Fault Tolerant Collective Communication Algorithms for Distributed Database Systems

Fehlertolerante Gruppenkommunikations Algorithmen für verteilte Datenbanksysteme

Master-Thesis von Jan Stengler aus Mainz

April 2017

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Distributed Systems Programming
Parallel Programming

Fault Tolerant Collective Communication Algorithms for Distributed Database Systems
Fehlertolerante Gruppenkommunikations Algorithmen für verteilte Datenbanksysteme

Vorgelegte Master-Thesis von Jan Stengler aus Mainz

Prüfer: Prof. Dr. Patrick Eugster, Prof. Dr. Felix Wolf
Betreuer: Jonathan Dees, Dipl.-Inform., Martin Weidner, M.Sc.

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den April 3, 2017

(Jan Stengler)

Zusammenfassung

Aktuelle Entwicklungen haben gezeigt, dass sich mittels spaltenbasierten Hauptspeicherdatenbanken komplexe Anfragen um mehrere Größenordnungen schneller verarbeiten lassen, als in traditionellen festplattenbasierten Datenbanken. Dabei ist der physisch nutzbare maximale Hauptspeicher pro Maschine beschränkt bzw. nur bis zu einer gewissen Grenze kosteneffizient. Daher ist es erforderlich, ab einer bestimmten Datengröße mehrere Maschinen zur Datenverarbeitung zu nutzen, wobei der Nutzer weiterhin die Sicht eines einzelnen logischen Datenbanksystems beibehält. Je mehr Maschinen verfügbar sind, desto mehr Daten können gespeichert und effizient verarbeitet werden.

Aber die Anzahl der verfügbaren Maschinen kann die Performance nur verbessern, wenn es eine Möglichkeit zur effizienten Kommunikation zwischen den Maschinen gibt. Zusätzlich sinkt die durchschnittliche Zeit bis zum Auftreten eines Systemfehlers, je mehr Maschinen vorhanden sind. Dies kann so weit führen, dass komplexe Berechnungen so lange dauern, dass diese nicht abgeschlossen werden können, ohne von Systemfehlern unterbrochen zu werden. Daher ist es notwendig, über eine effiziente und fehlertolerante Kommunikationsmöglichkeit zu verfügen.

Das Message Passing Interface (MPI) bietet eine Möglichkeit zur effizienten Kommunikation zwischen Maschinen, jedoch sind die Möglichkeiten Fehler zu tolerieren stark beschränkt. Mehrere Arbeiten beschäftigen sich mit Fehlertoleranz in MPI und einige Verfahren, wie z. B. das sogenannte Checkpoint-Restart Verfahren, wurden eingeführt. Diese Verfahren beruhen jedoch darauf, die Berechnung von einem bestimmten, stabilen Zeitpunkt des Systems wieder zu starten und können somit einen erheblichen Verlust von bereits ausgeführten Rechenleistungen haben. Zudem können in extremen Fällen Fehler so häufig auftreten, dass sich Berechnungen mit Checkpoint-Restart Verfahren endlos wiederholen.

In dieser Arbeit wird untersucht, ob fehlertolerante kollektive Kommunikationen die Fehlerbehandlung verbessern können und ob dieser Ansatz hinsichtlich Fehlertoleranz in zukünftigen System, wie z. B. Exascale-Systemen, relevant ist. Hierbei werden jedoch nicht alle von MPI bereitgestellten kollektiven Kommunikationen untersucht, sondern eine Teilmenge der möglichen Kommunikationen, die es ermöglicht komplexere Kommunikationen zu implementieren.

In verteilten Datenbanksystemen wird MPI heutzutage nur selten verwendet, obwohl es eine sehr effiziente Möglichkeit für Kommunikationen bietet und in den meisten High-Performance-Computing Anwendungen verwendet wird. Durch fehlertolerante kollektive Kommunikationens schließen wir eine Lücke, die den Einsatz von MPI in verteilten Datenbanksystemen behindert.

Wir implementieren einen Prototyp, der die fehlertoleranten kollektiven Kommunikationen ermöglicht und evaluieren diesen mit dem Fokus auf verteilten Datenbanksystemen. Hierfür evaluieren wir die implementierten Kommunikationen hinsichtlich der Laufzeit und testen deren Anwendbarkeit in verteilten Datenbanksystemen.

Wir zeigen, dass Fehlertoleranz in kollektiven Kommunikationen zusätzliche Kosten einführt und vergleichen diesen Ansatz mit dem Ansatz, die Berechnung neu zu starten, falls Fehler auftreten. Unsere Ergebnisse zeigen den Vorteil von fehlertoleranten kollektiven Kommunikationen.

Abstract

Current developments have shown that column-based main memory database management systems allow the usage of much more complex requests than traditional disk-based database management systems. Because of the limitation of the main memory in a computer the trend goes towards clusters of many interconnected machines, also called nodes, with the user interface of a single logical database management system. The more nodes are available in a cluster the more data can be stored and efficiently processed.

