipeline keeps track of the incoming data. The reducer processes it without merging it with the final output. So if a failure interrupts the transmission of a map task, then the reduce task discards the non-complete data and waits for the master to reschedule the map task. The master thread reschedules failed reduce tasks, which have to receive the intermediate data again from each map task. If the map and reduce workers would only communicate through pipelines, this would require the mapper to recompute nearly every task. Therefore, the map worker saves the data send over pipelines additionally on their local disk. In case a reduce worker fails, the new reduce task can request their corresponding intermediate key-value pairs again.
An additional problem of the Google MapReduce algorithm is that a reduce task has to wait until each map worker has sent the corresponding key-value pairs. This requires the reduce worker to store the intermediate data on a local disk instead of the map worker. Therefore, Condie et al.[32] introduce snapshots and the partial execution of reduce tasks. The general idea is to apply the user-defined reduce function immediately on the available data and save the results as snapshots. If a reduce task receives new data, then it completes the snapshot. Hadoop assigns each snapshot a progression status, which estimates their progression.
Hadoop uses these snapshots to construct a pipeline between two successive MapReduce operations mr_1 and mr_2. If mr_1 produces a snapshot, the corresponding reduce worker pipelines it immediately to a map task in mr_2. Additionally, the reducer saves the snapshot to the HDFS file system in case the worker containing the snapshots fail. The reduce worker sends new updates to the snapshot to the corresponding map worker and replaces it on the HDFS

file system. Note that the user-defined reduce function interface is changed to allow a partial execution.
By allowing pipelines between different MapReduce operations, we have to ensure the fault-tolerance. Hadoop handles failures inside a MapReduce operation as described above. If a map task in mr_2 fails, then the master reschedules it and the new mapper can load the snapshots with the highest progression from the distributed file system. The tasks in mr_2 cache the received snapshots and their progression. They wait for a new snapshot with higher progression if the corresponding reduce task in mr_1 fails.
All in all, the pipeline system enables a faster communication inside a MapReduce operation and between two successive operations. But the fault-tolerance and the limitation of TCP connections still requires to save data on the local disks and the HDFS file system. Finally, the introduction of snapshots enables the execution of reduce tasks without waiting for the completion of each map task.

3.3.3. MR-MPI and Mimir

The map reduce library MapReduce-MPI (MR-MPI) provides a similar functionality to Google MapReduce introduced in Section 3.3.1. This small and portable C++ framework only requires a C++ compiler and an MPI library installed on the target system [46, 77]. Plimpton et al. [77] developed MR-MPI to realize large-scale graph algorithms [77] on HPCs: random graph generator R-MAT [27], page rank [73], triangle enumeration [30], single-source shortest path and maximal independent set.
MR-MPI has a simple C++, C and Python interface providing functions to compute the map, shuffle and reduce phase. The library user needs to define a *map* function μ and *reduce* function ρ. Similar to Google MapReduce and Hadoop the input and output are key-value pairs as described in Sections 3.3.1 and 3.3.2. The main difference between this and previous frameworks is the type of the in and output parameters of the user-defined functions. MR-MPI requires the user to serialize and deserialize the inputs and outputs during the function definition. This may have performance advantages but makes it more difficult to use the framework. MR-MPI allows the user to access a user-defined object during the map and reduce phase. This object is local for each process. It is the same for each user-defined function executed on the same process.
MR-MPI requires an MPI environment therefore, the user has to call the map and reduce phases between an `MPI_Init` and `MPI_Finalize` command. This allows to combine MR-MPI with ordinary MPI commands. To allow this interleave of MPI and MR-MPI functions, the map reduce library synchronizes all processes after each of its functions with an `MPI_Barrier`. Note that the MR-MPI framework does not ensure fault-tolerance and does not provide data redundancy as Hadoop and Google MapReduce with their fault-tolerant file system. Since MR-MPI uses an ordinary MPI and not a fault-tolerant version like ULFM, we cannot detect faults and the application aborts in case of a process failure.
A map reduce operation in MR-MPI illustrated in Figure 6 consists of a map, reduce and shuffle phase, where the shuffle phase includes an aggregation and a convert phase. The MPI process applies the user-defined map function locally. MR-MPI divides the input and output key-value pairs into fixed-sized pages, which it distributes between the different processes. These pages are not related with the operating system pages, but are preallocated chunks of memory, which the library uses to avoid memory fragmentation. We can compare pages to the split files and tasks in Google MapReduce. Pages represent a collection of input elements to which the user-defined map function is applied at once. The page size is an optimization parameter with a default size of 64 MB. During the reduce phase MR-MPI applies the user-defined reduce function.

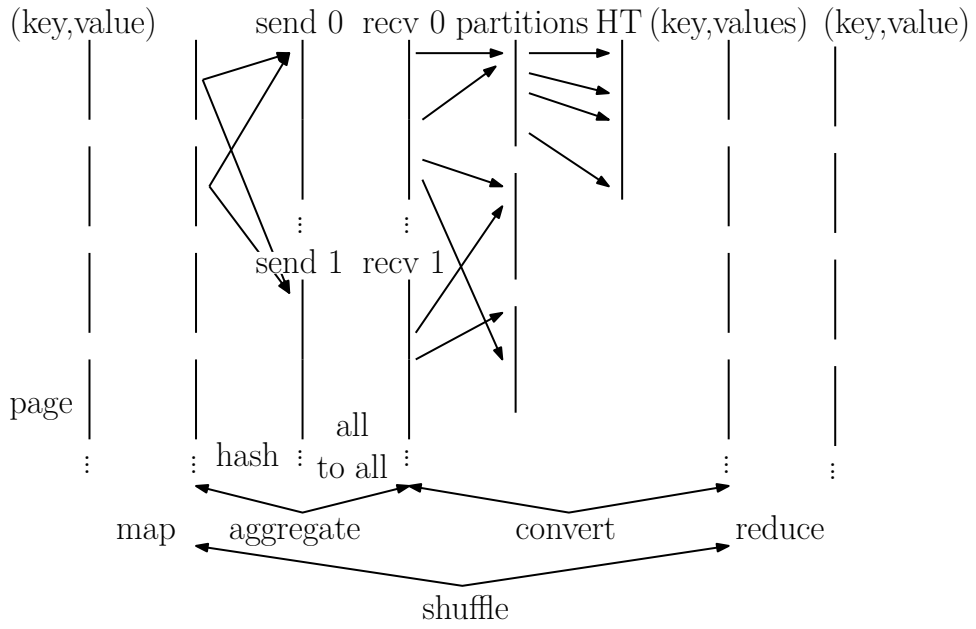


Figure 6: Illustration of a map and reduce phase executed by the MR-MPI library [77]. This figure presents the execution on one MPI process. The shuffle phase divides into the aggregate and convert phase. During the aggregate phase, the intermediate key-value pairs are sent between processes. The convert phase groups the key-value pairs on the same process by their key.

The aggregation phase distributes the key-value pairs to the different MPI processes by using a 32-bit hash function. The key-value pair is sent to process $(h(key) \mod n)$, where n is the number of MPI processes. After partitioning the pairs into send buffers, MR-MPI uses `MPI_All_to_allv` commands to exchange the pairs and ensures that each pair with the same key is on the same process.

During the convert phase, each MPI process groups the key-value pairs by their key. This algorithm uses a hash table of fixed size, which does not necessarily fit all keys. Therefore, the convert phase divides the pairs into partitions. First, MR-MPI fills the hash table and assigns each key to the first partition. After the hash table is full, MR-MPI scans over the remaining pairs and estimates the number and size of partitions. The algorithm assigns pairs, whose keys are already in the hash table to the first partition. By using the estimated partition number and size, MR-MPI assigns the remaining keys to a partition according to their hashed key. By using a hash table, the convert phase groups each partition by key and saves keys followed by their value in pages.

The advantage of this two phase shuffle is that the user can call the aggregate and convert functions individually. For instance, the user may need all key-value pairs on the same process but does not need to apply a reduce function. Furthermore, the user may need to group the key-value pairs locally without shuffling the pairs.

The main problem of MR-MPI is its memory efficiency [46]. On the one side, the shuffle phase allocates the send and receive buffer each time. On the other side, allocating pages of fixed size statically leads to more memory usage than needed. The map reduce library Mimir based on MR-MPI addresses these problems.

To avoid allocating large save and receive buffers for the aggregation phase, Mimir combines both phases into a MapReduce operation similar to the Google library. Mimir allocates a send and receive buffer of equal fixed size for each MPI process. After applying the user-defined map function, the algorithm saves the produced key-value pairs into the correct send buffer and de-

termines the destination process for each pair as described above. The map phase stops if one of the send buffers is full and exchanges the buffers by using an `MPI_All_to_all` operation. After exchanging the messages, Mimir copies the key-value pairs into a list, which is the input for the convert phase. After copying the pairs Mimir resumes the map phase.
This approach prevents to save all key-value pairs after the map phase before sending. Furthermore, Mimir requires only fixed sized send and receive buffers. Moreover the MapReduce framework exchanges the pages for a dynamic memory management system, which prevents allocating too much memory. Note that Mimir still saves all intermediate key-value before applying the reduce phase, but it does no longer require a large send buffer.

3.3.4. Fault-Tolerant MapReduce-MPI for HPC Clusters

Yanfei et al.[52] developed a fault-tolerant MapReduce framework (FT-MRMPI) to combine the MPI based implementation of MR-MPI in Section 3.3.3 and fault-tolerance of Google MapReduce in Section 3.3.1. FT-MRMPI provides two types of fault-tolerance. First, this framework uses the error handler of standard MPI libraries to abort the MapReduce execution. This permits a resubmission of the MapReduce job, which uses checkpoints to speedup the recovery and re-execution. Secondly, FT-MRMPI can use a fault-tolerant MPI implementation like ULFM to recover from failures without scheduling the MapReduce job again.
As for the previous frameworks, the user has to define a *map* function μ and *reduce* function ρ function. Similar to previous frameworks, the input and output types are key-value pairs. In contrast to MR-MPI, FT-MRMPI accepts data types instead of serialized elements as key-value pairs. Therefore, the user has to provide the serialization separately. In contrast to all previous frameworks FT-MRMPI allows the user to emit only one key-value pair during an application of μ or ρ.
Figure 7 illustrates the architecture of FT-MRMPI, which uses MPI or ULFM as communication library. FT-MRMPI saves the input, output and intermediate key-value pairs on the local disk or the shared HPC file system. This library does not allow in memory MapReduce like the Hadoop extension introduced in Section 3.3.1. Each process runs the same program using a distributed master and load balancer. The master divides the input data into small chunks which are distributed between the different processes using a hash function, similar to MR-MPI. Furthermore, the master takes track of the processed chunks by using a local and global task table. We can compare chunks to the MR-MPI pages or Google MapReduce splits.

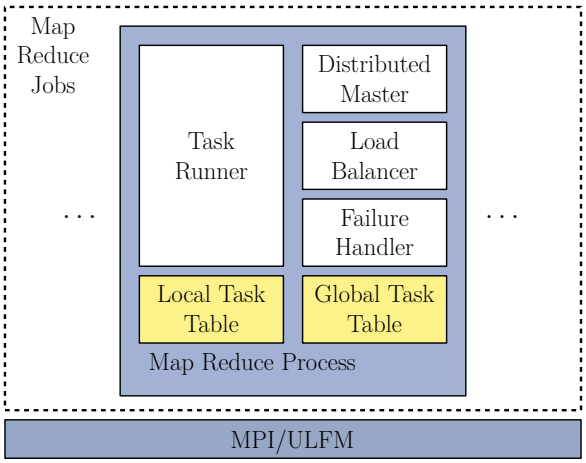


Figure 7: Illustration of the FT-MRMPI architecture [52]. This figure shows one of the p identical programs executed by the MPI library in parallel.

The task runners apply the user-defined map and reduce functions to the different data chunks. The local task table takes track of the chunks locally processed by the task scheduler and saves its completion state. The global task table contains information about chunks processed by other processes. Note that while the local task table is always updated, the global table may not contain all finished chunks. Each distributed master periodically broadcasts the state of their local task table, which allows each process to update their global task table. During the execution of a task, FT-MRMPI makes periodical checkpoints and tracks them with the task tables.
As soon as a task finishes, the task runner saves the resulting data on the global file system of the HPC. Checkpoints are usually small, while the HPC file systems are designed and optimized to handle large I/O operations. FT-MRMPI gathers the checkpoints on a local disk or in memory and transfers them together to the file system to minimize small I/Os. Note that usually a HPC nodes have no local disks and can therefore gather the checkpoints in memory. The algorithm needs to save the checkpoints directly on the shared file system if the memory is too small, which results in small inefficient I/O operations.
FT-MRMPI initially uses a static load balancing strategy by distributing the intermediate key-value pairs according to their hashed keys. A new load balancing problem arises during the recovery. The load balancer needs to redistribute the remaining work of a failed process equally. While unprocessed chunks should have similar execution times, the distributed scheduler does not know the remaining time for partially executed tasks, whose checkpoints are located on the shared file system. To estimate the remaining time, the load balancer takes track of the execution times of the different tasks and estimates the running time for the different chunks and checkpoints. FT-MRMPI uses this estimation to redistribute the remaining tasks during a recovery.
The FT-MRMPI failure handler distinguishes between a *checkpoint/restart* and *detect/resume* fault-tolerance. The former requires only a standard MPI implementation, which needs to detect failures and aborts the MapReduce execution. By using the saved results of the tasks during the map and reduce phase, as well as the additional checkpoints the failure handler can detect the remaining work, which the load balancer distributes. Note that this approach requires the user to manually resubmit the MapReduce job to the HPC, since FT-MRMPI aborted the entire execution. The input, intermediate and output data as well as the checkpoints are saved on the distributed file system of the HPC. Therefore, the data is still available even if a compute node crashes during the execution.
The *detect/resume* fault-tolerance relieves the user from having to resubmit the MapReduce job by using ULFM. In case of a failure, the ULFM library gathers the remaining processes and indicates which processes have crashed. By using the global task table, the load balancer can redistribute the remaining task of the failed processes. The algorithm then reloads the checkpoints from the file system and restores part of the lost data. FT-MRMPI suggests an alternative version, where the task runner makes no checkpoints. In case of a failure, the failure handler recomputes all tasks of the crashed processes. This requires no overhead to make checkpoints during the map and reduce phase, but leads to longer recovery times. Furthermore, the map tasks do not need to save the intermediate key-value pairs to the file system.
One advantage of the *detect/resume* compared to the *checkpoint/restart* method is that FT-MRMPI has to reload only the checkpoints of the failed processes. The *checkpoint/restart* method requires to reload the checkpoints of each MPI process, since the failure handler aborts the execution. This reduces the number of I/O operations and therefore the execution time.

3.3.5. Twister

Twister [39] is a fault-tolerant iterative MapReduce framework similar to the iterative Hadoop extension and using the Google MapReduce design. Ekanayake et al. [39] designed Twister to run on multiple compute nodes connected by a broker network. The target systems have a local disk per node, which contains the input and output data evenly distributed. Twister provides commands to manage this distributed file system composed of the local disks. For instance, Twister provides a function to partition files over multiple nodes. This files system is similar to GFS and HDFS but does not provide all their functionality. This MapReduce library provides fault-tolerance only for iterative algorithms. This means the data produced during one MapReduce operation is lost in case of a failure. The Twister framework requires the user to define a map and reduce function with the same signature as in Google MapReduce.
Twister uses NaradaBrokering [74]: a message interface following the publish–subscribe messaging pattern. The sender in a broker network does not specify the destination of the message, but divides the message into classes. In this case, Twister chooses the classes, so that the key-value pairs produced during the map phase are in the same class. A participant in a broker network can choose which class he wants to receive and subscribes to this class. Therefore, a node can receive all key-value pairs in a class and perform the reduce phase on this class.
Figure 8 illustrates the architecture of Twister. A user program starts the Twister library on the main node, which spawns a deamon process on each compute node. This MapReduce library distributes the input data for the map phase statically, evenly and randomly between the different nodes and saves it on their local disks. Each node has a worker pool for the map and reduce phase. These workers apply the user-defined map functions and push the resulting intermediate key-value into the broker network.
The reduce worker on the different nodes subscribe to a class of key-value pairs and receive the data as soon as the map worker classifies them. The reducer buffers the intermediate key-value pairs until all pairs for the reduce task are gathered. If the memory is not sufficient, then the intermediate can be stored temporarily on the local disk. During the reduce phase, Twister applies the user-defined reduce function and saves the results to the local disk. The

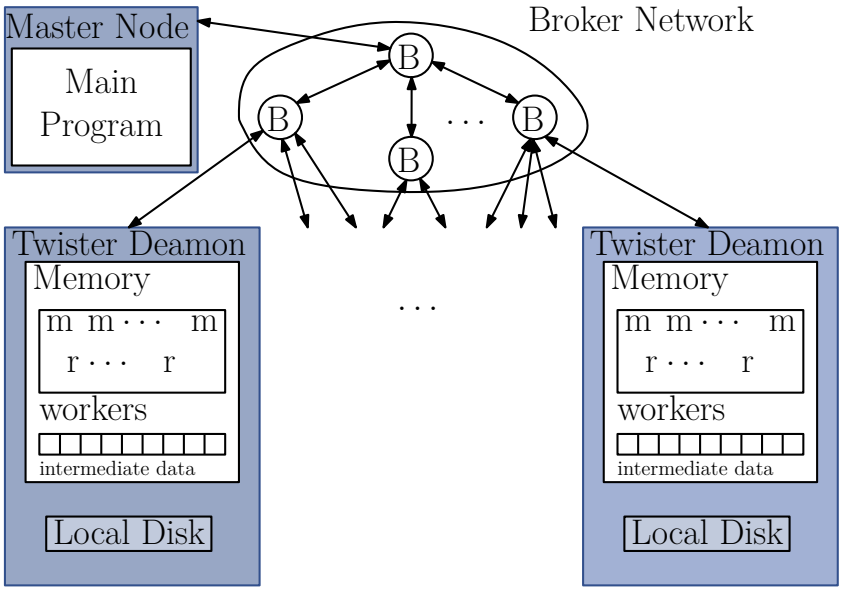


Figure 8: Illustration of the Twister architecture [39]. Each compute node starts a Twister Deamon consisting of multiple map and reduce workers. Twister exchanges messages through the broker network.

next MapReduce operation can use this output.
Furthermore, this library provides two new functions, to facilitate the programming of iterative algorithms. The Twister user can use these functions in their program to send data to the map and reduce tasks or get information from the output data. First, Twister has a broadcast operation, which sends the same data object to all twister deamon processes. The user can access this object in the user-defined functions. Secondly, Twister provides a combine function, which the user program can call after a MapReduce operation. This function computes a reduction over all output key-value pairs and provides a result that can indicate whether an iterative algorithm should stop. Note that the user has do define a reduction function for all output elements of the MapReduce operation.
In contrast to FT-MRMPI in Section 3.3.4, Twister does not save checkpoints during the map and reduce phase. Twister's approach at fault-tolerance is to save the application state between MapReduce iterations. Note that if checkpointing is expensive, then Twister does not have to save its state after each iteration. The number of MapReduce operations between checkpoints is a tuning parameter. This approach only works if it is possible to save the necessary information redundantly. If the computation system only provides a local disk per node and no global fault-tolerant file system, the checkpoints cannot be saved safely. If during a failure the non-redundant checkpoints are lost, then the recovery fails.
Twister has to save the data send by the user program directly to the map and reduce phase by the broadcast method. If a failure occurs, these data and the saved key-value pairs are sufficient to roll back to the last saved MapReduce operation and recompute all steps. Twister restarts the computation on a new set of compute nodes.
Furthermore, Twister provides fault-tolerance during the message exchange by using a fault-tolerant network interface. It assumes that no irreversible failures occur during the communication. Finally, this framework does not provide fault-tolerance in case the master node with the user program fails. The reasoning is that master failures are unlikely and if it occurs the user can restart the program.

3.3.6. Thrill and Spark

Apache Spark [100] is a library that came up to satisfy a larger set of algorithms than a map reduce framework. The main goal is still to provide an automatic work parallelization, data distribution and fault-tolerance for big data algorithms on large computer systems. While still implementing the map and reduce functions, Spark provides more operations which do not have to be necessarily parallel. The filter function, for instance removes elements that do not satisfy a certain condition. Furthermore Spark provides a machine learning library MLlib, which facilitates the design and implementation of scalable machine learning algorithms and pipelines [84].
Spark functions operate on *resilient distributed datasets* (RDD). These read-only data structures divide into multiple partitions distributed between multiple processes. If a RDD partition is lost, then Spark can recompute the lost data from other RDDs.
The Thrill library [22] is similar to Spark with three major differences. Thrill is C++ implementation and by exploiting template meta programming achieves faster execution times. The basic data structures of Thrill are arrays, which allows additional operations, for instance sorting, prefix sums, window scans, and zipping. Note that Thrill does not support fault-tolerance. The main Thrill data structure is a distributed immutable array (DIA) comparable to Spark RDDs. We can compare a DIA to a large array, which is distributed evenly over the compute nodes of a cluster. The user cannot modify this array manually but has to use the by Thrill provides operations. These operations can be divided into four categories. *Source* operations

create a DIA object, for instance by reading a file. *Local* operations have one or more DIAs as input and output and require no communication, for example a map phase which applies a user-defined map function. *Distributed* operations have DIAs as input and output and require communication between the different compute nodes, for instance a reduce phase or sorting. Finally, *actions* take a DIA as input but do not produce no new DIA., for instance getting the maximum element.
Moreover, Thrill provides a flat map operation, which correspond to a map phase and applies a user-defined map function μ. Furthermore, Thrill implements an associative reduce operation, which combines the shuffle and reduce phase in MapReduce frameworks. Note that in Thrill the user-defined reduce function has to be associative, which is not necessarily the case in MapReduce frameworks. Similar to FT-MRMPI in Section 3.3.4, the reduce phase of Thrill allows the user to crate only one key-value pair in the user-defined reduce function. Since Thrill does not require to perform the map and reduce phase with one operation, we can execute a map phase after a reduce phase to get the same functionality as Google MapReduce.
Furthermore, Thrill allows the user to chain different functions and create a pipeline. Due to the template meta programming, the compiler merges multiple *local* operations into a single operation without performance loss. This can reduce the total number of operations and the data flow between local operations.
We describe a MapReduce execution in Trill more precisely. First, the Thrill map phase consists of simply applying the user-defined map function. The Thrill reduce phase combines shuffling and reduction. First, Thrill divides the space of keys into w equally sized partitions, where w is the total number of workers distributed over the cluster. Each worker gets a partition. Each Thrill worker iterates over their DIA and streams the key-value pairs immediately to their corresponding worker. This framework uses MPI or TCP for the communication between workers. Each worker stores the received data in an in-memory vector. If this buffer is full, it is sorted and saved to the disk. After sorting all key-value pairs, Thrill merges them with a multiway merge algorithm. After sorting, we can apply the user-defined reduce function and save the results in a new DIA. Note that we need to define compare operation for the keys.

3.4. Aggregation: Hashing vs. Sorting

During the reduce phase of a MapReduce algorithm, we apply the user-defined reduce function to all keys and their corresponding values. We thus have to gather all key-value pairs with the same key and pass them to the user-defined reduce function (Algorithm 1 Line 5). Aggregating elements according to a key is a well-studied problem in relation with query algorithms in large database and SQL operations like `JOIN` and `GROUP BY` [15, 20, 49, 56, 65, 69].
In literature two major techniques are used to realize aggregation: hashing and sorting. The core of a hashing based technique is a hash table, which gathers the values of each key. The advantage of aggregation with hashing is the expected $\mathcal{O}(1)$ access time of hash tables [56]. The disadvantage is a performance loss with increasing number of table entries [56, 69]. To limit comparisons during hash table accesses, the table size should be two times larger than the number of input elements. This increases the memory requirement. For large hash tables, an operation can access memory whose page entries are not cached in the translation lookaside buffer (TLB). Increasing the number of TLB misses leads an increased runtime. Furthermore, duplicated keys lead to collisions in the hash table. Finally, poor hash functions decrease the viability of the hash table method drastically [49]. To avoid these disadvantages of hashing based aggregation, a common approach is to divide the input into partitions according to their hash value. Then we construct a hash table on each partition [20, 56, 69]. We chose the size of each partition so that their corresponding hash table fits into the cache.

Sorting based aggregation techniques sort the elements according to their key. After sorting, the elements with the same key are situated next to each other. A popular sorting algorithm for aggregation is a merge sort with a running time in $\mathcal{O}(n \log n)$, where n the number of input elements [15, 20, 56]. The input elements can be sorted using a radix sort to achieve linear running times [56].
The optimal aggregation technique seems to depend on the used hardware [15, 20, 56]. On Intel Core i7 from 2009, hash-based aggregation outperforms the sorting algorithm [56]. The authors [56] claim that future hardware will be more suited for merge sort aggregation, because of the increasing number of cores, decreasing memory bandwidth, and wider SIMD. A parallel aggregation executed on a NUMA architecture performs best with a version of the merge sort algorithm [15]. Furthermore, hardware with sufficiently wide SIMD instructions performs best with sorting based aggregation [20]. Geafe [20] claims that no technique outperforms the other in all situations and therefore the aggregation algorithm should be chosen depending on the use case.
An important optimization metric is the number of cache misses [69]. By efficiently using the CPU caches, we can speed up the aggregation process [65, 69]. In the case of cache efficient aggregation, the sorting and hashing approach are equivalent [69]. The external memory model consists of four parameters characterizing the cache and input. Let N be the amount of input rows, K the number of groups in the input, M the number of rows fitting into the cache and B the number of rows per single cache line. Aggregation with a cache efficient sorting algorithms, like the radix sort *IPS*^{2}*Ra* [17], requires roughly $2 \cdot \frac{N}{B} \left\lceil \log_{\frac{M}{B}} \min\left(\frac{N}{B}, K\right) \right\rceil + \frac{N}{B} + \frac{K}{B}$ cache line transfers [69]. Aggregation with a single hash table is efficient if the table fits into the cache ($K < M$) with $\frac{N}{B} + \frac{K}{B}$ cache line transfers. This approach performs worse with $\frac{N}{B} + 2 \cdot (1 - \frac{M}{K}) \cdot N$ cache line transfers if the hash table does not fit into the cache ($K \geq M$). Finally, the authors [69] propose and optimized sorting and hashing aggregation with $2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil - 1 \right) + \frac{N}{B} + \frac{K}{B}$ cache line transfers.

3.5. In-place Parallel Super Scalar Radix Sort

To group all intermediate key-value pairs, our MapReduce implementation uses sorting based aggregation as introduced in Section 3.4. We employ *IPS*^{2}*Ra*, an in-place cache efficient parallel MSD radix sort [17]. *IPS*^{2}*Ra* provides best results for near-uniform input distribution, small keys, or a sequential execution. A most significant digit (MSD) radix sort is a non-comparative sorting algorithm [17, 44] for unsigned integer data types. The authors of [89] extend radix sort for floating point data types.
Radix sort is related to bucket sort and recursively divides input elements into buckets. *IPS*^{2}*Ra* uses the bits of integer input elements to determine the buckets during sorting. A MSD radix sort starts with the most significant bit and proceeds to the least significant bit. *IPS*^{2}*Ra* starts with the most significant bit i. We distribute the input elements into two buckets, one where all elements have 0 as bit i and one for 1. We recursively partition each bucket according to bit $i - 1$. The input elements are sorted if we have partitioned according to the least significant bit.
IPS^2*Ra* is based on an in-place parallel sample sort algorithm *IPS*^{4}*o* [17]. Sample sort is a generalization of quicksort for multiple pivots. *IPS*^{2}*Ra* reuses the highly optimized, in-place, parallel, and cache efficient distribution operation of *IPS*^{4}*o* to divide elements into two buckets according to their bits. We can divide this operation into four phases. During **sampling**, the algorithm determines the bucket boundaries. The **classification** divides the input into blocks so that each element in a block belongs to the same bucket. During **permutation** the blocks belonging to the same bucket are gathered and placed in the correct order. The **clean-up**

phase handles partially filled blocks and blocks which cross bucket boundaries.
IPS^2*Ra* combines multiple sorting algorithms to optimize its runtime. For buckets with more than 2^{12} elements *IPS*^{2}*Ra* uses the distribution operation of *IPS*^{4}*o*. If a bucket has less than 2^{12}, then *IPS*^{2}*Ra* uses a sequential in-place MSD radix sort [89] as base case. This sequential radix sort uses quick sort for fewer than 2^7 elements, which falls back to insertion sort if less than 2^5.

3.6. Randomized Static Load Balancing

In our MapReduce algorithms (Sections 6 and 7), we use randomized static load balancing to distribute the input and intermediate key-value pairs between different parallel processes and threads. We adopt the abstract static load balancing model [85]. We distribute m subproblems with sizes $s_1, ..., s_m \geq 0$ among p parallel processes. The total work is represented by $w = \sum_i s_i$ with maximum work load $\bar{w} = \max_i s_i$. The sizes of subproblems are unknown to the load balancing algorithm. An adversary chooses the sizes s_i in order to maximize the running time. The different subproblems are independent of each other and can be scheduled in any order. A randomized static load balancing strategy distributes the subproblems s_i interdependently and uniformly between p processes at random.
Let S_i be the subproblems assigned to process i. We denote $L_i = \sum_{s \in S_i}^{s}$ the random variable representing the workload at process i. The adversary tries to maximize the maximum load $L_{max} = \max_i L_i$. This corresponds to the worst-case for the scheduling algorithm. The goal of a scheduling technique is to achieve an execution with $L_{max} \leq (w + \varepsilon)/p$ for $\varepsilon > 0$, while an adversary tries to maximize the expected running time.

Lemma 3.1. *Let m be the total number of subproblems. The worst-case for random static load balancing occurs if the total work w is divided into $\lceil w/\bar{w} \rceil$ subproblems with maximum subproblem size $\bar{w}$. The remaining $m - \lceil w/\bar{w} \rceil$ subproblems have a size of* 0.

Proof. This Lemma is proven in Lemma 1. [85]. □

Random static load balancing algorithm can compete with dynamic load balancing for a large number of subproblems [85]. This method can even outperform dynamic load scheduling techniques when the parallel machines have slow inter-process communication. In our runtime analyzes (Sections 6 and 7) we require Lemma 3.1 to prove their expected runtime.

4. MapReduce Benchmark Algorithms

In order to test the correctness and performance of our MapReduce implementation, we implement four MapReduce algorithms. We introduce the word count algorithm in Section 4.1. We study three iterative MapReduce algorithms: Page rank in Section 4.2, and connected component in Section 4.3, and R-MAT in Section 4.4.

4.1. Word Count

The word count algorithm computes the number of occurrences of each word in a text. This algorithm finds applications in data analytics, for instance to analyze research data. Sahane et al. study the research focus in the zoology and botany department by performing the word count algorithm on a collection of research papers [83]. Word count is a popular benchmark algorithm to test, improve, and compare the performance of MapReduce implementations [52, 79, 80, 82, 83].
The input for the map phase is a text, usually contained in one or more files. The word count MapReduce algorithm then saves each word followed by its occurrence count into one or multiple output files. The user-defined map function is illustrates in Algorithms 2 and takes a string as input. We split the input by space and emit each word w as key with 1 as value. This key-value pair corresponds to one occurrence of w in the input text. The user-defined reduce function shown in Algorithm 3 has a word *key* and list of ones as input. For each word *key* in the input, the list *values* contains a 1. Therefore, the sum of over all elements in *values* correspond to the occurrence count of *key*. For each key we emit itself and its occurrence during the reduce phase.

Algorithm 2: User-defined map function: Word Count

```
Input: text : String                          // the words in text are delimited by " "
1 foreach w ∈ split(text, " ") do             // iterate over all words in the text
2 |   emit(w, 1)                              // emit the word w as key and 1 as value
3 end
```

Algorithm 3: User-defined reduce function: Word Count

```
Input: key : String, values : list< ℕ >                          // key is a word
1 emit((key,sum(values)))   // sum up the elements in values to get the number of occurrence
```

4.2. PageRank

Google developed PageRank [73] as part of their search engine to rank different web pages according to their importance. This algorithm takes the world wide web (WWW) as input and provides a metric to determine the human interest of certain pages. The general idea behind PageRank is to use the topology of the WWW and relations between web pages to determine if a page is important for a user. In addition to its importance in search engines, PageRank has a MapReduce reduce implementation which is commonly used as a benchmark algorithm to test and compare different libraries [39, 52, 58, 22].
In our implementation and to facilitate notations, we represent the WWW as directed unweighted graph $G = (V, E)$. We represent each web page by a unique vertex $a \in V$ and an edge $(a, b) \in E$ represents a link from page a to $b \in V$. This allows us to represent pages with integer data types and we do not require long strings for links. If the input is a web graph with string links, we can map each link to a unique integer value in a preprocessing step.

PageRank determines the rank by using the web structure and links between web pages. Intuitively PageRank simulates a person clicking random links on the WWW. This corresponds to a random walk through the web, where the surfer clicks on successive links randomly and has a probability of $\alpha \in [0, 1]$ to visit a random web page instead. Let $a \in V$ be a web page, then the surfer visits each linked page $b \in O(a)$ next with a probability of

$$(1 - \alpha)\frac{1}{|O(a)|}.$$

PageRank determines for each page $a \in V$ the probability $p(a) \in [0, 1]$ of a random surfer to visit it. In other words, if a lot of web pages refer to a or if an important page links to a, then it is likely a random surfer would visit it. Since the ranks $p(a)$ of a page is a probability we have

$$\sum_{a \in V} p(a) = 0.$$

We can calculate the page rank iteratively by using the following formula [54]:

$$p_{n+1}(a) = \frac{1 - \alpha}{N} + \alpha \sum_{b \in I(a)} \frac{R_n(b)}{|O(b)|}, \tag{4.1}$$

where a is a web page, $p_n(a)$ is the rank of a at iteration n, $N = |V|$ the number of pages in the web, $I(a)$ the pages which contain a link to page a, $O(a)$ the pages to which u has a link and α a dampening constant. Note that the dampening constant corresponds to the probability of the random surfer to visit a random page.
We can implement PageRank using an iterative map reduce framework. The input for this MapReduce application is a graph $G = (V, E)$ representing a network of pages. We illustrate the user-defined map function $\mu : (\mathbb{N}, \mathbb{R}, 2^{\mathbb{N}}) \to \text{list} < \mathbb{N}, (\mathbb{R}, 2^{\mathbb{N}}) >$ and reduce function $\rho : (\mathbb{N}, \text{list} < (\mathbb{R}, 2^{\mathbb{N}}) >) \to (\mathbb{N}, \mathbb{R}, 2^{\mathbb{N}})$ in Algorithms 4 and 5. The input of the map and output of the reduce functions is a triple containing a page a, its current rank $p(a)$ and its outgoing neighbors $O(a)$. The keys are pages and the values are pairs consisting of a rank and a list of nodes.
During the map phase, we iterate over the outgoing neighbors $b \in O(a)$ of a page a and emit b as key and a pair as value. This pair consists of $\frac{p(a)}{|O(a)|}$ and an empty set. Finally, we emit a as key and $(\infty, O(a))$ as value. We have to send the outgoing neighbors to reduce phase because we need them again in the next map phase. During the reduce phase, we calculate the new

Algorithm 4: User-defined map function: PageRank

```
Input: (a, p(a), O(a))
foreach b ∈ O(a) do                                   // iterate over all outgoing links
    emit(b, (p(a)/|O(a)|, ∅))        // emit the current rank divide by the number of outgoing links
end
emit(a, (∞, O(a)))                                    // emit the outgoing links for future iterations
```

Algorithm 5: User-defined reduce function: PageRank

```
Input: b, values
p(b) = (1−α)/N + α · Σ_{(x,·)∈values∧x≠∞} x     // calculate the new rank according to Equation 4.1
O(b) = O with (x, O) ∈ values ∧ x = ∞                                  // extract the outgoing links
emit(b, p(b), O(b))          // emit the page, new rank and outgoing links for the next iteration
```

rank according to Equation 4.1 and gather the outgoing links for the next iteration. The triples emitted by the reduce phase are the input triples with updated page rank $p(a)$.
We repeat the map phase followed by the reduce phase for a fixed number of iterations. It is possible to implement PageRank so that the algorithm stops after the ranks between iterations differ only by a small constant $\epsilon > 0$. We initialize the ranks with $\frac{1}{N}$, where N is the total number of pages.
A common idea of the parallel MapReduce libraries introduced in Section 3.3 is to send the key-value pairs $\Big(a, (\infty, O(a))\Big)$ to process $i = (hash(a) \mod p)$, where p is the number of parallel processes. The pairs containing $O(a)$ have a larger serialization and the shuffle phase does not need relocate them. In Section B we introduce an alternative PageRank MapReduce algorithm with 3 consecutive MapReduce operations.

4.3. Connected Components

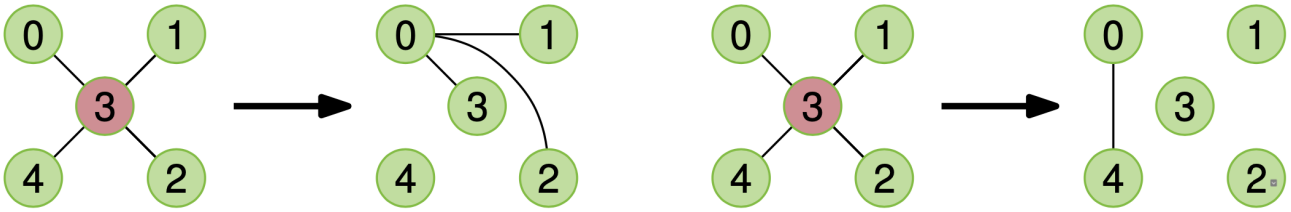
Connected component (CC) algorithms are basic tools to analyze and use large graphs in social, communication, and information networks [57]. An example is the use as a subroutine in clustering algorithms [33, 36]. Vitali et al.[96] use CC algorithms to analyze the structure of a corporate control network to determine the ownership between transnational corporations. Marina et al. [66] analyze the routing performance of unidirectional links in multi-hop wireless networks by calculating CCs.
Let $G = (V, E)$ be a directed graph. A strongly connected component [72, 24] is a maximum vertex set $C \subseteq V$ so that $G[C]$ is a connected component. For each distinct vertices $u, v \in V$ there is a path between u and v. There is no path form $x \in V \setminus C$ and u. A CC algorithm partitions V into pairwise disjoint vertex sets $\{U_1, ..., U_c\}$, where $G[U_i]$ is strongly connected for $i \in \{1, ..., c\}$.
Kiveris et al. [57] implement an iterative connected component algorithm using a MapReduce framework. The inputs to the following algorithms are a graph $G = (V, E)$ represented as an edge list and a unique label l_v for each vertex $v \in V$. The output is a set of tuples (v, c_v),

Algorithm 6: Large star MapReduce operation of the connected component algorithm

```
1 Map(u, v)                                   // use the vertex with larger label as key
2 |   if l_v ≤ l_u then emit(u, v)
3 |   else emit(v, u)
4
5 Reduce(u, N_s = {x ∈ O(u) | l_v ≤ l_u})
6 |   m = argmin_{v∈N_s∪{u}} l_v               // determine the vertex with minimal label m
7 |   foreach v ∈ N_s do emit((v, m));         // emit an edge for each value v pointing to m
8
```

Algorithm 7: Large star MapReduce operation of the connected component algorithm

```
1 Map(u, v)                                   // emit both edges (u, v) and (v, u)
2 |   emit(u, v) and emit(v, u)
3
4 Reduce(u, N = O(u) ∪ I(u))
5 |   m = argmin_{v∈N∪{u}} l_v                // determine the vertex with minimal label m
6 |   foreach v ∈ N ∧ l_v > l_u do emit((v, m));   // emit edges for each value v ∈ N_l
7
```



(a) small star (b) large star

Figure 9: Figures 9a and 9b illustrate the application of the small and large star to vertex $v = 3$. We represent the outgoing neighbors $O_+(3)$. The vertex u corresponds to its label l_u. The small star algorithm connects lower labeled neighbors to $m(3) = 0$. The large star algorithm connects higher labeled neighbors to $m(3) = 0$.

where c_v is the vertex with minimal label in the strongly connected component containing v. The alternating and two phase cc Algorithms 8 and 9 are composed of the small and large star MapReduce Algorithms 6 and 7.

Let $u \in V$ be a vertex and $m(u) = \text{argmin}_{v \in O_+(u)} l_v$ the neighbor with minimal label. The small start Algorithm 6 replaces edges (u, v) with $(v, m(u))$, where $v \in N_s = \{x \in O(u) \,|\, l_v \leq l_u\}$. The large star Algorithm 7 replaces edges (u, v) with $(v, m(u))$, where $v \in N_l = \{x \in O(u) \,|\, l_v > l_u\}$. We illustrate both algorithms applied to the vertex $v = 3$ (Figure 9a). A vertex's id corresponds to its label. The small star connects the neighbors of 3 with smaller label to the neighbor with minimal label 0. We apply the large star Algorithm 7 to vertex $v = 3$ in Figure 9b. The algorithm connects the neighbor with greater label 4 to the vertex with minimal label 0.

During the two phase cc Algorithm 8 we repeat the large star algorithm until convergence, followed by the small star. This procedure repeats until the large and small star algorithms stop changing the edge list. The two phase algorithm converges after $\mathcal{O}(\log^2(n))$ MapReduce operations, where n is the number of vertices [57]. The alternating Algorithm 9 repeats the large and small star until convergence. This algorithm needs $\mathcal{O}(n)$ iterations.

Both algorithms finish if during a consecutive call to the large and small star the edge list does not change. The large star algorithm does not modify the edge list if during the reduce phase the key u has the minimal label. Furthermore, the small star does not change the edge list if during the reduce phase the number of values $|N|$ is smaller or equal to 2.

Algorithm 8: Two Phase Connected Component Algorithm

```
Input: edge list, unique label l_v for each node
repeat
    repeat
        large star
    until convergence
    small star
until convergence
```

Algorithm 9: Alternating Connected Component Algorithm

```
Input: edge list, unique label l_v for each node
repeat
    large star
    small star
until convergence
```

Note that, the large star algorithm results in star graphs and high degree vertices, which leads to imbalance during the MapReduce execution [96]. The user-defined reduce function in the large star Algorithm 7 receives the entire neighbor hood as value list. During the final stages, this represents the entire connected component for a representative vertex $v \in V$. This leads to load balancing problems, especially for graphs with unevenly sized connected components. Therefore, the load balance depends on the number and size of the different connected components.