But the increasing size of a cluster can only improve the performance of the database system if the nodes in a cluster have the possibility to communicate efficiently with each other. Furthermore, the Mean Time To Failure (MTTF) of these systems is decreasing as the number of machines is increasing. The execution time for complex computing applications might become longer than the MTTF, so that they can not complete their computation without a node failure. Therefore, it is necessary to enable efficient and fault tolerant collective communications in a cluster.

The Message Passing Interface (MPI) provides an efficient way to enable the communication between nodes. However, the error handling in many MPI implementations is very limited. Still, there is a lot of research focusing on fault tolerant MPI including approaches like checkpoint restart protocols. Depending on the checkpoint granularity, falling back to a previous checkpoint can result in a significant loss of computation and in the extreme case even to an endless loop, if there is a continual error between two checkpoints. If checkpoints are too fine granular, the overhead for writing persistent checkpoints can become significant.

In this work, we analyze whether introducing fault tolerant collective communications can improve the error handling of computations and show that this approach can be a promising fault tolerance technique with regard to future systems, like exascale systems. Instead of supporting efficient error handling for all possible collective communication functions of MPI on an abstract level, this work focuses on a subset of collective communication algorithms which allow to build a more complex set of collective communications in a fault tolerant way. The examined collective communications are Broadcast, Scatter, Gather, Reduce and All-to-all.

Although MPI is a good abstraction layer for efficient collective communications and heavily used in high performance computing, it is rarely used in database systems today. Adding efficient error handling support for collective communications closes one feature gap required by distributed database systems.

We create a prototypical implementation for fault tolerant collective communications and evaluate it with focus on distributed database systems. We evaluate the implemented algorithms on the Lichtenberg High Performance computer of TU Darmstadt and show their applicability by implementing a TPC-H Benchmark query and a distributed sorting algorithm, the sample sort algorithm.

We show that introducing fault tolerance in collective communication algorithms is introducing overhead and compare this approach with the approach of restarting the computation in case of failures. Our results illustrate the advantage of resilient collective communications compared to restarting approaches.

Applications that use our proposed approach of fault tolerant collective communications can be secured. Therefore, we propose a general approach to application based fault tolerance.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Nowadays column-based main memory database management systems are widely used for efficient data processing. On the one hand because they have proven to be much more efficient than traditional row-oriented database systems [2], on the other hand the increasing availability of cheap, large memory supports the growth of their usage [43]. With the increasing performance of the database management systems, the size of the databases is increasing too. Because of the limited main memory in a single machine, the trend goes towards parallel and distributed databases, which consists of clusters of many (interconnected) processing nodes. Hereby a node is a machine with multiple cores and a shared-memory. With the demand for processing more and more data efficiently, the number of machines, and especially the number of computing cores in a cluster, is increasing steadily. Figure 1.1 shows the development of the amount of cores in the first ranked supercomputer of the TOP500 list[1] from the year 1993 to 2011.

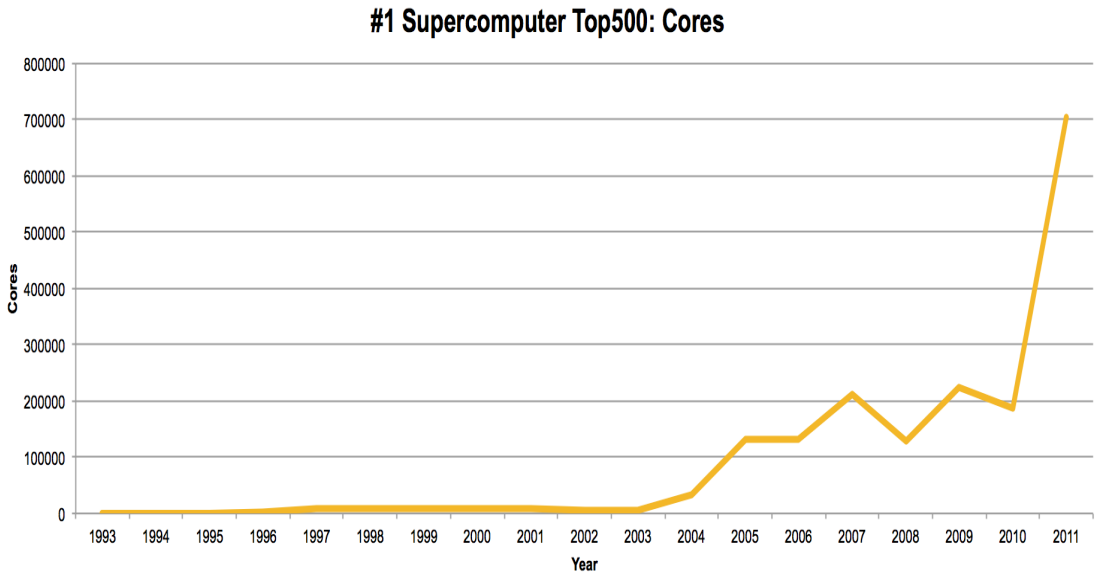

Figure 1.1: Number of Cores of the # 1 supercomputer (Data Source: TOP500[1])

With the increasing size of clusters many challenges are emerging. Efficient inter-node communication is one of these challenges. Without an efficient way to communicate in a cluster, the benefit of using many nodes can not be fully utilized. Another challenge lies in the decreasing Mean Time Between Failures (MTBF) [29]. The more machines are available in a cluster, the more likely it is, that a machine will experience a failure. A failure in this case can be software or hardware related and will hinder the machine to finish the computation successfully and correctly. Therefore, the need for fault tolerance in distributed systems is becoming more present, as the size of clusters is increasing. If we consider a very high, optimistic, Mean Time Between Failures of one century, a system with 100 000 nodes will experience a node failure every 9 hours [22]. This is leading as far as some complex computations might not even be able to finish their computation, without being interrupted by node failures.

To address the first challenge of the efficient inter-node communication, the Message Passing Interface (MPI) has been evolved. MPI is a specification for a standard library for message passing, that was defined by the MPI Forum [31]. Several MPI implementations are available today and MPI has been established as the prevalent programming paradigm for High-Performance-Computing (HPC) applications [37]. MPI provides an efficient way to enable the inter-node communication in a cluster and offers basic functions for sending and receiving messages as well as functions for collective communications, like

[1] https://www.top500.org

broadcast, all-to-all and many more. However, the support for fault tolerance in MPI is limited [13] and no clear strategy has been formed.

Since HPC applications are becoming more complex, several fault tolerance techniques have been established, most prominently the checkpoint-restart technique. In this technique checkpoints are taken periodically and in case of failures the status of the distributed system is rolled back to a checkpoint in which the system was stable. This technique is introducing a serious overhead, since checkpoints have to be taken periodically, have to be stored and the system has to be able to determine which checkpoint was the latest stable status of the system. Much research is focusing on improving the performance of checkpoint-restart techniques but recent studies outline that checkpoint-restart techniques will perform poorly in exascale systems, leading so far that replication techniques might become more efficient [11, 27].

1.1 Contributions

The work of this thesis is focusing on introducing fault tolerant collective communications. Since checkpoint-restart techniques might become irrelevant in exascale systems and are already introducing much overhead in current petascale systems, we will analyze techniques which do not rely on checkpointing. We will offer several fault tolerant collective communication algorithms, which can be used by developers to create fault tolerant applications without having to put much effort in resilience techniques. The focus of the application area of these algorithms is on distributed database systems, but they are also portable to different HPC scenarios. The main objective of these algorithms is the correctness of the result of the communication, even in case of failures. Hereby, the number of the failures or the point in time of their occurrence will not affect the correctness of the proposed algorithms. Furthermore, we require the algorithms to run efficiently. In case of no failures the algorithms shall still perform efficiently with a minimum overhead that is introduced by enabling fault tolerance.

1.2 Outline

In Chapter 2 we provide a basic overview of the background related to this work. We start by introducing parallel and distributed database systems, the main application area of this work. Then we give an overview about state-of-the-art fault tolerance techniques in High-Performance-Computing and continue by showing the possibilities for resilience in MPI.

In Chapter 3 we explorer how to build the most basic collective communication algorithms in an efficient way with regard to fault tolerance. We show their applicability by implementing the distributed sorting algorithm sample sort with the usage of the elaborated fault tolerant collective communications.

In Chapter 4 we show the enhancement of the interfaces of the collective communication functions to use fault tolerance. We compare the interfaces with the interfaces of the not fault tolerant MPI functions.

Chapter 5 shows the evaluation of the afore mentioned communications. The overhead, introduced by fault tolerance techniques, is evaluated also as their recovery possibilities and their correctness. We implement a TPC-H Benchmark query and evaluate the behavior in case of failures for the fault tolerant and not fault tolerant collective communications. We show that the implemented algorithms experience an overhead and have a good recovery performance, which shows the relevance and their applicability in HPC applications.

2 Background

This chapter provides an overview over the background of this thesis. First the system model is explained and parallel and distributed database systems are discussed in Section 2.1. Basic terminologies about failures in distributed systems are introduced in Section 2.2. A previous approach for fault tolerance in broadcast is shown in Section 2.3. Section 2.4 is presenting techniques for resilience in High-Performance-Computing (HPC) applications. These techniques can be applied to distributed systems in general. Thus, they are also relevant for distributed database systems.

The Message Passing Interface (MPI) is introduced in Section 2.5. In Section 2.6 we show several possibilities to apply the resilience techniques from Section 2.4 in MPI implementations.