4.4. Recursive Matrix Model (R-MAT)

Chakrabarti et al. [27] develop R-MAT: a random graph generator for graphs with power-law degree distributions. This algorithm produces graphs with a community structure similar to those of complex real-world networks [53]. R-MAT can generate real world graphs by specifying the number of vertices n, number of edges m and the parameters $a, b, c, d > 0$ summing up to 1. We discuss the R-MAT algorithm for n a power of two. R-MAT wit parameters $a = 0.57, b = 0.19, c = 0.19$ and $d = 0.05$ is part of the Graph 500 benchmarks [70]. Graph 500 is a list of large graphs to test and optimize large-scale graph algorithms, especially in high-permanence computing.
Let $G = (V, E)$ be a graph with adjacency matrix $A = (a_{i,j})_{i,j} \in \{0, 1\}^n$ and $n = |V|$. A tuple $(x, y) \in V \times V$ is part of G if and only if $a_{x,y} = 1$. The R-MAT algorithm generates G by successively computing and adding a random edge. R-MAT recursively divides the adjacency equally matrix into four (Figure 10). Each partition is associated with a probability a, b, c or d of an edge to be placed into a partition. The algorithm recursively assigns an edge into partitions and determines its in-and outgoing vertex. Note that generating an edge requires $\log_2(n)$ time. By using an alias table, it is possible to generate edges in constant time [53]. To keep our MapReduce algorithm simple we use the $\log_2(n)$ time version.
Plimpton et al. [77] propose a MapReduce algorithm to generate R-MAT graphs, which we modify to fit our framework. The general idea of this algorithm is to construct the random graph incrementally. During the map phase, we generate $\frac{R}{p}$ random edges, where R is the number of remaining edges and p is the number of parallel processes. During the reduce phase, we remove duplicated edges. We repeat this procedure until all m distinct edges are generated. We have to modify the algorithm since we are unable to determine the number of remaining edges R. Algorithms 10 and 11 illustrate the user-define map reduce functions of our MapRe-

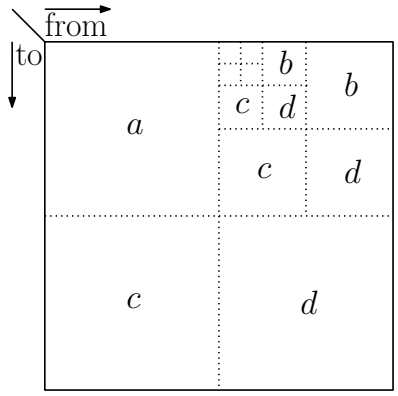


Figure 10: Illustration of the random edge generation of R-MAT [27]. The figure contains the adjacency matrix A of a graph $G = (V, E)$. We represent the in-vertices vertically and out-nodes horizontally. The figure illustrates the recursive subdivision of A according to the probabilities a, b, c and d.

Algorithm 10: User-defined map function: R-MAT

```
Input: e ∈ E
1 emit(e, 1)                                   // emit the edge e as key and a value 1
```

Algorithm 11: User-defined reduce function: R-MAT

```
Input: e ∈ E, A                                // edge e as and a list A of ones
1 c = ∑_{a∈A} a                                // compute the occurences of e
2 emit(e)                                      // emit edge e
3 repeat c − 1 times                           // emit random edges for each duplicated e
4 |  emit random edge
5 end
```

duce implementation of R-MAT. We generate $\frac{m}{p}$ random edges on each process, which our MapReduce operation takes as input. During the map phase, we emit the edge e as key and 1 as value. The user-defined reduce function takes an edge e and a set of ones A as input. For each edge e emitted during the map phase, A contains a 1. By summing um all values in A we get the number of duplicated edges e. First we emit the edge e (Line 2). Then we generate and emit a new random edge for each duplicated edge (Line 4). We repeat this MapReduce operation until no duplicated edge is identified during the reduce phase. To generate random edges, we use the $\log_2(n)$ time algorithm by Chakrabarti et al. [27].

5. Fault-Tolerant MapReduce

In Section 5.1 we describe the general structure of our MapReduce library. We introduce an in-memory fault-tolerance mechanism to recover from a single process or compute node failures in Section 5.2.

5.1. General

We design our MapReduce library for executions on high-performance computing (HPC) systems, while employing a large number of compute nodes. Our library handles multiple consecutive MapReduce operations in memory without saving each output to the file system.
In contrast to Google, Hadoop, and Twister MapReduce (Section 3.3) we do not adopt the master worker design pattern. We do not employ a single master process, which distributes and manages the entire workload. The communication between master and worker processes represents a bottleneck, which may lead to performance issues if we use a large number of processes.
We design our MapReduce library according to the Bulk Synchronous Parallelism (BSP) model [90, 95]. An algorithm following the BSP principle consists of three consecutive repeating phases. First, the algorithm executes locally without communication on multiple parallel processes. During the second phase, the processes exchange data between each other. Finally, all processes synchronize and start a new local execution phase.
Our implementation uses the message passing interface (MPI) for communication and parallelization. The map and reduce phases correspond to the local execution of the BSP model. The shuffle phase provides the inter-process exchange and synchronization. We adopt a similar approach to MRMPI, Mimir and FT-MapReduce-MPI (Section 3.3), where the libraries distribute the workload statically by using hash functions.
We use the definitions and notations introduced in Section 2. In the following sections, we use indices i to indicate the MapReduce operation MR_i to which an object corresponds. We use superscripts j or l to indicate the MPI processes. Objects without superscript are distributed between multiple processes and represent the entire state of a MapReduce operation.
Let's consider a sequence of l successive MapReduce operations $< \mathrm{MR}_1, ..., \mathrm{MR}_i, ..., \mathrm{MR}_l >$, where the output of MR_i equals the input of MR_{i+1} for $i \in \{1, ..., l-1\}$. Figure 11 illustrates the application of MapReduce operations $\mathrm{MR}_i = (\mu_i, \rho_i, s_i, h_i)$ and MR_{i+1} with MPI execution (S, P, γ), with HPC system S, processes P and bindings γ using p processes. Each MPI process $j \in P$ has a local multi-set of elements A_i^j as input, where $A_i = \bigcup_{j=1}^{p} A_i^j$ is the input multi-set for MR_i. During the computation of MR_i, each MPI process $j \in P$ executes Algorithm 12. This pseudo-code summarizes the three main phases: *map*, *shuffle*, and *reduce* phase.
During the map phase in Line 2, each process $j \in P$ applies the user-defined map function μ_i

Algorithm 12: MapReduce: overview execution on process j of MR_i

`Input`: $A_i^j \subseteq I_i,\ \mathrm{MR}_i = (\mu_i, \rho_i, s_i, h_i)$
1 save MR_i for fault-tolerance
2 $SB_i^j = \mathrm{map}(A_i^j)$ *// applay map μ_i and serialize (s_i) each element in A_i^j*
3 $RB_i^j = \mathrm{shuffle}(SB_i^j)$ *// gather the same keys at the same process*
4 $D_i^j = \mathrm{reduce}(RB_i^j)$ *// deserialize (s_i^{-1}) and reduce (ρ_i)*
5 `return` $D_i^j \subset O_i = I_{i+1}$

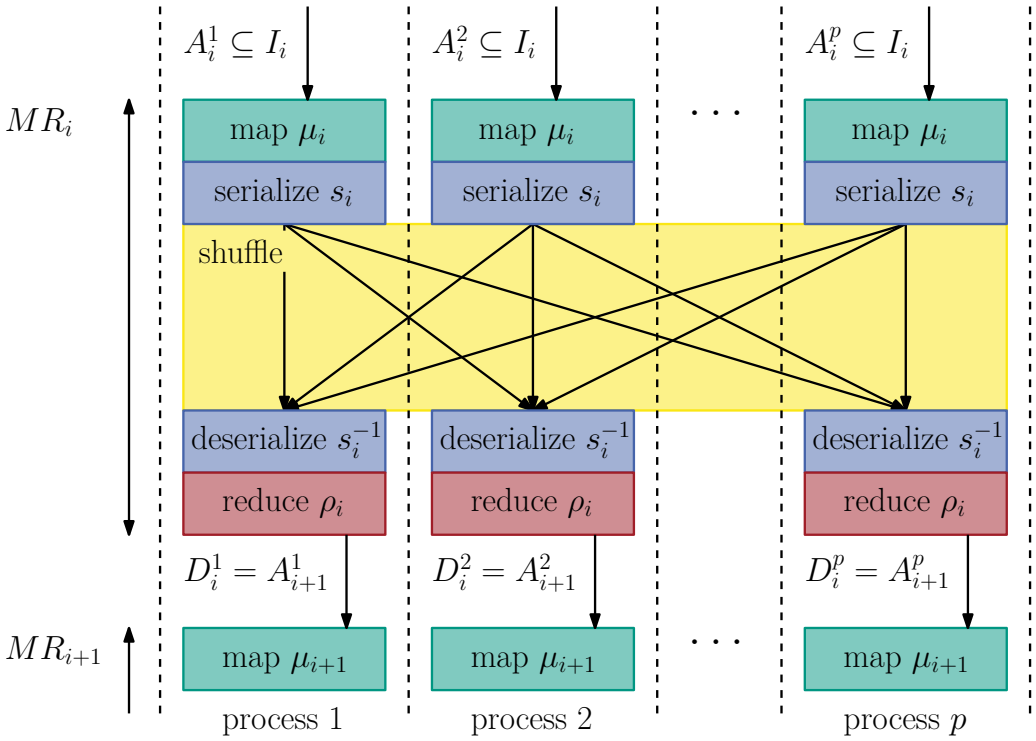


Figure 11: High-level illustration of our iterative MapReduce algorithm. This figure shows the three different phases during execution of the MapReduce operation MR_i: map, shuffle and reduce. We execute the MapReduce frame work on p MPI processes.

to each element in the multi-set A_i^j and gathers the local key-value pairs:

$$B_i^j = \bigcup_{e \in A_i^j} \mu_i(e) \subseteq K \times V.$$

Note that the union over all local key-value sets B_i^j corresponds to the multi-set of all key-value pairs B_i in MR_i (Section 2.2). We serialize the keys and values by apply s_i and save them into send buffers $SB_i^j = \{SB_i^{j,1}, ..., SB_i^{j,l}, ..., SB_i^{j,p}\}$. We save the serialized key-value pairs with key k into buffer $SB_i^{j,l}$, where $(h_i(k,x) \mod p) = l$ and $x \in \mathbb{N}$ the same seed on each process. During the shuffle phase (Line 3) each MPI process j sends the send buffer $SB_i^{j,l}$ to process $l \in P$. After the shuffle phase, process j has received a set of receive buffer RB_i^j, containing messages $SB_i^{l,j}$ from each process $l \in P$. We use the `MPI_All_to_allv` function to exchange the serialized key-value pairs. The reduce phase (Line 4) first applies s_i^{-1} to deserialize the key-value pairs. Note that no two different processes $j, l \in P$ contain key-value pairs with the same key. Each process j groups their key-value pairs by their key and creates a set of local key-values pairs

$$C_i^j = \{(k, X) : k \in K \wedge X = \{x : (k,x) \in B'^j_i\} \neq \emptyset\}.$$

The union of all local key-values sets C_i^j equals the list of key-values pairs C_i of MR_i (Section 2.2). Finally, we apply the user-defined reduce function ρ_i and produce local output multi-sets

$$D_i^j = \bigcup_{(k,X) \in C_i^j} \rho_i(k, X).$$

The output of the MapReduce operation MR_i is $D_i = \bigcup_{j=1}^p D_i^j$ and is distributed between the different processes. These local output sets are the input sets for the next MapReduce operation MR_{i+1}.

During the shuffle phase in Line 3 we send the key-value pair $(k, v) \in B_i$ to process $(h_i(k, x) \mod p)$ for a seed x. In Section 7 we use an alternative method to determine the process for each pair. Let $h_{max} = \max_{k \in K} h_i(k, x)$ be the maximum hash value and $s = \frac{h_{max}}{p}$. Then we send $(k, v) \in B_i$ to process

$$\left\lfloor \frac{h_i(k, x)}{s} \right\rfloor.$$

This method divides the range of hash values into p equally sized partitions.
Since the map and reduce phase require no communication, we detect and handle process failures during the shuffle phase. We provide a fault-tolerant pseudo-code for the shuffle phase (Section 5.2). In case of a failure we need to recompute different map and reduce steps, therefore we save the user-defined functions in Line 1.
In Section 6 we discuss map and reduce algorithms that run sequentially on each MPI process. We introduce a hybrid parallelization in Section 7, where we use an OpenMP parallelization per MPI process.

5.2. Single Node Fault-Tolerance

In Section 5.2.1 we present a fault-tolerance mechanism to handle single process failures for iterative MapReduce algorithms. Instead of saving checkpoints on a fault-tolerant file system, we send additional messages between processes. We generalize this approach for compute node failures in a HPC system (Section 5.2.2). A goal is to optimize our algorithm for low-overhead in case of no failure occurring and fault recovery in case of a failure.

5.2.1. Single MPI Process Failure

In this section we introduce a fault-tolerance mechanism for single process failures, which we extend to node or rack failures in Section 5.2.2. Let's consider two consecutive MapReduce operations MR_i and MR_{i+1}, where the output O_i of MR_i equals the input I_{i+1} of MR_{i+1}. Let (S, P, γ) be the MPI execution of the MapReduce algorithm. Figure 12 illustrates an MPI process failure during the reduce phase of MR_i or the map phase of MR_{i+1}. During the shuffle phase of MR_i, MPI exchanges the messages successfully. During the following reduce or map phase process $f \in P$ fails. MPI detects this during the shuffle phase of MR_{i+1}. We handle this failure restoring the lost data at process f and continue the execution of MR_{i+1}.
To recover the lost data, we need the messages sent to process f during the shuffle phase in MR_i:

$$RB_i^f = \{SB_i^{1,f}, ..., SB_i^{f,f}, ..., SB_i^{p,f}\}.$$

For this, we save the sent messages SB_i^j from each process $j \in P$ to other processes during the shuffle phase of MR_i^j. Additionally, we still need the message which process f sends to itself ($SB_i^{f,f}$). We call this message a *self-message*, which we cannot recover in case of a failure. Since the data saved at the failed process is lost, its self-message is no longer available. We therefore save the self-message $SB_i^{f,f}$ at another process, for instance at process $((f+1) \mod p)$. In case of a failure at process f, each process $j \in P \setminus \{f\}$ has saved message $SB_i^{j,f}$. Process $((f+1) \mod p)$ additionally contains $SB_i^{f,f}$. Therefore, we can reconstruct the receive buffer RB_i^f of process f.
Algorithm 13 describes the shuffle phase of MapReduce operation MR_i and illustrates how we provide fault-tolerance. Line 3 of Algorithm 12 calls this algorithm to exchange the messages provided by the map phase. Each MPI process $j \in P$ executes this shuffle phase. The input is a set of messages SB_i^j, one message for each process. In Line 4 we use an `MPI_all_to_allv`

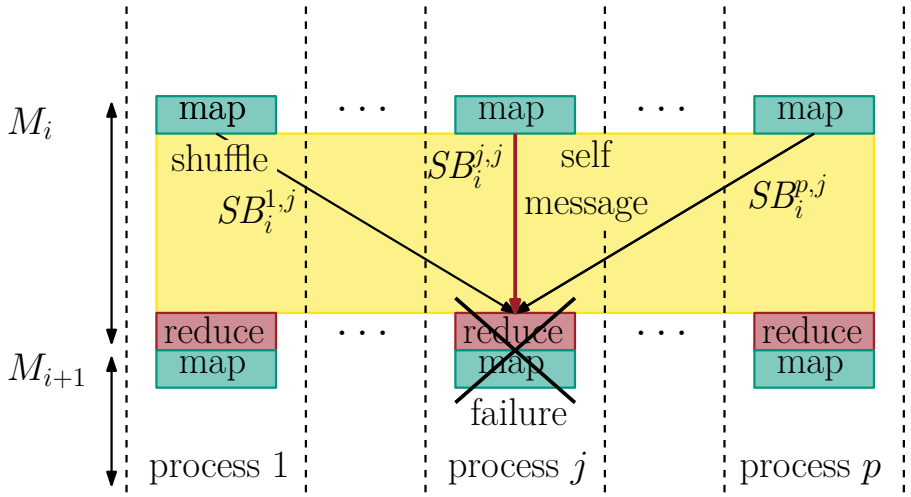


Figure 12: Illustration of a failure at process j during MapReduce operation MR_{i+1}. The implementation runs p processes. The process failure results in a loss of the self-message $SB_i^{j,j}$.

to sent exchange all messages at once. For each processes $j, l \in P$, we send message $SB_i^{j,l}$ from process j to process l. The MPI call in Line 4 requires the size of each receiving message, which we acquire with an all to all operation in Line 2. As indicated above, we save the send messages SB_i^j in Line 7 and exchange self-messages $SB_i^{j,j}$ by applying Algorithm 14.
The MPI calls in Lines 2 and 4 represent global synchronization steps, in which each MPI process participates together. During these calls, we can detect whether an MPI process has failed during a previous map or reduce step. We use ULFM (Section 3.1) to detect failures and restore the MPI functionality. We use Algorithm 15 in case of a detected failure in Lines 3 or 6. For each process $j \in P$, this recovery algorithm takes the current messages SB_i^j as input, recomputes the lost data by using previous messages and executes the shuffle phase again.
Algorithm 14 illustrates the save message algorithm. First we save all messages sent during the last shuffle phase SB_i^j in Line 1. We send the self-message $SB_i^{j,j}$ of process j to process $((j+1) \mod p)$, where p is the number of processes and save it there. We have to schedule the send and receive operations carefully to avoid deadlocks. We delete the old messages if no longer needed to decrease memory consummation. If the shuffle phase of the MapReduce operation MR_i has completed, we no longer need the saved messages of the previous MapReduce operation.
Algorithm 15 recomputes the data lost due to a process failure. Let MR_i, with $i > 0$, be the

Algorithm 13: Fault-Tolerant Shuffle of MapReduce operation MR_i on process j

```
Input: SB_i^j = {SB_i^{j,1}, ..., SB_i^{j,l}, ..., SB_i^{j,p}}   // send buffers: Line 3 in Algorithm 12
1 sizes : list<int>                                   // sized of the receive buffers required for AllToAllv
2 e = AllToAll(SB_i^j.sizes(), sizes)                 // exchange the size of SB_i^{j,i} and SB_i^{i,j} with process i
3 if e = ERROR then return Recover(SB_i^j) ;          // apply Algorithm 15
4 e = AllToAllv(SB_i^j, sizes, RB_i^j)                // exchange SB_i^{j,i} and SB_i^{i,j} with process i
5 RB_i^j = {SB_i^{1,j}, ..., SB_i^{l,j}, ..., SB_i^{p,j}}   // messeges received during Line 4
6 if e = ERROR then return Recover(SB_i^j) ;          // apply Algorithm 15
7 SaveMessage(SB_i^j)                                 // apply Algorithm 14
8 return RB_i^j
```

Algorithm 14: Save Messages Single Process Failure: during MR_i on process j

```
Input: SB_i^j = {SB_i^{j,1}, ..., SB_i^{j,l}, ..., SB_i^{j,p}}   // messages send during the last shuffle phase
1 save(SB_i^j)                                                   // save the messages in case a failure occures
2 send SB_i^{j,j} to ((j + 1) mod p)                             // send self-message
3 r = (j − 1) mod p                                              // process from which to recive the self-message
4 SB_i^{r,r} = receive the self-message from r
5 save(SB_i^{r,r})                                               // save the self-message in case a failure occures
```

MapReduce operation in which we detect the failure of process $f \in P$. This means process f failed during the reduce phase of MR_{i-1} or the map phase of MR_i. First, we determine the failed process and recover the MPI communicator with ULFM. If more than one process has failed, we cannot recover and refer to Section 5.2.2, where we handle some types of multi-process failures. ULFM modifies the MPI execution by removing the failed process. The MapReduce framework continues with a shrank MPI execution (S, P', γ') (Section 5.2.2).
We determine the messages, which MPI has sent during MR_{i-1} to the failed process (Line 2). Furthermore, we redistribute the saved messages equally between the remaining processes P' (Line 7). We construct redistribute send buffers RSB_{i-1}^j for each remaining process $j \in P'$. The algorithm sends the serialized key-values pairs with key k to process $l = h(k, y) \mod p'$, where y is another seed as used during the map phase. For each processes $j, l \in P'$, we send message $RSB_{i-1}^{j,l}$ from process j to process l (Line 8). This message exchange is similar to a shuffle during a successful MapReduce operation.
We reapply the reduce phase from MR_{i-1} (Line 9) and the map phase from MR_i (Line 10) as described in Section 5.1 (Algorithm 12). We merge the send messages SB_i^j with the recovered messages SB'^j_i into new send messages $NewSB_i^j$, where message $NewSB_i^{j,l}$ is sent from process j to process $l \in P'$. After merging, the message $NewSB_i^{j,l}$ contains the serialized key-value pairs with key k and $l = (h(k, y) \mod p')$. This corresponds to a valid map phase output of MR_i on the shrank MPI execution. Finally, we perform the shuffle of MR_i again and exit the error state (Line 12).
The MergeBuffer and Redistribute algorithms depend on the implementation of the map and reduce phase. We describe them in the Sections 6 and 7. If a process failure occurs during

Algorithm 15: Recover: during MR_i on process j

```
Input: SB_i^j = {SB_i^{j,1}, ..., SB_i^{j,l}, ..., SB_i^{j,p}}   // messages of the shuffle phase from MR_i
1  f = identify the failed process
2  if f = (j − 1) mod p then                                     // process j has saved the self-message of f
3  |  S_i^j = {SB_{i-1}^{j,f} ∪ SB_{i-1}^{f,f}}                 // messages to failed process
4  else
5  |  S_i^j = {SB_{i-1}^{j,f}}                                  // messages to failed process
6  end
7  RSB_{i-1}^j =Redistribute(S_i^j)                             // distribute saved messages evenly
8  RRB_{i-1}^j =shuffle(RSB_{i-1}^j)                            // exchange distribute saved messages evenly
9  RD_{i-1} = reduce(RRB_{i-1}^j)                               // deserialize (s_{i-1}^{-1}) and reduce (ρ_{i-1})
10 SB'^j_i = map(RD_{i-1})                                      // applay map μ_i and serialize (s_i) each element in A_i^j
11 NewSB_i^j = MergeBuffer(SB_i^j, SB'^j_i)                     // merge send messages
12 return shuffle(NewSB_i^j)
```

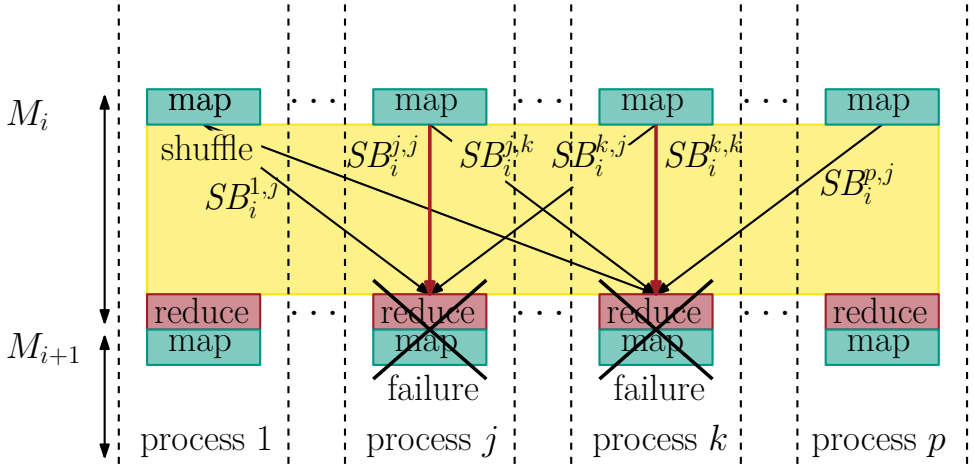


Figure 13: Illustration of a failure at process j during MapReduce operation MR_{i+1}. The implementation runs p processes. The process failure results in a loss of the self-message $SB_i^{j,j}$.

this recovery phase, then we cannot recover. Lets consider an MPI execution (S, P, γ), with $|P| > 2$ and two different process failures $j, k \in P$ as illustrated in Figure 13. If the failure k occurs during the recovery of failure j, we miss at least message $SB_i^{j,k}$, since the saved messages at process j are no longer available. Furthermore, we would need to send at least the entire p messages SB_i^j to another process instead of the self-message, because the second failure is arbitrary.This would double the communication volume of the fault-tolerant shuffle Algorithm 13. In Section 5.2.2 we propose a mechanism to recover from certain multi-process failures.

5.2.2. Single Node Failure

The single process failure fault-tolerance has limitations. On an HPC system, a failure usually implies that more than one MPI process fails. For instance, a compute node failure stops the execution of all MPI processes on this node. Since our fault-tolerance model only handles single process failures, we can only start one process per node. To make use of all cores on a node, we have to parallelize the execution locally by using a shared memory model (Section 7). This local parallelization is difficult because we have to take the NUMA architecture of the nodes into consideration (Section 2.1). We need to minimize memory access across different NUMA nodes. To avoid this problem we want to start one MPI process per NUMA node and parallelize locally.

We modify the single process fault-tolerance mechanism (Section 5.2.1) to allow single node failures. Therefore, we start the same number of processes on each compute node. We expand the definition of a self-message to include the messages, which MPI sends between processes on the same node. The following fault-tolerance mechanism saves all messages send between processes on the same compute node at a different node.

Let us consider a MapReduce operation MR_i with MPI execution (S, P, γ), with processes P, HPC system $S = (C, M, N)$, and bindings γ. First, we modify the save self-message Algorithm 14 and save all messages send on the same node (Algorithm 16). The input at process j are the messages produced by the map phase and exchanged during the shuffle phase SB_i^j (Algorithm 13). We save the messages send by j (Line 1). Then we determine the processes P^j bound to the same node as process j (Line 2). We gather the message S^j send between processes on the same node (Line 3).

Algorithm 16: Save Messages Single Node Failure: during MR_i on process j

```
Input: SB_i^j = {SB_i^{j,1}, ..., SB_i^{j,l}, ..., SB_i^{j,p}}, (S, P, γ)   // messages of the shuffle phase from MR_i
1 save(SB_i^j)
2 P^j = {p ∈ P | N_j ∈ N : γ(p) ⊂ N_j ∧ γ(j) ⊂ N_j}        // detrmine processes on same node as j
3 S^j = {SB_i^{j,l} | l ∈ P^j}                              // detrmine messages send to processes on same node as j
4 send S^j to process α_k(j) on the successive node
5 receive RS^j = S^l from process α_{k-1}^{-1}(j) = l on the proceeding node
6 save(RS^j)                                                // save the self-messages received from process α'_{k-1}(j)
```

Lets number the nodes $N = \{N_0, ..., N_n\}$ number consecutively and let N_k be the node hosting process j ($\gamma(j) \subseteq N_k$). Each process in N_k sends its local messages to a unique process on the successive node in N. We determine a bijection $\alpha_k : N_k \to N_{k'}$, where $k' = ((k+1) \mod n+1)$. This function α_k assigns each process a unique process in $N_{k'}$. We send S^j to the corresponding process $\alpha_k(j)$ (Line 4). Finally, we receive (Line 5) and save (Line 6) the self-message from a process on the proceeding node. Each process saves the self-messages from exactly one process on a different node. We have to schedule the message exchange in Lines 4 and 5 carefully to avoid deadlocks.
Figure 14 illustrates this procedure. On each node we run four processes. The self-message of process 0 contains the message send to itself as well as the message sent to the other processes on node i: 1, 2 and 3. These messages are sent to process 4 on node $i+1$. We sort the processes on each node by their identifier. We send the self-messages to the next node to the process with the same index in the sorted process list. In case node i fails, we have saved all messages send between processes on node i at node $i+1$.
Note that the save message Algorithm 16 has a super linear speedup and depends on the size of the serialized key-value pairs as shown in Lemma 5.1.

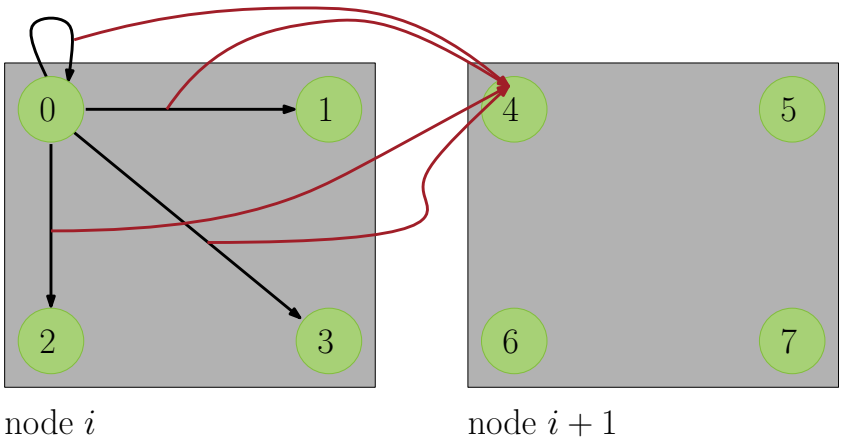


Figure 14: Illustration of saving self-messages for single node failures with more than one process per node. This figure shows the messages, which process 0 sends on the same node during the shuffle phase. These messages are sent to process 4. This ensures no data lost in case node i fails. We save the self-messages of a node j at node $(j+1 \mod p)$, where p is the process count.

Lemma 5.1. *Let's consider the save message Algorithm 16 performed after the shuffle phase on p processes and k processes per node. Let m be the number of machine words to save all key-value pairs and $\bar{m}$ the maximum number of machine words consumed by a user-defined reduce function. Then the expected runtime of Algorithm 16 is*

$$\mathcal{O}\left(k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p^2\right)\right).$$

The value $b(x, y)$ indicates the maximum expected number of balls in one bin, where x is the number of balls and y the number of bins. We place balls interdependently at random. Furthermore, if $m \in \Omega(\bar{m}p^2 \log(p))$, then the algorithm has a time complexity of:

$$\mathcal{O}\left(k\frac{m}{p^2}\right).$$

Proof. As described in Section 5.1 the map phase distributes the key-value pairs uniformly by their hashed key. If we assume that the used hash function is a truly random mapping, then the pairs are distributed uniformly at random into p messages on each MPI process. All in all the pair are distributed between p^2 messages. A user-defined reduce function processes at most $\bar{m}$ machine words and therefor at most $\bar{m}$ pairs. Since the pairs are distributed uniformly at random by key, we can employ the random static load balancing Lemma 3.1. The worst case for random static load balancing occurs if $\lceil m/\bar{m} \rceil$ user-defined reduce function process $\bar{m}$ key-value pairs. The remaining functions process no pairs. This results in a memory distribution which is as sewed as possible [86].
Hence, the expected maximum number of pairs in one of p^2 messages lies in $\mathcal{O}\left(\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p^2\right)\right)$. Each process sends and receives exactly k self-messages. This results in an expected runtime of $\mathcal{O}\left(k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p^2\right)\right)$ for Lines 2 to 6.
Furthermore, since we do not need the send messages after the shuffle phase we do not need to copy the `MPI_All_to_allv` send buffers in Line 1. We can perform this save operation in constant time. All in all Algorithm 16 has an expected runtime of

$$\mathcal{O}\left(k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p^2\right)\right).$$

If $m \in \Omega(\bar{m}p^2 \log(p))$, then $\mathcal{O}\left(\bar{m} \cdot b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p^2\right)\right) = \mathcal{O}\left(\frac{m}{p^2}\right)$ (Lemma 2.2) and Algorithm 16 has an expected runtime of

$$\mathcal{O}\left(k\frac{m}{p^2}\right).$$

□

We have to modify the recover Algorithm 15 only slightly. Instead of determining only one failed process in Line 1 we have to find all processes $N_f \in N$ on the failed node. The messages we need for recovery include $SB_{i-1}^{j,l}$ destined to each failed process $l \in N_f$. Furthermore, we determine the processes saving the self-messages send on the failed node. We can execute the recover algorithm from Line 7 as for the single process failure case (Section 5.2.1).
We can expand the single node failure fault-tolerance to rack or even HPC failures. HPC racks group multiple compute nodes together. A rack failure causes multiple nodes to fail. We can employ a similar technique by saving all messages send between nodes on one rack to another. This allows us to handle rack failures. Furthermore, we can use the same approach if we want to execute on multiple HPC systems together and an entire HPC cluster fails. We study only single node failures as a proof of concept. We do not have the resources to test the approach on multiple racks.

6. MPI Parallelized MapReduce

In the following sections, we introduce algorithms for the map (Section 6.1), reduce (Section 6.2), and recovery (Section 6.3) phases for our MapReduce framework with a single node fault-tolerance of Section 5. We parallelize our MapReduce algorithm with MPI by starting p parallel processes.

6.1. Map

6.1.1. Algorithm

As described in Section 5.1, the map phase applies the user-defined map function, serializes the key-value pairs and saves the results in an `MPI_All_to_allv` send buffer. We parallelize the map phase by using MPI. Each process processes a subset of the input elements. We illustrate the general idea of our map phase in Figure 15. On each process, we apply the user-defined map function and sort the resulting key-value pairs according to their hashed key. Then we group the pairs according to their key. We determine the destination process of each aggregated pair by its hashed key and save the serialized keys followed by its values in a send message.
Algorithm 17 illustrates the map phase executed one process. Let MR$= (\mu, \rho, s, h)$ be a MapReduce operation with user-defined map function μ, serialization function s, and hash function h (Section 2.2). The MapReduce operation runs on an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ (Section 2.1). We execute Algorithm 17 on each process $j \in P$ in parallel. The inputs at process j are a subset of input elements $A^j \subseteq I$ and the map reduce operation MR. First, we iterate over the elements in A^j and apply the user-defined map function μ. For each resulting key-value pair (k, v), we save the key-serialized value pair $(k, s(v))$

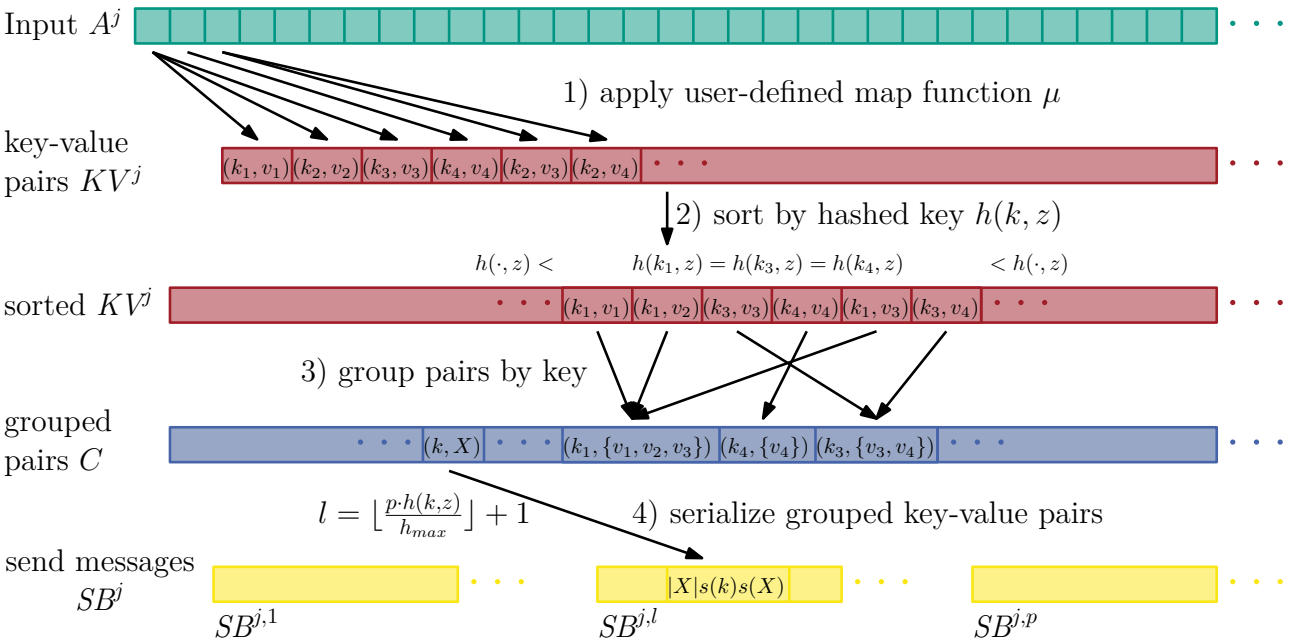


Figure 15: Illustration of the map phase executed on MPI process j. Let z be the same seed for the hash function on all processes. (1) First, we apply the user-defined map function. (2) Secondly, we sort the pairs according to their hashed key. (3) For all pairs with the same hashed key, we gather all values with the same key. (4) Finally, we save the size of values, followed by the serialized key and values at a send message destined for the corresponding process.

Algorithm 17: Purely MPI parallelized map phase executed on process j

```
Input: A^j ⊆ I, MR = (μ, ρ, s, h)  // multiset of input elements A^j, MapReduce operation MR
1  KV, HI = ∅                      // initialize an empty list of key-value pairs and hash-index pairs
2  foreach e ∈ A^j do              // iterate over the input elements
3  |  foreach (k, v) ∈ μ(e) do     // apply the user-defined map function
4  |  |  HI += (h(k, z), |KV|)     // save the hashed key h(k, z) and the index of (k, s(v)) in KV
5  |  |  KV += (k, s(v))           // save the key k and the seralized value s(v)
6  |  end
7  end
8  sort HI by the first pair element
9  SB^j = {SB^{j,1}, ..., SB^{j,l}, ..., SB^{j,p}}   // initialize p empty send messages
10 while |KV| > 0 do
11 |  h = min_{(k,·)∈KV} h(k, z)   // determine the minimum hash value
   |  // determine the key-value pairs with minimal hashed key and group them by key
12 |  C = {(k, X) | k ∈ K ∧ h(k, z) = h ∧ X = {s(v) | (k, s(v)) ∈ KV}}
13 |  l = ⌊(p·h)/h_max⌋ + 1          // determine destination process l for pairs with hash h
14 |  foreach (k, X) ∈ C do
15 |  |  SB^{j,l} += |X| + s(k)     // add the size of values and the serialized key
16 |  |  foreach s ∈ X do SB^{j,l} += s   // add the serialized values
17 |  end
18 |  KV = KV \ {(k, s) | (k, X) ∈ C ∧ s ∈ X}   // remove the processed pairs
19 end
20 return SB^j                      // return a message for each node
```

in KV. Let z be the same seed on each process and i the index of $(k, s(v))$ in KV. We save the pair of hashed key $h(k, z)$ and index i in HI.

To avoid cache efficiency problems, we group the key-value pairs by key with a cache efficient sorting algorithm (Section 3.4). We use the IPS^2Ra radix sort (Section 3.5) to sort the pairs according to their hashed key and achieve a linear time complexity (Section 6.1.2). Note that we sort the hash-index pairs HI to avoid the reevaluation of hash functions during sorting and prevent problems with large keys and values (Line 8). We can use the sorted index sequence of HI to get sorted key-value pairs.

Let h_{max} be the maximum possible hash value plus one. During the shuffle phase, we send a key-value pair (k, v) with hashed key $h(k, z) \in [(l-1) \cdot \lfloor h_{max}/p \rfloor, l \cdot \lfloor h_{max}/p \rfloor)$ to process l. The map phase at process j saves all serialized key-value pairs destined to process l in message $SB^{j,l}$. Finally, we iterate over the sorted hash-index pairs HI (Line 10). We process the key-value pairs with current minimal hash value h (Lines 11-12). We add the pairs in a hash table according to their key. This allows us to gather all values with the same key. Note that we have to choose a different seed for the hash function used in the hash table. I we would choose the same hash function this would result in hash collisions since all keys have the same hash value with seed z. The algorithm iterates over all pairs $(k, X) \in C$, where k is the key and X the set of all serialized values with key k. We save the size of all serialized values $|X|$, followed by the serialized key $s(k)$ and its serialized values X in message $SB^{j,l}$ (Line 14). We determine the destination process l by computing $\lfloor p \cdot h(k, z)/h_{max} \rfloor + 1$ (Line 13). Note that we still have to concatenate the send messages into a continuous send buffer for the `MPI_All_to_allv` operation performed during the shuffle phase.

6.1.2. Complexity

We adopted the general structure of the BSP MapReduce algorithm introduced in [86] and achieve the same expected running time (Lemma 6.1). We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4.

Lemma 6.1. *The input elements A are distributed randomly between p processes. Then we can implement the map phase (Algorithm 17) with an expected runtime of*

$$\mathcal{O}\left(\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, p\right) + \bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, p\right)\right).$$

If $w \in \Omega(\bar{w}p\log(p))$ and $m \in \Omega(\bar{m}p\log(p))$, then we have an expected time complexity of:

$$\mathcal{O}\left(\frac{w+m}{p}\right).$$

Proof. We follow the proof of Theorem 4.1 in [86]. We assume that the input elements A are chosen randomly between the different processes. During the map phase, each process applies the map function once for each input element (Line 3). For each intermediate key-value pair, we apply the serialization function once for each key and value (Lines 5 and 15). We compute and save the hash value of each key once in Line 4. Since the input elements are distributed at random, we can apply the static load balancing Lemma 3.1. Therefore, the worst case occurs, if the execution time is zero for all input elements except for $\lceil w/\bar{w} \rceil$ elements, which require $\bar{w}$ time. Randomly assigning $\lceil w/\bar{w} \rceil$ elements between p processes corresponds to the balls in bins problem (Section 2.4). Then let $b\left(\lceil w/\bar{w} \rceil, p\right)$ be the maximum expected number of elements per process.
Hence, Applying the user-defined map function requires $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$ time. Each user-defined map function can produce at most $\bar{w}$ pairs. By using the same reasoning as above, the maximum expected number of key-value pairs per process lies in $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$. We use a linear time radix sort algorithm (Section 3.5) in Line 8, which results in a runtime of $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$. In Line 10 we iterate over the hash-index pairs and process the key-value pairs in ascending order of their hashed key. For all pairs with the same hashed key, we group the pairs with the same key by using a hash table, which requires linear time. All in all, Lines 10 to 19 require $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$ time.
Finally, we have to copy the send messages SB^j into a single `MPI_All_to_allv` send buffer. Since the input elements are distributed randomly between the different processes, the m machine words produced during the map phase are distributed randomly and we can use Lemma 3.1. Hence, the worst case occurs if the machine words are divided into $\lceil m/\bar{m} \rceil$ elements with a size of $\bar{m}$. This results in an expected runtime of $\mathcal{O}\left(\bar{m} \cdot b\left(\lceil m/\bar{m} \rceil, p\right)\right)$ for the send buffer construction. All in all, we have an expected runtime of

$$\mathcal{O}\Big(\underbrace{\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, p\right)}_{\text{Lines (2–7),8,(10–19)}} + \underbrace{\bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, p\right)}_{\text{construct send buffer}} \Big) \tag{6.1}$$

If $w \in \Omega(\bar{w}p\log(p))$ and $m \in \Omega(\bar{m}p\log(p))$, then we can apply Lemma 2.2 and receive an expected runtime of

$$\mathcal{O}\left(\frac{w+m}{p}\right).$$

□

6.2. Reduce

6.2.1. Algorithm

As described in Section 5.1 the reduce phase receives key-value pairs from the shuffle phase. During the map phase, we ensure that the shuffle phase sends the key-value pairs with the same key to the same process. The reduce phase then gathers all values for a given key and applies the user-defined reduce function. In Figure 16 we illustrate the general idea of our reduce phase. First we have to deserialize the keys and values received from the shuffle and map phase. We call the key followed by a set of values produced during the map phase *key-values pairs*. We sort these pairs according to their hashed key. Then the reduce phase processes the pairs in increasing order of their hashed key. For all key-values pairs with the same hashed key, we group the pairs by their key. Finally, we have all values for a given key and we can apply the user-defined reduce function.
Algorithm 18 describes the reduce phase of our MapReduce library. Let MR= (μ, ρ, s, h) be a MapReduce operation with user-defined map function μ, serialization function s, and hash function h (Section 2.2). We parallelize the reduce phase with an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ (Section 2.1). Let z be the same seed used for the hash functions during the previous map phase and h_{max} the maximum possible hash value plus one. Let (k, X) be a *key-values* pair with key k and values X aggregated during the map phase. The shuffle phase sends (k, X) with hashed key $h(k, z)$ to process $l \in P$, if $h(k, z) \in [(l-1) \cdot \lfloor h_{max}/p \rfloor, l \cdot \lfloor h_{max}/p \rfloor)$. Therefore, we send the key-values pairs with the same key to the same process. The messages RB^j represent all serialized key-values pairs send to process $j \in P$.
Let's consider the execution of the reduce phase on process $j \in P$. First, we deserialize the key-values pairs and save them in an array KV. Similar to the map phase, we save for each

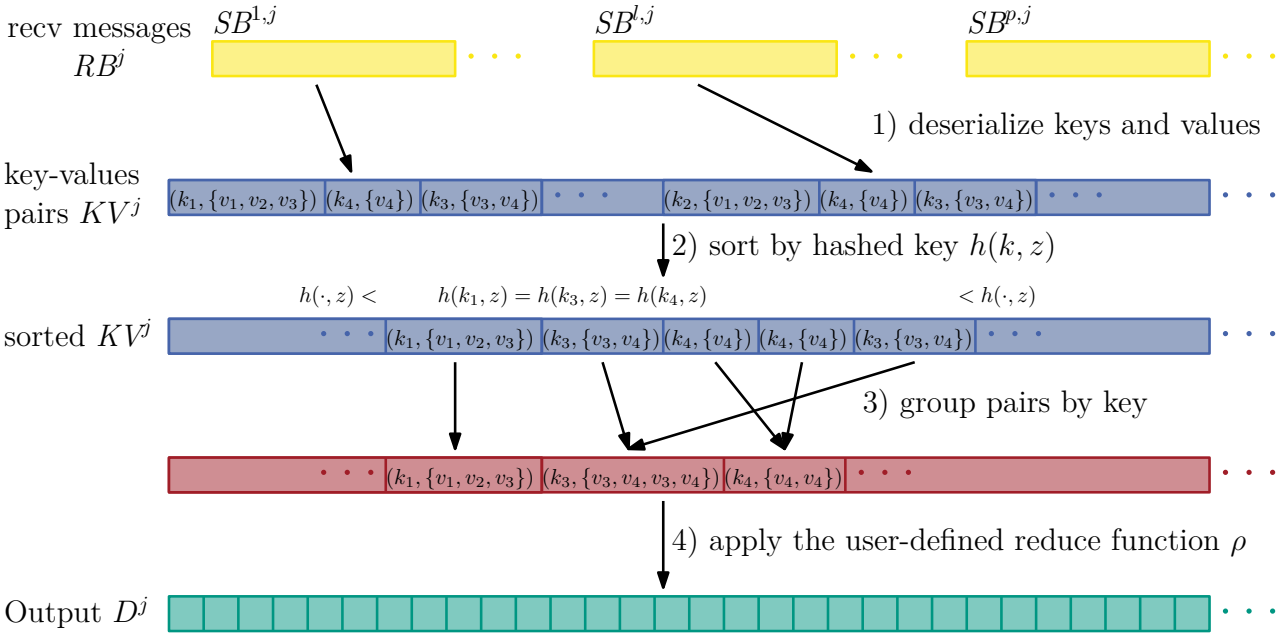


Figure 16: Illustration of the reduce phase executed on MPI process j. Let z be the same seed for the hash function as used during the map phase. (1) First, we deserialize the key-values pairs produced during the map phase. (2) Secondly, we sort the pairs according to their hashed key. (3) For all pairs with the same hashed key, we gather all values with the same key. (4) Finally, we apply the user-defined reduce function.