In contrast to most of the existing works about fault tolerance for distributed systems, this work is focusing on fault tolerant algorithms for collective communications and not on introducing fault tolerance for a specific type of application. Using these fault tolerant communications allows developers to implement High-Performance-Computing applications in general resilient. This work provides a general approach for applications to handle failures.

2.1 System Model

Distributed systems are the application area of this work with a special focus on distributed database management systems. A distributed system consists of clusters, which have multiple computing nodes. The nodes in a cluster are all interconnected by a network, e.g. InfiniBand, and represent a self-contained system, which has several cores that can access the same memory area, also called a shared-memory system. The more cores are available in a system, the more tasks can be performed in parallel. For that reason the number of available cores in clusters is increasing steadily. Since the possibilities to increase the number of cores and the main memory of a system (scale-up) are limited or too costly, the computing power of a cluster is increased by adding more nodes (scale-out) [44]. Distributed systems have been proven applicable to improve the performance of database systems in form of parallel database management systems and distributed database management systems [59].

2.1.1 Parallel Database Systems

In [47] a parallel database system is defined as a database system that seeks to improve performance through parallelization of various operations. These operations can be loading data, building indexes, and evaluating queries. Although data may be stored in a distributed fashion in such a system, the distribution is governed solely by performance considerations.

For multiprocessor high transaction rate systems three main architectures have been established [56], which are shown in Figure 2.1: the shared-nothing, shared-memory and shared-disk systems. These architectures are also the most common architectures for parallel database systems.

In shared-nothing environments every CPU has a private memory and a private disk area. This means that data can only be exchanged by communicating over the interconnection network. In shared-memory environments many CPUs can access the same memory and all disks. In a shared-disk environment every processor has its own memory area but is able to access every disk via the interconnection network.

2.1.2 Distributed Database Systems

In [47] a distributed database system is defined as a database system in which data is physically stored across several sites. Each site is typically managed by a database management system, capable of running independent of the other sites. The location of data items and the degree of autonomy of individual sites have a significant impact on all aspects of the system, including query optimization and processing, concurrency control, and recovery. In contrast to parallel databases, the distribution of data is governed by factors such as local ownership and increased availability, in addition to performance issues.

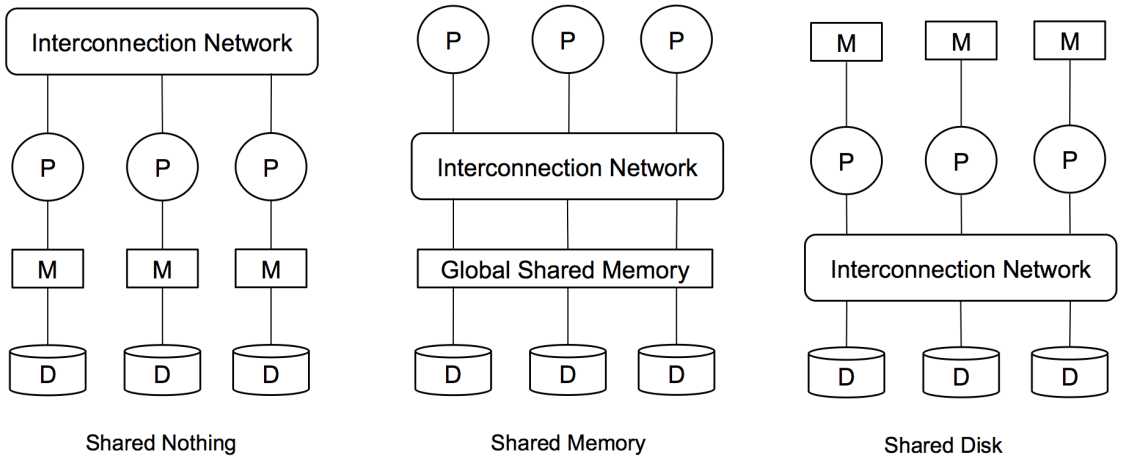

Figure 2.1: Physical Architectures for Parallel Database System [47]

This means that a distributed database system can consist of multiple (parallel) database management systems. Each node has its own database management system which is organizing the data access for the processors on this node.

The application area of this work focuses on distributed database systems that follow a shared-nothing environment. This means that every database management system in the distributed database system is following a shared-nothing environment and data exchange has to be performed with communications.

2.2 Failures in Distributed Systems

In order to discuss fault tolerance, we first need to discuss failure types and to define basic terms like *fault tolerance* and *resilience*. In distributed systems many different types of failures are possible. Therefore, we give an overview of possible errors and state on which error types this work is focusing. Furthermore, we analyze how to model the distribution of errors in distributed systems.

2.2.1 Terminology

For a long time in High-Performance-Computing many terms related to failures and fault tolerance have been used vaguely. This is why basic terms were defined in [6, 53, 54], with the purpose to create a common basic understanding. In the following we name the most important definitions which are used in this work.