Algorithm 18: Reduce phase executed on process j

```
Input: RB^j, MR = (μ, ρ, s, h)          // received messages RB^j, MapReduce operation MR
1  KV, HI = ∅                           // initialize an empty list of key-values pairs and hash-index pairs
2  foreach (s(k), s(X)) ∈ RB^j do       // iterate over the received messages
3  |  HI += (h(k, z), |KV|)             // save the hashed key and the index of (k, X) in KV
4  |  KV += (k, X)                      // deserialize and save the key-values pairs
5  end
6  sort HI by the first pair element
7  D^j = ∅                              // initialize an empty set of output elements
8  while |KV| > 0 do
9  |  h = min_{(k,·)∈KV} h(k, z)        // determine the minimum hash value
   |  // determine the key-value pairs with minimal hashed key and group them by key
10 |  C = {(k, Y) | k ∈ K ∧ h(k, z) = h ∧ Y = ⋃{X | (k, X) ∈ KV}}
11 |  foreach (k, Y) ∈ C do
12 |  |  D^j = D^j ∪ ρ(k, Y)           // apply the user-defined reduce function
13 |  end
14 |  KV = KV \ {(k, s) | (k, X) ∈ C ∧ s ∈ X}   // remove the processed pairs
15 end
16 return D^j                           // return the output elements
```

pair (k, X) the pair (h, i) in HI, where $h = h(k, z)$ is the hash value of key k and i is the index of (k, X) in KV. Then we sort the hash-index pairs HI according to the hashed key (Line 6). We use the cache efficient and linear time radix sort implementation IPS^2Ra (Section 3.5). In Lines 8 to 15 we gather all values to their corresponding keys and apply the user-defined reduce function. We iterate over the sorted hash-index list HI and process the key-values pairs in increasing order of their hashed key. First, we determine all key-values pairs with hashed key equal to the current minimal hash value determined in Lines 9. By using a hash table and different hash function, we group the pairs by their key in C. Finally, we have determined all values for a given key and we can apply the user-defined reduce function (Line 12).

6.2.2. Complexity

We adopted the general structure of the BSP MapReduce algorithm introduced in [86]. We achieve the same expected runtime (Lemma 6.2). We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4. Note that if we assume that the hash function h is a truly random mapping, then our map and reduce phase distribute the keys interdependently uniformly at random between all processes.

Lemma 6.2. *Let the key-value pairs be distributed uniformly at random by their key between p processes. Then the reduce phase (Algorithm 18) has an expected running time of:*

$$\mathcal{O}\left(\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, p\right)\right).$$

Furthermore, if $w \in \Omega(\bar{w}p\log(p))$, then the reduce phase has an expected time complexity of:

$$\mathcal{O}\left(\frac{w}{p}\right).$$

Proof. We follow the proof of Theorem 4.1 in [86]. Note that the user-defined reduce function processes key-values pairs, hence the maximum key-value pair size is $\bar{w}$. The term w indicates the time needed to apply the user-defined reduce function to all pairs, hence we have at most w pairs. Since keys are distributed uniformly at random between p processes, we can use the static load balancing Lemma 3.1. The worst key-values pairs distribution occurs if we have $\bar{w}$ pairs of size $\lceil w/\bar{w} \rceil$. Therefore, the maximum expected number of pairs per process is $b\left(\lceil w/\bar{w} \rceil, p\right)$. We use a linear time radix sort algorithm in Line 6, which results in an expected runtime of $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$.
In Lines 8 to 15 we iterate once over all key-values pairs in ascending order of their hashed key. We can perform the grouping by key in linear time with a hash table. Hence, Lines 8 to 15 without Line 12 have an executed runtime of $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$.
Since the shuffle phase distributed the keys uniformly at random we can apply Lemma 3.1 for the user-defined reduce function load balancing. The worst case occurs, if there are exactly $\lceil w/\bar{w} \rceil$ function calls with a non-zero execution time of $\bar{w}$. Since w includes deserialization and hashing, Lines 2 to 5 and 12 have an expected runtime of $\mathcal{O}\left(\bar{w} \cdot b\left(\lceil w/\bar{w} \rceil, p\right)\right)$. All in all, Algorithm 18 has an expected runtime of

$$\mathcal{O}\left(\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, p\right)\right).$$

If $w \in \Omega(\bar{w}p\log(p))$, then we can apply Lemma 2.2 and receive an expected runtime of

$$\mathcal{O}\left(\frac{w}{p}\right).$$

□

6.3. Fault-Tolerance

6.3.1. Redistribute Algorithm

To realize the fault-tolerance mechanism, we provide an algorithm for the Redistribute operation used in Algorithm 15 (Section 5). First, each process gathers all messages sent to the failed compute node during the previous shuffle phase. The algorithm includes the self-messages of the failed node. We distribute the key-values pairs uniformly between the remaining processes. This allows us to parallelize the recovery phase. The redistribution returns an `MPI_All_to_allv` send buffer used to exchange the key-value pairs.
Let p_{old} and p_{new} be the number of processes before and after a node failure. Then $k = p_{old} - p_{new}$ indicates the number of failed processes. Let $\text{MR} = \text{MR}_i = (\mu, \rho, s, h)$ be the previous MapReduce operation with serialization function s and hash function h (Section 2.2). The MapReduce operation runs on an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ, with $|P| = p_{new}$ (Section 2.1). Let $F = \{f_1, ..., f_2\}$ be the list of processes on the failed node, with $|F| = k$. The input of the Redistribute Algorithm 19 is a list of messages S send to the failed processes F this includes the saved self-messages.
During the save message operation after the shuffle phase in Section 5, each process has saved exactly one message send to one of the failed processes. Furthermore, exactly k processes have saved the self-messages of one of the failed processes. Therefore, $p_{new} - k$ processes have saved k messages and k processes have saved $2 \cdot k$ messages. The general idea of the redistribution algorithm is to divide each message into p_{new} buckets and send each bucket to a different process. Hence, we divide each message between the remaining processes. More precisely, we partition the serialized key-values pairs according to their hashed key. A message

$S^{j,f}$ send from process j to a failed process f contains the key-value pairs with hashed key in $[(f-1)\cdot\lfloor h_{max}/p_{old}\rfloor, f\cdot\lfloor h_{max}/p_{old}\rfloor)$, where h_{max} is the maximum hash value plus one. We divide $S^{j,f}$ into p_{new} buckets $b \in \{1,...,p_{new}\}$, where bucket b contains the key-values pair (k, X), if

$$b = \left\lfloor \frac{h(k,z) - (l-1)\frac{h_{max}}{p_{old}}}{\frac{h_{max}}{p_{new}}} \right\rfloor + 1.$$

After dividing each message into buckets, we save bucket b into a message $RSB^{j,b}$ for process b (Lines 10 and 14). We distinguish between two different cases. Process j has saved the self-messages of a failed process in Line 14 or not in Line 10. If we have saved no self-messages, then there are no two key-values pairs in different messages. Therefore, we copy the bucket b of each message $S^{j,f}$ into the redistribution buffer $RSB^{j,b}$ destined for process b. If we have saved the self-messages, then for each failed process $f \in F$ we have two messages $S^{j,f}$ and $S^{i,f}$ previously destined to process f. There may be key-value pairs with the same key in both messages. To combine the values with the same key we perform a merge for each bucket b in Line 14.
We produced $S^{j,f}$ and $S^{i,f}$ during a map phase and are therefore lists of serialized key-values pairs sorted according to their hashed key. For each bucket b we process the pairs in ascending order of their hashed key. By using a hash table, we gather for each key with the same hash value all its corresponding serialized values. We save the resulting key-values pair into the redistribution message $RSB^{j,b}$ destined for process b as described above.
Note that we have to deserialize the keys once to determine its hash value and to add the key into the hash table during the merge. We do not need to deserialize the values. During the map

Algorithm 19: Redistribute: shared memory

// saved messages S send to the failed processes $\{f_1,...,f_2\}$, new and old number of processes p_{old}, p_{new} and MapReduce operation MR

Input: $S = \{S^{j,f_1},...,S^{j,f_2},S^{i,f_1},...,S^{i,f_2}\}, p_{old}, p_{new} \in \mathbb{N}, \text{MR} = (\mu,\rho,s,h)$

1 **foreach** $S^{l,f} \in S$ **do** *// divide each saved message into p_{new} buckets*
2 **foreach** $b \in \{1,...,p_{new}\}$ **do** *// construct bucket b*
3 $S^{l,f}(b) = \left\{(s(k),s(X)) \in S^{l,f} | k \in K, X \subseteq V \wedge \left\lfloor \frac{h(k,z)-(l-1)\frac{h_{max}}{p_{old}}}{\frac{h_{max}}{p_{new}}} \right\rfloor = b-1\right\}$
4 **end**
5 **end**
6 $RSB^j = \{RSB^{j,1},...,RSB^{j,x},...,RSB^{j,p_{new}}\}$ *// initialize messages $RSB^{j,x}$ send from j to x*
7 **foreach** $f \in \{f_1,...,f_2\}$ **do** *// iterate over the failed processes $\{f_1,...,f_2\}$*
8 **if** $S^{i,f} \notin S$ **then** *// process j has not saved the self-message of a failed process*
9 **foreach** $b \in \{1,...,p_{new}\}$ **do** *// iterate over buckets b*
10 $RSB^{j,b} = RSB^{j,b} \cup S^{j,f}(b)$ *// copy $S^{j,f}(b)$ into the new bucket $RSB^{j,b}$*
11 **end**
12 **else**
13 **foreach** $b \in \{1,...,p_{new}\}$ **do** *// iterate over buckets b*
  // merge buckets $S^{j,f}(b)$ and $S^{i,f}(b)$
14 $RSB^{j,b} = RSB^{j,b} \cup \left\{(s(k),X_1 \cup X_2) | (s(k),X_1) \in S^{j,f}(b) \wedge (s(k),X_2) \in S^{i,f}(b)\right\}$
15 **end**
16 **end**
17 **end**
18 **return** RSB^j

phase, we have saved the serialized value size. This allows us to determine the next key-value pairs in a message $S^{j,f}$ without deserializing all its values (Section 6.1.1).

6.3.2. MergeBuffer Algorithm

Algorithm 20 performs the merge buffer operation needed during the single node failure fault-tolerance mechanism (Section 5.2). During recovery, we recompute the reduce phase and map phase for the key-value pairs lost during the failure. This results in a recovery send buffer RSB^j at each process j. The merge buffer algorithm takes RSB^j and the output of the map phase SB^j as input. It merges the sorted key-values pairs of both buffers and distributes the resulting pairs between the still working processes.
Let p_{new} be the number of remaining processes after recovery and p_{old} be the number of processes before failure. Then $k = p_{old} - p_{new}$ represents the number of processes lost due to the node failure. Let $\text{MR} = \text{MR}_i = (\mu, \rho, s, h)$ be the current MapReduce operation with user-defined map function μ, serialization function s, and hash function h (Section 2.2). The MapReduce operation runs on an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ, with $|P| = p_{new}$ (Section 2.1). Let SB^j be the messages produced during the map phase and RSB^j be the messages produced during the recovery phase at process j. Since we have produced these messages during the map phase, they are sorted by their hashed key.
Algorithm 20 illustrates the MergeBuffer operation during the recovery phase. We process the serialized key-values pairs in SB^j and RSB^j in ascending order of their hashed key (Line 2). First, we determine the current minimal hashed key h (Line 3). Then we gather all pairs with the same hashed key and group them according to their keys (Line 4). We perform this aggregation by adding these pairs in a hash table. Similar to the map phase (Section 6.1), we determine the destination process $l = \lfloor p \cdot h/h_{max} \rfloor$ of each pair by their hashed key (Line 5). Then we save the merged pairs in a message for process l (Line 6). This results in sorted key-values pairs inside the new messages $NewSB^j$, which the shuffle phase distributes.
Note that our reduce phase does not necessarily require a sorted list of key-values pairs (Section 6.2.1), but the redistribution phase does (Section 6.3.1). Since we exchanged the messages $NewSB^j$ during the next shuffle phase, we need to save the send buffer and self-messages for our fault-tolerance mechanism. If two node failures occur on to successive MapReduce

Algorithm 20: MergeBuffer: MPI algorothm (execution on process j)

// send buffer from map SB^j and RSB^j recovery phase,MapReduce operation MR
`Input`: $SB^j, RSB^j, \text{MR} = (\mu, \rho, s, h)$
1 $NewSB^j = \left\{NewSB^{j,1}, ..., NewSB^{j,l}, ..., NewSB^{j,p_{new}}\right\}$ *// initialize p_{new} empty messages*
2 `while` $|SB^j| > 0 \wedge |RSB^j| > 0$ `do`
3 $h = \min_{A \in \{SB^j, RSB^j\}} \min_{(s(k), s(X)) \in A} h(k, z)$ *// determine the current minimum hash value*
// group key-values pairs with the same the hased key
4 $C = \{(s(k), s(Y)) | k \in K \wedge h(k, z) = h \wedge Y = \bigcup\{X | (s(k), s(X)) \in SB^j \cup RSB^j\}\}$
5 $l = \lfloor \frac{hp_{new}}{h_{max}} \rfloor + 1$ *// determine the destination process l*
// save the grouped key-values pairs into a new send buffer
6 `parallel_foreach` $(s(k), s(Y)) \in C$ `do` $NewSB^{j,l} += |Y| + s(k) + s(Y)$
$SB^j = SB^j \setminus \{(s(k), s(X)) \in SB^j | h(k, z) = h\}$ *// remove processed pairs*
7 $RSB^j = RSB^j \setminus \{(s(k), s(X)) \in RSB^j | h(k, z) = h\}$ *// remove processed pairs*
8 `end`
9 `return` $NewSB^j$

operations, we require sorted pairs for Algorithm 19.

6.3.3. Complexity

To analyze the complexity of Algorithms 19 and 20, we adopt a similar approach as during the previous sections. We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4. Let p_{old} and p_{new} be the number of processes before and after a node failure. Then $k = p_{old} - p_{new}$ indicates the number of failed processes.

Lemma 6.3. *The redistribution Algorithm 19 performed after a node failure can be implemented with an expected runtime of:*

$$\mathcal{O}\left(k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p_{old}^2\right) + k\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, p_{old}^2\right)\right).$$

If $m \in \Omega(\bar{m}p_{old}^2 \log(p_{old}))$ and $w \in \Omega(\bar{w}p_{old}^2 \log(p_{old}))$, then we get an expected runtime of:

$$\mathcal{O}\left(k\frac{w+m}{p_{old}^2}\right).$$

Proof. We perform the redistribution phase after a map phase and each process partitioned their key-value pairs into p_{old} messages. Hence, after the map phase all intermediate key-values pairs are distributed between p_{old}^2 messages. If the hash function h is a truly random mapping, then the pairs are distributed uniformly at random according to their keys (Section 6.1.1). Each call of this function processes at most $\bar{w}$ pairs. According to Lemma 3.1 the worst case for static load balancing occurs if the workload is skewed and we have $\lceil w/\bar{w} \rceil$ different keys with $\bar{w}$ values each. Therefore the maximum expected number of key-value pairs per message and processed by each process in Line 3 is $\bar{w}b\left(\lceil w/\bar{w} \rceil, p_{old}^2\right)$. By applying a similar reasoning, the maximum expected number of machine words per process is $\bar{m}b\left(\lceil m/\bar{m} \rceil, p_{old}^2\right)$.
Processes which have saved no self-message of a failed node have k messages S. The remaining processes have $2k$ input messages. We can determine the buckets in Line 3 with a linear scan by deserializing keys and applying the hash function. Therefore, Lines 1 to 5 have an expected runtime of $\mathcal{O}\left(k\bar{w}b\left(\lceil w/\bar{w} \rceil, p_{old}^2\right)\right)$. Copying the buckets into the new messages in Line 10 has an expected runtime of $\mathcal{O}\left(\bar{m}b\left(\lceil m/\bar{m} \rceil, p_{old}^2\right)\right)$. Merging messages in Line 14 requires to add each key-values pair once into a hash table and then copying the grouped pairs into the send message. This results in an expected runtime of $\mathcal{O}\left(\bar{m}b\left(\lceil m/\bar{m} \rceil, p_{old}^2\right) + \bar{w}b\left(\lceil w/\bar{w} \rceil, p_{old}^2\right)\right)$. Finally, we need to gather all messages into an `MPI_All_to_allv` send buffer, which requires expected $\mathcal{O}\left(k\bar{m}b\left(\lceil m/\bar{m} \rceil, p_{old}^2\right)\right)$ time. The expected overall runtime of Algorithm 19 is

$$\mathcal{O}\Big(\underbrace{k\,\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, p_{old}^2\right)}_{\text{Lines (3),(10),(14)}} + \underbrace{k\,\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p_{old}^2\right)}_{\substack{\text{Lines (10),(14)}\\ \text{construct send buffer}}}\Big).$$

If $m \in \Omega(\bar{m}p_{old}^2 \log(p_{old}))$, $w \in \Omega(\bar{w}p_{old}^2 \log(p_{old}))$ and by using Lemma 2.2 we get a runtime of

$$\mathcal{O}\left(k\frac{w+m}{p_{old}^2}\right).$$

□

Lemma 6.4. *The merge buffer Algorithm 20 performed after a node failure has an expected runtime of:*

$$\mathcal{O}\left(\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, p_{new}\right) + \bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, tp_{new}\right)\right).$$

If $w \in \Omega(\bar{w}p_{new}\log(p_{new}))$ *and* $m \in \Omega(\bar{m}p_{new}\log(p_{new}))$*, then we get an expected runtime of:*

$$\mathcal{O}\left(\frac{w+m}{p_{new}}\right).$$

Proof. During the map phase, the input data have been divided into p_{old}^2 messages, p_{old} messages for each process. During the redistribution Algorithm 19 the key-values pairs are redistributed uniformly at random between the remaining processes. After redistribution and before the merge Algorithm 20 we have no data exchange. Furthermore, the pairs in SB^j have been produced during a map phase initialized with a uniformly and randomly distributed input. Hence, all key-values pairs $\bigcup_j SB^j \cup RSB^j$ have been distributed uniformly at randomly between p_{new} process. The user-defined reduce function consumes the intermediate key-value pairs. Each call of this function processes at most $\bar{w}$ pairs. According to Lemma 3.1 the worst case for static load balancing occurs if the workload is skewed and we have $\lceil w/\bar{w}\rceil$ different keys with $\bar{w}$ values each. Therefore, the maximum expected number of key-value pairs per process is $\bar{w}b\left(\lceil w/\bar{w}\rceil, p_{new}\right)$. By applying a similar reasoning, the maximum expected number of machine words per process is $\bar{m}b\left(\lceil m/\bar{m}\rceil, p_{new}\right)$.
Lines 2 to 8 result in a linear scan over the pairs in SB^j and RSB^j. In Line 4 we need to deserialize the keys, apply the hash function and use a hash table for aggregation. This results in an expected runtime of $\mathcal{O}\left(\bar{w}b\left(\lceil w/\bar{w}\rceil, p_{new}\right)\right)$. In Line 6, we copy the pairs into their new messages, which requires an expected runtime of $\mathcal{O}\left(\bar{m}b\left(\lceil m/\bar{m}\rceil, p_{new}\right)\right)$. Finally, we have to construct an `MPI_All_to_allv` send buffer, which requires expected $\mathcal{O}\left(k\bar{m}b\left(\lceil m/\bar{m}\rceil, p_{old}^2\right)\right)$ time. The expected overall runtime of Algorithm 19 is

$$\mathcal{O}\Bigg(\underbrace{\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, p_{new}\right)}_{\text{Line (4)}} + \underbrace{\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, p_{new}\right)}_{\substack{\text{Line (6)}\\ \text{construct send buffer}}}\Bigg).$$

If $m \in \Omega(\bar{m}p_{new}\log(p_{new}))$, $w \in \Omega(\bar{w}p_{new}^2\log(p_{new}))$ and by using Lemma 2.2 we get a runtime in

$$\mathcal{O}\left(\frac{w+m}{p_{new}}\right).$$

□

7. Hybrid Parallelized MapReduce

In Section 6 we introduce a MapReduce algorithm, which applies on each MPI processes the user-defined map and reduce functions sequentially to the input. We want to design our library to run on high-performance computers consisting of multiple compute nodes including multiple processors. We engineer a MapReduce framework, which starts one MPI process per compute node or NUMA node and parallelizes locally on each node using OpenMP.
For the following sections we use *process* to indicate an MPI process, *node* to refer to a compute node and *thread* to refer OpenMP threads. Each node contains the same number of processes k, where each process consists of t parallel threads.
In Section 7.1 we introduce our hybrid parallel implementation of the map phase. We propose a hybrid parallel algorithm for the reduce phase in Section 7.2. In Section 7.3 we provide the necessary algorithm to ensure the single node fault-tolerance.

7.1. Shared Memory Parallelized Map

7.1.1. Algorithm

We describe a shared memory parallelization of the map phase taking the NUMA architecture into account. The following algorithm implements Line 2 during a MapReduce operation

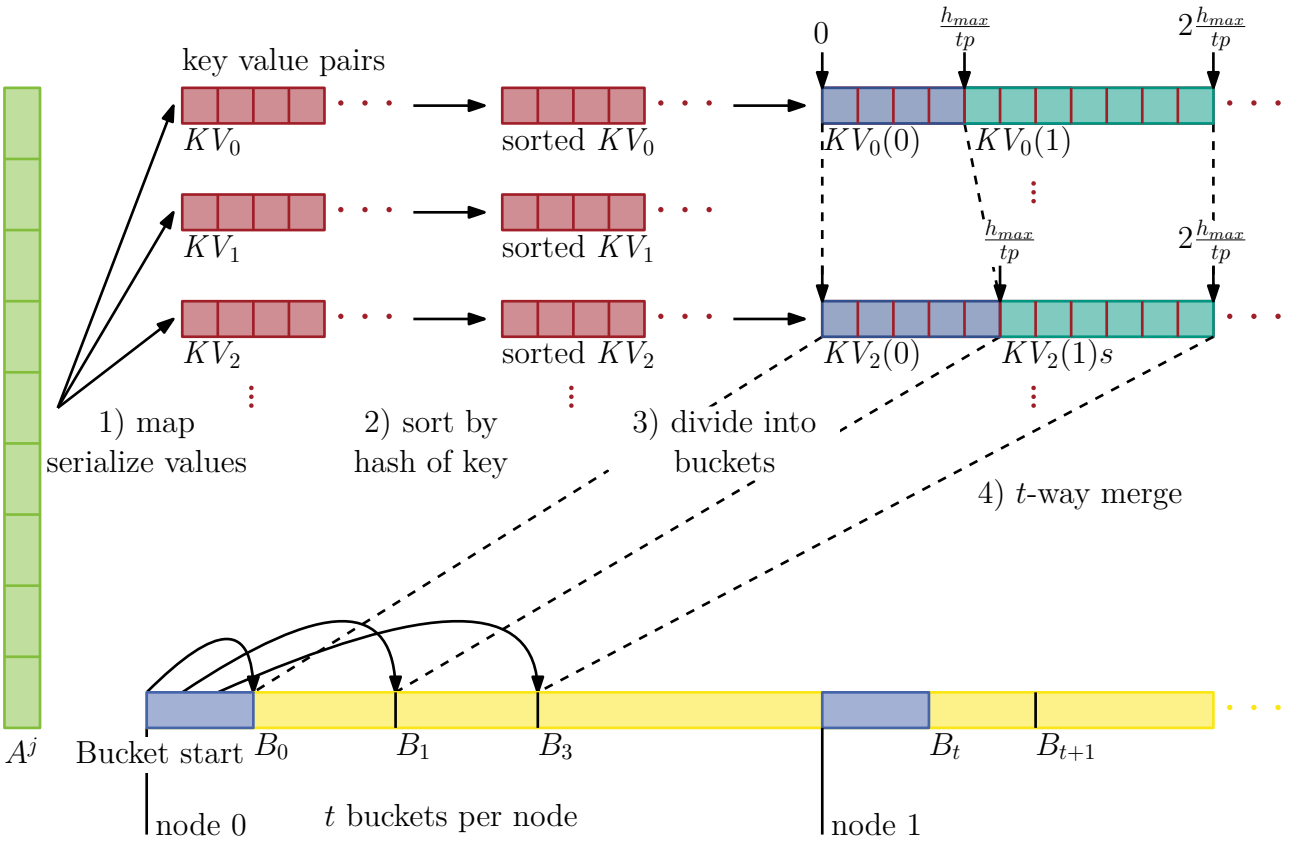


Figure 17: Illustration of the parallel shared memory implementation of the map phase. We follow the following steps: (1) First, we parallelize the application of the user-defined map function and save the pair in t arrays. (2) Each thread sorts the pairs of one array according to their hashed key. (3) Each thread divides its array into $p \cdot t$ buckets according to their hashed key. (4) We parallelize over the buckets and merge them into the send buffer.

(Algorithm 12). The map phase parallelizes the application of the user-defined map function on each MPI process using a shared memory model. We follow the general idea of the map phase introduced in Section 6.1. First, the map phase applies the user-defined map function to the input elements. Then we group the key-value pairs by their key. The algorithm serialized the keys and their values and saves the results in an `MPI_All_to_allv` send buffer for the shuffle phase. We illustrate the map phase in Figure 17 and Algorithm 21.
We use aggregation with sorting to avoid the cache efficiency problems as described in Section 3.4. To sort the key-value pairs we use a combination of the in-place cache efficient radix sort IPS^2Ra (Section 3.5) and multi-way merge.
Let MR$= (\mu, \rho, s, h)$ be a MapReduce operation with user-defined map function μ, serialization function s, and hash function h (Section 2.2). The MapReduce operation runs on an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ (Section 2.1). We parallelize the map phase on process $j \in P$ using t parallel threads, one for each core $\gamma(j)$. Algorithm 21 consists of the following steps on process j:

0 The input is a multi-set $A^j \subseteq I$ of input elements and the MapReduce operation tuple MR$= (\mu, \rho, s, h)$.

1 First, we initialize a list of key-serialized value pairs KV_x and a hash-index list HI_x. We distribute the input elements dynamically between the different threads. Each thread x applies the user-defined map function μ, serializes the value, and saves the key-serialized value pairs $(k, s(v))$ in KV_x. We associate $(k, s(v))$ with a hash-index pair $(h(k, z), i)$, which we save in HI_x. The index i indicates the position of $(k, s(v))$ in KV_x. We compute the hash value of the key $h(k, z)$ with the same seed on each process. (Lines 1-8)

2 On each thread x we sort the hash-index pairs HI_x by their hash value in ascending order. By using the sorted hash-index pairs HI_x, we can sort the list of key-serialized value pairs KV_x. (Lines 10)

3 For each thread x we partition the list of key-serialized value pairs KV_x according to their hashed key into $t \cdot p$ buckets. We divide the range of hash values $[0, h_{max})$ into $t \cdot p$. The bucket $KV_x(i)$ contains the pairs with hashed key $h(k, z) \in [i \cdot \frac{h_{max}}{t \cdot p}, (i+1) \cdot \frac{h_{max}}{t \cdot p})$, where h_{max} is the maximum possible hash value plus one. For a given hash value h we can determine its bucket $i = \lfloor \frac{t \cdot p \cdot h}{h_{max}} \rfloor$. (Line 11)

4 OpenMP distributes the buckets dynamically between the different threads using a parallel for-loop. For each bucket i we gather all buckets $KV_x(i)$ from the different threads x. We perform a t-way merge to gather for each key k all its serialized values. We iterate over the different buckets $KV_x(i)$ and process the keys in ascending order of their hash value. We save the result of the t-way merge of bucket x in B_x.
First, we determine the minimal hash value h_{min} and all corresponding key-serialized value pairs $(k, s(v))$ with $h(k, z) = h_{min}$. These pairs can have different keys with the same hash value. Therefore, we add these pairs into a hash table, with k as key and $s(v)$ as value. We save the serialized key $s(k)$, the serialized values and the serialization size of values into a send buffer B_x corresponding to bucket x. (Lines 13-25)

5 Finally, the algorithm merges the serialized buckets $B = \{B_0, ..., B_i, ..., B_{n \cdot t-1}\}$ into an MPI send buffer for an execution of `MPI_All_to_allv`. For each MPI process i we construct a send message containing the serialized buckets j, where $\left\lfloor \frac{j}{t} \right\rfloor = i$. This means we send the keys with hash values in $[i \cdot \frac{h_{max}}{p}, (i+1) \cdot \frac{h_{max}}{p})$ to MPI process i during the shuffling phase. (Lines 26-30)

During the entire map phase, we do not need the values of the key-value pairs. Therefore, we do not need to save the value but can save the serialized value instead. We need the keys in Step 4 (Line 18) to group the key-serialized value pairs by their key. We use hash tables to

Algorithm 21: Shared memory parallelized Map executed on process j

```
Input: A^j ⊆ I, MR = (μ, ρ, s, h)  // multiset of input elements A^j, MapReduce operation MR
1  KV = {KV_0, ..., KV_x, ..., KV_{t-1}}   // initialize an empty list of key-value pairs for each thread
2  HI = {HI_0, ..., HI_x, ..., HI_{t-1}}   // initialize an empty list of hash-index pairs for each thread
3  parallel_foreach e ∈ A^j do   // iterate parallel over the input data and execute on thread x
4  |  foreach (k, v) ∈ μ(e) do                          // apply the user-defined map function
5  |  |  HI_x += (h(k, z), |KV_x|)        // save the hash of k and the index of (k, s(v)) in KV_x
6  |  |  KV_x += (k, s(v))                              // save the key k and the seralized value
7  |  end
8  end
9  parallel                                             // execute in parallel for each thread x
10 |  sort HI_x by the first pair element
   |  // Divide the pairs in KV_x into t · n buckets according to the hashed key by using HI_x
11 |  KV'_x = {KV_x(i) | 0 ≤ i < t · p}, where KV_x(i) = {KV[y] | (h, y) ∈ HI_x ∧ ⌊(t*p*h)/h_max⌋ = i}
12 end
13 B = {B_0, ..., B_l, ..., B_{p·t-1}}   // initialize p · t empty buffers of serialized key-values pairs
   // iterate over all buckets in parallel and execute on thread x
14 parallel_for i ∈ {0, ..., p · t − 1} do
15 |  KV(i) = ⋃_{x∈{0,...,t-1}} KV_x(i)   // gather all buckets produced by the different threads
16 |  while KV(i) conatins pairs do
17 |  |  h_min = min_{x∈{0,...,t-1}} min_{(k,·)∈KV_x(i)} h(k, z)   // determine the minimum hash value
   |  |  // determine the key-value pairs with minimal hashed key and group them by key
18 |  |  C = {(k, X)|k ∈ K ∧ h(k, z) = h_min ∧ X = {s(v)|(k, s(v)) ∈ KV(i)}}
19 |  |  foreach (k, X) ∈ C do
20 |  |  |  B_i += |X| + s(k)                  // add the size of values and the serialized key
21 |  |  |  foreach s ∈ X do  B_i += s                    // add the serialized values
22 |  |  end
23 |  |  KV(i) = KV(i) \ {(k, s)|(k, X) ∈ C ∧ s ∈ X}   // remove the processed pairs
24 |  end
25 end
26 SB^j = {SB^{j,1}, ..., SB^{j,l}, ..., SB^{j,p}}   // initialize p empty send messages
27 parallel_for i ∈ {0, ..., p · t − 1} do   // iterate over all buckets in parallel
28 |  l = ⌊i/t⌋   // we send buckets y · t to ((y + 1) · t − 1) to node y
29 |  SB^{j,l} = SB^{j,l} ∪ B_l   // add the serialized bucket l to the send message
30 end
31 return SB^j                                           // return a message for each node
```

aggregate the serialized values, which require hashed keys. Furthermore, we use a seed z for the hash function h to determine the process of a key-value pair for the reduce phase. Since all keys we add to the hash table have the same hash value with seed z, we need to choose a different seed. Otherwise, the hash table accesses would cause hash collisions.
In Step 2 we use a sequential sorting algorithm to sort the pairs in HI_x according to their first value. This approach has the disadvantage that we cannot perform load-balancing between the different threads during sorting. Since each thread has mapped an equal number of elements, the different HI_x have similar sizes. An alternative approach is to save all hash-index pairs into a single list and use a parallel sorting algorithm. This would require to save all HI_x into a single list. This requires additional memory. Furthermore, the NUMA architecture of compute nodes

makes copying elements between threads slow.
We sort hash-index pairs HI_x instead of key-serialized value pairs KV_x on each thread x. The key and value types are user-defined and can be large. Sorting large elements can lead to longer sorting times if the algorithm needs to copy or swap large data types. Thus, we do not sort by key, but by the hashed key. Sorting the pairs in KV_x directly requires to evaluate the hash function multiple times. Therefore, for each key-serialized value pair $(k, s(v)) \in KV_x$ we compute $h(k, z)$ only once and save $(h(k, z), i)$ in HI_x. After sorting, we can use the index i of $(k, s(v))$ in KV_x to sort KV_x.
During the application of the user-defined map function, we distribute the input data equally between the different threads. This should lead to similar execution times for each thread in Steps 1-3.
Note that we sort and aggregate the key-value pairs during the map phase before shuffling through the network. The alternative is to serialize the pairs without aggregation. Our approach has two advantages. First, we have to serialize each key only once per process (Line 20). Secondly, serialized keys can be large, for instance during a word count algorithm (Section 4.1). By aggregating before shuffling, we have to send each key only once from a process. By sorting during the map phase, the reduce phase amounts to a multi-way merge to finish the aggregation (Section 7.2).
From Line 14 to 30 we perform a multi-way merge algorithm and save the serialized buckets B into send buffers SB^j. Since we use `MPI_All_to_allv` calls during the shuffle phase, we have to save the send messages for the different processes successively into a single buffer (Figure 17 Step 4). To minimize memory usage and copy operations we store the serialized key-values directly into the MPI send buffer. To determine the start index of each bucket B_i, we require their sizes. We perform the multi-way merge from Line 14 to 25 twice. During the first merge, we determine the size of each bucket, by serializing the key and using the already serialized values. Then we determine the start index of each bucket in the MPI send buffer by computing a prefix sum over the bucket sizes. Finally, we merge the serialized key-value pairs into the send buffer.
A message $SB^{j,l}$ sent from process j to process i contains the t serialized buckets $\{B_{t \cdot i}, ..., B_{t \cdot (i+1)-1}\}$. The message starts with $t + 1$ indexes indicating the start and end index of each bucket. At position x we save the start index of bucket x. Position $x + 1$ contains the end index of bucket x and start index of bucket $x + 1$. Following the bucket start indexes, we save the serialized buckets consecutively.

7.1.2. Complexity

We adopted the general structure of the BSP MapReduce algorithm introduced in [86]. We achieve the same expected running time (Lemma 7.1). We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4.

Lemma 7.1. *The input multi-set A is distributed randomly between p processes. Let t be the number of threads per process. We assume the hash function h is a truly random mapping. Then the map phase (Algorithm 21) can be implemented with expected running time:*

$$\mathcal{O}\left(\bar{w} \cdot \log(t) \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right) + \bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, tp\right) + p\right).$$

If $w \in \Omega(\bar{w} \log(t) tp \log(tp))$ and $m \in \Omega(\bar{m} tp \log(tp))$, then we get an average complexity of:

$$\mathcal{O}\left(\frac{w + m}{tp} + p\right).$$

Proof. We follow the proof of Theorem 4.1 in [86]. We assume that the input elements $A \subseteq I$ are distributed randomly between the different processes. At each process j we use a parallel for-loop to distribute the local input elements $A^j \subseteq A$ between the different threads. Therefore, we can assume that the input elements are distributed randomly between all tp threads.
The running time of applying the user-defined map function (Line 4), evaluating the hash function (Line 5), serializing values (Line 6), and serializing keys (Line 20) is worst if the work is as skewed as possible (Lemma 3.1). Therefore, we assume the execution time is zero for all input elements except for $\lceil w/\bar{w} \rceil$ elements, which require $\bar{w}$ time. Assigning $\lceil w/\bar{w} \rceil$ elements randomly between pt threads corresponds to the balls in bins problem [78], where $\lceil w/\bar{w} \rceil$ is the number of balls and tp the number of bins. The expected maximum number of elements per thread is $b(\lceil w/\bar{w} \rceil, tp)$. Since each of these elements has a running time of $\bar{w}$, the expected running time of Lines 3 to 8 is $\mathcal{O}(\bar{w} \cdot b(\lceil w/\bar{w} \rceil, tp))$.
The user-defined map function produces the intermediate key-value pairs. Therefore, each map function can produce at most $\bar{w}$ keys. In total there are at most w keys, which are distributed between tp threads. Since the input elements are distributed randomly between all threads, the key-value pairs are produced randomly.(Line 4). As before we can apply Lemma 3.1. Hence, the expected maximum number of key-value pairs on each thread is $b(\lceil w/\bar{w} \rceil, tp)$. In Line 10 we sort the pairs according to the hashed key-value. We represent the hash values using 64Bits. By using a radix sort algorithm [17, 44] we can sort in expected $\mathcal{O}(64 \cdot b(\lceil w/\bar{w} \rceil, tp))$ time. Dividing the pairs into equally sized blocks (Line 11) requires a linear scan and therefore runs in expected $\mathcal{O}(b(\lceil w/\bar{w} \rceil, tp))$ time.
On each thread we have divided the key-value pairs into tp buckets. Over all processes we have tp^2 buckets. We divide the pairs by dividing the hashed key into $\lfloor h_{max}/t/p \rfloor$ sized partitions, where h_{max} is the maximum hash value. On each process we assign p buckets to each thread. Since we assume that the hash function is truly random, each thread processes expected $\mathcal{O}(b(\lceil w/\bar{w} \rceil, tp))$ pairs (Lines 14-25).
We perform a multi-way merge over buckets $KV_x(i)$ for $i \in \{0, ..., t-1\}$, which has a complexity of $\mathcal{O}(\log(t) \cdot b(\lceil w/\bar{w} \rceil, tp))$ on each thread. We group the pairs in Line 18 by using a hash table, which requires linear time. The worst case for saving the serialized pairs in a bucket (Line 19-22) occurs if the serialized data m is as skewed as possible (Lemma 3.1,[86]). This means if we have $k = \lceil m/\bar{m} \rceil$ elements with a size of $\bar{m}$ (Lemma 1 [85]). The expected maximum serialized data per thread is $b(\lceil m/\bar{m} \rceil, tp)$. All in all Lines 14 to 25 have an expected running time of $\mathcal{O}(\log(t) \cdot b(\lceil w/\bar{w} \rceil, tp) + \bar{m} \cdot b(\lceil m/\bar{m} \rceil, tp))$.
We save the buckets B into a single send buffer SB^j used by an `MPI_All_to_allv` command. We can use a parallel prefix sum to compute the start and end index of each bucket B_x in SB^j. A parallel prefix sum over tp elements running on t parallel threads requires $\mathcal{O}(p + \log(t))$ operations [88]. By using the same reasoning as before, the worst case for saving the buckets B_x in SB^j occurs if the expected maximum number of machine words handled by each thread is $\bar{m} \cdot b(\lceil m/\bar{m} \rceil, tp)$. Since we have the positions of each bucket in SB^j, copying results in a linear scan, which has an expected running time of $\mathcal{O}(\bar{m} \cdot b(\lceil m/\bar{m} \rceil, tp))$.
All in all Algorithm 21 has a complexity of:

$$\mathcal{O}\Bigg(\underbrace{\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right)}_{\text{Lines } 3-13} + \underbrace{\bar{w} \cdot \log(t) \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right)}_{\text{Lines } 14-25} + \underbrace{\bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, tp\right)}_{\text{Lines } 26-30} + \underbrace{p}_{\text{prefix sum}} \Bigg).$$

With Lemma 2.2, $w \in \Omega(\bar{w} \log(t) tp \log(tp))$, and $m \in \Omega(\bar{m} tp \log(tp))$ we get the expected time

$$\mathcal{O}\left(\frac{w+m}{tp} + p \right).$$

□

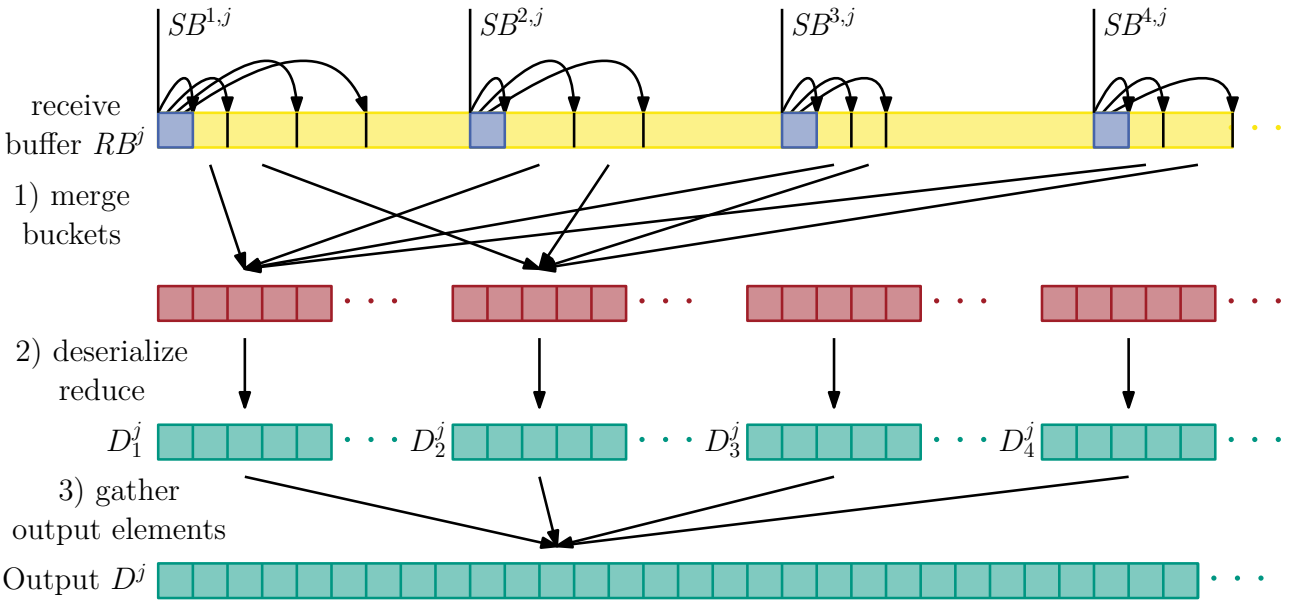


Figure 18: Illustration of the parallel shared memory implementation of the reduce phase. (1) First, each thread x gets the bucket x of each received message and merges the key-values pairs according to their keys. (2) Each thread deserialized its values and applies the user-defined reduce function. (3) We gather all output elements into one large array.

7.2. Shared Memory Parallelized Reduce

7.2.1. Algorithm

The following shared memory algorithm parallelizes the application of the user-defined reduce function on each MPI process. The general idea is to use the buckets constructed during the map phase described in Section 7.1. Each thread deserializes and merges the buckets received from the different MPI processes during the shuffle phase. After aggregation, we apply the user-defined reduce function.

Let $\text{MR} = (\mu, \rho, s, h)$ be a MapReduce operation with user-defined reduce function ρ, serialization function s, and hash function h (Section 2.2). We execute the MapReduce operation on p parallel processes. We use a shared memory algorithm to parallelize the reduce phase. We assign t threads for each process $j \in \{1, ..., p\}$. We illustrate this Algorithm 22 in Figure 18. The algorithm consists of the following steps corresponding to the numbering in Figure 18:

0 The input of the reduce phase at process j are the received messages from the shuffle phase RB^j and the MapReduce operation $\text{MR} = (\mu, \rho, s, h)$. The previous map phase (Section 7.1) at process l has produced the message $SB^{l,j} \in RB^j$. This message contains the serialized key-values pairs sorted in ascended order according to their hashed key value. (Line 1)

1 The map phase has divided each message $M \in RB^j$ into t buckets. At process j the bucket $x \in \{1, ..., t\}$ in M contains the key-values pairs (k, X) with:

$$h(k,z) \in \left[(x-1)\left\lfloor\frac{h_{max}}{t \cdot p}\right\rfloor + (j-1)\left\lfloor\frac{h_{max}}{p}\right\rfloor, x\left\lfloor\frac{h_{max}}{t \cdot p}\right\rfloor + (j-1)\left\lfloor\frac{h_{max}}{p}\right\rfloor\right)$$

The seed z is the same as used during the map phase. We assign a thread to a bucket. Each thread x iterates in parallel over the received messages $M \in RB^j$ and deserialize the key-values pairs in bucket x. We save the resulting from process l in C_x^l. (Line 6)

2 Each thread x merges the deserialize buckets $C_x = \{C_x^1, ..., C_x^l, ..., C_x^p\}$. We perform a multi-way merge by scanning over the p buckets simultaneously. We gather the key-values pairs with the same minimal hashed key value h_{min}. We group these pairs according to their key. We apply the user-defined reduce function ρ and save the results in a thread-local output set D_x^j. We remove the key-values pairs (k, X) with $h(k, x) = h_{min}$ and repeat Step 2 until the buckets C_x are all empty. (Lines 8-15)

3 Finally, we gather the thread-local output elements D_x^j into a single set D^j (Line 17).

During the map phase, we save the start and end index of a bucket x in message $SB^{l,j}$ send from process j to l. The map phase append these start indexes at the start of $SB^{l,j}$. This allows us to determine the bucket boundaries in constant time (Step 1, Line 6).

Note that each thread works exactly on one bucket, which can lead to imbalanced execution times. We have constructed these buckets by dividing the key-value pairs by their key hash value. If our hash function maps the keys equally distributed and there are no imbalances in the key-value pairs, then each thread should get an equal workload. Note that this does not work if there are some keys with not evenly distributed values, for instance if there is one key with more values than the rest.

To lower the memory usage we combine the deserialization and merging in Step 1. We perform a multi-way merge algorithm over the serialized buckets x situated in messages $RB^j = \{SB^{1,j}, ..., SB^{l,j}, ..., SB^{p,j}\}$. Thread x scans over the buckets x in the received messages simultaneously. We deserialize the keys and values with minimal hashed key value h_{min} (Line 9). We group the values with the same key (Line 10). For this aggregation, we use a hash table. We add the key-values pairs with the same hashed key in the table. This allows us to gather all values of a given key. If we choose a sufficiently large hash value space, then hash collisions are improbable. Therefore, the number of keys with the same hash values should be

Algorithm 22: Reduce: shared memory

```
Input: RB^j, MR = (μ, ρ, s, h)   // output from shuffle phase RB^j, MapReduce operation MR
1  RB^j = {SB^{1,j}, ..., SB^{l,j}, ..., SB^{p,j}}
2  parallel_foreach x ∈ {1, ..., t} do                    // process in parallel on each thread x
3      C_x = {C_x^1, ..., C_x^l, ..., C_x^p}   // initialize a set of key-values pairs for each receive message
4      D_x^j = ∅                                  // initialize an empty output set for each thread
5      foreach l ∈ {1, ..., p} do                          // iterate over received messages RB^j
           // deserialize the key-values pairs in bucket x and message SB^{l,j}
6          C_x^l = {(k, X) | k ∈ K, X ⊆ V, (s(k), s(X)) ∈ SB^{l,j} ∧ (x − 1) · h_max/(t·p) ≤ h(k, z) < x · h_max/(t·p)}
7      end
8      while ∃∅ ≠ E ∈ C_x do
9          h_min = min_{E∈C_x} min_{(k,·)∈E} h(k, z)                  // determine the minimum hash value
           // determine the key-values pairs with minimal hashed key and group them by key
10         C = {(k, X) | k ∈ K ∧ h(k, z) = h_min ∧ X = ⋃_{E∈C_x} {X | (k, X) ∈ E}}
11         foreach (k, X) ∈ C do
12             D_x^j = D_x^j ∪ ρ(k, X)                          // apply the user-defined reduce function
13         end
14         C_x = {E \ C | E ∈ C_x}                              // remove the processed key-values pairs
15     end
16 end
17 D^j = {D_1^j, ..., D_x^j, ..., D_p^j}                        // gather the locally reduced output
18 return D^j
```

small. This results in small hash tables, which can fit into the cache. We use another seed for the hash function than during the map phase. Using the same hash function would ensure hash conflicts during the group by key in Line 10.
Finally, in Line 17 and Step 3 we use a prefix sum and parallel copy to gather all local output elements D_x^j into a single list D^j. We compute the prefix sum over the local thread output list sizes $|D_x^j|$. This provides us with a start and end index for each local list in the final list, as well as the size of the final list. Each thread x can then copy their elements D_x^j in parallel into the final list D^j.

7.2.2. Complexity

We analyze the complexity of Algorithm 22 executed after the shuffle phase and Algorithm 21. We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4.

Lemma 7.2. *Let the serialized key-value pairs be distributed uniformly at random between tp buckets. We can implement the reduce phase (Algorithm 22) with expected running time of:*

$$\mathcal{O}\left(\log(p) \cdot \bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right) + \bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, tp\right) + \log(t)\right).$$

If $w \in \Omega(\bar{w} tp \log(tp))$ and $m \in \Omega(\bar{m} tp \log(tp))$, then the algorithm has a time complexity of:

$$\mathcal{O}\left(\log(p) \cdot \frac{w+m}{tp} + \log(t)\right).$$

Proof. We follow the proof of Theorem 4.1 in [86]. The map (Algorithm 21) and shuffle phase distribute the workload uniformly at random between tp buckets for the reduce phase. Each thread processes exactly one bucket in Lines 2 to 16. According to Lemma 3.1 the worst case for random static load balancing occurs if $\lceil w/\bar{w} \rceil$ subproblems have a size of $\bar{w}$, while the remaining sizes are 0. The expected maximum number of subproblems per thread is $\bar{w} \cdot b(\lceil w/\bar{w} \rceil, tp)$.
In Lines 5 to 7 each thread deserializes the key-value pairs of its corresponding bucket. In Lines 11 to 13 we apply the user-defined reduce functions. This results in an expected running time of $\mathcal{O}(\bar{w} \cdot b(\lceil w/\bar{w} \rceil, tp))$.
Each thread has to merge its bucket from p messages, each provided by a different process. We perform a multi-way merge in Lines 8 to 15. The key-value pairs correspond to an application of the user-defined reduce function. Each reduce function can process at most w_i pairs, where $w_i \leq \bar{w}$ is its runtime. Therefore, we have at most w pairs, where each reduce function consumes at most $\bar{w}$ pairs. By applying Lemma 3.1, the worst case expected maximum number of key-value pairs per thread is $\bar{w} \cdot b(\lceil w/\bar{w} \rceil)$. In Line 10 we group the key-value pair with the same hashed key h_{min} by using a hash table, which results in a linear execution time. Performing the $p-$way merge requires $\mathcal{O}(\log(p) \cdot \bar{w} \cdot b(\lceil w/\bar{w} \rceil, tp))$ time.
In Line 17 we save the output elements into a single array on each process j. We determine the start and end indexes of each thread-local output elements D_x^j by performing a parallel prefix sum over the sizes of D_x^j. This parallel prefix sum over t elements requires $\mathcal{O}(\log(t))$ time. Each thread can now copy its local output D_x^j into the global output D^j. We can apply Lemma 3.1 and assume the worst case with $\lceil m/\bar{m} \rceil$ output elements with a size of $\bar{m}$. Then the expected copy time lies in $\mathcal{O}(\bar{m} \cdot b(\lceil m/\bar{m} \rceil, tp))$. All in all, Algorithm 22 has an expected running time in

$$\mathcal{O}\Bigg(\underbrace{\bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right)}_{\text{Lines 5–7}} + \underbrace{\log(p) \cdot \bar{w} \cdot b\left(\left\lceil \frac{w}{\bar{w}} \right\rceil, tp\right)}_{\text{Lines 8–15}} + \underbrace{\bar{m} \cdot b\left(\left\lceil \frac{m}{\bar{m}} \right\rceil, tp\right) + \log(t)}_{\text{Line 17}} \Bigg).$$

By applying Lemma 2.2 with $w \in \Omega(\bar{w}tp\log(tp))$ and $m \in \Omega(\bar{m}tp\log(tp))$ we get an expected runtime of

$$\mathcal{O}\left(\log(p) \cdot \frac{w+m}{tp} + \log(t)\right).$$

□

7.3. Shared Memory Parallelized Fault-Tolerance

The following sections introduce the algorithms needed for the single node fault-tolerance mechanism in Section 5. We present a shared memory implementation of the Redistribute algorithm in Section 7.3.1. Section 7.3.2 presents the MergeBuffer algorithm. We analyze their runtime complexity in Section 7.3.3.

7.3.1. Redistribute Algorithm

To realize the single node fault-tolerance, we have to provide a shared memory implementation for the Redistribute operation used in Algorithm 15 (Section 5). The redistribution algorithm takes the messages send to a process on the failed node during the previous shuffle phase as input. This algorithm redistributes the key-value pairs contained in these messages uniformly at random between the remaining p_{new} processes. The output is a send buffer, which an `MPI_All_to_allv` operation exchanges between all processes. Let p_{old} be the number of processes before a node failure. During the failure $k = p_{old} - p_{new}$ MPI processes stop working. The number of parallel threads per process is t. Let MR$= (\mu, \rho, s, h)$ be the previous MapReduce operation with user-defined map function μ, serialization function s, and hash function h (Section 2.2). The MapReduce operation runs on an MPI execution (S, P, γ) with HPC system S, processes P and bindings γ, with $|P| = p_{new}$ (Section 2.1).

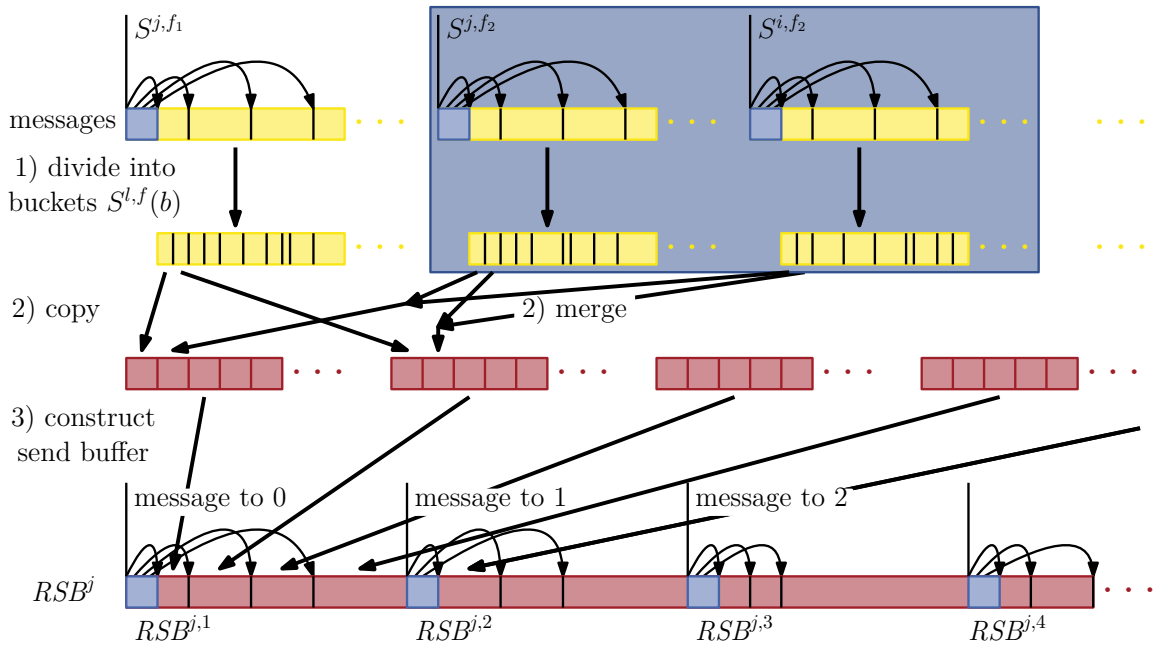


Figure 19: This figure illustrates the redistribution of the saved messages to reconstruct the lost data. (0) Gather all messages to processes on a failed node. (1) Divide each message into $p_{new} \cdot t$ buckets. (2) Copy and merge buckets together from all messages. (3) Construct an `MPI_All_to_allv` send buffer.

We parallelize the redistribution Algorithm 23 on process $j \in P$ using t parallel threads, one for each core $\gamma(j)$. Figure 19 illustrates the three main steps:

0 Let $F = \{f_1, ..., f_2\}$ be the list of processes on the failed node, with $|F| = k$. The input is a list of messages S send to the failed processes F. The map phase (Section 7.1) has constructed these messages. Each message $S^{l,f}$ contains t buckets, where bucket $x \in \{1, ..., t\}$ contains the key-value pairs with hashed key in $\left[\left\lfloor \frac{((l-1)t+x-1)h_{max}}{tp_{old}} \right\rfloor, \left\lfloor \frac{((l-1)t+x)h_{max}}{tp_{old}} \right\rfloor\right)$. The previous map phase has sorted the serialized pairs according to their hashed key. k different processes contain the self-messages exchanged during the shuffle phase on the failed node.

1 First, we divide each message $S^{l,f}$ into tp_{new} buckets. The algorithm iterates over all messages in S and each thread processes one of its t buckets in parallel. A thread x divides the old bucket $x \in \{0, ..., t-1\}$ into p_{new} sub-buckets $y \in \{1, ..., p_{new}\}$. The new bucket $S^{l,f}(b)$ in message $S^{l,f}$, with $b = xp_{new} + y$, contains the key-value pairs with hashed key in $\left[\left\lfloor \frac{((l-1)tp_{new}+b-1)h_{max}}{tp_{old}p_{new}} \right\rfloor, \left\lfloor \frac{((l-1)tp_{old}+b)h_{max}}{tp_{old}p_{new}} \right\rfloor\right)$. We perform a scan over the sorted bucket, deserialize their keys and determine the boundaries of the new buckets $S^{l,f}(b)$. (Lines 1 to 8)

2 For each saved message $S^{l,f}$ on process j, we distribute their tp_{new} buckets $S^{l,f}(b)$ between global buckets $B = \{B_1, ..., B_x, ..., B_{p_{new} \cdot t}\}$ in parallel. We parallelize over buckets $b \in \{1, ..., p_{new}t\}$. We distinguish between copying and merging buckets.

a) **copy**: Process j has not saved self-messages of the failed processes. Two distinct messages in S have no key-value pairs with the same hashed key in common.We only need to copy bucket $S^{l,f}(b)$ into the global bucket B_b (Lines 11 to 15)

b) **merging**: Process j has saved self-messages of the failed processes. For two distinct messages $S^{j,f}, S^{i,f} \in S$ send to the same failed process f, we have to merge their buckets $S^{j,f}(b)$ and $S^{i,f}(b)$. These two buckets contain key-value pairs with the same keys. We perform a 2-way merge of $S^{j,f}(b)$ and $S^{i,f}(b)$, which are sorted according to their hashed keys. We save the result in the global bucket B_b. (Lines 15 to 19)

3 Finally, we construct a message $RSB^{j,l}$ send from each process j to l. Each message contains t buckets of B. $RSB^{j,l}$ saves buckets $B_{(l-1)t+1}$ to B_{lt}. We parallelize the send message construction over the buckets in B. (Lines 22 to 25)

In Algorithm 23 in Line 5 we determine the start and end index of each new bucket $S^{l,f}(b)$ and save them for later use. For each key-values pair (k, X) we have saved their serialization size $|(k, X)|$ followed by the serialized key $s(k)$ and the serialized values $v \in X$. By using the serialization size and the deserialized key k we can determine their bucket

$$b = \left\lfloor \frac{h(k,z) - (l-1)\frac{h_{max}}{p_{old}}}{\frac{h_{max}}{p_{new} \cdot t}} \right\rfloor .$$

We compute this partitioning with a linear scan over the message. We parallelize over the t buckets in $S^{l,f}$ produced during the map phase (Section 7.1). Since we have saved the start and end index of each bucket in the message during the map phase, each thread can determine their bucket it constant time.

In Step 2b we parallelize over the buckets determined in Step 1. If process j has saved self-messages of the failed node, then we need to merge two buckets $S^{j,f}(b)$ and $S^{i,f}(b)$. We perform a linear scan over both buckets, which are sorted according to their hashed key. Then we determine all key-values pairs with the same hash value and use a hash table to gather all values with the same key. Note that we have to use a different hash function as during the map phase. We save the serialized keys followed by their values in $S^{j,f}(b)$ and $S^{i,f}(b)$ into the

Algorithm 23: Shared memory parallelized Redistribute on process j

// saved messages S send to the failed processes $\{f_1, ..., f_2\}$, new and old number of processes p_{old}, p_{new} and MapReduce operation MR

Input: $S = \{S^{j,f_1}, ..., S^{j,f_2}, S^{i,f_1}, ..., S^{i,f_2}\}, p_{old}, p_{new} \in \mathbb{N}, \text{MR} = (\mu, \rho, s, h)$

1 `foreach` $S^{l,f} \in S$ `do` *// divide each saved message into $p_{new} \cdot t$ buckets*

2 `parallel_foreach` $x \in \{0, ..., t-1\}$ `do` *// each thread x divides one of the old buckets*

3 `foreach` $y \in \{1, ..., p_{new}\}$ `do` *// divide bucket x into p_{new} new buckets*

4 $b = x \cdot p_{new} + y$ *// determine the new bucket b in message $S^{l,f}$*

5 $S^{l,f}(b) = \left\{(s(k), s(X)) \in S^{l,f} | k \in K, X \subseteq V \wedge \left\lfloor \frac{h(k,z) - (l-1)\frac{h_{max}}{p_{old}}}{\frac{h_{max}}{p_{new} \cdot t}} \right\rfloor = b \right\}$

6 `end`

7 `end`

8 `end`

9 $B = \{B_1, ..., B_x, ..., B_{p_{new} \cdot t}\}$ *// initialize $p_{new} \cdot t$ new empty buckets*

10 `foreach` $f \in \{f_1, ..., f_2\}$ `do` *// iterate over the failed processes $\{f_1, ..., f_2\}$*

11 `if` $S^{i,f} \notin S$ `then` *// process j does not contain the self-message of a failed process*

12 `parallel_foreach` $x \in \{1, ..., p_{new} \cdot t\}$ `do` *// parallelize over the new buckets x*

13 $B_x = B_x \cup S^{j,f}(x)$ *// copy $S^{j,f}(x)$ into the new bucket B_x*

14 `end`

15 `else`

16 `parallel_foreach` $x \in \{1, ..., p_{new} \cdot t\}$ `do` *// parallelize over the new buckets x*

// merge buckets $S^{j,f}(x)$ and $S^{i,f}(x)$

17 $B_x = B_x \cup \left\{(s(k), X_1 \cup X_2) | (s(k), X_1) \in S^{j,f}(x) \wedge (s(k), X_2) \in S^{i,f}(x)\right\}$

18 `end`

19 `end`

20 `end`

21 $RSB^j = \{RSB^{j,1}, ..., RSB^{j,x}, ..., RSB^{j,p_{new}}\}$ *// initialize messages $RSB^{j,x}$ send from j to x*

22 `parallel_foreach` $B_x \in B$ `do` *// iterate over buckets B_x in parallel*

23 $l = \lfloor \frac{x}{p_{new}} \rfloor + 1$ *// determine the message l for bucket x*

24 $RSB^{j,l} = RSB^{j,l} \cup B_x$ *// copy B_x into message $RSB^{j,l}$*

25 `end`

26 `return` RSB^j

new bucket B_b. Finally, we send the messages with an `MPI_All_to_allv` command. We use a prefix sum algorithm to determine the position of each bucket B_x in the MPI send buffer RSB^j and parallelize over the buckets B_x to construct it.

7.3.2. MergeBuffer Algorithm

Algorithm 24 performs the merge buffer operation needed during the single node failure fault-tolerance mechanism (Section 5.2). The merge buffer algorithm operates on the send messages produced by the map phase and the messages produced during the recovery. We merge both buffers together and create tp_{new} buckets which we distribute between the p_{new} remaining processes. We can divide Algorithm 24 into the following steps:

0 At process j, the map phase has produced the p_{old} send messages SB^j. A message $SB^{j,l} \in SB^j$ send from process j to $l \in \{1, ..., p_{old}\}$ contains t buckets. Let h_{max} be the maximum possible hash value. The bucket $x \in \{1, ..., t\}$ in $SB^{j,l}$ contains key-values

pairs with sorted hashed key in $\left[\left\lfloor\frac{((l-1)t+x-1)h_{max}}{tp_{old}}\right\rfloor, \left\lfloor\frac{((l-1)t+x)h_{max}}{tp_{old}}\right\rfloor\right)$. The recovery phase has produced p_{new} messages $RSB^{j,l} \in RSB^j$ send from process j to $l \in \{1, ..., p_{new}\}$. Bucket $x \in \{1, ..., t\}$ in $RSB^{j,l}$ contains key-values pairs with sorted hashed key in $\left[\left\lfloor\frac{((l-1)t+x-1)h_{max}}{tp_{new}}\right\rfloor, \left\lfloor\frac{((l-1)t+x)h_{max}}{tp_{new}}\right\rfloor\right)$.

1 Algorithm 24 constructs tp_{new} new buckets, which the recovery has distributed equally between the remaining processes. On each process j, we parallelize over the new buckets $B_b \in B$ with t threads. We determine the key-values pairs $SB^j(b) \in SB^j$ and $RSB^j(b) \in RSB^j$ with sorted hashed key in $\left[b\left\lfloor\frac{h_{max}}{tp_{new}}\right\rfloor, (b+1)\left\lfloor\frac{h_{max}}{tp_{new}}\right\rfloor\right)$. (Lines 4 and 5)

2 We perform a 2−way merge on $SB^j(b)$ and $RSB^j(b)$ into a new bucket B_b. We group the values according to their key. The algorithm scans over $SB^j(b)$ and $RSB^j(b)$. We add the key-values pairs with the same hash value into a hash table. This allows us to aggregate the pairs according to their key. We save the serialization size of a merged key-values pairs followed by the serialized key and serialized values from $SB^j(b)$ and $RSB^j(b)$ into B_b. (Line 6)

3 Finally, we construct send messages $NewSB^{j,l}$ send from process j to l, each containing t buckets B_x. The message $NewSB^{j,l}$ contains the buckets $B_{(l-1)t}$ to B_{lt-1}. (Lines 8 to 12)

A map phase (Section 7.1) performed during recovery (Section 5.2.2) constructs the send messages RSB^j. The recovered key-value pairs are distributed uniformly at random between tp_{new} buckets. Therefore, we have saved and computed the bucket boundaries needed in Line 5 during the map phase. This is not the case for the messages contained in SB^j, which the previous map phase has partitioned into tp_{old} buckets. We have to determine the start index of $SB^j(b)$ in Line 4. The set $SB^j(b)$ contains the key-values with sorted hashed key in $\left[b\left\lfloor\frac{h_{max}}{tp_{new}}\right\rfloor, (b+1)\left\lfloor\frac{h_{max}}{tp_{new}}\right\rfloor\right)$. We determine the bucket x in SB^j containing the key-value pairs with hashed key $h = b\left\lfloor\frac{h_{max}}{tp_{new}}\right\rfloor$, with $x = \left\lfloor\frac{htp_{old}}{h_{max}}\right\rfloor$. We scan over bucket x in SB^j to determine the start index of $SB^j(b)$.
We do not need to save the buckets B explicitly before saving them into the `MPI_All_to_allv` send buffer. First We perform the merge in Line 6 without saving the merged key-values pairs

Algorithm 24: Shared memory parallelized MergeBuffer on process j

`Input`: $SB^j = \{SB^{j,1}, ..., SB^{j,p_{old}}\}, RSB^j = \{RSB^{j,1}, ..., RSB^{j,p_{new}}\}, \text{MR} = (\mu, \rho, s, h)$

1 $B = \{B_0, ..., B_b, ..., B_{p_{new} \cdot t-1}\}$ *// initialize $p_{new} \cdot t$ new empty buckets*
2 `parallel_foreach` $b \in \{0, ..., p_{new} \cdot t - 1\}$ `do`
3 $x = \lfloor \frac{b}{t} \rfloor$, $y = b \mod t$ *// determine the new process x and local bucket y of bucket b*
// gather the key-values pairs in SB^j and RSB^j corresponding to the new bucket b
4 $SB^j(b) = \left\{(s(k), s(X)) \in M \middle| M \in SB^j, k \in K, X \subseteq V, \left\lfloor\frac{p_{new} \cdot t \cdot h(k,z)}{h_{max}}\right\rfloor = b\right\}$
5 $RSB^j(b) = \left\{(s(k), s(X)) \in M \middle| M \in RSB^j, k \in K, X \subseteq V, \left\lfloor\frac{p_{new} \cdot t \cdot h(k,z)}{h_{max}}\right\rfloor = b\right\}$
6 $B_b = \left\{(s(k), X_1 \cup X_2) \middle| (s(k), X_1) \in SB^j(b) \wedge (s(k), X_1) \in RSB^j(b)\right\}$ *// merge buckets*
7 `end`
8 $NewSB^j = \left\{NewSB^{j,1}, ..., NewSB^{j,l}, ..., NewSB^{j,p_{new}}\right\}$ *// initialize messages from process j to l*
9 `parallel_foreach` $B_b \in B$ `do` *// iterate over buckets B_b in parallel*
10 $l = \lfloor \frac{b}{p_{new}} \rfloor$ *// determine the message l for bucket x*
11 $NewSB^{j,l} = NewSB^{j,l} \cup B_b$ *// copy B_b into message $NewSB^{j,l}$*
12 `end`
13 `return` $NewSB^j$

$(s(k), X_1 \cup X_2)$ but we compute the size of each bucket B_x. Then we perform a prefix sum over the sizes of B_x to determine its position in the send buffer $NewSB^j$. Finally, we perform the merge in Line 6 again and save the results immediately in $NewSB^j$. Note that after Algorithm 24 the key-values pairs contained in the send messages $NewSB^j$ are sorted according to their hashed key.

7.3.3. Complexity

To analyze the complexity of Algorithms 19 and 20, we adopt a similar approach as during the previous sections. We use the MapReduce runtime parameters $w, \bar{w}, m, \bar{m}$ introduced in Section 2.2 and balls in bins model of Section 2.4. Let p_{old} and p_{new} be the number of processes before and after a node failure. Then $k = p_{old} - p_{new}$ indicates the number of failed processes.

Lemma 7.3. *Let's consider the redistribution Algorithm 23 performed after a node failure. Let the serialized key-value pairs be distributed uniformly at random by key between tp_{old} buckets during the map phase (Algorithm 21). We can implement the redistribution Algorithm 23 with an expected running time of:*

$$\mathcal{O}\left(k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, tp_{old}^2\right) + k\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, tp_{old}^2\right) + p_{new} + \log(t)\right).$$

If $m \in \Omega(\bar{m}tp_{old}^2 \log(tp_{old}))$ and $w \in \Omega(\bar{w}tp_{old}^2 \log(tp_{old}))$, then we get an expected complexity of:

$$\mathcal{O}\left(k\frac{w+m}{tp_{old}^2} + p_{new} + \log(t)\right).$$

Proof. Let's consider Algorithm 23. Processes which have saved a self-message of a failed node have $2k$ input messages S. The remaining processes process k messages. Each message consists of t buckets. Each process in the map phase partitioned their key-value pairs into tp_{old} buckets. This results in an overall uniform distribution of pairs into tp_{old}^2 buckets by their keys. In Line 2 each thread processes a bucket. Furthermore, the user-defined reduce function consumes the intermediate key-value pairs and each call of this function processes at most $\bar{w}$ pairs. According to Lemma 3.1 the worst case for static load balancing occurs if the workload is skewed and we have at most $\lceil w/\bar{w} \rceil$ different keys with $\bar{w}$ values each. Therefore the expected number of key-value pairs per bucket and processed by each thread is $\bar{w}b\left(\lceil w/\bar{w} \rceil, tp_{old}^2\right)$.
In Line 5 each thread scans over the sorted intermediate pairs and divides the bucket into p_{new} partitions. Since each process processes at most $2k$ messages. Lines 1 to 8 require expected $\mathcal{O}\left(k\bar{w}b\left(\lceil w/\bar{w} \rceil, tp_{old}^2\right)\right)$ time.
From Lines 11 to 19 each thread processes p_{new} buckets generated in the previous step. By applying Lemma 3.1 and the previous reasoning, each thread processes expected at most $\bar{w}b\left(\lceil w/\bar{w} \rceil, tp_{old}^2\right)$ pairs. We use Lemma 3.1 and reason that the maximum expected number of machine words per thread is $\bar{m}b\left(\lceil m/\bar{m} \rceil, tp_{old}^2\right)$. If the process does not contain self-messages of the failed node, we only perform copy operations which results in expected $\mathcal{O}\left(k\bar{m}b\left(\lceil m/\bar{m} \rceil, tp_{old}^2\right)\right)$ time for all k messages. Otherwise we perform a 2-way merge and copy operations which requires expected $\mathcal{O}\left(k\bar{m}b\left(\lceil m/\bar{m} \rceil, tp_{old}^2\right) + k\bar{w}b\left(\lceil w/\bar{w} \rceil, tp_{old}^2\right)\right)$ time.
Finally, we compute a prefix sum on each process over the new bucket B sizes to get the start and end indexes in the `MPI_All_to_allv` send buffer. This requires $\mathcal{O}(p_{new} + \log(t))$ operations. By using the same reasoning as before each thread processes expected $\bar{m}b\left(\lceil m/\bar{m} \rceil, tp_{old}^2\right)$ machine words in Line 22. All in all, Algorithm 23 has an expected runtime of

$$\mathcal{O}\Big(\underbrace{k\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, tp_{old}^2\right)}_{\text{Lines (1–8)(9–20)}} + \underbrace{k\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, tp_{old}^2\right)}_{\text{Lines (9–20)(22–25)}} + \underbrace{p_{new} + \log(t)}_{\text{prefix sum}}\Big).$$

Furthermore, if $m \in \Omega(\bar{m}tp_{old}^2 \log(tp_{old}))$, $w \in \Omega(\bar{w}tp_{old}^2 \log(tp_{old}))$ and by applying Lemma 2.2 we get an expected runtime in

$$\mathcal{O}\left(k\frac{w+m}{tp_{old}^2} + p_{new} + \log(t)\right).$$

□

Lemma 7.4. *Let's consider the merge Algorithm 24 performed after a node failure. Let the serialized key-value pairs be distributed uniformly at random by key between tp buckets during the map phase (Algorithm 21) and redistribution (Algorithm 23). We can implement the merge Algorithm 24 with an expected runtime of:*

$$\mathcal{O}\left(\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, tp_{new}\right) + \bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, tp_{new}\right) + p_{new} + \log(t)\right).$$

If $w \in \Omega(\bar{w}tp_{new} \log(tp_{new}))$ and $m \in \Omega(\bar{m}tp_{new} \log(tp_{new}))$, then we get an expected runtime of:

$$\mathcal{O}\left(\frac{w+m}{tp_{new}} + p_{new} + \log(t)\right).$$

Proof. Over all processes j, the send SB^j and recovered messages RSB^j contain all serialized key-value pairs produced during the map phase. In Line 6 we distribute all pairs into tp_{new}^2 buckets. We use a hash function to assign pairs interdependently and uniformly at random between these buckets by their key. Each parallel thread processes p_{new} buckets in Line 6. During the reduce phase, each user-defined reduce function can process at most $\bar{w}$ key-value pairs. By applying Lemma 3.1, the expected number of pairs processed by each thread is $\bar{w}b\left(\lceil w/\bar{w}\rceil, tp_{new}\right)$ in Line 6. By using a similar reasoning, the expected number of machine words is $\bar{m}b\left(\lceil m/\bar{m}\rceil, tp_{new}\right)$.
The bucket boundaries in Line 5 can be determined in constant time, since we have saved them during the recovery map phase. In Line 4 we can determine the start of the new bucket by scanning an additional bucket in the worst case as described in the previous section. Performing the 2-way merge in Line 6 and aggregating pairs with the same key and hash value can be done in Linear time. Therefore, Lines 2 to 7 have an expected runtime of $\mathcal{O}\left(\bar{w}b\left(\lceil w/\bar{w}\rceil, tp_{new}\right) + \bar{m}b\left(\lceil m/\bar{m}\rceil, tp_{new}\right)\right)$.
We perform a prefix sum over the sizes of buckets B to determine their positions in the `MPI_All_to_allv` send buffer. This requires $\mathcal{O}\left(p_{new} + \log(t)\right)$ operations. By using the same reasoning as before each thread processes expected $\bar{m}b\left(\lceil m/\bar{m}\rceil, tp_{new}\right)$ machine words in Line 11. All in all, the expected runtime for Algorithm 24 lies in

$$\mathcal{O}\Big(\underbrace{\bar{w}b\left(\left\lceil\frac{w}{\bar{w}}\right\rceil, tp_{new}\right)}_{\text{Lines (4–6)}} + \underbrace{\bar{m}b\left(\left\lceil\frac{m}{\bar{m}}\right\rceil, tp_{new}\right)}_{\text{Lines (6)(8–12)}} + \underbrace{p_{new} + \log(t)}_{\text{prefix sum}}\Big).$$

Furthermore, if we have $w \in \Omega(\bar{w}tp_{new} \log(tp_{new}))$ and $m \in \Omega(\bar{m}tp_{new} \log(tp_{new}))$ we can apply Lemma 2.2 and get a runtime of

$$\mathcal{O}\left(\frac{w+m}{tp_{new}} + p_{new} + \log(t)\right).$$

□

8. Experimental Setup

The following sections contain our experimental setup used to evaluate our MapReduce framework and its fault tolerance mechanism (Section 9). We run all our experiments on the SuperMUC-NG supercomputer (Section 8.1). Section 8.2 contains the used software, compiler and libraries. We describe the failure generation and our MapReduce configurations in Section 8.3. In Section 8.4 we describe the timing of the different parts of our algorithms. Additionally, we list the benchmark data sets used in our experiments (Section 8.5). We use these sets to run the benchmark algorithms introduced in Section 4. Finally, we give an overview of our used plots (Section 8.6)

8.1. Hardware

We perform the experiments in the following sections on the SuperMUC-NG high performance computing system [10]. The Leibniz Supercomputing Center (LRZ) of the Bavarian Academy of Sciences and Humanities situated in Munich maintains and runs this supercomputer. Figure 20 illustrates the architecture of SuperMUC-NG. This HPC system consists of 6336 thin and 144 fat compute nodes [6]. These two types of nodes only differ by the available memory. Thin nodes have 96 GiByte of memory, while fat nodes have 768 GiByte of memory. Each node consists of two sockets with a total of 48 cores, where each socket is an Intel Skylake Xeon Platinum 8174 processor [3, 8]. Intel launched this processor in 2017. It consists of 24 cores, and 48 threads. The processor base frequency is 3.10 GHz, with am maximum turbo frequency of 3.90 GHz. Each core has a L1 cache of 64 KiByte and L2 cache of 1024 KiByte. All cores on a possessor share an LLC cache with 33 MiByte. SuperMUC-NG employs four login nodes through which we can access the compute nodes and start our experiments [1].
SuperMUC-NG groups the thin compute nodes into 8 domains or islands. The remaining fat nodes constitute the last island. A fast OmniPath network [10, 94] connects the nodes inside an island. This network consists of a fat tree optimized for highly efficient communication [76, 99]. Between islands, the supercomputer adopts an OmniPath network topology, which is pruned

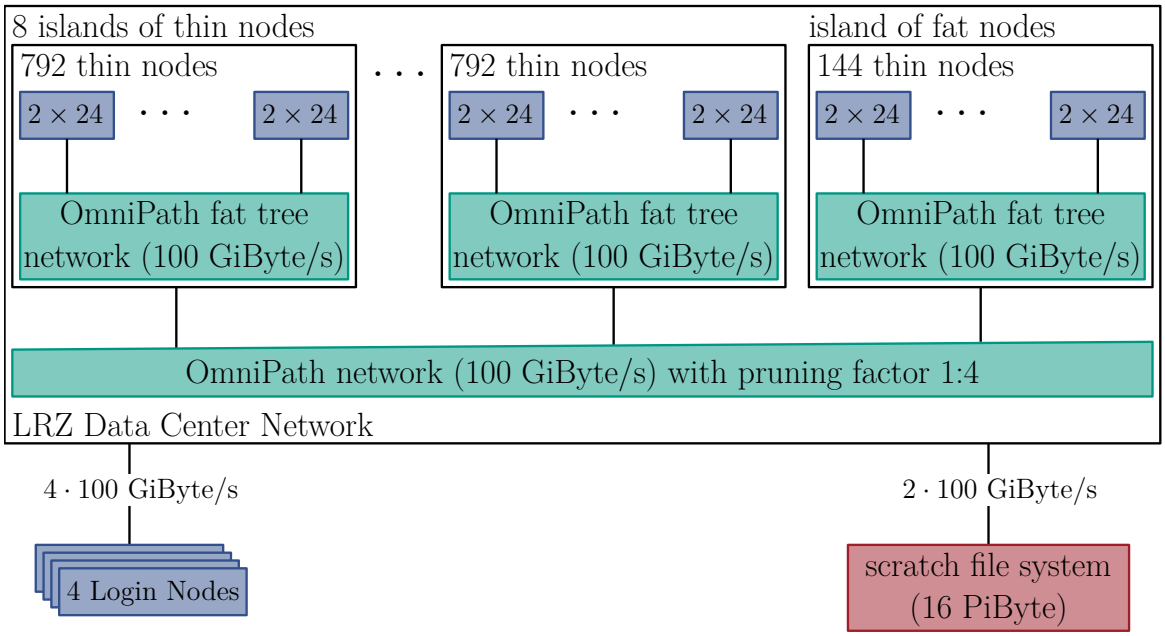


Figure 20: Illustration of the SuperMUC-NG architecture including the compute nodes, network typology, file system, and login nodes [2, 3, 4, 10].

with pruning factor 1:4. A pruned OmniPath network omits intermediate switches [12]. The communication bandwidth over the OmniPath network is 100 Gbit/s. During our experiments we ensure to perform our tests on one single island. We perform the experiments on the thin compute nodes with 96 GiByte of memory. We use the base process frequency, disable the turbo frequency, and disable automatic frequency scaling.
The SuperMUC-NG storage consists of Lenovo DSS-Gs for IBM Spectrum Scale [4]. Note that compute notes do not have local disks and use the scratch file system to replace the `/tmp` directory [4]. Instead, one has to use the parallel scratch file system, which is not situated on the compute nodes. We can access this file system with an average bandwidth of 200 GiByte/s. This scratch system handles all temporary data sets and we have access to 1 PiByte of storage.

8.2. Software and Compilation

The login and compute nodes run SUSE Linux Enterprise Server 15 SP1. SuperMUC-NG schedules the experiments using the Slurm 20.11.7 batch system. We compile our MapReduce frame work with GCC 8.4.0 and use the `C++17` standard. We construct the make files with Cmake 3.16.5. We use the OpenMPI 4.0.4 and OpenMP 4.5 libraries. OneTBB (branch: master, commit: 9e15720b) [13] is a GitHub project, which we use to include the Intel oneAPI Threading Building Blocks library [7]. For sorting we use *IPS*^{2}*Ra* (branch: master, commit: ee6103c) [68] introduced in Section 3.5. We use xxHash (branch: dev, commit: 4aa3d59) [31] as hash functions in our library. We compile with optimization `-O3` and compilation parameter `-DNDEBUG`.

8.3. Parameters Used and Failure Generation

We designed our framework with ULFM (Section 3.1) in mind, but this library did not run reliably on SuperMUC-NG. Periodically, ULFM processes failed without generating any failures. Therefore, we use OpenMPI 4.0.4 and simulate failures. We chose this MPI implementation, because ULFM is based on OpenMPI 4.0 [25]. During our experiments testing the failure recovery, we simulate the failure of 10% of a jobs compute nodes if not specified otherwise. We distribute the failures equidistantly and uniformly between all successive MapReduce operations of a benchmark algorithm. We use the number of MapReduce operations occurring during the execution without failure generation to distribute failures. Since we detect failures during the shuffle phase, we simulate them during the `MPI_All_to_allv` message exchange. We use `rand` and `srand` of the `<stdlib.h>` with seed 25 to generate the same failed node on all MPI processes,

Table 1: List of MapReduce framework configurations and their designation used during the experimental evaluation in Section 9. `MPI` represents the pure MPI algorithm and `HYB` the hybrid MPI and OpenMP algorithm. The designation `ft` indicates that we save self-messages for fault tolerance and `gf` specifies that we generate node failures.