A *failure* is defined as the termination of the ability of an item, to perform a required function [54, 58]. To restore the ability of the item to perform the required function an external action is required, e.g., a manual reboot. They differentiate between *failures* and *interrupts*. *Failures* regard the components of a system and *interrupts* regard the work flow of a system.

A *fault* is defined as an accidental condition that causes an item to fail to perform its required function and is therefore the cause of a *failure*. [6, 53] differentiate between *fault* and *error*. An *error* is the part of a total state that may lead to a failure (e.g. a bad value) and is caused by a *fault*. For reasons of simplicity we will use the terms *fault* and *error* equivalently. *Faults* can be active, when they are causing a *failure*, or inactive, when they are not affecting the system. Definitions of more general terms like supercomputer, process, etc., can also be found in [54].

Terms that define techniques for failure handling are defined in [53]. *Avoidance* is defined as reducing the occurrence of failures. *Detection* means to detect failures as soon as possible after their occurrence. *Recovery* is the ability to overcome detected failures and *repair* means that the failed components will be repaired or replaced.

Another important definition is the *persistence* of a failure. The *persistence* is the behavior in case of a failure, if the failed system may halt (fail-stop) or may exhibit erratic behavior. In this work we

mainly focus on fail-stop failures since these failures are hard to recover and interrupt the execution of the application. Other failure types are important too but can also be handled in other areas, like the network protocol. Silent errors, errors which can not be detected directly, are supposed to be rare and thus are not examined in this work.

In [53] *resilience* is defined as the collection of techniques for keeping applications running until a correct solution in a timely and efficient manner, despite underlying system faults, is delivered. Possible solutions for *resilience* in future exascale systems are divided into three options. The *base option* in which resilience is handled with the same approach as of today, by using checkpointing (see Section 2.4.1), with the drawback of more costs with regard to added hardware and power consumption. The *system option* in which resilience is a combination between hardware and system software techniques and the *application option* in which developers have to handle resilience. The *application option* is beneficial with regard to the costs of the platform itself. As outlined in [53], resilience techniques without software changes may become too expensive for exascale systems. Thus, the *application option* is the most promising approach for the future. However, the drawback of fault-tolerant applications is that they are specific to one or to a family of algorithms. For exascale systems a more general approach is needed that can apply to all computations. For this reason, we focus in this work on the *application option* and propose a new, general approach of algorithmic fault tolerance, fault tolerant collective communications. Using these fault tolerant collective communications will allow any computation to resist failures.

Tolerance is the avoidance of service failures in the presence of faults and therefore, the main focus of this work. The application should be able to continue correctly in case of failures. Other important means are *forecasting*, *prevention* and *removal*. *Forecasting* is estimating the present and future number of faults. It can be combined with several fault tolerance techniques in order to improve their performance. *Prevention* is describing the group of techniques that focus on preventing failure occurrence. *Removal* is focusing on reducing the number and severity of faults. Fault tolerance techniques can be categorized in *error detection* and *recovery*, where *error detection* stands for identifying errors and *recovery* stands for preventing faults from causing failures. *Recovery* is categorized in *error-handling* which means to eliminate the errors and *fault-handling* which stands for preventing faults from reactivating.

A failure will cause a change in the state of a supercomputer. The different states of a supercomputer can be seen in Figure 2.2.

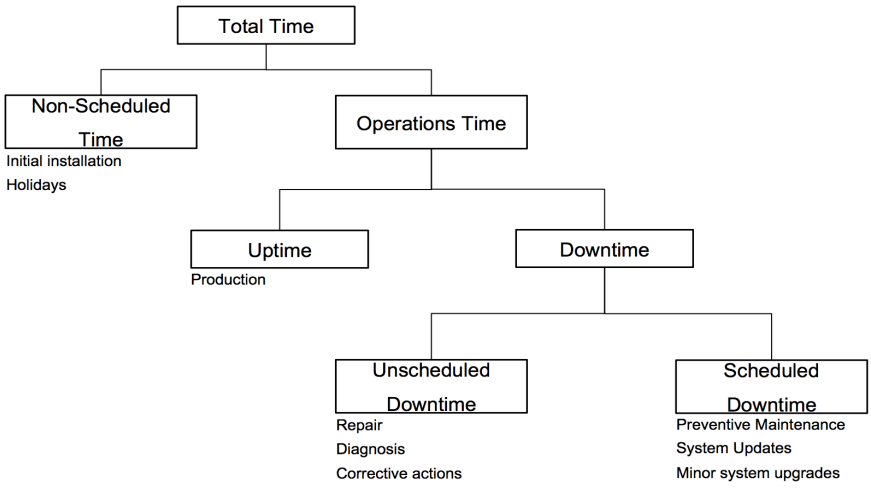

Figure 2.2: Possible states of a supercomputer [29]

In the lifecycle of a supercomputer there is a non-scheduled time. This time can be used for the initial installation or in holidays when there is no computation. All other times belong to the operations time. While the supercomputer is running, it is in the uptime, if not it is in the downtime. The downtime can be a scheduled or an unscheduled downtime. The scheduled downtime can be used, e.g., for system

updates. A failure can cause an unscheduled downtime. This time is used for actions like repairing, diagnosis and other corrective actions.