Algorithm	Description	Sections
`MPI-MR`	MPI MapReduce without fault tolerance	6
`MPI-MR-ft`	MPI MapReduce with fault tolerance (save self-messages)	5.2.2+6
`MPI-MR-gf`	MPI MapReduce with fault tolerance and 10% node failure	5.2.2+6
`HYB-MR`	Hybrid MapReduce without fault tolerance	7
`HYB-MR-ft`	Hybrid MapReduce with fault tolerance (save self-messages)	5.2.2+7
`HYB-MR-gf`	Hybrid MapReduce with fault tolerance and 10% node failure	5.2.2+7

which stop their normal execution. We determine all processes situated on the node provided by `rand` and use `MPI_Split` to remove them from the MPI communicator. All nodes not situated on the failed node perform the recovery phase . Furthermore, we perform the experiments for the benchmark algorithms with and without saving self-messages (Section 5) without generating node failures. In Table 1 we show the designations of the different MapReduce framework configurations used during our experiments. Note that `MPI(HYB)-MR-ft`=`MPI(HYB)-MR`, since we do not exchange self-messages. Furthermore, `MPI(HYB)-MR-ft`=`MPI(HYB)-MR-gf`, because we start failure generation after 10 nodes.
For our Hybrid Parallelized MapReduce algorithm (Section 7) we pin each MPI process to a different socket. This results in two processes per compute node. We pin the OpenMP threads used by an MPI process to the each core on the corresponding socket. Each process has access to 24 cores and OpenMP threads. For our pure MPI implementation (Section 6) we start a process for each core, which results in 48 processes per node. We bind each process to a different core. We do not perform experiments for hyperthreading or MPI overcommitment.

8.4. Timing

Since our MapReduce framework is an in-memory implementation and we are interested in the performance of the fault tolerance mechanism, we exclude the time spend performing the I/O. We start monitoring after reading the data from the parallel file system and distributing the input randomly between the different processes. For all our configurations, each MPI process

Table 2: Designations used to identify the runtime of the different code segments of our MapReduce implementation during the experiments in Section 9. We take the maximum runtime over all MPI processes performing the different phases in parallel.

Label	Description	Algorithm
`MapReduce`	time to perform a MapReduce operation including failure recovery	Algorithm 12
`map`	time to perform the map phase, this includes the key-value pair serialization and the local group by key	Algorithm 12 Line 2
`shuffle`	time to exchange all messages during the shuffle phase	Algorithm 12 Line 3
`reduce`	time to perform the reduce phase, this includes the key-values pair deserialization and gathering all values per key	Algorithm 12 Line 4
`save`	runtime to save the send messages and exchange self-messages	Algorithm 16
`recovery`	time to recover from a node failure	Algorithm 15
`redistribute`	time to distribute the saved messages during recovery	Algorithm 15 Line 7
`recovery shuffle`	time to shuffle the redistributed data during recovery	Algorithm 15 Line 8
`recovery reduce`	time to perform the reduce phase during recovery	Algorithm 15 Line 9
`recovery map`	time to perform the map phase during recovery	Algorithm 15 Line 10
`merge buffer`	time to merge buffers during recovery	Algorithm 15 Line 11

uses `<sys/time.h>` to stop the runtime of the different phases: map phase, shuffle phase, reduce phase, save self-messages time, and recovery time. In our results we only represent the maximum execution times over all processes. Table 2 lists the labels which we use in our experiments. Furthermore, we visualize the maximum runtime across all processes per phase during our experiments.

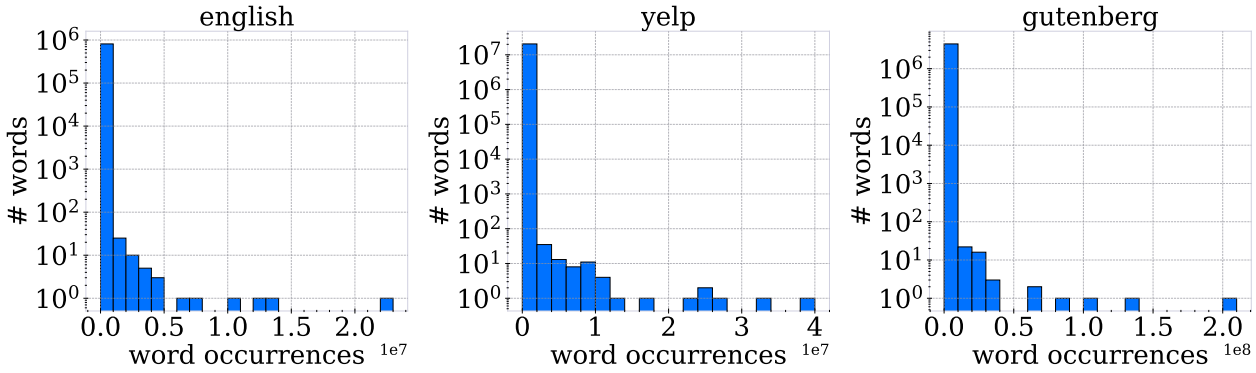


Figure 21: These tree histograms show the distribution of word occurrences of each text listed in Table 3. The x-axes contain the word occurrences and the y-axes represent the number of words in each bucket. We have chosen a logarithmic scale on the y-axes.

8.5. Benchmark Sets

In Section 8.5.1 we introduce four text benchmark data sets used during our Word Count experiments. Most of our benchmark algorithms (Section 4) require graphs as input, which we present in Section 8.5.2.

8.5.1. Text Data Sets

We use the texts in Table 3 to perform runtime experiments of our MapReduce framework with the Word Count algorithm (Section 4.1). We take the `english` [43] and `gutenberg` [9] data sets from the Gutenberg Project [9]. The `english` text is a concatenation of english text files from the etext02 to etext05 collections. The Gutenberg Project [9] contains over 60000 free books in plain text format. The `yelp` data set is a collection of Yelp reviews [11].
In Figure 21 we illustrate the occurrences of words in each text listed in Table 3. Note that for `english`, `yelp`, and `gutenberg` most words have a low occurrence. There are few words

Table 3: This table gives an overview of our text benchmark data sets, their sources, and their sizes. We use these texts to test our MapReduce framework with the Word Count algorithm (Section 4.1).

Text	Description	Source	Size (GB)
`english`	concatenation of open source english books	[43]	2.21
`yelp`	Yelp reviews as json file	[11]	6.95
`gutenberg`	Gutenberg bible in text form	[9]	22
`randtext-`y	x GiBytes of random text generated by the Hadoop RandomTextWriter with words chosen from the distinct text y	[5]	x

with a high occurrence. The text `english` contains 811 644 distinct words with a maximum occurrence of 22 804 033 and `gutenberg` consist of 4 436 574 distinct words with a maximum word occurrence of 208 499 702 for *"the"*. Additionally, the maximum word occurrence of `yelp` is 20 541 926.
The Hadoop random text generator RandomTextWriter [5] is a standard benchmark set for Word Count experiments [22]. This text generator creates texts by choosing words independently and uniformly at random from a list of 1000 distinct words until we reach a given size. The authors of [22] consider this benchmark bad for the reduce phase in MapReduce algorithms, because of the small word number. In our Word Count MapReduce implementation, the keys are words. Since we assign each key a process during the reduce phase, this may lead to imbalances during the shuffle and reduce phase (Section 9.1). The `randtext-`y text is a text generated by the RandomTextWriter [5] algorithm, where we choose the words from the unique words in text y. For instance, the `gutenberg` data set contains 4436574 distinct words.

8.5.2. Graph Data Sets

During our experiments with PageRank (Section 9.2), cc (Section 9.3), and R-MAT (Section 9.4) we use the graph datasets listed in Table 4. We use the `orkut` and `twitter` graphs provided by the Stanford Network Analysis Platform (SNAP) [60]. Orkut is a social network, where users are linked to their friends. We use the `orkut` friendship social network graph with $n = 3072441$ vertices and $m = 117185083$ edges [59]. The `twitter` graph consists of the follower relations from a snapshot taken in 2010 [61]. It contains $n = 41652230$ vertices and $m = 1468364884$ edges. An edge (x, y) is part of this graph, if y is a follower of x. The histograms in Figure 22 illustrate the distribution of the outgoing vertex degrees of `orkut` and `twitter`. Note that most vertices have a low degree. Most vertex degrees in `orkut` are smaller then 1 000, but the graph has one vertex with degree 33 007. Furthermore, most vertices in `twitter` have a degree smaller than 100 000, but one vertex has 2 997 487 neighbors.
We use the KaGen random graph generator [45] to generate random hyperbolic graphs [50] `rhg-d8-g3` with an average vertex degree of 8 and a gamma factor of 3. In random hyperbolic graphs the number of vertices with degree a vertex degree d is proportional to d^{γ}, where $\gamma > 0$ is the gamma factor [97]. We can use these random hyperbolic graphs to generate and simulate large networks, for instance the World Wide Web. KaGen generates us these graphs for our week scaling experiments for PageRank. Since KaGen allows us to specify the number of vertices and edges, we can generate larger graphs by increasing the node number and keeping

Table 4: This table gives an overview of our graph benchmark data sets, their sources, their number of nodes n and number of edges m. We use these graphs to benchmark our MapReduce framework with the graph algorithms in Section 4.

Graph	Description	n	m	Source
`orkut`	online social network graph	$3 \cdot 10^3$	10^8	[59]
`twitter`	snapshot of Twitter in 2010	$4 \cdot 10^7$	$1.4 \cdot 10^9$	[61]
`rhg-d8-g3`	Random Hyperbolic Graph with n vertices, average degree 8, and gamma factor 3	n	$8 \cdot n$	[45, 50]
`ErdősRényi-d`x`-n`n	random Erdős Rényi graph with n vertices and average degree of x	n	$x \cdot n$	[40]
`rmat(a,b,c,d)-n`n`-d`x	R-MAT graph with n vertices and average degree of x	n	$x \cdot n$	[27]

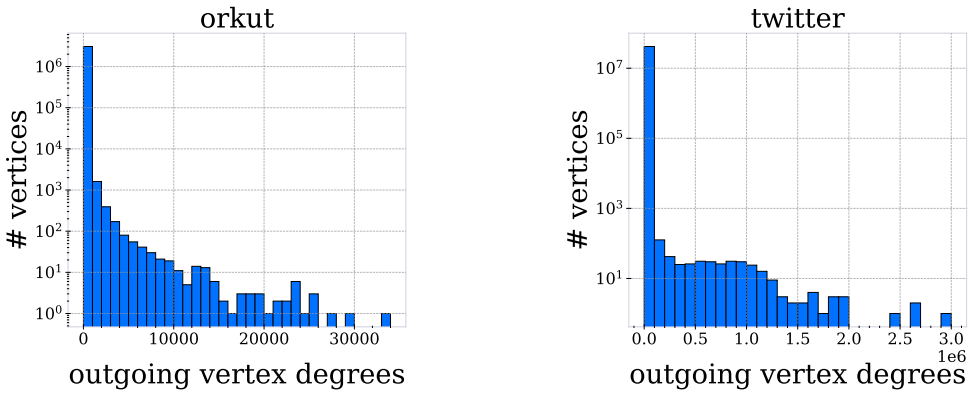


Figure 22: These two histograms show the distribution of outgoing vertex degrees of the `orkut` (left) and `twitter` (right) graphs (Table 4). The x-axes contain the word occurrences and the logarithmic y-axes represent the number of words in each bucket.

an average degree of 8.
A random Erdős Rényi graph [40] $G_{n,m}$ is a graph chosen randomly between all graphs with n vertices and m edges. We use our R-MAT MapReduce implementation (Section 4.4) with parameters a=b=c=d=0.25 to generate Erdős Rényi graphs [27]. We use the `ErdősRényi-d`x`-n`n graph with n vertices and average degree of x during our cc experiments. This allows us to generate small connected components for small x (Section 4.4). We use `rmat(a,b,c,d)-n`n`-d`x to indicate a R-MAT graph with parameters a, b, c, and d with n vertices and $m = x \cdot n$ edges.

8.6. Statistics

In Section 9 we perform strong and weak scaling experiments to analyze the behavior of our MapReduce framework with increasing number of compute nodes. For strong scaling experiments we use a single data set as input, while increasing the number of node. We plot the number of nodes on the x-axes and the runtime on the y-axes. We often need to choose logarithmic y-axes to analyze the behavior for large node numbers. An optimal scaling behavior occurs if the runtime decreases proportionally to the increasing node number. Doubling the compute nodes should half the runtime.
We use relative speedup plots to visualize the scaling behavior during strong scaling experiments. The y-axis illustrates the speedups $\frac{t_1}{t} \cdot n_1$, where t_1 is the time of the execution with lowest node number n_1. The term t is the execution time for the different node numbers illustrated on the x-axis. We achieve an optimal speedup if the data points are situated on the line $y = x$. If the different MapReduce phase speedups correspond to a linear function of the form $r(x) = ax + b$, we can compute a linear regression. The slope a of this regression represents the observed speedup [101]. Let D be a set of data points with $(n, s) \in D$, where n is the number of nodes and s its speedup calculated as above. Then the linear regression minimizes the mean square error

$$e(r, D) = \frac{1}{|D|} \cdot \sum_{(n,s)\in D} (r(n) - s)^2.$$

We use *liner* or *optimal* speedup to determine a speedup with slope $a = 1$ and super linear speedup for $a > 1$.
During our weak scaling experiments we increase the input size proportionally to the number of nodes. We use double the amount of input data if we double the number of nodes. Ideally this should result in no runtime increase. We represent the number of nodes on the x-axes and the execution time on the y-axes.

9. Experimental Results

We analyze our MapReduce frameworks with and without hybrid parallelization (Sections 6 and 7) by performing runtime experiments on the MapReduce algorithms introduced in Section 4. We run strong and weak scaling experiments to analyze their scaling behavior and the overhead of our fault-tolerant mechanism. We perform our experiments on the Supermuc-NG high performance computing system (System 8.1) and use the experimental setup described in Section 8.
Section 9.1 shows our results with the Word Count algorithm. Furthermore, we apply the PageRank and connected components algorithms on different graphs in Sections 9.2 and 9.3 respectively. We illustrate the results of the R-Mat graph generation with our MapReduce library in Section 9.4. Finally, Section 9.5 gives a brief summary of all our experiments.

9.1. Word Count

We apply the Word Count algorithm implemented in our MapReduce framework as indicated in Section 4.1 to real world text samples (Table 3). Initially, we distribute the texts uniformly between the different processes. By increasing the number of compute nodes, we want to analyze the scaling behavior and the overhead of sending self-messages for our fault-tolerant mechanism (Section 5). Since Word Count consists of a single MapReduce operation, we cannot test the recovery from a failed node. Furthermore, we compare the MapReduce implementations with (`HYB-MR(-ft)`) and without (`MPI-MR(-ft)`) hybrid parallelization.
Figure 23 shows the execution times of our MapReduce implementations with and without fault-tolerance applied on the `english`, `yelp`, and `gutenberg` data sets. First of all, we observe that increasing the number of nodes decreases the execution time for the hybrid MapReduce implementation. This is also the case for `MPI-MR(-ft)` until 80 nodes for `english` and `yelp`. We detect an increase from $0.31s$ on 80 nodes to $0.56s$ on 160 nodes for `english`. Fort `gutenberg` this increase starts at 128 nodes from $2.04s$ to $2.51s$ (256). We analyze this behavior further in Figure 24a on the `gutenberg` text, since we have results for a higher number of nodes and

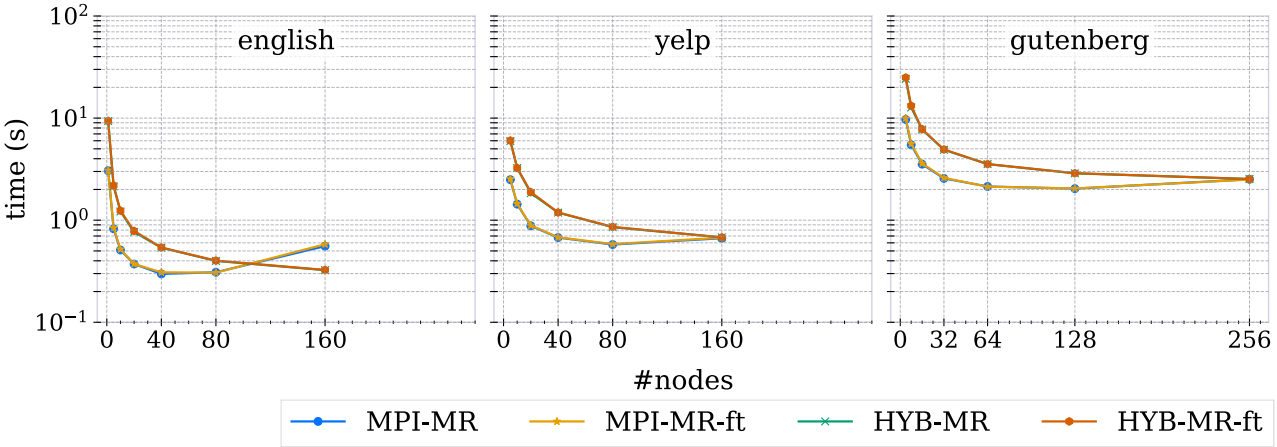
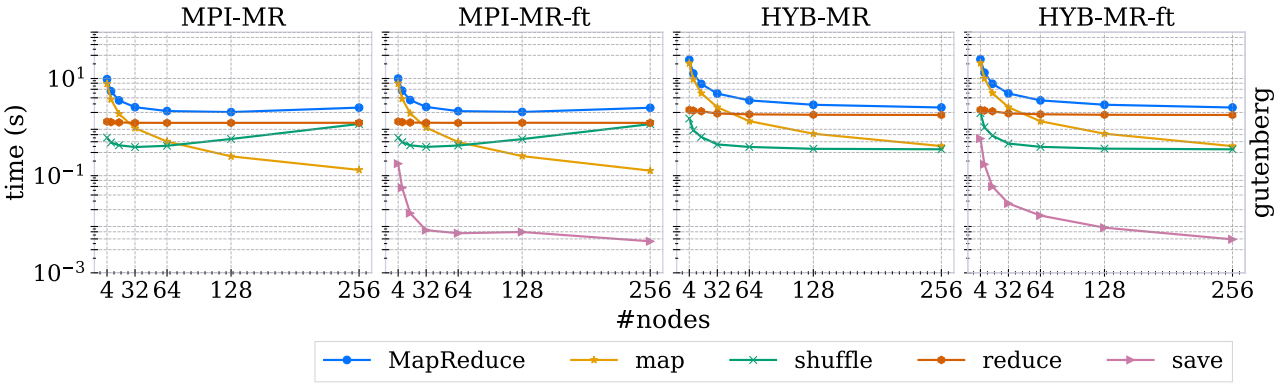
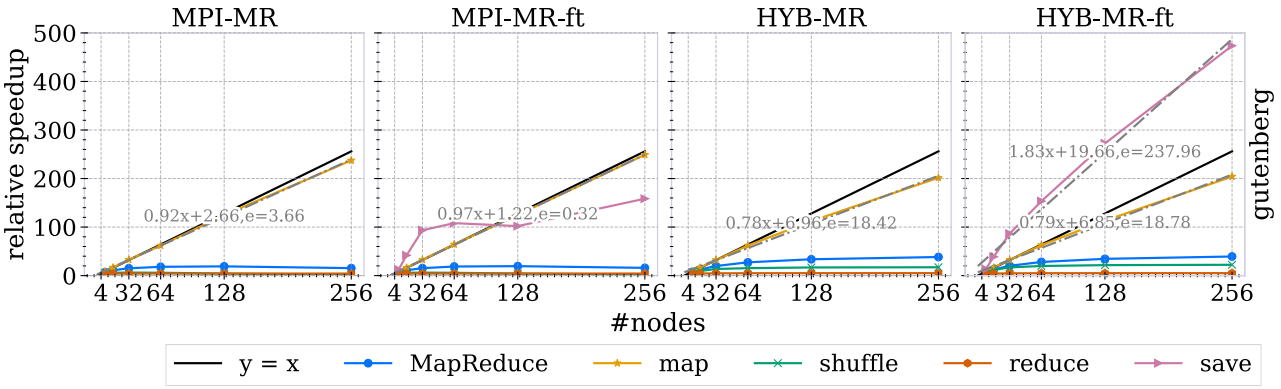


Figure 23: Illustration of strong scaling experiments executed on the texts in Table 3 and MapReduce configurations listed in Table 1. We apply the Word Count algorithm on `english` (nodes in $\{1, 5, 10, 20, 40, 80, 160\}$, 7 runs), `yelp` (nodes in $\{5, 10, 20, 40, 80, 160\}$, 7 runs), and `gutenberg` (nodes in$\{4, 8, 16, 32, 64, 128, 256\}$, 4 runs). The x-axes show the node number, while the logarithmic y-axes show the average runtime over all runs.

its our largest text data set. The experiments for `english` and `yelp` yield the same results and we illustrate them in Section C.1 (Figures 48 and 49).
The execution time of the shuffle phase starts increasing for `MPI-MR(-ft)` from $0.39s$ on 32 nodes to 1.16 on 256 nodes (Figure 24a). Since the time of the other phases does not increase, the shuffle phase causes the execution time increases for the entire MapReduce operation observed in Figure 23. Let n be the number of nodes, then `MPI-MR(-ft)` employs $48n$ MPI processes, while `HYB-MR(-ft)` uses $2n$ (Section 8). Furthermore, the shuffle phase uses the collective `MPI_All_to_allv` operation to exchange the key-value pairs between processes. This MPI operation is part of the irregular MPI collective, which are known to have scaling issues for large process numbers [18, 19]. Moreover, collective operations require synchronization, which can lead to significant overheads due to operating system or network noise [18]. Note that the shuffle phase of our hybrid MapReduce implementation decreases from $0.44s$ to $0.35s$ between 32 and 256 nodes.
Additionally, the reduce phase of `MPI-MR(-ft)` decreases until 32 nodes, after which it remains constant at $1.22s$. On the other hand, the reduce phase of `HYB-MR(-ft)` keeps decreasing between 32 and 256 nodes from $1.90s$ to $1.78s$. We can explain this with the word occurrences in



(a) The logarithmic y-axes show the average runtime.



(b) The linear functions $ax + b$ are a linear regression for the map phases (`map`) and the save self-message phase of `HYB-MR-ft` (`save`). The term e indicates the mean square error. The y-axes show the average speedups.

Figure 24: We perform 4 runs of each MapReduce configuration (Table 1) executing Word Count (Section 4.1) on `gutenberg` with nodes in $\{4, 8, 16, 32, 64, 128, 256\}$. The x-axes show the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), and the entire MapReduce execution (`MapReduce`)

Table 5: This table shows the relative average MapReduce phase times $r_x = \frac{t_x}{t_{mr}} \cdot 100\%$ during the Word Count experiments on `gutenberg` (Figure 24). t_x is the average execution time of phase x and t_{mr} the MapReduce time over all runs and node counts.

MapReduce	`map`(%)	`reduce`(%)	`shuffle`(%)	`save`(%)
`MPI-MR-ft`	54.33	30.54	14.16	0.97
`HYB-MR-ft`	67.95	23.16	7.45	1.44

the different texts. As indicated in Figure 21 and Section 8.5, `gutenberg` has a maximum word occurrence of 208 499 702 for *"the"*, where 99.99% of all distinct words occur less then 10^7 times. This leads to an imbalanced reduce phase, since one process has to handle the key *"the"*. The entire phase is bound by this slower process. Increasing the number of nodes cannot decrease the execution time because of this bottleneck workload. Therefore, we see nearly no speedups for the reduce phase (Figure 24b).
The map phase, on the other hand, decreases with increasing number of nodes (Figure 24a). We use a linear regression to analyze the speedups of the different map phases (Figure 24b). This results in a nearly linear speedup for `MPI-MR(-ft)`, which has a slope of 0.92(0.97). The speedup of our hybrid MapReduce implementation is lower with a slope of 0.78 for `HYB-MR`. This matches the theoretical results in Sections 6.1 and 7.1. We only achieve the optimal speedup with our random static load-balancing strategy, if the problem size is large enough compared to the number of used processes and parallel threads. In this case, the used data sets may be too small for `HYB-MR`. Note that its theoretical runtime has an additional factor p.
Moreover, the execution times with fault-tolerance mechanism are close to those without (Figure 24a). For `MPI-MR-ft` saving self-messages constitutes 0.97% of the average execution time (Table 5), while the save phase of `HYB-MR-ft` requires 1.44% of its average runtime. Furthermore, we observe a super linear speedup for the save self-message times of `HYB-MR-ft` (Figure 24b). The linear regression over the save phase times has a slope $1.38 > 1$. This result corresponds to the theoretical results proven in Lemma 5.1, which proves an expected runtime of $\mathcal{O}\left(k\frac{m}{p^2}\right)$, where m is the total data send size, p the number of processes, k the process count per node, $\bar{m}$ the maximum message size, and $m \in \Omega(\bar{m}p^2 \log(p))$. The speedups of `MPI-MR-ft`, on the other hand, are greater than one until 64 nodes and smaller after 128. Note that `MPI-MR-ft` employs 24 times more processes than `HYP-MR-ft`, which requires a larger input data set m to achieve the optimal runtime.
We illustrate the relative execution times for the different phases during a MapReduce operation for the `yelp` and `english` data sets in Section C.1 in Tables 9 and 10. Additionally, for both frameworks, the map phase constitutes more than half of the entire MapReduce operation (Table 5). This can be explained by the group by key performed during mapping (Section 6 and 7), which transforms pairs into aggregated key-values pairs. Hence, the group by key performed during the reduce phase works on potentially fewer pairs.
As described above and in accord to the theoretical results our MapReduce frame work may not scale optimally, if the data set is too small for the number of used nodes or the data set contains a large bottleneck workload. To test our library in a more favorable environment we perform weak scale experiments with randomly generated texts. We use the Hadoop random text generator (Section 8.5), used by Word Count implementations during testing [22]. This generator creates a text from a pool of 1000 words. Since this number of words is too small for scalability during the reduce phase, we choose words from the `gutenberg` data set (Section 8.5).
As during our strong scale experiments we notice that the runtime with and without fault-tolerance are similar (Figure 25). For `HYB-MR-ft` the save self-message time constitutes 1.11% of its total execution time and for `MPI-MR-ft` 1.41% (Table 8). Furthermore, for both MapRe-

Table 6: This table shows the relative average MapReduce phase times $r_x = \frac{t_x}{t_{mr}} \cdot 100\%$ during the Word Count experiments on `gutenberg` (Figure 24). t_x is the average execution time of phase x and t_{mr} the MapReduce time over all runs and node counts.

MapReduce	`map`(%)	`reduce`(%)	`shuffle`(%)	`save`(%)
`MPI-MR-ft`	68.06	20.68	9.85	1.41
`HYB-MR-ft`	90.12	1.97	6.8	1.11

duce configurations the execution time of the save time is slower than the other phases and decreases with increasing number of nodes. The save time decreases from 0.2404s at 2 nodes to 0.0277s at 256 nodes for `HYB-MR-ft` and 0.3096s to 0.0033s for `MPI-MR-ft`. This matches the theoretical results as described above.
The map phases represent more than 68% of the entire runtime (Table 8), which can be explained by the fact that it performs the most work as described above. Furthermore, its execution time remains nearly constant with increasing node number. The map phase of `MPI-MR(-ft)` remains between 3.26s and 3.52, while the map phase of `HYB-MR(-ft)` is between 5.02s and 5.27s. Moreover, the shuffle phase increases 2.37 times for `HYB-MR` and 9.36 times for `MPI-MR` from 1 to 256 nodes (Figure 26). The time of the shuffle phase increases faster for `MPI-MR` and `MPI-MR-ft`. As described above this may indicate scalability issues for the `MPI_All_to_allv` operation.
Furthermore, the reduce phase of `MPI-MR` (avg. 1.04s) is slower than that of `HYP-MR` (avg. 0.11s) (Figure 26). The hybrid MapReduce implementation uses a $p-$way merge during the reduce phase, where p is the number of processes, while the `MPI-MR(-ft)` uses a sorting algorithm. Note that `HYP-MR(-ft)` has a worse theoretical execution time due to this merge (Section 7.2.2), but it consists of a single scan over the data. Whereas the radix sort used by `MPI-MR(-ft)` may has a larger constant time factor (Section 6.2.2).
The map phase of `MPI-MR` (avg. 3.31s) is faster than the map phase of `HYP-MR` (avg. 5.19s) even though their general idea remains the same (Figure 26). This may be caused by additional scans over the data, the t-way merge of buckets, and prefix sum, which we require to enable the parallel shared memory implementation of `HYP-MR(-ft)` (Section 7.1).

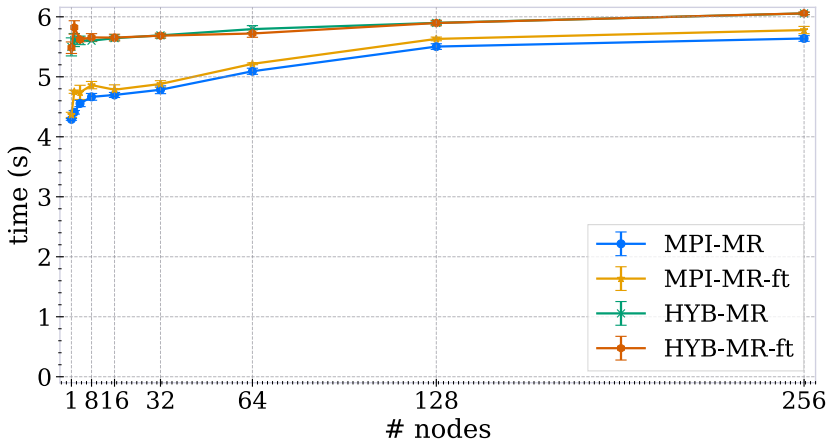


Figure 25: We apply the different MapReduce configurations with Word Count on the randomly generated text `randtext-gutenberg` (2GiByte of text per nodes) with seeds in $\{0, 1, 2, 3, 4\}$ and nodes in $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$. The x-axis shows the number of nodes, while the y-axis show the average runtime with error bars illustrating the standard deviation.

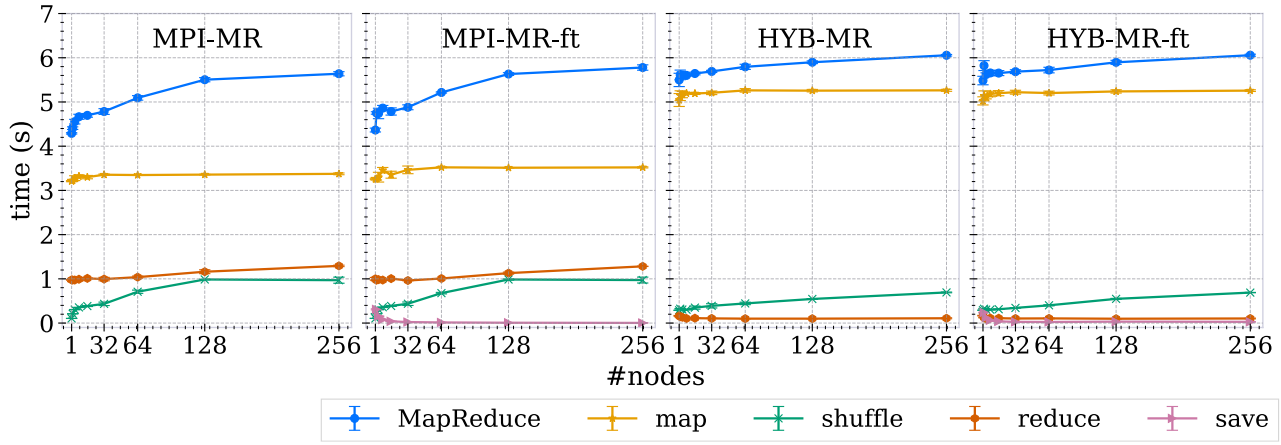
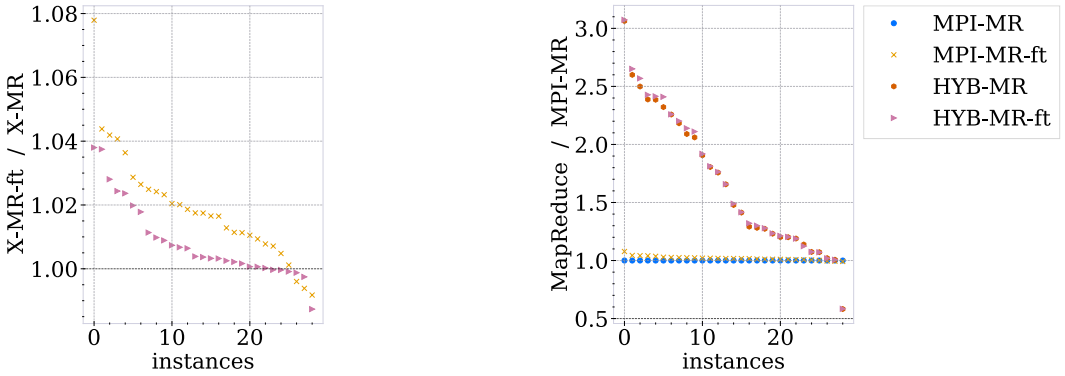


Figure 26: For each seed in $\{0, 1, 2, 3, 4\}$, nodes in $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$ we apply Word Count on `randtext-gutenberg` (2GiByte of text per nodes). We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), and the entire MapReduce execution (`MapReduce`). The x-axis shows the number of nodes, while the y-axis show the average runtime with error bars illustrating the standard deviation.

Over all our Word Count experiments saving and sending self-messages requires at most 1.08 times more time for `MPI-MR` and 1.04 for `HYB-MR-ft` (Figure 27a). Furthermore, `HYB-MR-ft` is $\times 1.01$ lower and `MPI-MR-ft` $\times 1.02$ for 19 instances out of 29. As we observe in Figures 25 and 24 `HYB-MR(-ft)` is slower than `MPI-MR`. In Figure 27b we compare all MapReduce configurations with `MPI-MR`. `HYB-MR(-ft)` are at most 3.07 times slower and for 15 instances it is less than 1.5 times slower.



(a) The y-axis shows the runtime of the MapReduce configurations divides by `MPI-MR-ft`. A value of 1.04 indicates for `X-MR-ft` that there is an instance, where `HYB-MR-ft` is 1.04 times slower than `HYB-MR`

(b) The y-axis shows the runtime with saving self-message divided by the MapReduce without. A value of 1.04 indicates for `HYB-MR-ft` that there is an instance, where `HYB-MR-ft` is 3.07 times slower than `MPI-MR`

Figure 27: These plots contain the values of the `gutenberg`, `yelp` and `english` experiments (Figure 24) as well as the week scaling experiments (Figure 25). The x-axes are Word Count applied on the different data sets with their corresponding node numbers. We average over runs. We compare the different configurations to a base line.

To sum up, the saving self-messages is at most 1.08 times slower for `MPI-MR` and 1.04 for `HYB-MR-ft`. Furthermore, the map and reduce phase have nearly linear speedups if the processed data is large enough compared to the number of nodes and the bottleneck workload. For a small number of nodes `MPI-MR(-ft)` is faster, which may change for larger node numbers due to its shuffle phase. Due to the large node number and scheduling times on the SuperMUC-NG (Section 8.1) we could not confirm this hypothesis.

9.2. Page Rank

We use the PageRank algorithm (Section 4.2) to analyze the scaling behavior and fault-tolerance overheads. Hence, we compute the page ranks for the graphs listed in Table 4 with a fixed number of 100 iterations. We compare our MapReduce libraries with and without hybrid parallelization (Sections 6 and 7). We perform the experiments with and without fault-tolerance mechanisms. Additionally, we generate 10% node failures. We list the corresponding abbreviations and configurations in Table 1. During our speedup experiments we compute linear regressions, to avoid overloading the plots we do not represent them all, but we use their slops to determine the speedups of the phases (Section 8.6).
The strong scaling experiments in Figure 28 compare the execution times of our MapReduce libraries with and without hybrid parallelization. First, the execution time decreases with increasing number of nodes but we do not achieve an optimal linear speedup (Figures 29b and 30b). The linear regression slope of the `MPI-MR` speedups is 0.09 for `twitter` (`orkut` 0.43). The slope for `HYB-MR` is 0.3 for `twitter` (`orkut` 0.35). Hence the hybrid parallelization has better speedups for `twitter`. Furthermore, `HYB-MR` requires $0.65s$ on average per MapReduce operation with 256 nodes, while `HYB-MR` has a runtime of $1.50s$. This can be explained by the dynamic scheduling used during the map phase (Algorithm 21) during the parallel for-loop (Line 3). This allows a better load balancing during the map phase, since the implementations without shared memory parallelization only use static load balancing. Note that we have to use static load balancing during our reduce phase (Section 7.2), which explains its low speedup of 0.08 compared to the map phase with 0.36.

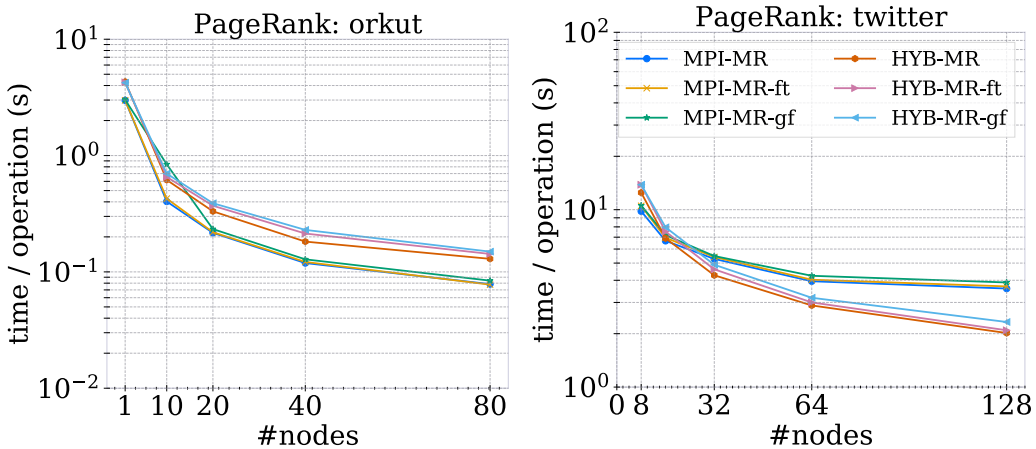
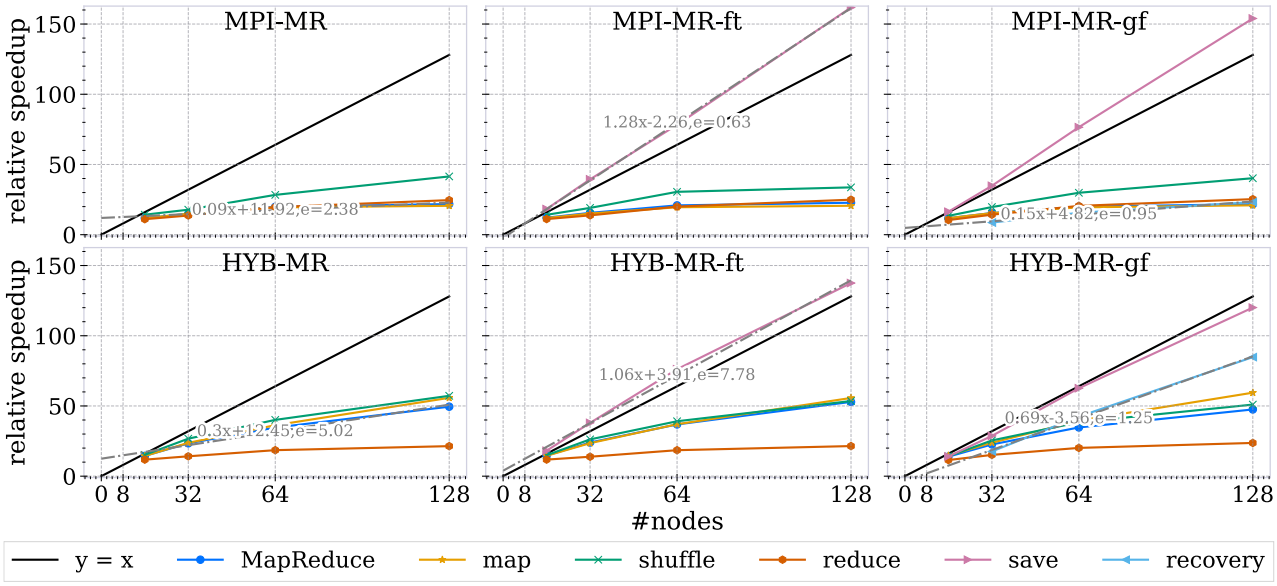
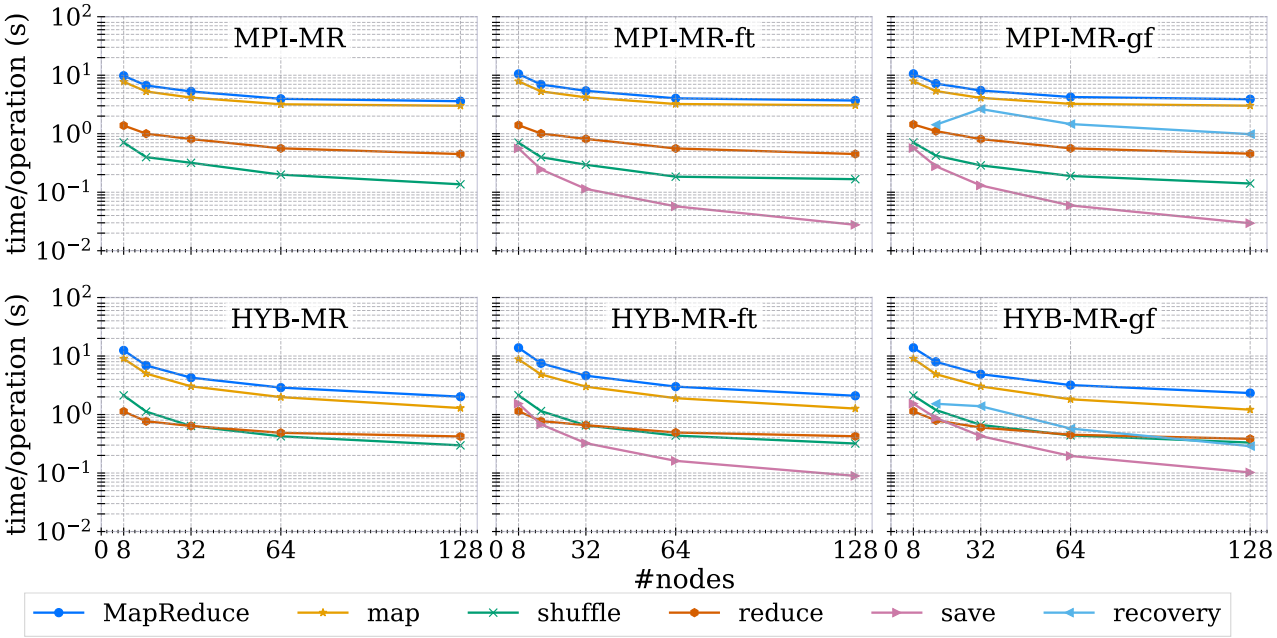


Figure 28: We perform strong scaling experiments with our MapReduce configurations (Table 1) for the PageRank algorithm (Section 4.2). We perform 100 iterations for `orkut` (left) with nodes in $\{1, 10, 20, 40, 80\}$ and `twitter` (right) with nodes in $\{8, 16, 32, 64, 128\}$. The x-axes contain the number of nodes, while the logarithmic y-axes show the runtime per MapReduce operation.

All in all, we observe smaller speedups as during the Word Count strong scale experiments (Section 9.1, Figure 24b). This behavior could be explained by the graph typologies and the bottleneck workload. Let v be a vertex, $O(v)$ its outgoing neighbors, and $I(v)$ its incoming neighbors. The runtime of the user-define map function (Algorithm 4) during PageRank has a time com-



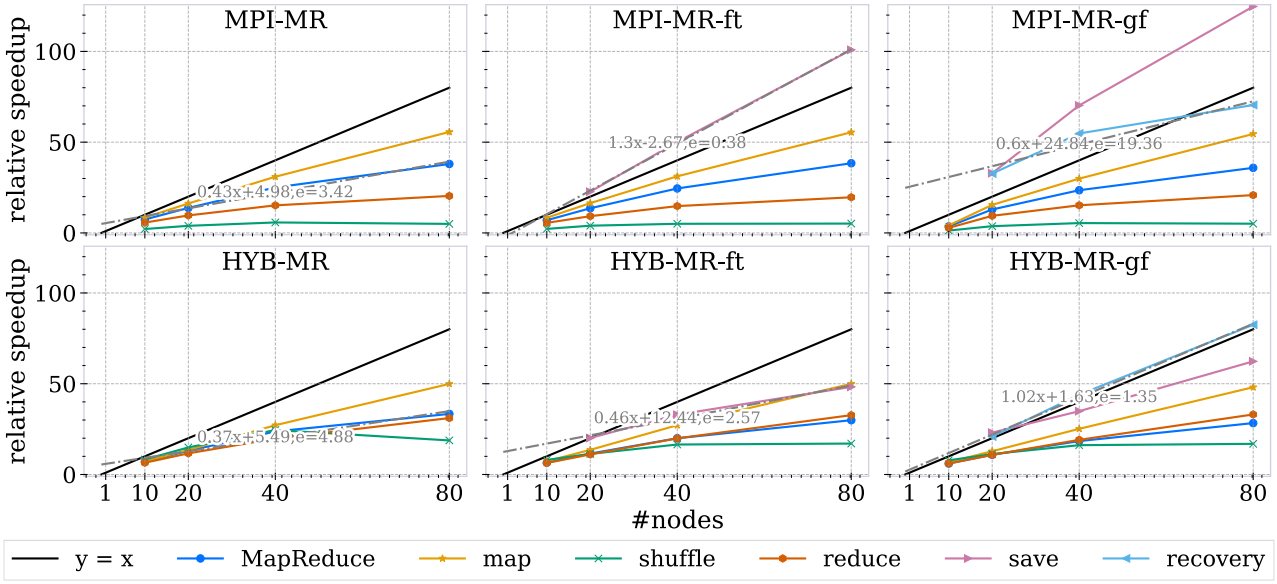
(a) The y-axes show the relative speedups. We plot linear regressions $ax + b, e$, where e is the mean square error. We compute the regression for the speedup of `MPI(HYB)-MR`, the save self-message phase (`save`) of `MPI(HYB)-MR-ft`, and the speedup for the recovery of `MPI(HYB)-MR-gf`.



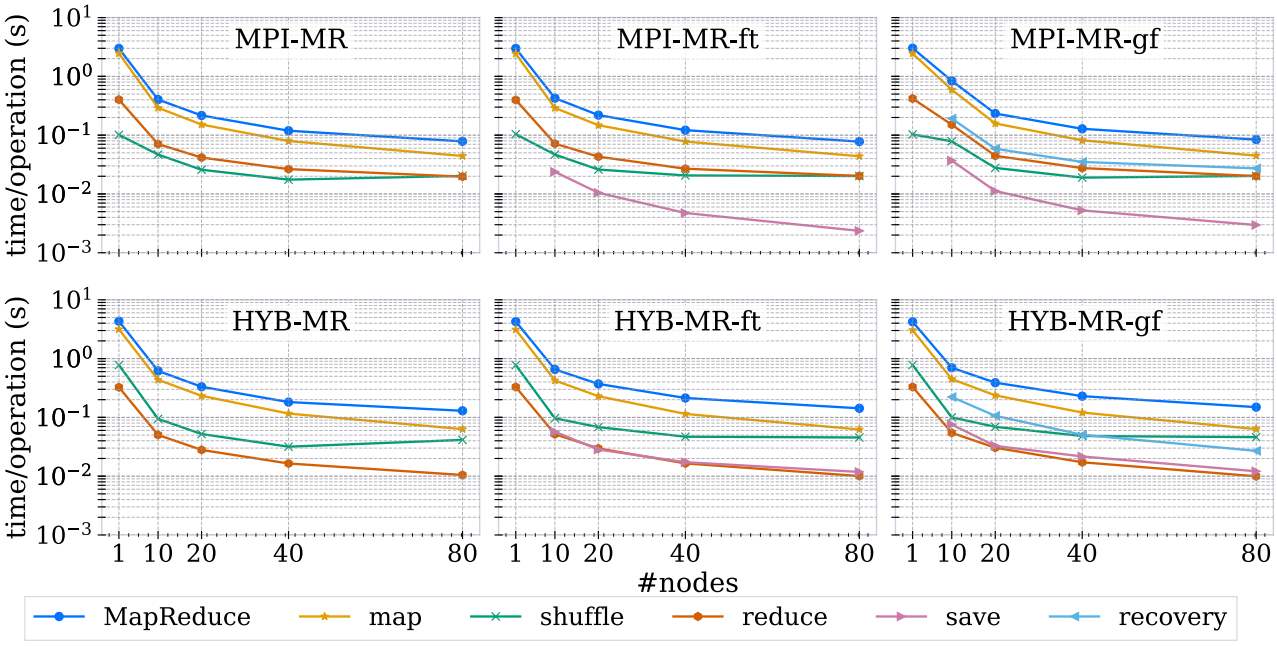
(b) The logarithmic y-axes show the average runtime per MapReduce operation.

Figure 29: We execute 100 PageRank iterations (Section 4.2) on `twitter` implemented in our MapReduce framework (Table 1) with nodes in $\{4, 8, 16, 32, 64, 128\}$. The x-axes show the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

plexity of $\mathcal{O}(|O(v)|)$, while the runtime of the user-defined reduce function (Algorithm 5) lies in $\mathcal{O}(|I(v)|)$. Therefore, the bottleneck workload $\bar{w}$ lies in $\mathcal{O}(|O(v)| + |I(v)|)$. As proven in Sections 6 and 7, the runtime and scaling behavior of our MapReduce framework depends on the bottleneck workload $\bar{w}$. We can only expect an optimal linear speedup if the processed data



(a) The y-axes show the relative speedups. We plot a linear regression $ax + b, e$, where e is the mean square error. We compute the regression for the speedup of `MPI(HYB)-MR`, the save self-message phase (`save`) of `MPI(HYB)-MR-ft`, and the speedup for the recovery of `MPI(HYB)-MR-gf`.



(b) The logarithmic y-axes show the average runtime per MapReduce operation.

Figure 30: We execute 100 PageRank iterations on `twitter` implemented in our MapReduce framework (Table 1) with nodes in $\{1, 10, 20, 40, 80\}$. The x-axes show the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

is large enough compared to $\bar{w}$. In Section 8.5.2 we analyze the vertex degrees of both graphs (Figure 22) and `twitter` has a bottleneck vertex with 2 997 487 neighbors and `orkut` one with 33 007. All processes have to wait that one process computes the results of the high degree vertex. The maximum vertex degree is larger for `twitter`. Furthermore, the maximum vertex degree of `orkut` is closer to the low vertex degrees (Figure 22). This means the bottleneck workload $\bar{w}$ of `twitter` is larger. This can explain the fact that `orkut` has a better speedup (Figures 29b and 30b).

This is a well-known issue for MapReduce algorithms operating on large graphs, especially for our naive PageRank implementation [62]. Lin and Schatz [62] propose an alternative PageRank algorithm without these scaling issues. They require additional data structures and MapReduce related operations, which our framework does not support. Implementing those features into our framework would make our fault-tolerance mechanism insufficient.

We perform weak scaling experiments on `rhg-d8-g3` and `ErdősRényi-d38` (Table 4) to analyze the behavior of our PageRank implementation for larger data sets and node counts. In Figure 31, we plot the average execution times per MapReduce operation for both data sets. Since both experiments have similar results, we analyze `rhg-d8-g3` in detail since we have data for larger compute nodes. We illustrate the results for `ErdősRényi-d38` in Figure 50 in Section C.2.

First of all, the runtime of all our implementations rises with increasing number of nodes. For `rhg-d8-g3` the execution time of `MPI-MR` increases from 0.22s at one node to 0.45s at 400 nodes. The runtime of `HYB-MR` increases from 0.40s at one node to 0.76s at 400 nodes. To analyze this behavior further we plot the runtime of the different MapReduce phases (Figure 32). The map phase of `MPI-MR` has an increase of 26.0%, while the reduce phase increases by 66.9% (Figure 11). Furthermore, the shuffle time gets 20.427 times slower from 1 to 400 nodes. Note that the shuffle time gets only 5.366 times slower from 2 nodes to 400. On one node, processes do not need to use the network to exchange messages, they can be sent inside the node, which is faster. The bad shuffle scaling could be caused by scaling issues of the `MPI_All_to_allv` operation as explained in Section 9.1. We also observe an increase in runtime for the phases

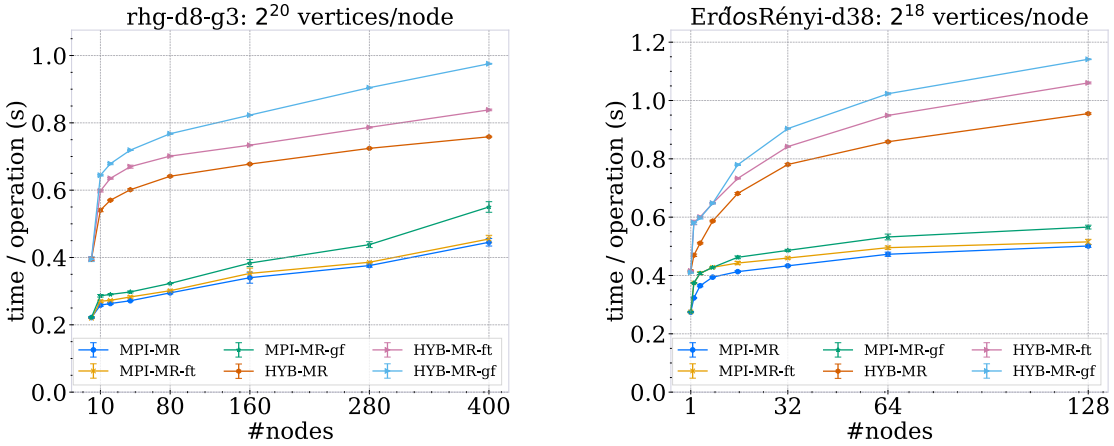


Figure 31: We perform 100 PageRank iterations with our MapReduce configurations (Table 1) for seeds in $\{1, 2, 3, 4, 5\}$ on $\{1, 10, 20, 40, 80, 160, 280, 400\}$ nodes for `rhg-d8-g3` (left, 2^{20} vertices per node) and $\{1, 2, 4, 8, 16, 32, 64, 128\}$ nodes for `ErdősRényi-d38` (right, 2^{18} vertices per node) (Table 4). We generate 10% node failures for `MPI(HYB)-MR-gf`. The x-axes show the number of nodes, while the y-axes show the average runtime per MapReduce operation. We include error bars with standard deviation.

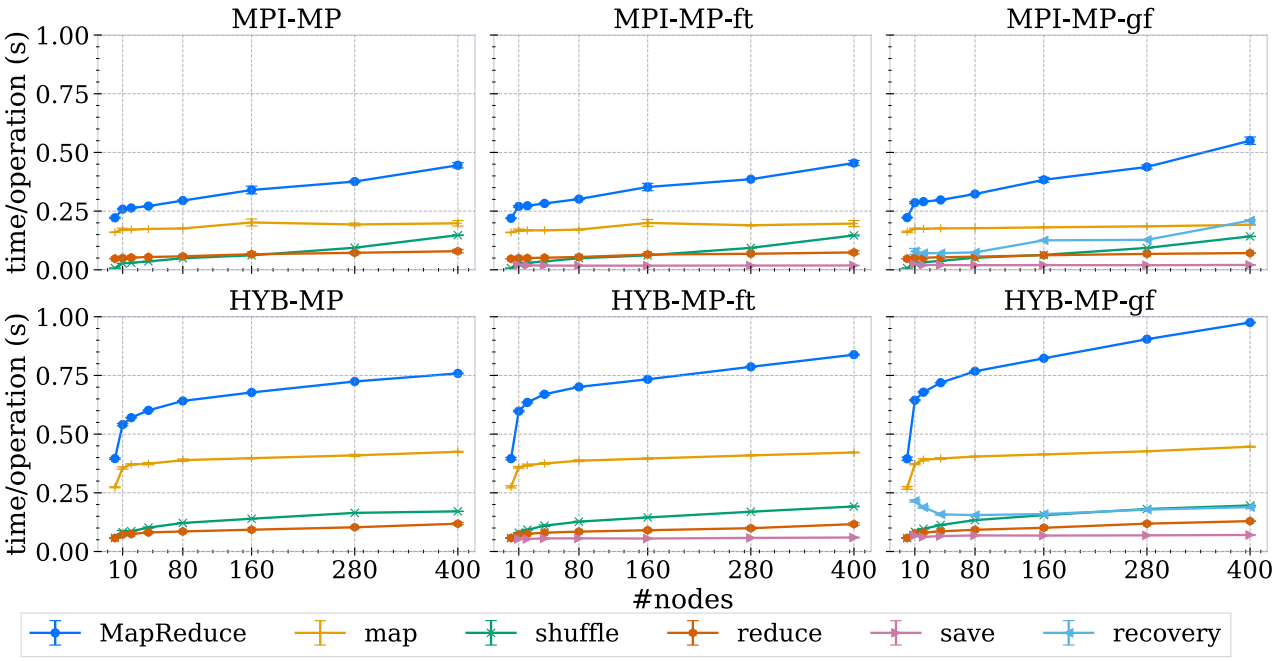


Figure 32: We perform 100 PageRank iterations with our MapReduce configurations (Table 1) on `rhg-d8-g3` (seeds in $\{1, 2, 3, 4, 5\}$ and nodes $\{1, 10, 20, 40, 80, 160, 280, 400\}$). We generate 2^{20} vertices per node. Furthermore, we generate 10% node failures for `MPI(HYB)-MR-gf`. The x-axes show the number of nodes, while the y-axes show the average runtime per MapReduce operation including error bars with standard deviation. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

of `HYB-MR` (Table 11). Since we use random static load balancing, we only achieve optimal speedups if the data set is large enough compared to the number of used processes (Sections 6 and 7).

Despite using the same general algorithm, `MPI-MR` requires on average $0.18s$ during the map phase, while the map phase of `HYB-MR` has an average runtime of $0.37s$ (Figure 32). We observe the same behavior during our Word Count experiments (Figure 9.1). The main difference between both map phases is that `HYB-MR` uses a shared memory parallelization on each process with t threads. This approach requires additional data structures, a t-way merge and scans over the data to avoid read/write conflicts (Section 7.2.1). Furthermore, we need to gather all data into a single send buffer.

First of all, the additional time needed to send these messages is lower than the other phases (Figures 29b, 30b, and 32). Figure 33 (left) illustrates the overheads of the save self-message phase. For instance, the largest data point 1.239 of `HYB-MR-ft` indicates that over all our PageRank instances, the largest overhead for saving the self messages was 23.9%. In other words, `HYB-MR-ft` was 23.9% slower then `HYB-MR`. The largest overhead for `MPI-MR-ft` is 16.0%. Note that for 21 of all 24 instances, `HYB-MR-ft` has overheads lower than 12% and an average overhead of 10.4. `MPI-MR-ft`, on the other hand, has an average overhead of 4.7%. All in all, `MPI-MR-ft` has lower overheads than `HYB-MR-ft` (Figure 33).

Furthermore, the average and median overhead for `rhg-d8-g3` and `MPI-MR-ft` is approximately 3 times lower than the overhead for `HYB-MR-ft`. `MPI-MR-ft` has approximately 1.5 times lower overheads for `ErdősRényi-d38`. Overall, the average overhead of `MPI-MR-ft` is 2.21 times lower

than that of `HYB-MR-ft`.
We can explain this phenomenon with the theoretical runtime of the save self-message phase (Section 5.2.2). Let n be the number of nodes. Each node has 48 cores and we start 2 processes per node for `HYB-MR-ft` and 48 for `MPI-MR-ft`. Therefore, the expected runtime is $\mathcal{O}\left(\frac{m}{2n^2}\right)$ for `HYB-MR-ft` and $\mathcal{O}\left(\frac{m}{12n^2}\right)$ for `MPI-MR-ft`. The constant factor of our hybrid MapReduce is 6 times larger. In other words `HYB-MR-ft` has 24 times fewer processes than `MPI-MR-ft` and must send 24 times the data volume. We cannot speedup this message exchange with shared memory techniques, because we send the messages between different nodes with non-shared memory.
For `twitter` the save self-message phase has a spear linear speedups with a slope of 1.28 for `MPI-MR-ft` and 1.06 for `HYB-MR-ft` (Figure 29a). According to Lemma 5.1, this phase has an expected runtime of $O(k\frac{m}{p^2})$, where k is the number of processes and m the size of all messages. Hence, an even better speedup of 2 is possible if m is large enough compared to the bottleneck workload. As described above, this is not the case for our strong scale experiments.
In Figure 33 we illustrate the overheads of the 10% failure generation (`MPI(HYB)-MR-ft`) and `MPI(HYB)-MR-gf`. Note that this overhead does not only include the time needed to recover from a failure and recompute lost data, but also the execution on fewer nodes, since they fail. For `MPI-MR-gf` is maximally 1.970 times slower than `MPI-MR-ft` and has an average overhead of 12.0%, while for 16 of 19 instances `MPI-MR-gf` is at most 1.098 times slower. On the other hand, `HYB-MR-gf` is at most 1.163 times slower than `HYB-MR-ft` with a maximum overhead of 8.3%. Notice that the overheads for both configurations are similar (Figure 33).
All in all, the time needed for the recovery is comparable to the other phases. The recovery

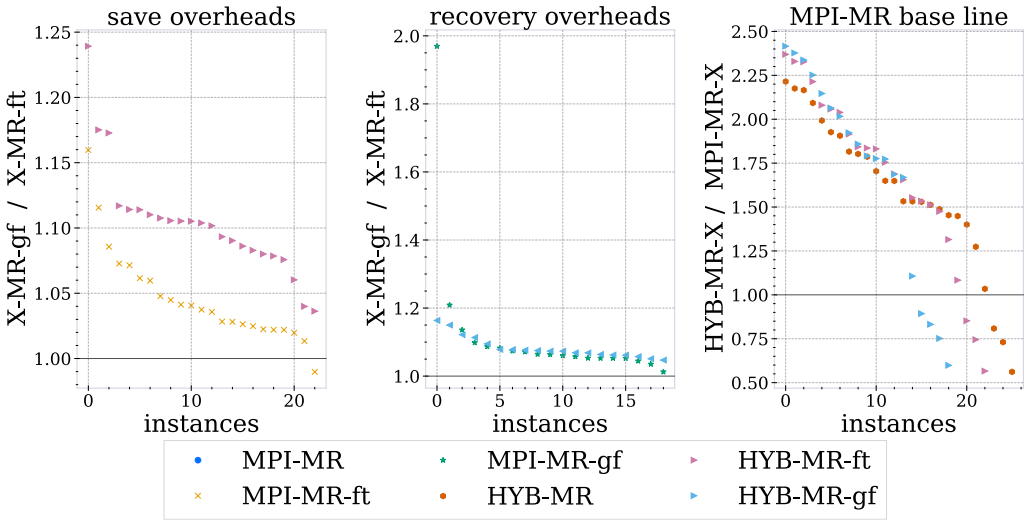


Figure 33: These plots contain the values of the `orkut` and `twitter` experiments (Figure 28) as well as the week scaling experiments (Figure 31). The x-axes are PageRank applied on the different data sets with their corresponding node numbers. We average over runs. We compare the different configurations to a base line and plot the results in descending order. In the following definitions we divide the total execution times of PageRank. The left plot shows the save self-message overheads on the y-axis ($\frac{\texttt{MPI-MR-ft}}{\texttt{MPI-MR}}$ and $\frac{\texttt{HYB-MR-ft}}{\texttt{HYB-MR}}$). We omit instances with no save self-messages (on 1 node). The left plot shows the recovery overheads on the y-axis ($\frac{\texttt{MPI-MR-gf}}{\texttt{MPI-MR-ft}}$ and $\frac{\texttt{HYB-MR-gf}}{\texttt{HYB-MR-ft}}$). We omit instances with no error generation. The right plot compares the MapReduce implementations with and without hybrid parallelization. The y-axis shows $\frac{\texttt{HYB-MR-X}}{\texttt{MPI-MR-X}}$, where `X` indicates a configuration in Table 1.

phase (`recovery`) is faster than the map phase (`map`), but it is slower than the remaining phases (Figures 29b, 30b, and 32). Furthermore, the recovery time for `rhg-d8-g3` with `HYB-MR-gf` decreases with increasing number of nodes. During the recovery, we have to recompute the map and reduce phase for a p-th of the key-value pairs (Section 7). Hence, the map and reduce phase should be p-times faster. Note that this is only the case if the data volume is large enough compared to the number of processes and the bottleneck workload. Furthermore, the redistribute phase (Section 7.3.1) requires expected $\mathcal{O}(k\frac{w+m}{tp_{old}^2} + p_{new} + \log(t))$ time, where k is the number of processes per node, p_{old} the number of processes before failure, p_{new} the number of processes after failure and t the number of threads per process.
Note that the recovery time has a constant runtime for the remaining configurations and graphs during our week scale experiments (Figures 32 and 50). The expected merge buffer time during recovery is $\mathcal{O}(\frac{w+m}{tp_{new}} + p_{new} + \log(t))$ (Section 7.3.2). This is p_{new} times slower than the remaining operations during the recovery. Moreover, the runtime of the different operation during recovery depends on the size of the processes data. If the data is too small compared to the number of used processes and bottleneck workload, we do not observe the optimal speedup, as discussed in Sections 6 and 7.
As during the Word Count experiments, `MPI-MR` has mostly the lowest runtime. Therefore, we compare the different MapReduce configurations to `MPI-MR` in the right plot in Figure 33. Notice that `HYB-MR` is at most 2.215 times slower than `MPI-MR` and on average 1.584 times. The configuration including saving self-messages `HYB-MR-ft` 2.215 times slower than `MPI-MR-ft` on average 0.86 times. Moreover, `HYB-MR-gf` is on average 0.789 times slower. The instances with negative values correspond to the `orkut` experiments, where our hybrid implementation performed better (Figure 28).
To sum up, the runtime of all our MapReduce configurations decreases with increasing number of nodes. We only see significant speedups if the bottleneck workload is small enough compared to data size and number of processes. Furthermore, we observe linear and even super linear speedups for the save self-message phase, while its overhead is larger for our hybrid implementation. The recovery phase scales if the bottleneck workload is small enough. All in all, the fault-tolerance mechanisms have faster execution times than the map phase.

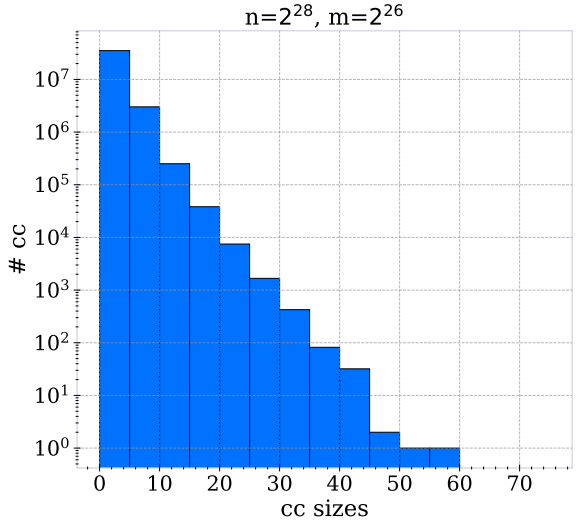


Figure 34: This histogram illustrates the number of connected components (logarithmic y-axis) for the different cc sizes (x-axis). The histogram contains bins with size 5. We apply the cc alternating MapReduce algorithm with `HYB-MR` to `ErdősRényi-d0.25-n`2^{28} with seed $s = 1$.

9.3. Connected Components

During the following experiments, we use the connected component (cc) algorithm (Section 4.3) to test the scalability of our framework and the fault-tolerance mechanism. We could only implement the two phase and alternating cc algorithms introduced in [57]. We could not implement the remaining algorithms due to the limitations of our frameworks. An optimized version of the MapReduce cc algorithm requires the neighbors of each node during the map phase. So would have to access and save the neighbors on locally on a process, which we would not be fault tolerant. Alternatively, we could attach the neighbors of a vertex to itself, similar to the procedure used during PageRank (Section 4.3). This would result in unnecessary data transfer during the shuffle and send self-message phase.
The main issue with our cc MapReduce implementations is that data sets with skewed nonuniform distributions lead to imbalances [57]. The large star MapReduce operation results in large degree vertices during the graph manipulation. Furthermore, the last small star operation gathers all the vertices of a cc at a representative vertex. This is a problem for many real world data sets [57]. For instance, the `orkut` graph (Table 4) consists of a single connected component [59]. Therefore, we generate graphs with a lot of small connected components.
We generate an `ErdősRényi-d0.25-n`2^{28} graph (Table 4) with $n = 2^{28}$ and $m = 2^{26}$ edges. Erdős-Rényi graphs are a well studied class of random graphs [40]. Let $G_{n,N(n)}$ be an Erdős-Rényi graph with n vertices and $N(n)$ edges. If $\frac{N(n)}{n} \xrightarrow{n\to\infty} c < \frac{1}{2}$, then the graph $G_{n,N(n)}$ converges to a graph with $n - N(n) + o(1)$ connected components, where the largest cc has a size in $\mathcal{O}(\log(n))$ [40]. For our random graph `ErdősRényi-d0.25-n`2^{28}, we have $\frac{m}{n} = \frac{1}{4} < \frac{1}{2}$. Therefore, we expect a graph with small connected components, with which our cc MapReduce algorithm should scale. Figure 34 illustrates the number and sized of connected components in `ErdősRényi-d0.25-n`2^{28}. The graph has a maximum cc with size 55. Most connected components are simple edges (64.08%) or consist of less than 10 vertices (99.22%). Since the bottleneck work load of our cc implementations (Section 4.3) is the size of the largest cc, our MapReduce framework should scale with those graphs.

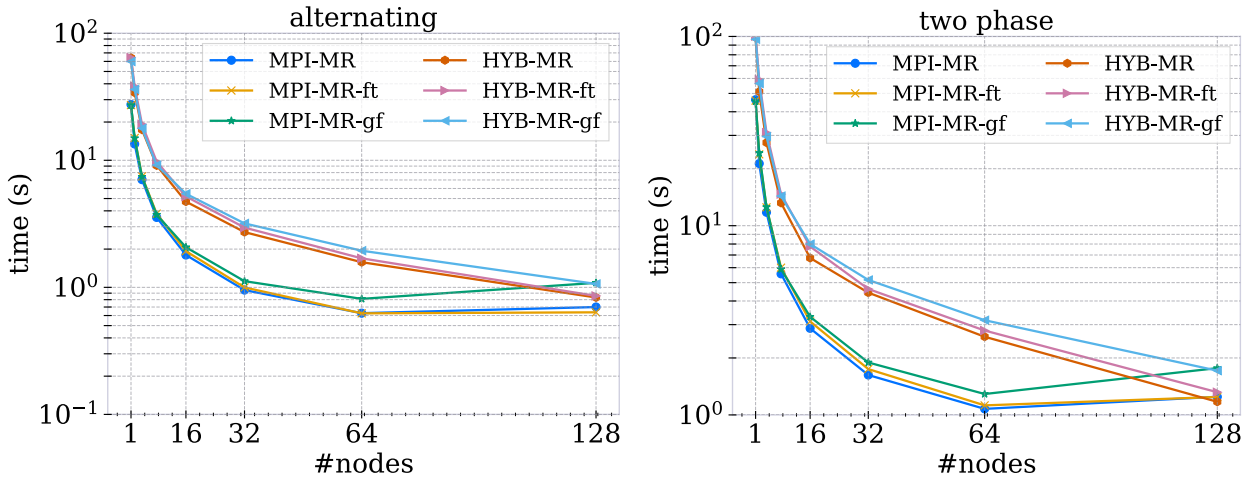
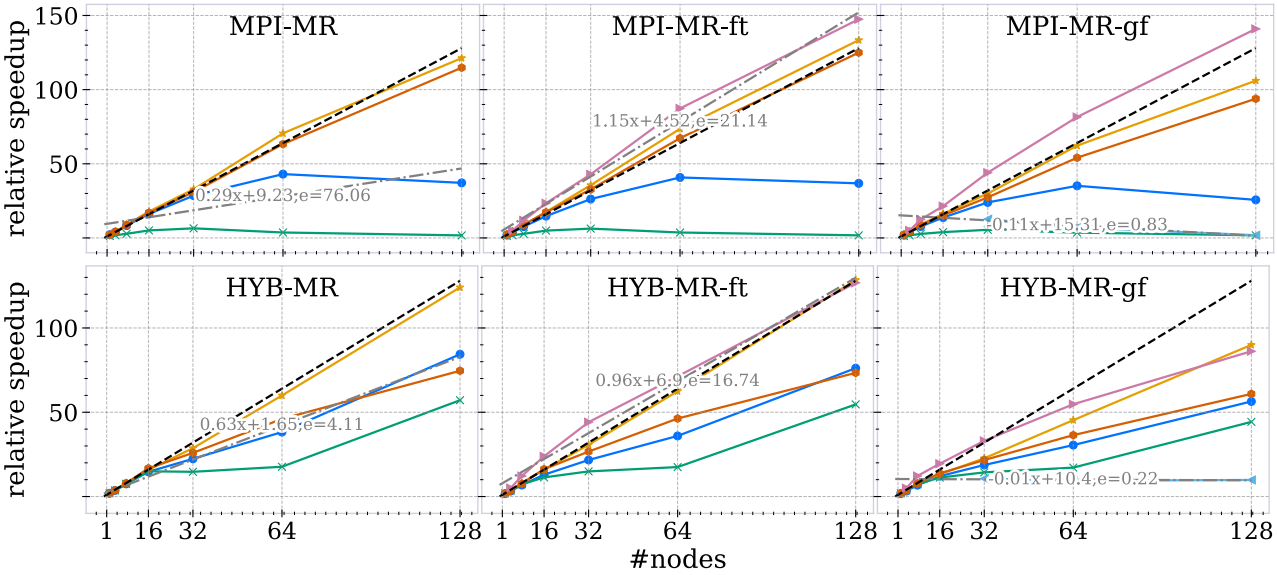
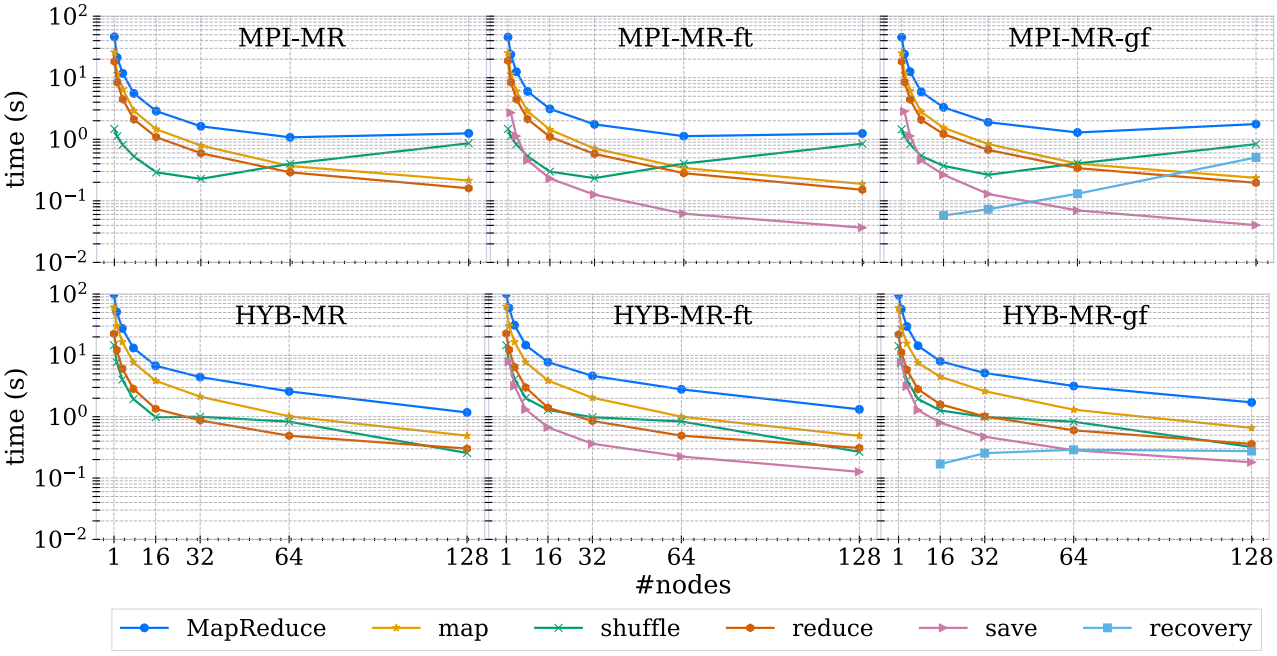


Figure 35: We execute the the alternating (left) and two phase (right) cc algorithm (Section 4.3) on `ErdősRényi-d0.25-n`2^{28} (Table 4) for seeds $s \in \{1, 2, 3, 4\}$ and nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. Table 13 illustrates the amount of large star and small star operations for the two phase cc algorithm. The alternating algorithm requires exactly 6 iterations. We generate 10% node failures. The x-axes contain the number of nodes, while the logarithmic y-axes show the runtime for the entire cc computation.

We apply the two phase and alternating cc MapReduce algorithm (Section 4.3) to `ErdősRényi-d0.25-n`2^{28} and perform strong scaling experiments (Figure 35). We compute the graph with our R-MAT MapReduce implementation (Section 4.4). Since each process requires a random number generator with a different seed, the graphs generated differ with the number of used processes. Hence, the number of large star and small star operations performed during the cc computation may differ. For the alternating algorithm with `MPI-MR(-ft/gf)` and `HYB-MR(-ft/gf)`, each cc computation requires 6 small star and large star operations. The two phase algorithm requires 14 to 18 large star and 4 or 5 small star MapReduce operations. The exact values can be found in Section C.3 in Table 13. The R-MAT graph computation required a single iteration. We first apply the cc algorithms without fault-tolerance and determine the number of large and small star operations needed during failure generation. This allows us to generate the failures uniformly. Since the algorithm consists of two different MapReduce operations, we illustrate the combined runtime instead of the runtime per operation.
Figure 35 illustrates the strong scaling experiments with the alternating and two phase cc MapReduce algorithms. Since both algorithms show a similar behavior, we add the plots (Figures 51 and 52) for the alternating algorithm in Section C.3. We consider the two phase cc algorithm. Note that the runtime decreases with increasing number of nodes until 64 nodes, while the time for `MPI-MR(-ft/gf)` increases for 128 nodes it decreases for `HYB-MR(-ft/gf)`. This is due to the shuffle time increase for `MPI-MR`, which is 3.423 times slower at node 128 ($0.484s$) then at node 32 ($0.141s$) (Figure 36b). We do not observe the same behavior for our hybrid MapReduce implementation. This matches the results of our previous experiments, which indicates that since `MPI-MR(-ft/gf)` uses 24 times more MPI processes, the `MPI_All_to_allv` call during the shuffle phase has scaling issues. Furthermore, the shuffle phase has nearly no speedup for `MPI-MR(-ft/gf)` (Figure 36a).
Moreover, the recovery time remains nearly constant for `HYB-MR-gf` with a speedup of 0.01 (Figure 36). Furthermore, the recovery time for `MPI-MR-gf` increases with a negative speedup of -0.11. Note that during these experiments, we do not visualize the recovery time for a single note failure as during our previous experiments. We plot the summed execution times. Since the number of failures increases linearly with the number of nodes, we would expect that the recovery time remains constant with increasing node number (Sections 6.3 and 7.3). The shuffle phase performed during the recovery distributes the saved data uniformly between the remaining processes. This `recovery shuffle` phase of `MPI-MR-gf` constitutes 83.19% of the total recovery time on 128 nodes (Figure 37). This recovery shuffle phase increases from $0.0038s$ on 16 nodes to $0.3400s$ on 128 nodes. We do not observe the same increase for `HYB-MR-gf`. This behavior matches that of the shuffle phase in Figure 36.
As observed during our previous experiments, the save self-message phase is faster than the other phases (Figure 36). Furthermore, we observe a super linear speedup of 1.15 for `MPI-MR-gf` and a near linear speedup of 0.96 for `HYB-MR-gf`. This matches the results of our previous experiments and the theoretical result in Section 5.2.2.
The following speedups are computed using the linear regression method of Section 8.6. Contrary to our previous examples, we observe nearly optimal speedups for the map (0.95) and reduce phase (0.90) of `MPI-MR-ft`. `HYB-MR-ft` has a speedup of 0.97 for the map and 0.57 for the reduce phase. Note that overall `MPI-MR` has a worse speedup, because of the bad scaling behavior of its shuffle phase as described above. `ErdősRényi-d0.25-n`2^{28} contains only small connected components, hence no large bottleneck workload as during the PageRank (Section 9.2) and Word Count (Section 9.1) experiments. This underlines our theoretical results in Sections 6 and 7, that our MapReduce library achieves linear speedups if the data volume is large enough compared to the number of processes and bottleneck workload. Large bottlenecks, on the other hand, lead to no speedup.



(a) The y-axes show the relative speedups. We plot a linear regression $ax + b, e$, where e is the mean square error. We compute the regression for the speedup of `MPI(HYB)-MR`, the save self-message phase (`save`) of `MPI(HYB)-MR-ft`, and the speedup for the recovery of `MPI(HYB)-MR-gf`.



(b) The logarithmic y-axes show the average runtime per MapReduce operation.