But a fault can not only impact the component it was raised on. Furthermore, the problem of cascading failures is present in distributed systems. This means that an error on one component can cause a failure on another component and the failure of one component can cascade to another component. This behavior is shown in Figure 2.3.

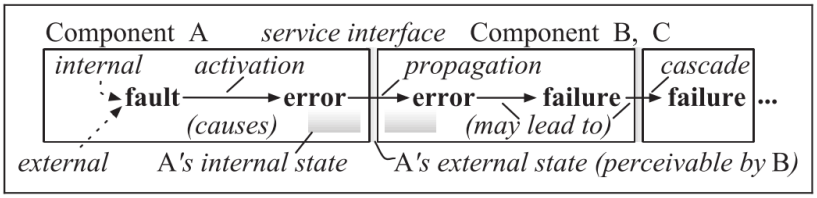

Figure 2.3: Cascading Error [53]

2.2.2 Characteristics

In [29] a survey of failure characteristics can be found. We specify the terms *mean time between failures*, *mean time to failures* and *mean time to repair* accordingly. These time characteristics can be represented by the following formulas.

- MTBF (Mean time between failures) $= \frac{TotalTime}{\#Failures}$
- MTTF (Mean time to failures) $= \frac{Uptime}{\#Failures}$
- MTTR (Mean time to repair) $= \frac{UnscheduledDowntime}{\#Failures}$

These terms allow us to specify some key characteristics about the resilience of a distributed system. It is also important to model the failures hitting a distributed system empirically in order to provide a compact evaluation of a system. One method is to calculate the three metrics, mean, median and squared coefficient of variation for the analysis. Other methods are using empirical cumulative distribution functions. The most used empirical cumulative distribution function was the Poisson distribution. However recent studies have found that the Poisson distribution is rather fitting poorly and a Weibull or log-normal distribution is a much better fit [29]. But no matter which distribution is chosen, the goodness of fit has always to be tested. There are several tests available, first of all it has to be tested that the failures are occurring randomly, after that the fit of the distribution can be tested, e.g., with the Anderson-Darling, Kolmogorov–Smirnov or the chi-square test.

Failures can be usually categorized into software, hardware, network, facility/environment, unknown or heartbeat failures. An overview about the failure types and their occurrence in famous distributed systems is shown in the table of Figure 2.4.

Category	Blue Waters (%)	Blue Gene/P (%)	LANL systems (%)
Hardware	43.12	52.38	61.58
Software	26.67	30.66	23.02
Network	11.84	14.28	1.8
Facility/Environment	3.34	2.66	1.55
Unknown	2.98	–	11.38
Heartbeat	12.02	–	–

Figure 2.4: Failure Types [29]

2.3 Fault Tolerant Broadcast

Several works have analyzed and proposed approaches for fault tolerant broadcasting. Best to our knowledge broadcast is the only collective communication for which fault tolerance has been studied. The reason for this is that a fault tolerant broadcast protocol is mainly used to inform all non failed entities in a distributed environment about the occurred failures.

In [52] an approach for a fault tolerant broadcast protocol is introduced and represents the basis of most works about fault tolerant broadcast. As in our work an environment of several processors is assumed, in which a root process wants to send the data to every other process. The fault tolerant broadcast protocol is applicable independent of the used broadcasting strategy. This means that the way the message is broadcasted through the network does not affect the correctness of the proposed fault tolerant protocol. Similar to our work, this approach is focusing on process failures.

The general approach for handling process failures is performed on the network level. If a process is not receiving an acknowledgment of another process, it has sent the message to, it knows that this process failed and it will send the message again to the, in the broadcast strategy successional, processes of the failed process. The sending process is taking the responsibility of the role of the failed process. Therefore, the message is distributed to the subtree of the failed process and every surviving process is receiving the message. Failed processes are not allowed to participate in the communication after they have been repaired because this can falsify the status of the fault tolerant broadcast protocol. The approach for fault tolerance in collective communications of this work is following a similar strategy to this protocol.

In Figure 2.5 the fault tolerant broadcast protocol is visualized. In this broadcasting strategy Process 0 is sending the message to Process 1, Process 1 forwards the message to Process 2 and Process 2 distributes the message to the Processes 3, 4 and 5. In the depicted scenario Process 1 is failing and can not distribute the message. Process 0 recognizes this error, the acknowledgment of Process 1 is not delivered. Therefore Process 0 is forwarding the message to the subsequent Processes of 1, which is the Process 2. Process 2 is distributing the message further and acknowledges Process 0 that is has received the message.