Figure 36: We perform strong scaling experiments with our MapReduce configurations (Table 1) for the two phase cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n`2^{28} for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages. We generate 10% node failures. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axes show the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

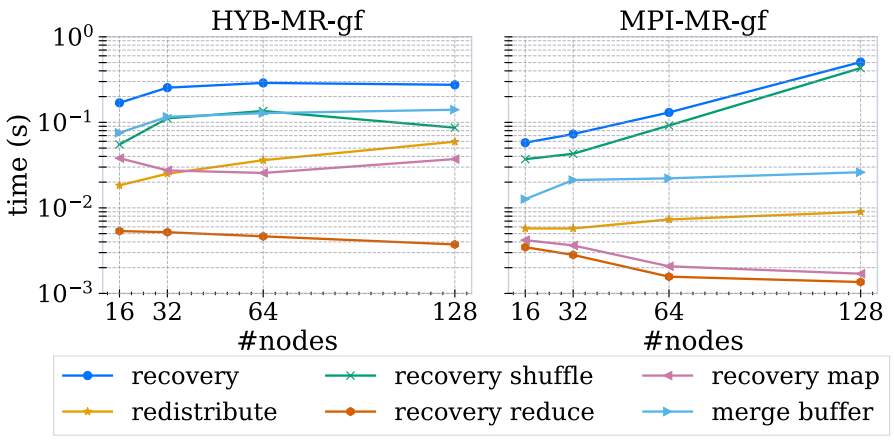


Figure 37: Illustration of the phases occurring during the recovery (Table 2). We show the results for our two phase cc algorithm applied on `ErdősRényi-d0.25-n`2^{28}. We generate 10% node failures. The x-axes contain the number of nodes, while the logarithmic y-axes show the runtime. These times correspond to the recovery time in Figure 36b

Note that the reduce phase of our hybrid parallelization has worse speedups than `MPI-MR`. This can be explained by the p-way merge, which we perform during the reduce phase of `HYB-MR(-ft/gf)` (Section 7.2). This results in an expected runtime of $\mathcal{O}\left(\log(p)\frac{w+m}{tp} + \log(t)\right)$, where p is the number of processes, t the number of threads, w the total workload, m the total memory used by the user-defined functions, $w \in \Omega(\bar{w}tp\log(tp))$, and $m \in \Omega(\bar{m}tp\log(tp))$. `MPI-MR(-ft/gf)` have an expected runtime of $\mathcal{O}\left(\frac{w}{p}\right)$, for $w \in \Omega(\bar{w}p\log(p))$. This additional $\log(p)$ factor could explain the speedup of 0.57 for the reduce phase.
Furthermore, we perform weak scaling experiments by generating Erdős-Rényi graphs with $n = x \cdot 2^{18}$ vertices and $m = x \cdot 2^{16}$ edges. As discussed above this results in graphs with

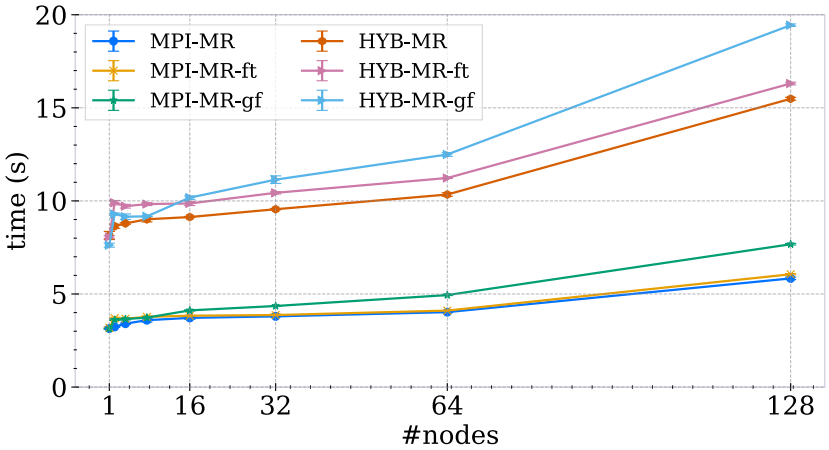


Figure 38: We perform weak scaling experiments with our MapReduce configurations (Table 1) for the alternating cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n`x for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages and include the standard deviation as error bar. For each node we generate 2^{18} vertices. Table 14 illustrates the amount of large star and small star operations. We generate 10% node failures. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axis shows the number of nodes, while the y-axis shows the runtime for the entire cc computation.

small connected components. The largest cc produced for our experiments has fewer than 100 vertices. For the alternating algorithm with `MPI-MR(-ft/gf)` and `HYB-MR(-ft/gf)`, each cc computation requires 6 small star and large star operations. The two phase algorithm requires 14 to 18 large star and 4 or 5 small star MapReduce operations. The exact values can be found in Section C.3 in Table 14. Since both algorithms have a similar behavior and the error bars are larger for the two phase algorithm on 32 node, we analyze the alternating algorithm in this section. Section C.3 contains the results for the two phase algorithm in Figures 53 and 54. In Figures 38 and 39, we generally observe the same behavior as during our PageRank (Section 9.2) and Word Count (Section 9.1) experiments. The fault-tolerance mechanism requires less time than the other phases and the map phase requires most time. Furthermore, the reduce phase is faster than the map phase and the shuffle time increases with the number of nodes. Additionally, the MapReduce configuration with no hybrid parallelization is faster (Figure 38).
The main difference between the PageRank and cc weak scaling experiments consists in how the execution times increase. During the PageRank experiments on `ErdősRényi-d30-n`x the runtime increases from 64 nodes to 128 by 5.98% for `MPI-MPI` and by 11.25% for `HYB-MPI` (Figures 31 and 50). During our alternating cc experiments, on the other hand, we observe an increase of 44.99% for `MPI-MPI` and 49.71% for `HYB-MPI` from 64 to 128 nodes. Furthermore, the `ErdősRényi-d30-n`x for 128 nodes and seed 1 has no connected component with a size greater than 70. Moreover, the map and reduce phases of `MPI-MPI` and map phase of `HYB-MPI`

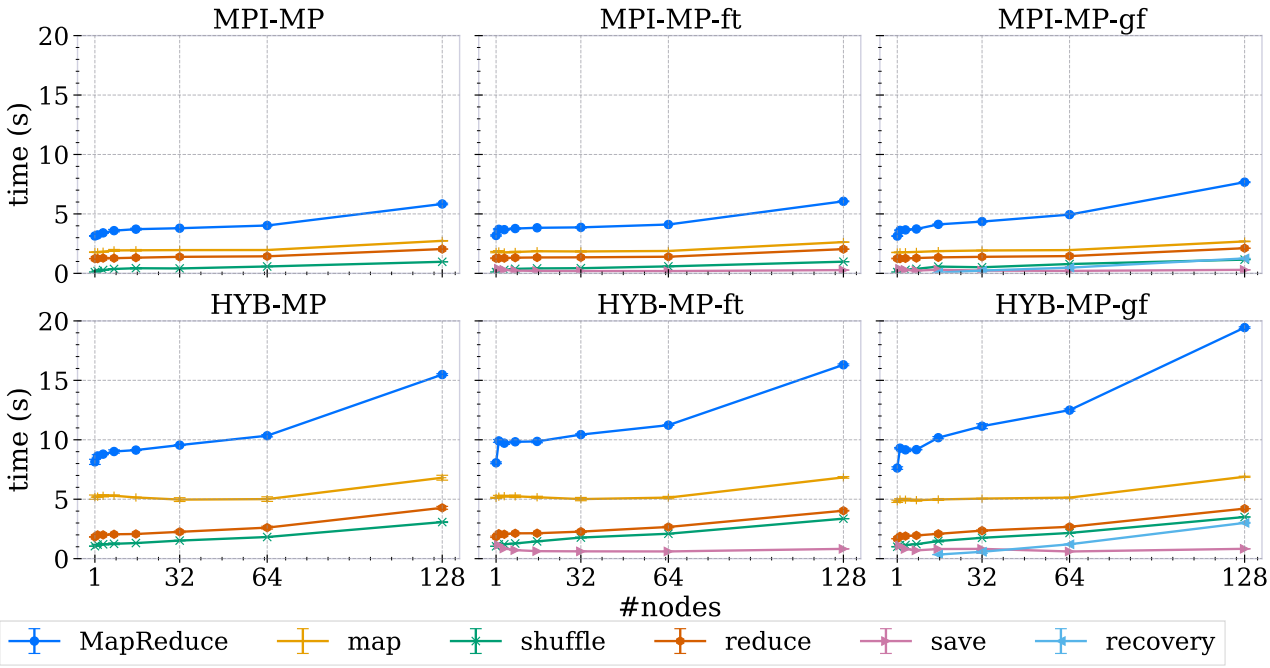


Figure 39: We perform weak scaling experiments with our MapReduce configurations (Table 1) for the alternating cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n`x for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages. For each node we generate 2^{18} vertices. We generate 10% node failures. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axes show the number of nodes, while the y-axes show the runtime for the entire cc computation. We illustrate the speedups of the map phase (`map`), reduce phase (`reduce`), shuffle phase (`shuffle`), save self-message time (`save`), and recovery time (`recovery`). `MapReduce` indicates the speedup of the entire MapReduce operation.

have at most an increase of 15% from 1 to 64 nodes. From 64 to 128 nodes, on the other hand, we observe a minimum increase of 35%.
During our weak scaling experiments, we increase the input linearly with the number of nodes. To achieve optimal speedups our MapReduce framework requires that the total workload w lies in $\Omega(\bar{w}p\log(p))$, where $\bar{w}$ is the bottleneck workload (Sections 6 and 7). Therefore, we require a more than linear workload increase to maintain the optimal speedups. We may have reached this point, where the input data is too small for the number of nodes.
We illustrate the save self-message overheads of `MPI(HYB)-MR-ft` in Figure 40 on the left. Similar to our PageRank experiments (Section 9.2) the overheads of `MPI-MR-ft` are smaller. For both configurations the maximum overhead is approximately 15%, where `MPI-MR-ft` has an average overhead of 5.6% and `HYB-MR-ft` n average overhead of 10.4%. Similar to our PageRank experiments, the overheads are larger for our hybrid parallelized MapReduce implementations. As described during the previous section, the save self-message phase of `MPI-MR-ft` has an expected 6 times faster runtime (Section 5.2.2) on our setup.
The center plot in Figure 40 shows the overheads of the failure recovery. Note that this includes the execution on fewer processes. For both `MPI-MR-gf` and `HYB-MR-gf`, the recovery has an overhead of 19.1%, while the maximum overhead of `MPI-MR-gf` is 70.6%. The average overhead of `HYB-MR-gf` is 36.7%. Furthermore, the MapReduce implementations with hybrid parallelization are approximately 2.3 times slower on average than the MapReduce implementation without (Figure 40). As discussed during the previous chapters, this may be caused

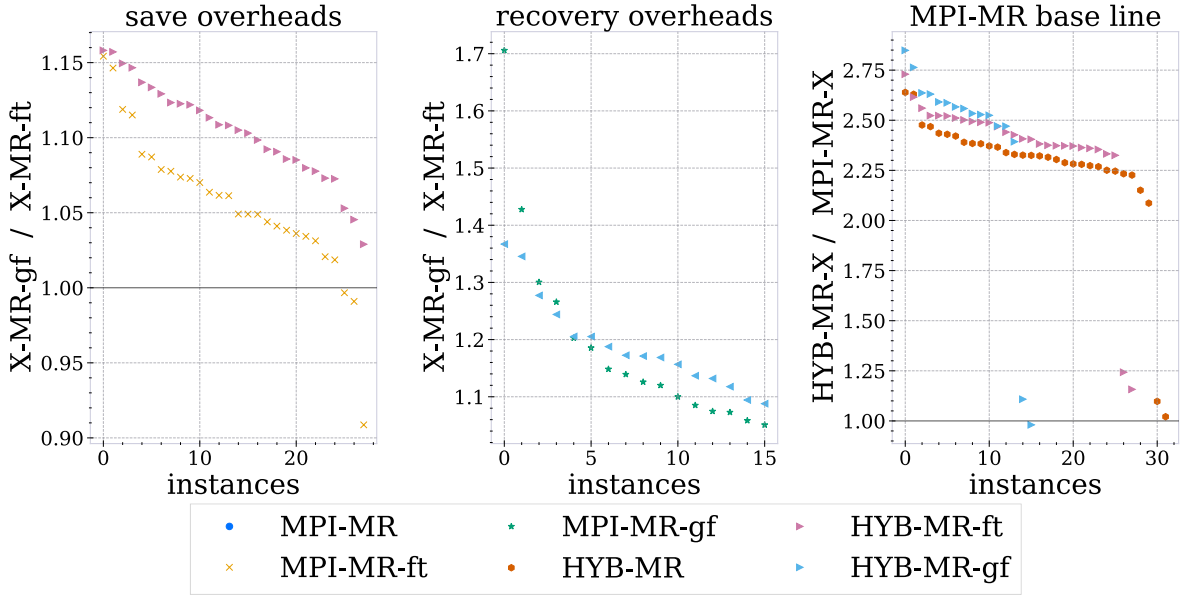


Figure 40: These plots contain the experiments performed in Section 9.3 (Figures 35, 38 and 53). The x-axes are both cc algorithms applied on the different data sets with their corresponding node numbers. We average over runs. We compare the different configurations to a base line and plot the results in descending order. During the following definitions, we use execution times of the cc algorithm. The left plot shows the save self-message overheads on the y-axis ($\frac{\texttt{MPI-MR-ft}}{\texttt{MPI-MR}}$ and $\frac{\texttt{HYB-MR-ft}}{\texttt{HYB-MR}}$). We omit instances with no save self-messages (on 1 node). The left plot shows the recovery overheads on the y-axis ($\frac{\texttt{MPI-MR-gf}}{\texttt{MPI-MR-ft}}$ and $\frac{\texttt{HYB-MR-gf}}{\texttt{HYB-MR-ft}}$). We omit instances with no error generation. The right plot compares the MapReduce implementations with and without hybrid parallelization. The y-axis shows $\frac{\texttt{HYB-MR-X}}{\texttt{MPI-MR-X}}$, where `X` indicates a configuration in Table 1.

by the reduce phase, which does not scale well. Moreover the map phase requires additional scans over the data and addition shared memory synchronization steps such as prefix sums (Section 7).
All in all, we achieve optimal speedups during our strong scaling experiments by choosing favorable graphs for the cc algorithm. The `MPI-MR(-ft/gf)` achieves faster results than our hybrid implementation. Moreover, we observe a smaller speedup for the hybrid parallelization, which may be caused by the p-way merge algorithm. The recovery time during our strong scale experiments is dominated by the recovery shuffle phase. This leads to a slower runtime for large number of nodes. We have nearly constant execution times for the map and reduce phases during our cc weak scale experiments, which increases for larger nodes, which may be caused ba our random static load balancing strategy. Finally, `MPI-MR-ft` and `MPI-MR-gf` have a lower save self-message overheads than `HYB-MR-ft` and `HYB-MR-gf`.

9.4. R-MAT

We test our MapReduce frameworks with an R-MAT MapReduce algorithm (Section 4.4), by computing random graphs with parameters $(a, b, c, d) = (0.57, 0.19, 0.19, 0.05)$. The Graph500 benchmark includes R-MAT graphs with those parameters [70]. For our strong scaling experiments we chose to generate a graph with $n = 2^{22}$ vertices and $m = 30 \cdot 2^{22}$. These graph sizes are similar to those generated for cc experiments by Kiveris et al. [57]. Since our R-MAT graph generator uses a different seed for the random number generator on each process, the graph generated varies with the number of nodes and processes. Furthermore, generating a failure results in a different graph than without failure generation. After a failure, the MapReduce framework redistributes the data and processes and generates the edges new on different processes with different seeds. This prevents us from determining the number of iterations before the execution and from distributing the failures uniformly over all MapReduce operations. We approximate the number of iterations with those observed during `MPI-MR-ft` and `MPI-MR-gf`. Contrary to the experiments with the cc MapReduce algorithm (Section 9.3), R-MAT repeats the same MapReduce operation. This allows us to visualize the average runtime of the different phases over all operations executed during a call of R-MAT.

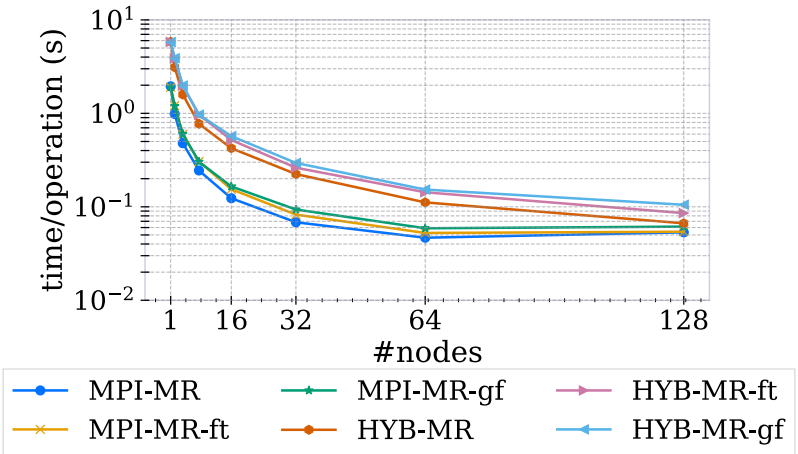
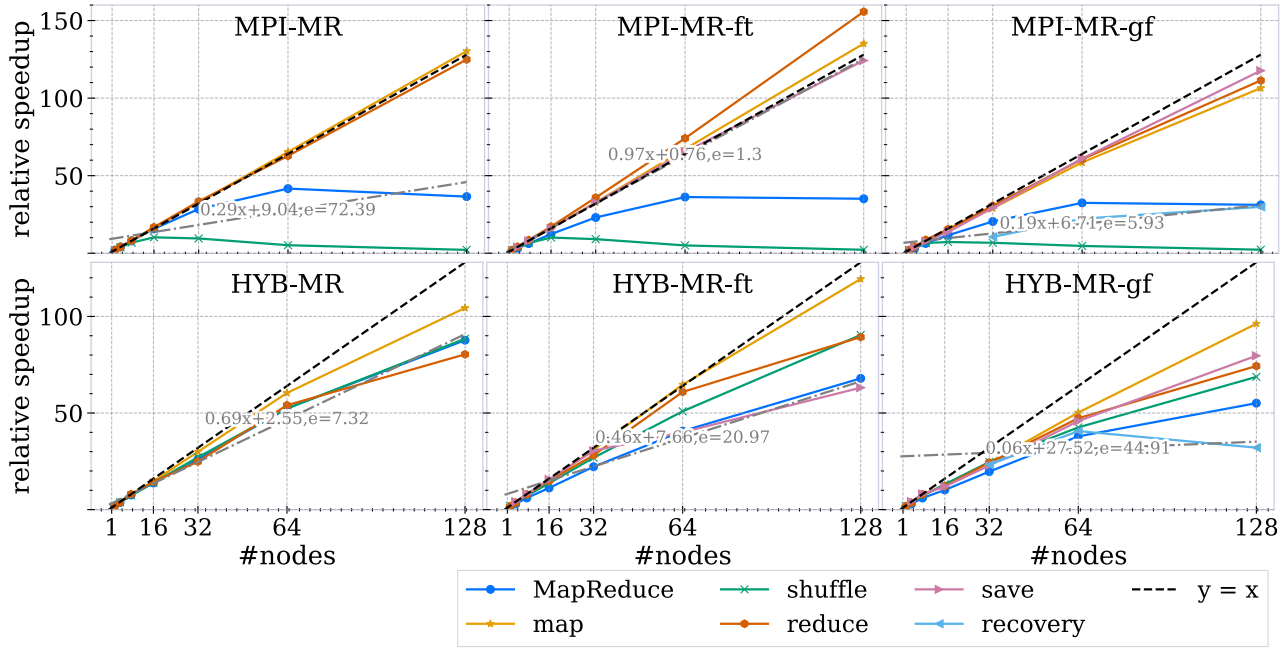
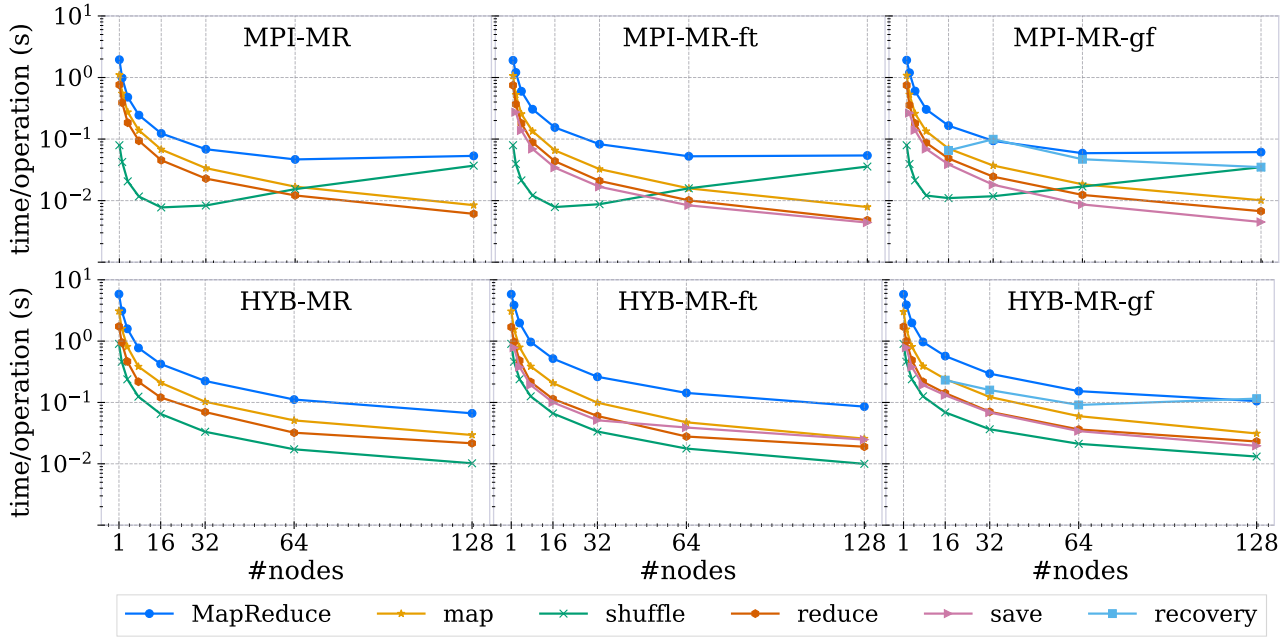


Figure 41: We generate an `rmat(0.57,0.19,0.19,0.05)`-$n2^{22}$-$d30$ graph (Table 22) with our R-MAT implementation (Section 4.4). We generate the graph on nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$ and seeds $s \in \{1, 2, 3, 4\}$. We plot the average times over all seeds. The x-axis contains the number of nodes, while the logarithmic y-axis shows the runtime per MapReduce operation.



(a) The y-axes show the relative speedups. We plot a linear regression $ax + b, e$, where e is the mean square error. We compute the regression for the speedup of `MPI(HYB)-MR`, the save self-message phase (`save`) of `MPI(HYB)-MR-ft`, and the speedup for the recovery of `MPI(HYB)-MR-gf`.



(b) The logarithmic y-axes show the runtime per MapReduce operation averaged over seeds

Figure 42: We generate the `rmat(0.57,0.19,0.19,0.05)`-$n2^{22}$-$d30$ graph, by using the R-MAT algorithm (Section 4.4) for seeds $\{1, 2, 3, 4\}$ and nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. We generate 10% node failures. The x-axes contain the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

We illustrate the results of our strong scaling experiments for R-MAT in Figure 42b. Overall, we observe the same behavior as during the cc experiments in Section 4.3. The runtime of or hybrid parallelization decreases with increasing number of nodes. The execution times of `MPI-MR`, on the other hand, decreases until 64 nodes and increases for 128 nodes. As during the cc experiments this is caused by the increase of shuffle time, which increases 4.768 times from nodes 16 ($7.75 \cdot 10^{-3} s$) to 128 nodes ($36.97 \cdot 10^{-3} s$). As discussed previously, this can be caused by scaling issues of the `MPI_All_to_allv` operation (Section 9.2 and 9.3). Note that we do not have the same issue for our hybrid implementation, which uses 24 times fewer processes.
We use a linear regression to determine the slops of speedups in Figure 42a (Section 8.6). We have nearly optimal speedups for the map (1.01) and reduce (0.97) phases of `MPI-MR`, but an execution time increases after 64 nodes due to the shuffle phase. Our hybrid parallelization, on the other hand, has an overall speedup of 0.69, it has a lower speedup for the map (0.82) and reduce (0.63) phases than `MPI-MR`. As explained in Section 9.3, this may be causes by the worse expected runtime of the p-way merge algorithm.
Furthermore, the save self-message time has only a speedup of 0.46 for `HYB-MR-ft` and is slower than the shuffle phase (Figure 42b). `MPI-MR-ft` also has a slower save time than shuffle time for node numbers smaller than 64 and lower speedups (0.97) than during previous experiments.
Let's consider and edge e during the R-MAT algorithm (Section 4.4). The map phase emits e and sends it to process i, which is determined by a hash value $h(e)$. During the reduce phase at process i, the user-defined reduce function generates a random edge for each duplicate of e and emits e. Since e is already at process i, the next map phase does not have to send e during the shuffle phase but during the save self-message phase. Therefore, the size of self-messages increases over time. After each MapReduce operation, we have to exchange large self-messages and smaller messages during the shuffle phase. This could explain the slower shuffle times compared to self-message times.
Finally, the recovery time does not decrease with increasing number of nodes, as observed during the previous experiments and has no linear speedup (Figure 42a). The time required for the merge buffer phase (Section 7.3) of `HYB-MR-gf` increases (Figure 43). The merge buffer phase on 128 nodes constitutes 82.0% of the entire recovery. For `MPI-MR-gf` the recovery shuffle phase requires most of the execution time followed by the redistribution and merge buffer

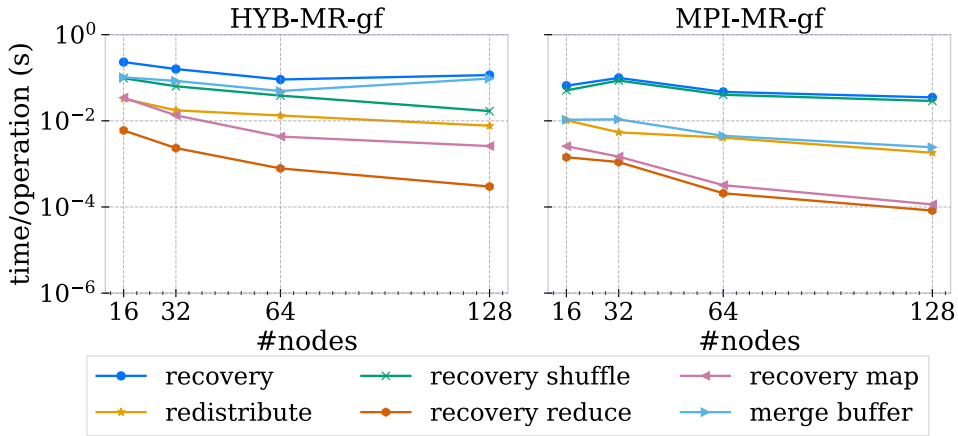


Figure 43: Illustration of the phases occurring during the recovery (Table 2). We show the runtime of the R-MAT algorithm applied on `rmat(0.57,0.19,0.19,0.05)`-$n2^{22}$-$d30$. We compute the graph with `HYB-MR-gf` (left) and `MPI-MR-gf` (right). The x-axes show the number of nodes, while the logarithmic y-axes show the runtime per recovery phase. These times correspond to the recovery time in Figure 42b

phases (Section 6.3). Note that the recovery phase works on n times fewer elements than the other phases, where n is the number of nodes. We have to recover the data lost during the failure of one node. Since all our algorithms use random static load balancing, we only achieve the optimal expected runtime, if the total workload is large enough compared to the number of processes and bottleneck workload (Sections 6 and 7). Scaling issues due to a small total workload could explain the behavior of the recovery phase.
We perform weak scaling experiments for our R-MAT implementation, by generating a graph with 2^{18} vertices per compute node, an average degree of 30, and parameters $(a, b, c, d) = (0.57, 0.19, 0.19, 0.05)$. As explained previously, we cannot set a fixed number of iterations, hence we display the average runtime per MapReduce operation. Furthermore, we use the number of iterations needed for `MPI-MR-ft` or `HYB-MR-ft` do distribute the node failures between the MapReduce operations in `MPI-MR-gf` or `HYB-MR-gf`.
The MapReduce implementation without hybrid parallelization is faster (Figure 44). As observed during the cc experiments in Section 9.3, the runtime increases slightly with increasing number of nodes. For our MapReduce framework without hybrid parallelization `MPI-MR`, the shuffle time gets 7.97 times slower from 1 to 128 nodes, while the map phase increases by 19.4% and reduce phase by 27.9% (Figure 45). Our MapReduce framework with hybrid parallelization `HYB-MR`, on the other hand, has a slight increase for the map (11.1%) and shuffle (17.5%) phases. The reduce phase has an increase of 51.5%. As before, the behavior of `MPI-MR(-fg/gf)` can be explained by scaling issues of the MPI calls during the shuffle phase. The increasing reduce phase can be explained by the scaling issues of the p-way merge as described in Section 9.3.
The recovery time is faster than an execution of a MapReduce operation (Figure 45). But in contrast to previous experiments the recovery phase is slower than the map phase. Furthermore, the runtime of the map phase at 16 nodes for `HYB-MR-gf` (Figure 45) increases but remains constant after this first increase. This increase is due to the loss of a compute note. After the 16 nodes, the number of failures and nodes increase proportionally, which explains no further rise of the map phase.

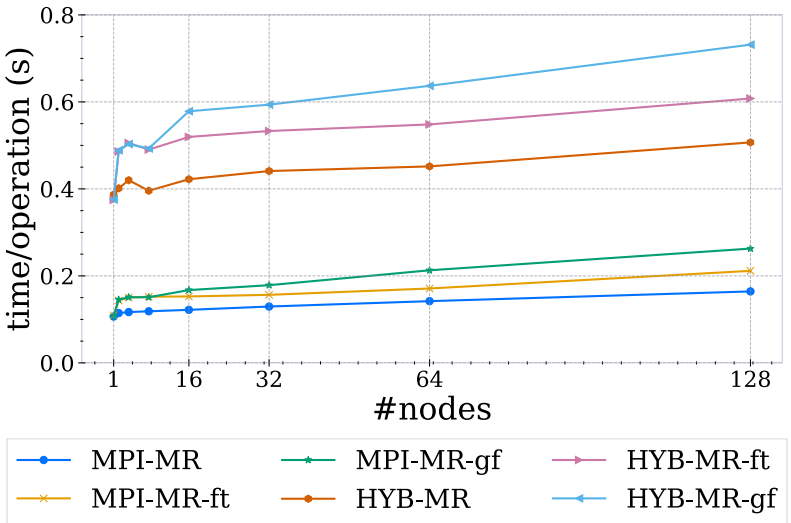


Figure 44: We generate a `rmat(0.57,0.19,0.19,0.05)-n`x`-d30` graph (Table 22) with our R-MAT implementation (Section 4.4). For each node we generate 2^{18} vertices. We compute the graph on nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$ and seeds $s \in \{1, 2, 3, 4\}$. We plot the average times over all seeds. The x-axis contains the number of nodes, while the y-axis shows the runtime per MapReduce operation.

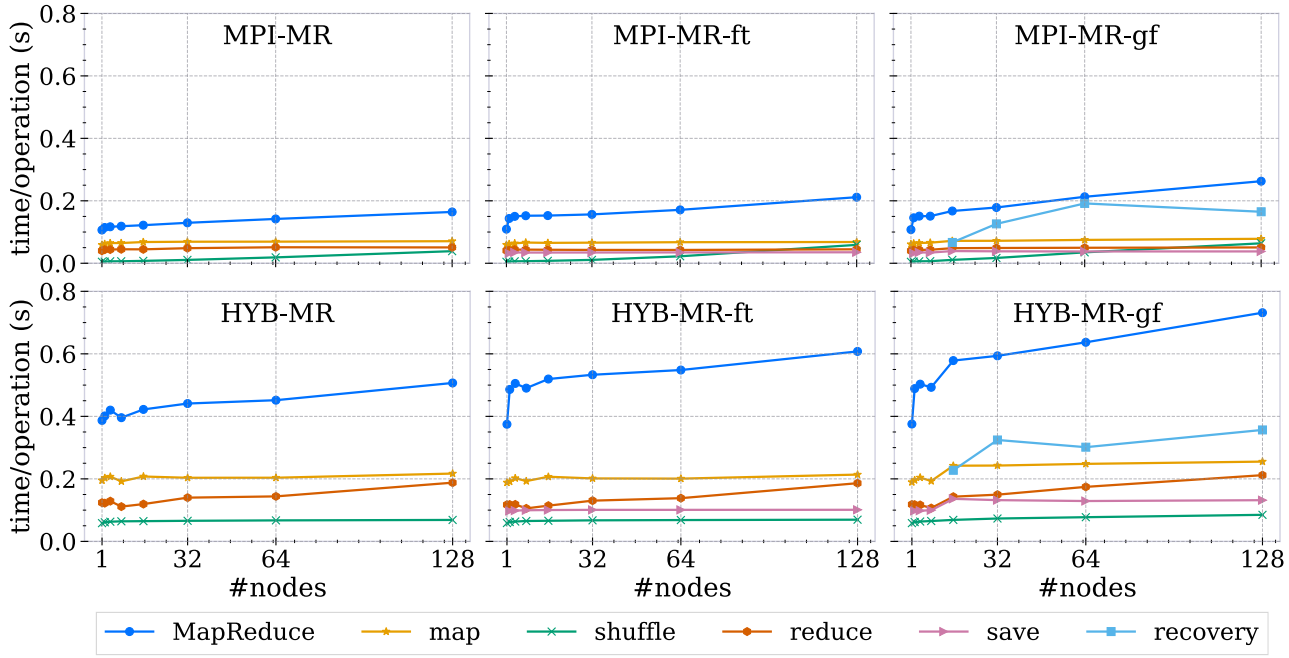


Figure 45: We perform weak scaling experiments with our MapReduce configurations (Table 1) for the R-MAT algorithm (Section 4.4). We run the experiments on `rmat(0.57,0.19,0.19,0.05)-n`x`-d30` for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages. For each node we generate 2^{18} vertices. We generate 10% node failures. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axes contain the number of nodes, while the y-axes show the runtime per operation. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

Figure 46 (left) shows the overheads of the save self-message phase. Contrary to previous experiments we observe higher overheads for `MPI-MR-ft` and `HYB-MR-ft`. For the hybrid MapReduce implementation, we have an average overhead of 22.86%, which is approximately 2 times slower than the overheads during our cc experiments (Section 9.3). Furthermore, `MPI-MR-ft` has an average overhead of 22.07%, which is approximately 4 times slower than the average overhead for the cc algorithm. Moreover, the recovery overhead in Figure 46 (center) is lower compared to our cc experiments. We have an average recovery overhead of 14.81% for `MPI-MR-ft` and 13.95% for `HYB-MR-ft`. In Figure 46 (right), we compare our MapReduce implementations with and without hybrid parallelization. As during our previous experiments, the hybrid implementation is slower. As explained in Section 9.2 this can be caused the additional time needed to perform the t-way merge and prefix sum during the map phase (Section 7).
To sum up, we could observe the same shuffle time increase for `MPI-MR` as during the cc experiments. Furthermore, we have nearly optimal speedups for the map and reduce phase of `MPI-MR` and the map phase of `HYB-MR`. As observed during the cc experiments, the reduce phase of `HYB-MR` has no optimal speedup, which may be caused by the p-way merge.

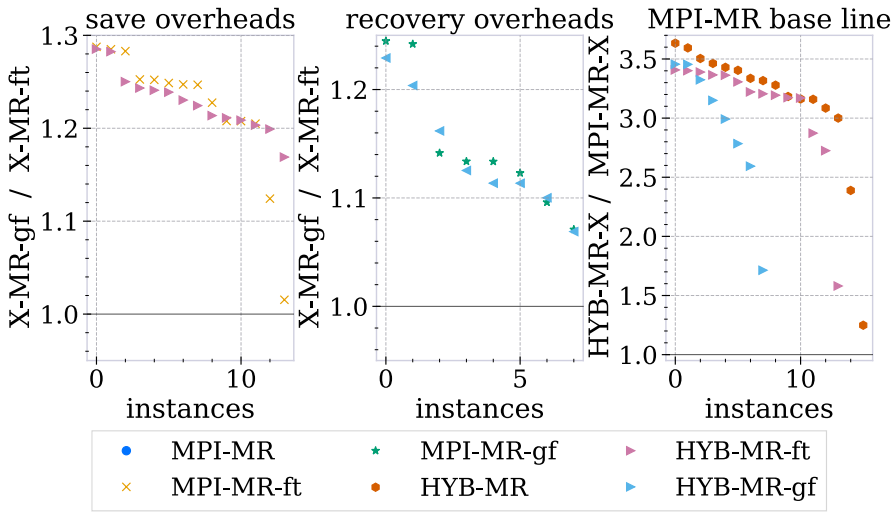


Figure 46: These plots contain the experiments performed in Section 9.4 (Figures 41 and 45). The x-axes are both R-MAT algorithms applied on the different data sets with their corresponding node numbers. We average over runs. We compare the different configurations to a base line and plot the results in descending order. During the following definitions, we use execution times of the R-MAT algorithm. The left plot shows the save self-message overheads on the y-axis ($\frac{\texttt{MPI-MR-ft}}{\texttt{MPI-MR}}$ and $\frac{\texttt{HYB-MR-ft}}{\texttt{HYB-MR}}$). We omit instances with no save self-messages. The left plot shows the recovery overheads on the y-axis ($\frac{\texttt{MPI-MR-gf}}{\texttt{MPI-MR-ft}}$ and $\frac{\texttt{HYB-MR-gf}}{\texttt{HYB-MR-ft}}$). We omit instances with no error generation. The right plot compares the MapReduce implementations with and without hybrid parallelization. The y-axis shows $\frac{\texttt{HYB-MR-X}}{\texttt{MPI-MR-X}}$, where `X` indicates a configuration in Table 1.

9.5. Summary

In this section we summarize our experimental results and observations from Sections 9.1 to 9.4. First of all, we notice an increase in shuffle time during our strong scaling experiments for Word Count, cc and R-MAT with `MPI-MR(-ft/gf)`. This increase occurs after 32 nodes and 1536 processes. Note `HYB-MR(-ft/gf)` employs 24 times fewer processes, executes the same shuffle method, and does not suffer from an increased shuffling. This behavior could be explained by the scaling issues of the collective irregular `MPI_All_to_allv` operation [18, 19].
Furthermore, we do not observe optimal speedups for our real world data sets during the PageRank and Word Count experiments. This is due to a large bottleneck workload, which constitutes a well-known problem for MapReduce algorithms on large graphs [57, 62]. By modifying and extending the MapReduce frameworks, such problems can be solved. For instance, process local data structures on which each user-defined map function on the same process has access could be used to optimize MapReduce algorithms [57, 62]. We could not employ these techniques, because those data structures are not fault-tolerant.
On the other hand, if the total workload is large enough compared to the number of processes and bottleneck workload, then the non hybrid parallelization has approximately an optimal speedup of 1 for the map and reduce phase. We observe this during our R-MAT and cc experiments. Furthermore, the map phase of our hybrid parallelization has the same behavior. This matches the theoretical expected execution times of Sections 6 and 7. Note that the reduce phase of `HYB-MR` has no linear speedups (0.63 for R-MAT and 0.57 for cc). This can be explained by the p-merge used during its reduce phase, which results in a worse time complexity (Section 7). Since `MPI-MR` uses a radix sort for the group by key during the reduce phase we have a better expected time complexity (Section 6).

Table 7: Comparison of our MapReduce implementations with and without hybrid parallelization. The following values correspond to the minimum, maximum and average values of Figure 47 (right).

MapReduce	minimum	maximum	average
HYB-MR	0.562	3.634	2.213
HYB-MR-ft	0.566	3.407	2.278
HYB-MR-gf	0.599	3.456	2.184

In Figure 47 (right), we compare HYB-MR(-ft/gf) to its corresponding non hybrid parallelized MapReduce implementation MPI-MR(-ft/gf). During all our experiments, except for PageRank on twitter (Figure 28), our hybrid parallelization is slower. HYB-MR is at worst 3.634 times slower and at best 1.779 times faster than MPI-MR (Table 8). On average HYB-MR is 2.213 slower. As described in previous sections, this is due to the additional scans, t-way merge and prefix sums necessary to implement our shared memory parallelization.
We analyze the overheads of 10% failure generation and recovery of MPI(HYB)-MR-gf compared to MPI(HYB)-MR-ft in Figure 47 (center). MPI-MR-gf has an average overhead of 15.2% and a maximum of 97.0%, while HYB-MR-gf has an average overhead of 13.4% (Table 8). We have observed that the shuffle phase necessary to redistribute the key-value pairs during the recovery phase does not scale for MPI-MR-gf (Section 9.3).
The overheads for our save self-message phase depend on the MapReduce algorithm. For instance, we observe an average overhead of approximately 2% during our Word Count experi-

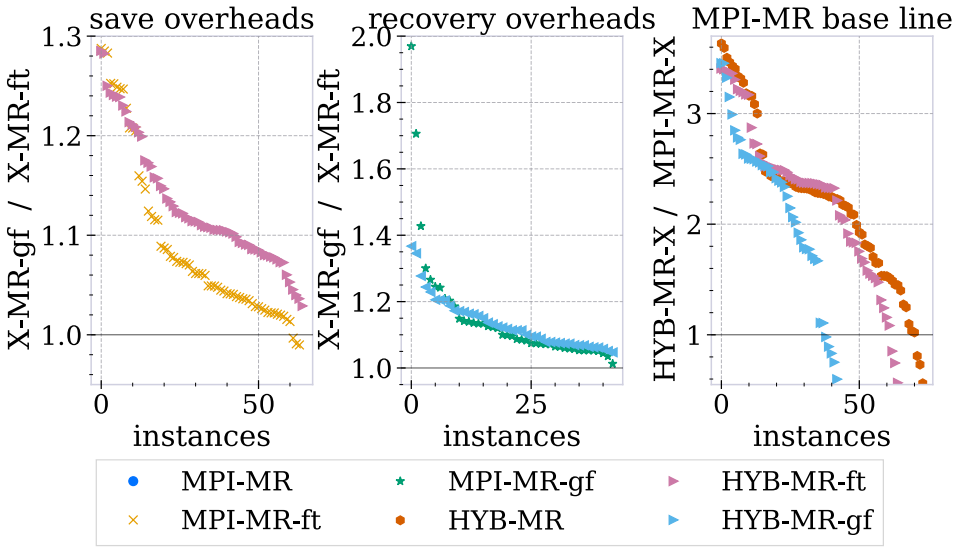


Figure 47: These plots contain all experiments performed in Section 9. The x-axes show the different algorithms executed on different data sets and node counts. We average over runs. We compare the different configurations to a base line and plot the results in descending order. During the following definitions, we use execution times of the R-MAT algorithm. The left plot shows the save self-message overheads on the y-axis ($\frac{\texttt{MPI-MR-ft}}{\texttt{MPI-MR}}$ and $\frac{\texttt{HYB-MR-ft}}{\texttt{HYB-MR}}$). We omit instances with no save self-messages. The left plot shows the recovery overheads on the y-axis ($\frac{\texttt{MPI-MR-gf}}{\texttt{MPI-MR-ft}}$ and $\frac{\texttt{HYB-MR-gf}}{\texttt{HYB-MR-ft}}$). We omit instances with no error generation. The right plot compares the MapReduce implementations with and without hybrid parallelization. The y-axis shows $\frac{\texttt{HYB-MR-X}}{\texttt{MPI-MR-X}}$, where $\texttt{X} \in \{\texttt{""}, \texttt{"ft"}, \texttt{"gf"}\}$.

Table 8: This table contains the minimum, maximum and average sent self-message and recovery overheads. The following values correspond to the left and center Figures 47.

MapReduce	minimum	maximum	average
`MPI-MR-ft`	0.909	1.288	1.088
`HYB-MR-ft`	1.029	1.285	1.131
`MPI-MR-gf`	1.013	1.970	1.152
`HYB-MR-gf`	1.047	1.367	1.134

ments, while we have an overhead of approximately 22% for R-MAT. As described in Section 9.4, this largely depends on the size of self-messages and the data, which a process does not need to send. Note that the overheads of `MPI-MR-ft` are lower than those of our hybrid parallelization (Figure 8 (left)). Overall our experiments the save self-message phase has an average overhead of 8.8% for `MPI-MR-ft` and 13.1% for `HYB-MR-ft`. Furthermore, during all our experiments we observe super linear speedups for the save self-message phase, which matches its theoretical expected runtime of $\mathcal{O}\left(\frac{m}{p^2}\right)$, where the sum of all exchanged message sized m is large enough compared to the number of processes p (Section 5.2.2).

10. Discussion

10.1. Conclusion

This thesis introduces an in-memory fault-tolerant MapReduce framework for iterative algorithms, designed to run on high performance computing (HPC) systems. We describe a fault-tolerance mechanism to handle compute node failures by sending additional messages during the shuffle phase. Furthermore, we implement a purely MPI based and a hybrid parallelized MapReduce framework and prove their expected runtime.
We implement a Word Count, a PageRank, a connected component, and a R-MAT MapReduce algorithm in our framework and perform runtime experiments. Since we use random static load balancing, we only observe optimal speedups if the input data is large enough compared to the number of processes and the bottleneck workload. Moreover, the purely MPI based implementation has on average a 2.21 times shorter runtime, but suffers from a performance loss during the shuffle phase for more than 64 nodes. Since the hybrid MapReduce implementation requires 24 times less processes, it does not suffer from MPI scaling issues.
Moreover, the recovery from node failures scales if the input data is large enough and its runtime depends on the benchmark algorithm. While generating 10% node failures, we observe an average overhead of 15.2% for our purely MPI based and 13.4% for our hybrid MapReduce implementation.
Furthermore, the phase sending and saving self-messages scales well with increasing number of processes and is faster than the other phases. We observe super linear speedups. This matches its theoretical runtime of $\mathcal{O}\left(\frac{m}{p^2}\right)$, where p is the number of processes and m the large enough size of all messages. For our purely MPI based implementation saving self-messages has an overhead of 8.8% on average. For our hybrid MapReduce framework this overhead is 13.1%.

10.2. Future Work

First of all, an optimized and well scaling MapReduce algorithm on real world graphs requires extensions to the MapReduce interface [57, 62]. One could introduce fault-tolerant data structures, which a user-defined map or reduce function could access locally on each process. Furthermore, a MapReduce operation for an associative user-defined reduce function could minimize the data sent during the shuffle phase. The map phase could apply the reduce function locally before the shuffle phase, similar to the `ReduceByKey` operation in Thrill [22].
Moreover, one could adopt a parallel radix sort during the reduce phase of the hybrid MapReduce implementation to address its scaling issues. This approach scales well for the purely MPI based implementation.
Furthermore, we currently use random static load balancing to distribute the workload between processes, which only achieves optimal speedups if the input data is large enough compared to the number of processes and bottleneck workload. Our framework could benefit for instance from a work stealing algorithm to better distribute the workload [86]. Note that one would have to modify the fault-tolerance mechanism.
Additionally, one could support more than one compute node failure, by periodically saving checkpoints of the entire MapReduce state to a fault-tolerant file system. Since these checkpoints would be costly, the MapReduce framework would perform them rarely. In case of a multi node failure, one could restart from the checkpoint and would not have to start the computation from the beginning.
Furthermore, it was not possible to compare our framework to other fault-tolerant MapReduce frameworks. Most libraries introduced in Section 3.3 are not designed to run on HPC systems.

The FT-MRMPI library (Section 3.3.4) is the most similar to our framework, but the code was not available and we were not able to contact the authors. However, an efficient implementation of FT-MRMPI would be a good comparison for our framework.
Finally, the fault-tolerant MPI library ULFM did not run stably on the SuperMUC-NG system, which forced us to simulate failures. A possible solution would be to use a different fault-tolerant MPI implementation and perform fault-tolerance experiments by killing MPI processes via system calls. Alternatively, one could run experiments on an HPC system on which ULFM works reliably.

A. Fundamentals: Balls in Bins

This section contains the proof of Lemma 2.2 in Lemma A.1. We use the same notations and definitions as introduced in Section 2.4.