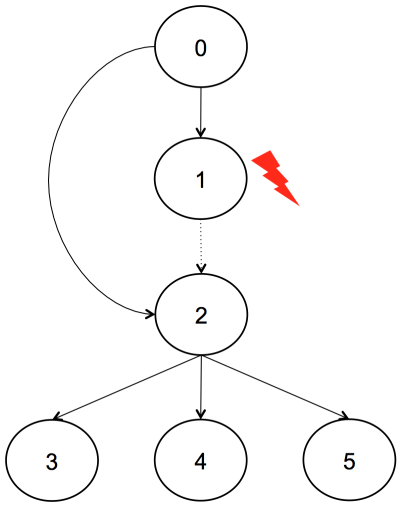

Figure 2.5: Fault Tolerant Broadcast Protocol

2.4 Fault Tolerance Techniques for High-Performance-Computing

In this section several techniques regarding resilience in High-Performance-Computing are discussed. The benefits and drawbacks are analyzed and we state which of these techniques are relevant for this work.

2.4.1 Checkpointing

Checkpointing, Checkpoint/Restart or Checkpoint and Rollback-Recovery are all describing the same fault tolerance technique in HPC and are currently the most used technique for providing fault tolerance in HPC applications. Checkpointing belongs to the group of general-purpose techniques. General-purpose techniques correct failures at a given level of the software stack and are masking these failures to higher levels of the software stack [22]. The key feature of checkpointing is that applications are regularly taking checkpoints (of processes) in which the current state of the application is saved and are restoring their state to the last stable one, if an error is detected. Checkpointing protocols can be categorized by the way they are creating their checkpoints, how they are rolling back the system state and on which level of the software stack they are used.

The lowest level of the software stack is the system level. Examples for checkpointing on system level are CRAK [62] which is a checkpointing framework for linux kernels and BLCR (Berkeley lab checkpoint/restart) [33] as a checkpointing framework for linux clusters. The advantage of system level checkpointing frameworks is that only the system has to be modified but not the application and therefore these techniques are highly compatible.

Higher in the software stack checkpointing techniques for compilers can be found. Examples for checkpointing techniques in compilers are CATCH (compiler-assisted techniques for checkpointing) [39], CAME (compiler-assisted memory exclusion) [46], Compiler-Assisted Full Checkpointing [40], Compiler-Assisted Static Checkpoint Insertion [41] and Compiler-Enhanced Incremental Checkpointing for OpenMP Applications [18].

On top of the software stack is the user level. In [8] and [35] checkpointing techniques on the user level can be found.

All checkpointing techniques can be classified into two main categories, the coordinated checkpointing and the uncoordinated checkpointing. In coordinated checkpointing a consistent global checkpoint on stable storage is taken [19]. This means that the state of every process is taken periodically and in case of a failure the system is restarted from a consistent global state, determined by the checkpoints of each process. A state of a system is considered consistent if no orphan or missing messages are existing.

Orphan messages are messages that have been sent by one process after the checkpoint and received by the other process before the checkpoint. If the system is rolled back to the checkpoint, the message will be sent, but no process will receive this message. A message that is sent before the checkpoint of the sending process and received by the corresponding process after its checkpoint is a missing message. In case of a rollback the receiving process wants to receive the message, but the message is never sent. Figure 2.6 is visualizing this problem.

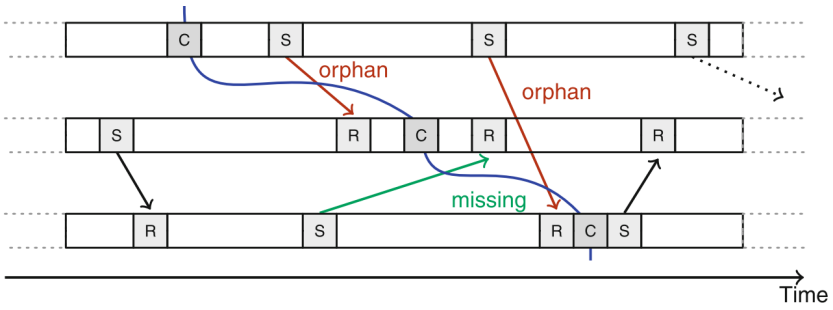

Figure 2.6: Orphan and missing messages [22]

The variants of coordinated checkpointing are wide, examples can be found in [36, 60]. Furthermore, coordinated checkpointing can be categorized into blocking and non-blocking techniques. In blocking coordinated checkpointing the computation is stopped before the checkpoint is taken. This shall allow better control of taking checkpoints for the processes. In the non-blocking coordinated checkpointing the computation can continue without having to stop for a checkpoint. The blocking method is easier

to implement, while the non-blocking technique might require modifications which can introduce an overhead [20]. The benefit of coordinated checkpointing is that it is easy to use, developers do not need to put additional effort into making applications fault tolerant.

The main drawback of every coordinated checkpointing technique is the overhead introduced by it. Checkpoints of every process have to be taken periodically, checkpoints have to be saved on a resilient storage and the complete system has to be restarted from a checkpoint in case of failures.