Lemma A.1. *Given the balls in bins problem with m balls and p bins with expected maximum occupancy $b(m,p)$, $m \in \mathbb{N}$ and $p \in \mathbb{N} \setminus \{1,2\}$. If $m \in \Omega(p\log(p))$, then $b(m,p) \in \mathcal{O}\left(\frac{m}{p}\right)$.*

Proof. Let log be the logarithmus dualis, $m \in \Omega(p\log(p))$, and $\alpha > 1$. There is a $\beta > 0$ so that $m \geq \beta p \log(p)$ and $\frac{m}{\beta p \log(p)} \geq 1$, since $p \geq 3$. Furthermore, we have $\log\log(p) \geq 0$ and $\log(p) \geq \log\log(p)$, therefore $\left(1 - \frac{1}{\alpha}\frac{\log\log(p)}{2\log(p)}\right)$ lies in $[0,1]$ and we can deduce:

$$\alpha\sqrt{\frac{2m\log(p)}{p}\left(1 - \frac{1}{\alpha}\frac{\log\log(p)}{2\log(p)}\right)} \leq \alpha\sqrt{\frac{2m\log(p)}{p}} \leq \alpha\sqrt{\frac{m}{\beta p\log(p)}}\sqrt{\frac{2m\log(p)}{p}} \leq \alpha\sqrt{\frac{2}{\beta}}\frac{m}{p}.$$

So the variable k_α in Theorem 2.1 lies in $\mathcal{O}\left(\frac{m}{p}\right)$. Let M be a random variable counting the maximum number of balls in a bin. Then $P(M \leq k_\alpha)$ indicates the probability that the maximum number of balls in a bin lies in $\mathcal{O}\left(\frac{m}{p}\right)$, if $m \in \Omega(p\log(p))$. By applying Theorem 2.1 [78] we can deduce:

$$P(M \leq k_\alpha) = 1 - P(M > k_\alpha) = 1 - o(1).$$

Finally, we prove Lemma 3.1 with

$$b(m,p) = \sum_{i=0}^{m} i \cdot P(M=i) \leq k_\alpha P(M \leq k_\alpha) + mP(M > k_\alpha) = k_\alpha(1-o(1)) + mo(1) \in \mathcal{O}\left(\frac{m}{p}\right).$$

□

B. MapReduce Benchmark Algorithms: PageRank

We introduce an alternative PageRank implementation to Section 4.2 using 3 different MapReduce operations. We can implement the PageRank without sending the outgoing neighbors $O(a)$ of a page a explicitly. Algorithm 25 simulates Algorithms 4 and 5 by chaining three different MapReduce operations. The input of this algorithm is a list of triples $(a, b, p(a))$, where $(a,b) \in E$ and $p(a)$ is the current page rank of a. So instead of working on a graph, where each vertex has an adjacency list as before, we use the edge list E. The map and reduce phases applying the user-defined functions Map1, Reduce1, and Map2 correspond to the map phase of Algorithm 4. Then we apply Equation 4.1 in Reduce2. Finally, during the last MapReduce operation applying Map3 and Reduce3, we send the new page rank $p(a)$ to each triple corresponding to edge (a,x) for $x \in V$.
Note that the advantage of the PageRank Algorithm 25 is that we do not need to send the neighbor vertices explicitly, which leads to equally sized input, output and intermediate elements. On the other hand, we need recompute the neighbors each time by using Map1 and Reduce2. Algorithm 25 requires more MapReduce operations, which can lead to longer running times. An advantage of PageRank with Algorithms 4 and 5 is that the outgoing neighbors $O(a)$ are usually not sent during the shuffle phase.

Algorithm 25: PageRank: Edge List Implementation

```
1  Map1(a, b, p(a))                                   // emit triples with a as key
2  |  emit(a, (b, p(a)))
3
4  Reduce1(a, values)                                 // aggregate outgoing neighbors
5  |  foreach (b, p(a)) ∈ values do  emit((a, b, p(a)/|values|))
6
7  Map2(a, b, p(a))                                   // emit triples with b as key
8  |  emit(b, (a, p(a)))
9
10 Reduce2(b, values)                                 // apply Equiation 4.1
11 |  p(b) = (1−α)/N + α · Σ_{(a,p(a))∈values∧p(a)≠∞} x
12 |  foreach (a, p(a)) ∈ values do  emit((a, b, ∞))
13 |  emit((b, ∞, p(b)))
14
15 Map3(a, b, p(a))                                   // emit triples with a as key
16 |  emit(a, (b, p(a)))
17
18 Reduce3(a, values)                                 // save p(a) at each triple corresponding to edge (a, b)
19 |  p(a) = p, with (∞, p) ∈ values ∧ p ≠ ∞
20 |  foreach (b, ∞) ∈ values do  emit((a, b, p(a)))
21
```

C. Experimental Evaluation

C.1. Word Count

In this section we provide additional plots and tables complementing the results of Section 9.1. Figures 48 and 49 contain the results of strong scale experiments on the `english` and `yelp` data sets (Table 3). These figures correspond to Figure 24 and provide the same results as discussed in Section 9.1. Furthermore, Tables 9 and 10 contain the average percentages of the MapReduce phases during an MapReduce operation.

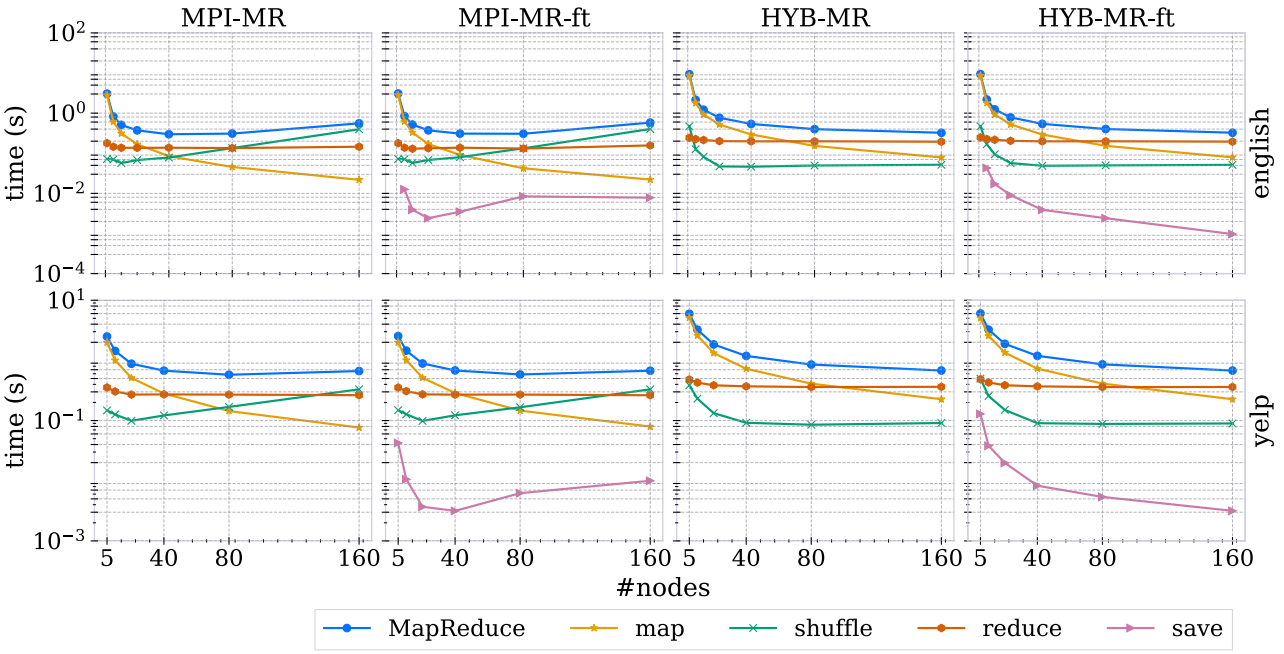


Figure 48: We illustrate the runtime division of the MapReduce operation during the Word Count algorithm (Section 4.1). We apply the Word Count algorithm on `english` (nodes in $\{1, 5, 10, 20, 40, 80, 160\}$, 7 runs) and `yelp` (nodes in $\{5, 10, 20, 40, 80, 160\}$, 7 runs). The x-axes show the number of nodes, while the logarithmic y-axes show the average runtime. We profile the runtime of the map phase including serialization and group by key (`map`), reduce phase including deserialization (`reduce`), shuffle phase (`shuffle`), and save self-message time (`save`). `MapReduce` indicates the runtime of the entire MapReduce operation.

Table 9: This table shows the relative average MapReduce phase times r_x of the different MapReduce phases during the Word Count experiments with MPI-MR-ft (Section 9.1). We compute $r_x = \frac{t_x}{t_{mr}} \cdot 100\%$, where t_x is the average execution time of phase x and t_{mr} over all runs and node counts.

text	`map`(%)	`reduce`(%)	`shuffle`(%)	`save`(%)
`english`	68.04	16.66	14.65	0.65
`yelp`	58.93	25.42	14.53	1.11
`gutenberg`	54.33	30.54	14.16	0.97
`randtext-gutenberg`	68.06	20.68	9.85	1.41
average	62.81	24.03	11.98	1.18

Table 10: This table shows the relative average MapReduce phase times r_x of the different MapReduce phases during the Word Count experiments with HYB-MR-ft (Section 9.1). We compute $r_x = \frac{t_x}{t_{mr}} \cdot 100\%$, where t_x is the average execution time of phase x and t_{mr} over all runs and node counts.

text	`map`(%)	`reduce`(%)	`shuffle`(%)	`save`(%)
`english`	84.49	10.09	4.88	0.54
`yelp`	74.11	17.21	7.21	1.46
`gutenberg`	67.95	23.16	7.45	1.44
`randtext-gutenberg`	90.12	1.97	6.8	1.11
average	78.46	13.39	6.92	1.23

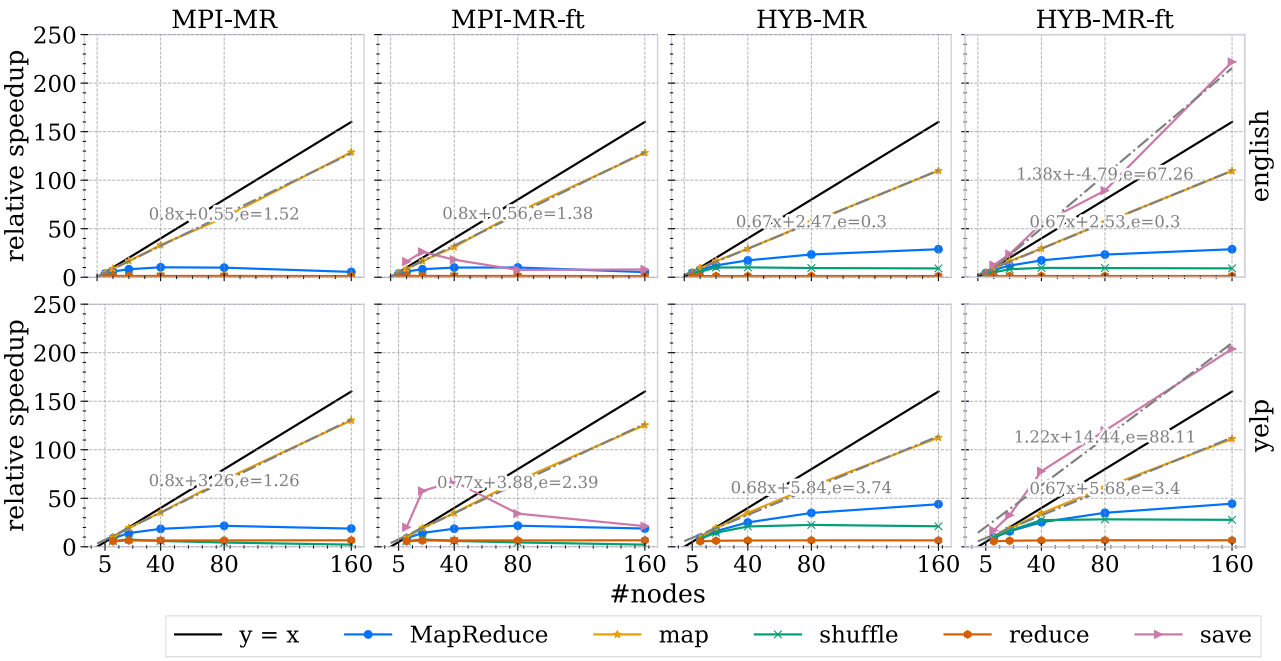


Figure 49: We illustrate the relative speedups of the MapReduce operation during the Word Count algorithm (Section 4.1). We show the results of the different MapReduce configurations listed in Table 1. We apply the Word Count algorithm on `english` (nodes in $\{1, 5, 10, 20, 40, 80, 160\}$) and `yelp` (nodes in $\{5, 10, 20, 40, 80, 160\}$). We perform 7 runs for each text and plot the averages. The x-axes show the number of nodes, while the y-axes show the relative speedups. We profile the runtime of the map phase including serialization and group by key (`map`), reduce phase including deserialization (`reduce`), shuffle phase (`shuffle`), and save self-message time (`save`). `MapReduce` indicates the runtime of the entire MapReduce operation. The linear functions $ax + b$ indicate linear regression with mean square error e.

C.2. PageRank

The following plots and tables correspond to the PageRank experiments described in Section 9.2. Figure 50 shows the results of the week scale experiments with PageRank on `ErdősRényi-d38`. The results are similar to those with `rhg-d8-g3` (Figure 32). Tables 11 and 12 contain the increases in runtime of the different phases during the week scale experiments in Section 9.2.

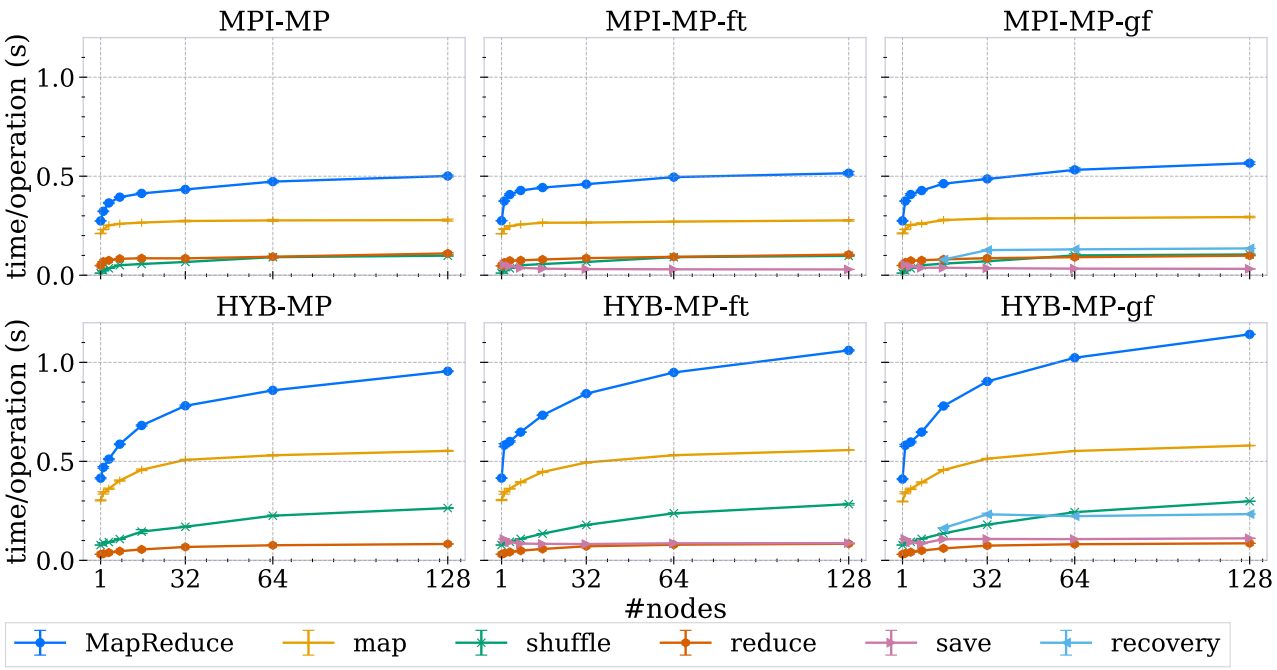


Figure 50: We perform 100 PageRank iterations with our MapReduce configurations (Table 1) on `ErdősRényi-d38` (seeds in $\{1, 2, 3, 4, 5\}$ and nodes $\{1, 2, 4, 8, 16, 32, 64, 128\}$). We generate 2^{18} vertices per node. Furthermore, we generate 10% node failures for `MPI(HYB)-MR-gf`. The x-axes show the number of nodes, while the y-axes show the average runtime per MapReduce operation including error bars with standard deviation. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

Table 11: Illustrate the increase in runtime for the different phases. We indicate the increase $\frac{t_{max}}{t_{min}}$ between the minimum and maximum runtime of a phase during the PageRank experiments on `rhg-d8-g3` (Figure 32). The column "all" indicates the overall increase.

MapReduce	map	shuffle	reduce	all
MPI-MR	1.26	20.427	1.669	2.009
MPI-MR-ft	1.255	20.279	1.574	2.072
MPI-MR-gf	1.186	19.685	1.511	2.474
HYB-MR	1.552	2.965	2.067	1.916
HYB-MR-ft	1.531	4.374	2.029	2.119
HYB-MR-gf	1.648	5.935	2.249	2.47

Table 12: Illustrate the increase in runtime for the different phases. We indicate the increase $\frac{t_{max}}{t_{min}}$ between the minimum and maximum runtime of a phase during the PageRank experiments on `ErdősRényi-d38` (Figure 50). The column "all" indicates the overall increase.

MapReduce	map	shuffle	reduce	all
MPI-MR	1.32	9.223	2.262	1.826
MPI-MR-ft	1.319	9.138	2.187	1.873
MPI-MR-gf	1.39	9.774	2.005	2.061
HYB-MR	1.824	3.413	2.702	2.302
HYB-MR-ft	1.829	3.667	2.731	2.552
HYB-MR-gf	1.949	3.859	2.805	2.781

C.3. Connected Components

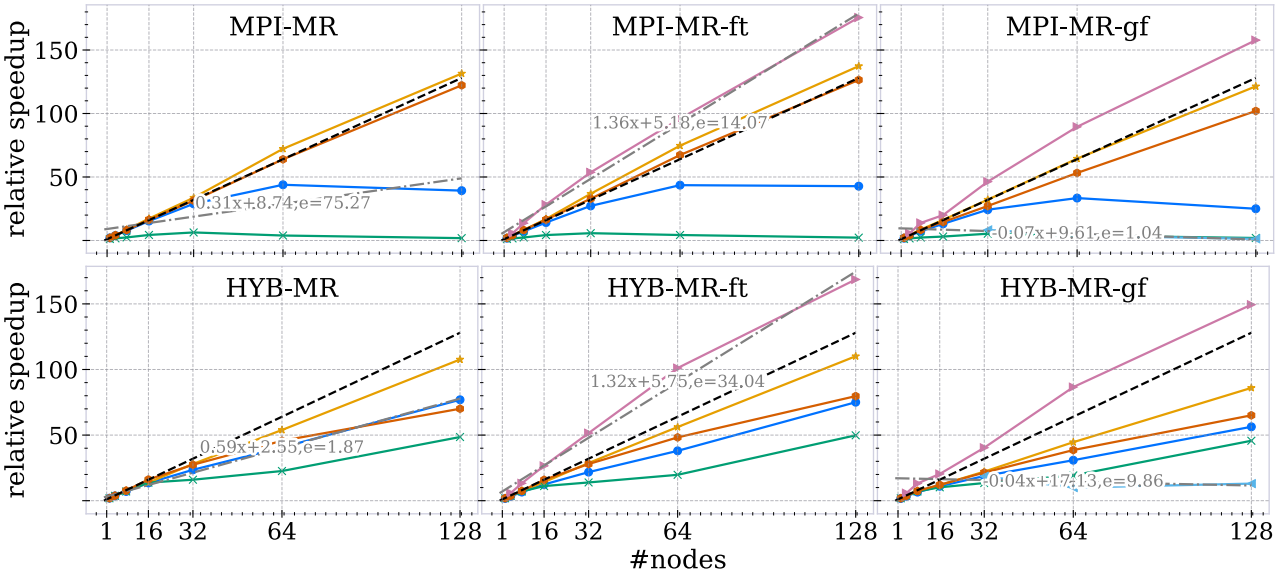
This section provides additional information to the connected component experiments performed in Section 9.3. We adopt the same parameters and naming conventions as introduced in Sections 8 and 9

Table 3 contains the number of large star and small star MapReduce operations performed during the execution of the two phase cc algoithm (Section 4.3). We apply the cc algorithm on the `ErdősRényi-d0.25-n`2^{28} graph (Table 4). Figures 52 and 51 analyze the experiments of the alternating cc algorithm using the same parameters as Figures 36 and 37.

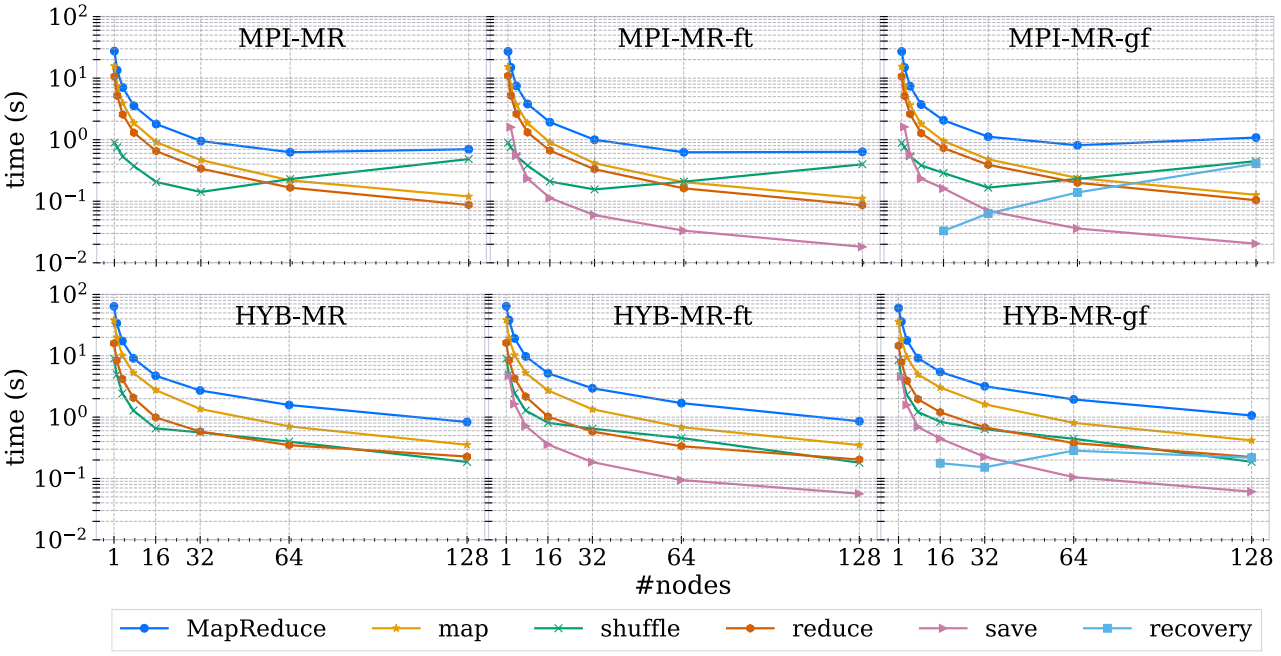
Moreover, we perform week scaling experiments with our cc algorithm on `ErdősRényi-d0.25-n`x (Table 4) for 2^{18} vertices per compute node. Table 14 contains all large star and small star counts for the different MapReduce configurations during our experiments with the two phase algorithm (Figures 53 and 54).

Table 13: This table contains the number of small star and large star MapReduce operations executed during the two phase cc computation for `ErdősRényi-d0.25-n`2^{28} (Table 4) with seeds $s = 1$.

	s	large star								small star							
nodes		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
`MPI-MR(-ft/gf)`	1	16	13	16	16	16	17	14	17	5	4	5	5	5	5	4	5
	2	15	15	17	14	15	17	17	17	5	5	5	4	5	5	5	5
	3	17	14	14	14	14	14	17	18	5	4	4	4	4	4	5	5
	4	14	16	16	15	15	16	16	17	4	6	4	5	5	5	5	5
`HYB-MR(-ft/gf)`	1	17	13	16	16	14	17	14	17	5	4	5	5	4	5	4	5
	2	15	15	17	14	15	16	17	15	5	5	5	4	4	5	5	4
	3	15	14	14	14	15	14	17	15	4	4	4	4	4	4	5	4
	4	14	16	16	15	13	16	15	14	4	5	4	4	4	5	5	4



(a) The y-axes show the relative speedups. We plot a linear regression $ax + b, e$, where e is the mean square error. We compute the regression for the speedup of `MPI(HYB)-MR`, the save self-message phase (`save`) of `MPI(HYB)-MR-ft`, and the speedup for the recovery of `MPI(HYB)-MR-gf`.



(b) The logarithmic y-axes show the average runtime per MapReduce operation.

Figure 51: We perform strong scaling experiments with our MapReduce configurations (Table 1) for the alternating cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n2`^{28} for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages. We generate 10% node failure. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axes show the number of nodes. We illustrate the map phase including grouping by key and serialization (`map`), the reduce phase including deserialization (`reduce`), the shuffle phase (`shuffle`), the save self-message time (`save`), recovery time (`recovery`), and the entire MapReduce execution (`MapReduce`).

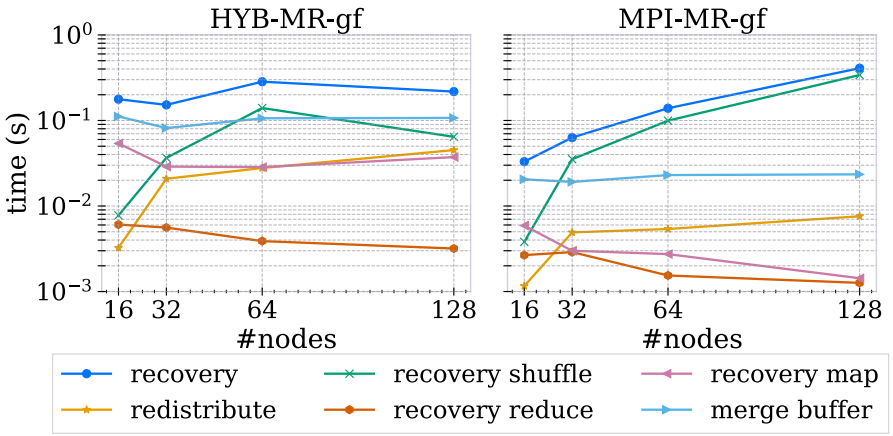


Figure 52: Illustration of the phases occurring during the recovery (Table 2). We show the results for our alternating cc algorithm applied on `ErdősRényi-d0.25-n`2^{28}. We generate 10% node failure. The x-axes contain the number of nodes, while the logarithmic y-axes show the runtime. These times correspond to the recovery time in Figure 51

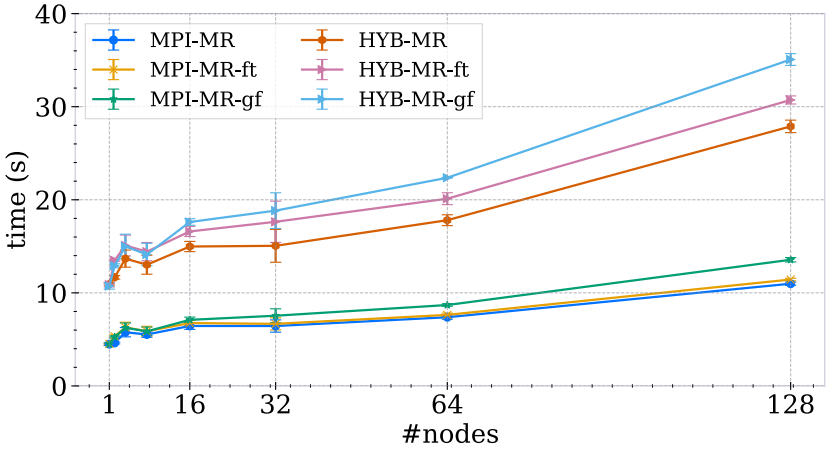


Figure 53: We perform weak scaling experiments with our MapReduce configurations (Table 1) for the two phase cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n`x for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages and include the standard deviation as error bar. For each node we generate 2^{18} vertices. Table 14 illustrates the amount of large star and small star operations. We generate 10% node failure. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axis shows the number of nodes, while the y-axis shows the runtime for the entire cc computation.

Table 14: This table contains the number of small star and large star MapReduce operations executed during the two phase cc computation for `ErdősRényi-d0.25-n`x (Table 4) during our strong scale experiments with seeds $s = 1$. We choose x so that for each node we have 2^{18} vertices and 2^{16} edges.

	s	large star								small star							
nodes		1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
`MPI-MR`	1	16	13	16	16	16	17	14	17	5	4	5	5	5	5	4	5
	2	15	15	17	14	15	17	17	17	5	5	5	4	4	5	5	5
	3	17	14	14	14	14	14	17	18	5	4	4	4	4	4	5	5
	4	14	16	16	15	15	16	16	17	4	5	4	4	5	5	5	5
`MPI-MR-ft`	1	16	13	16	16	16	17	14	17	5	4	5	5	5	5	4	5
	2	15	15	17	14	15	17	17	17	5	5	5	4	4	5	5	5
	3	17	14	14	14	14	14	17	18	5	4	4	4	4	4	5	5
	4	14	16	16	15	15	16	16	17	4	5	4	4	5	5	5	5
`MPI-MR-gf`	1	16	13	16	16	16	17	14	17	5	4	5	5	5	5	4	5
	2	15	15	17	14	15	17	17	17	5	5	5	4	4	5	5	5
	3	17	15	14	14	14	14	17	18	5	5	4	4	4	4	5	5
	4	14	16	16	15	15	16	16	17	4	5	4	4	5	5	5	5
`HYB-MR`	1	16	13	16	16	14	17	14	17	5	4	5	5	4	5	4	5
	2	15	15	17	14	15	16	17	15	5	5	5	4	4	5	5	4
	3	15	14	14	14	15	14	17	15	4	4	4	4	4	4	5	4
	4	15	16	16	15	13	16	15	14	4	5	4	4	4	5	5	4
`HYB-MR-ft`	1	16	13	16	16	14	17	14	17	5	4	5	5	4	5	4	5
	2	15	15	17	14	15	16	17	15	5	5	5	4	4	5	5	4
	3	15	14	14	14	15	14	17	15	4	4	4	4	4	4	5	4
	4	15	16	16	15	13	16	15	14	4	5	4	4	4	5	5	4
`HYB-MR-gf`	1	16	13	16	16	14	17	14	17	5	4	5	5	4	5	4	5
	2	15	15	17	14	15	16	16	15	5	5	5	4	4	5	5	4
	3	15	14	14	14	15	14	17	15	4	4	4	4	4	4	5	4
	4	15	16	16	15	13	16	15	14	4	5	4	4	4	5	5	4

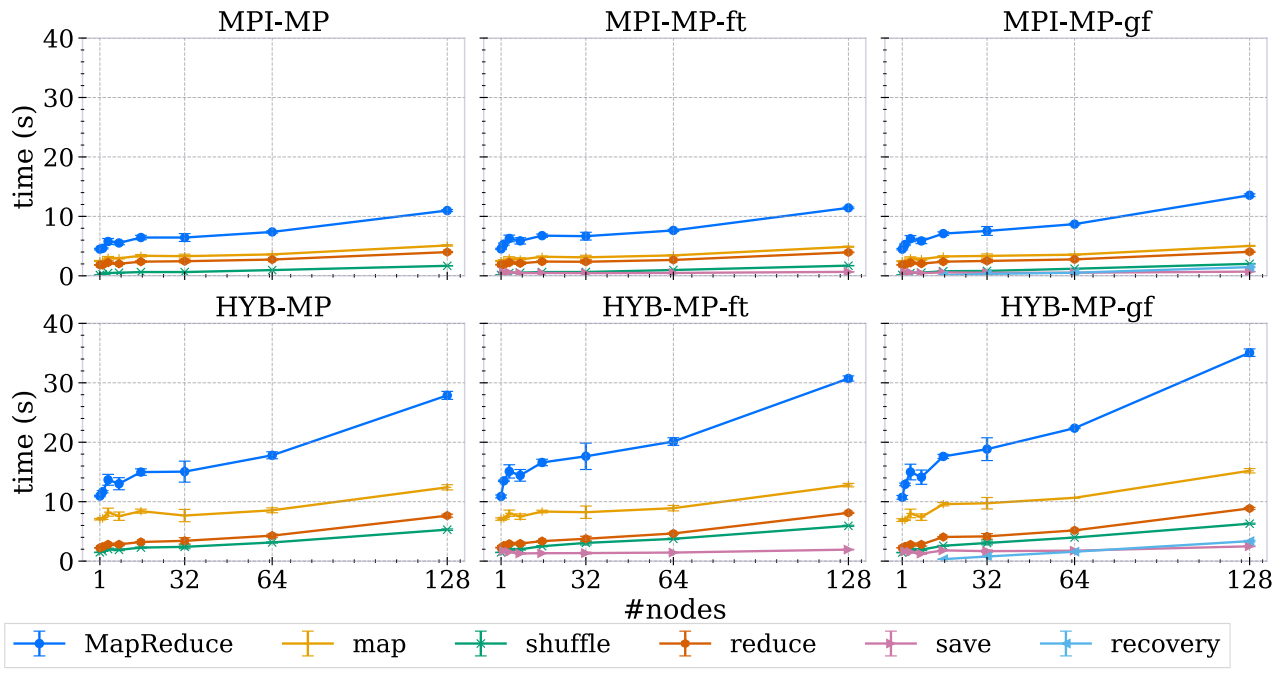


Figure 54: We perform weak scaling experiments with our MapReduce configurations (Table 1) for the two phase cc algorithm (Section 4.3). We run the experiments on `ErdősRényi-d0.25-n`x for seeds $s \in \{1, 2, 3, 4\}$ and plot the averages. For each node we generate 2^{18} vertices. We generate 10% node failures. We chose nodes in $\{1, 2, 4, 8, 16, 32, 64, 128\}$. The x-axes show the number of nodes, while the y-axes show the runtime for the entire cc computation. We illustrate the speedups of the map phase (`map`), reduce phase (`reduce`), shuffle phase (`shuffle`), save self-message time (`save`), and recovery time (`recovery`). `MapReduce` indicates the speedup of the entire MapReduce operation.

References

[1] Access and login to SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/Access+and+Login+to+SuperMUC-NG. Accessed: 2021-08-04.

[2] Data transfer options on SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/Data+Transfer+Options+on+SuperMUC-NG. Accessed: 2021-08-04.

[3] Details of compute nodes. https://doku.lrz.de/display/PUBLIC/Details+of+Compute+Nodes. Accessed: 2021-08-04.

[4] File systems of SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/File+Systems+of+SuperMUC-NG. Accessed: 2021-08-04.

[5] Hadoop randomtextwriter. https://github.com/facebookarchive/hadoop-20/blob/master/src/examples/org/apache/hadoop/examples/RandomTextWriter.java. Accessed: 2021-08-08.

[6] Hardware of SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG. Accessed: 2021-08-04.

[7] Intel oneapi threading building blocks. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html#gs.88b67g. Accessed: 2021-08-05.

[8] Intel xeon platinum 8174 processor. https://ark.intel.com/content/www/us/en/ark/products/136874/intel-xeon-platinum-8174-processor-33m-cache-3-10-ghz.html. Accessed: 2021-08-04.

[9] Project gutenberg is a library of over 60,000 free ebooks. https://www.gutenberg.org/. Accessed: 2021-08-08.

[10] SuperMUC-NG. https://doku.lrz.de/display/PUBLIC/SuperMUC-NG. Accessed: 2021-08-04.

[11] Yelp open dataset. https://www.yelp.com/dataset. Accessed: 2021-08-08.

[12] Intel omni-path fabric suite fabric manager. https://www.intel.com/content/dam/support/us/en/documents/network/omni-adptr/sb/Intel_OP_FabricSuite_Fabric_Manager_UG_H76468_v1_0.pdf, 2015.

[13] OneTBB. https://github.com/oneapi-src/oneTBB, 2021. commit: 9e15720b, branch: master.

[14] A. M. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469)*, pages 167–176. IEEE, 1999.

[15] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145*, 2012.

[16] Apache. Apache hadoop. http://hadoop.apache.org/, 2021.

[17] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. Engineering in-place (shared-memory) sorting algorithms. Computing Research Repository (CoRR), Sept. 2020.

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. Mpi on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[19] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. Mpi on a million processors. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 20–30. Springer, 2009.

[20] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[21] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–32, 2011.

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

[23] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[24] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.

[25] A. Bouteiller. Ulfm 4.0.2u1.

[26] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *The International Journal of High Performance Computing Applications*, 20(3):319–333, 2006.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[28] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 32(3):e4863, 2020.

[29] L. Clarke, I. Glendinning, and R. Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[30] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[31] Y. Collet. xxhash-extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash, 2021. commit: 4aa3d59, branch: dev.

[32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[33] E. Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36(2):205–240, 2000.

[34] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

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

[36] C. Doll, T. Hartmann, and D. Wagner. Fully-dynamic hierarchical graph clustering using cut trees. In *Workshop on Algorithms and Data Structures*, pages 338–349. Springer, 2011.

[37] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, et al. The international exascale software project: a call to cooperative action by the global high-performance community. *The International Journal of High Performance Computing Applications*, 23(4):309–322, 2009.

[38] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In *Fault-tolerance techniques for high-performance computing*, pages 3–85. Springer, 2015.

[39] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818, 2010.

[40] P. Erdos, A. Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[41] G. E. Fagg and J. J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 346–353. Springer, 2000.

[42] G. E. Fagg and J. J. Dongarra. Building and using a fault-tolerant mpi implementation. *The International Journal of High Performance Computing Applications*, 18(3):353–361, 2004.

[43] P. Ferragina and G. Navarro. English texts. pizzachili.dcc.uchile.cl/texts/nlang/. Accessed: 2021-08-08.

[44] E. H. Friend. Sorting on electronic computer systems. *Journal of the ACM (JACM)*, 3(3):134–168, 1956.

[45] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

[46] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer. Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 1098–1108. IEEE, 2017.

[47] G. Georgakoudis, L. Guo, and I. Laguna. Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In *International Conference on High Performance Computing*, pages 536–554. Springer, 2020.

[48] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

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

[50] L. Gugelmann, K. Panagiotou, and U. Peter. Random hyperbolic graphs: degree sequence and clustering. In *International Colloquium on Automata, Languages, and Programming*, pages 573–585. Springer, 2012.

[51] L. Guo, G. Georgakoudis, K. Parasyris, I. Laguna, and D. Li. Match: An mpi fault tolerance benchmark suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 60–71. IEEE, 2020.

[52] Y. Guo, W. Bland, P. Balaji, and X. Zhou. Fault tolerant mapreduce-mpi for hpc clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[53] L. Hübschle-Schneider and P. Sanders. Linear work generation of r-mat graphs. *Network Science*, 8(4):543–550, 2020.

[54] L. Jiang, B. Ge, W. Xiao, and M. Gao. Bbs opinion leader mining based on an improved pagerank algorithm using mapreduce. In *2013 Chinese Automation Congress*, pages 392–396. IEEE, 2013.

[55] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[56] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[57] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2014.

[58] A. Koschel, F. Heine, I. Astrova, F. Korte, T. Rossow, and S. Stipkovic. Efficiency experiments on hadoop and giraph with pagerank. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 328–331. IEEE, 2016.

[59] J. Leskovec. Orkut social network and ground-truth communities. https://snap.stanford.edu/data/com-Orkut.html. Accessed: 2021-08-20.

[60] J. Leskovec. Stanford network analysis project. https://snap.stanford.edu/index.html. Accessed: 2021-08-20.

[61] J. Leskovec. Twitter follower network. https://snap.stanford.edu/data/twitter-2010.html. Accessed: 2021-08-20.

[62] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85, 2010.

[63] S. Maitrey and C. Jha. Mapreduce: simplified data analysis of big data. *Procedia Computer Science*, 57:563–571, 2015.

[64] N. Manchanda and K. Anand. Non-uniform memory access (numa). *New York University*, 4, 2010.

[65] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB journal*, 9(3):231–246, 2000.

[66] M. K. Marina and S. R. Das. Routing performance in the presence of unidirectional links in multihop wireless networks. In *Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing*, pages 12–23, 2002.

[67] P. Messina. The exascale computing project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[68] L. H.-S. Michael Axtmann, Sascha Witt. In-place Parallel Super Scalar Radix Sort (IPS2Ra). https://github.com/ips4o/ips2ra, 2021. commit: ee6103c, branch: master.

[69] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1123–1136, 2015.

[70] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[71] J. R. Neely and B. R. de Supinski. Application modernization at llnl and the sierra center of excellence. *Computing in Science & Engineering*, 19(5):9–18, 2017.

[72] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information processing letters*, 49(1):9–14, 1994.

[73] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[74] S. Pallickara and G. Fox. Naradabrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer grids. In *ACM/IFIP/USENIX Interna-*

tional Conference on Distributed Systems Platforms and Open Distributed Processing, pages 41–61. Springer, 2003.

[75] A. B. Patel, M. Birla, and U. Nair. Addressing big data problem using hadoop and map reduce. In *2012 Nirma University International Conference on Engineering (NUiCONE)*, pages 1–5. IEEE, 2012.

[76] F. Petrini and M. Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Proceedings 11th international parallel processing symposium*, pages 87–93. IEEE, 1997.

[77] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[78] M. Raab and A. Steger. "balls into bins"—a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[79] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[80] K. Rattanaopas and S. Kaewkeeree. Improving hadoop mapreduce performance with data compression: A study using wordcount job. In *2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 564–567. IEEE, 2017.

[81] D. Richards, O. Aaziz, J. Cook, S. Moore, D. Pruitt, and C. Vaughan. Quantitative performance assessment of proxy apps and parentsreport for ecp proxy app project milestone adcd-504-9. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2020.

[82] S. Rivas-Gomez, S. Markidis, E. Laure, K. Brabazon, O. Perks, and S. Narasimhamurthy. Decoupled strategy for imbalanced workloads in mapreduce frameworks. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 921–927. IEEE, 2018.

[83] M. Sahane, S. Sirsat, and R. Khan. Analysis of research data using mapreduce word count algorithm. *Internl. Journal of Advanced Research in Computer and Commn. Engg*, 4, 2015.

[84] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3):145–164, 2016.

[85] P. Sanders. On the competitive analysis of randomized static load balancing. In *Proceedings of the first Workshop on Randomized Parallel Algorithms, RANDOM*. Citeseer, 1996.

[86] P. Sanders. Connecting mapreduce computations to realistic machine models. *arXiv preprint arXiv:2002.07553*, 2020.

[87] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[88] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *European Conference on Parallel Processing*, pages 682–694. Springer, 2007.

[89] M. Skarupke. I Wrote a Faster Sorting Algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/, 2016.

[90] D. B. Skillicorn, J. Hill, and W. F. McColl. Questions and answers about bsp. *Scientific Programming*, 6(3):249–274, 1997.

[91] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[92] M. Snir12. Toward exascale resilience: 2014 update.

[93] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of International Conference on Parallel Processing*, pages 526–531. IEEE, 1996.

[94] D. Türei, T. Korcsmáros, and J. Saez-Rodriguez. Omnipath: guidelines and gateway for literature-curated signaling pathway resources. *Nature methods*, 13(12):966–967, 2016.

[95] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[96] S. Vitali, J. B. Glattfelder, and S. Battiston. The network of global corporate control. *PloS one*, 6(10):e25995, 2011.

[97] M. von Looz, H. Meyerhenke, and R. Prutkin. Generating random hyperbolic graphs in subquadratic time. In *International Symposium on Algorithms and Computation*, pages 467–478. Springer, 2015.

[98] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[99] N. Wolfe, M. Mubarak, N. Jain, J. Domke, A. Bhatele, C. D. Carothers, and R. B. Ross. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 258–261. IEEE, 2017.

[100] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[101] K. H. Zou, K. Tuncali, and S. G. Silverman. Correlation and simple linear regression. *Radiology*, 227(3):617–628, 2003.