Contrary, the uncoordinated checkpointing tries to overcome the drawback of the coordinated checkpointing. The uncoordinated checkpointing needs no coordination of checkpoints and does not require all processes to restart in case of a failure. In the optimal case only the processes hit by an error are restarted. It requires no synchronization between processes at checkpoint time but has the major drawback of the *domino effect*, if no set of checkpoints form a consistent global state, the application has to be restarted from the beginning in the event of a failure [24, 32]. In general this problem is solved by logging all messages and therefore making this technique expensive. Message logging can be classified into optimistic and pessimistic message logging. In pessimistic logging the protocol is ensuring that every event is logged safely before any send operation can proceed. Optimistic message logging buffers events in the process memory and logs them asynchronously [15]. Other classifications of message logging can be casual and optimal [4].

The range of modifications and improvements of checkpoint restart techniques is very wide. Often checkpointing is combined with other fault tolerance techniques, like fault prediction (see Section 2.4.3). But even with optimizations checkpointing is not a feasible technique for this work, to design fault tolerant collective communication algorithms. Furthermore, with regard to the MTBF of exascale systems, checkpoint restart techniques can perform poorly and other fault tolerant techniques can become more relevant.

2.4.2 Replication

Replication describes a technique in which resources are multiple and redundant. These resources can, for example, be processes or memory. In case of process replication a computational task is handled by several process cores. If a task is handled by two processes, both will perform the same computational steps and therefore introduce the doubled amount of computational power. In case of no process failures the additional computational power is wasted. But in case of a failure, the failed process interrupts its computation and is not able to continue. In this case the additional process simply continues the task and no restart is required. The technique of process replication is often called state machine replication [51]. Scenarios in which more than one additional resource is used are also possible especially in systems with a high failure rate. The benefit of this fault tolerance technique is its efficiency. In case of failures no restart is required and no time is spent, e.g., for writing checkpoints. The major drawback is the overhead introduced by using resources redundantly. Furthermore, the replicas have to remain strongly synchronized, which is also introducing additional overhead [11].

In [27] an evaluation of process replication can be found. It states that process replication will be one of the major fault tolerance techniques in future exascale systems, since other techniques like checkpoint-restart are predicted to double the time needed by an application [27].

2.4.3 Fault Prediction

Fault prediction describes a technique which is not offering fault tolerance for a distributed system itself, but is often used in combination with other fault tolerance techniques to improve their performance. A fault predictor is a mechanism that warns users about upcoming failures in the system. A predictor can be classified by two parameters, the recall and the precision. The recall is the fraction of faults that are predicted and the precision is the fraction of correct predictions [22]. These parameters allow to estimate the usefulness of a predictor and to combine it with different resilience techniques, see Section 2.4.4.

2.4.4 Migration

Migration combines advanced, proactive failure detectors with some other form of fault tolerance, like checkpointing [11]. With these techniques the performance of the base fault tolerance technique can be increased. In the case of checkpointing the checkpoints have to be taken periodically and thus a huge overhead is introduced. With a fault prediction technique the points in time on which errors are likely to strike can be predicted. Therefore, the best point in time to take a checkpoint is predicted and the huge overhead of taking checkpoints periodically can be decreased significantly.

But not only checkpointing techniques can be improved by this technique. In migration, it is also possible that a process will be moved or replicated to a different node if it is likely that the old node will fail.

2.4.5 Transaction

In transactions the algorithm of the computation is divided into blocks. After every block a check is performed to indicate if all communications in this block have succeeded. If so, the algorithm continues. Otherwise, the application returns to the status before the execution of this block. Since this technique is repeating computations in case of failures, it can be seen as a lightweight checkpointing technique [11].

2.4.6 Algorithm-Based Fault Tolerance (ABFT)

Algorithm-based fault tolerance, also called application-specific fault tolerance, describes a technique in which fault tolerance is introduced by modifying the algorithm used for the computation. The benefit of this technique is that almost no performance degradation is introduced. The main drawback is the effort which is needed by the developer to change the algorithmic behavior for resilience.

In [21] an exemplary implementation of algorithm-based fault tolerance can be found. A technique without checkpointing is used to provide fault tolerance for the Linpack Benchmark, which is a benchmark for measuring the floating-point computation power of a system. This technique is based on the mathematical property that a checksum which is added to the matrix before factorization can still be maintained after it. With the help of this checksum in every step of the computation errors can be detected and repaired. No rollback and no periodic checkpoints are needed which is why this method is introducing a negligible overhead for fault tolerance.

Another prominent example of algorithm-based fault tolerance in matrix factorization can be found in [16, 23]. In this method a combination of checkpointing and algorithm-based fault tolerance is used. The key characteristic about this method is that it is still able to complete the computation even under extreme conditions, in which the checksum might be lost. Even in this method a negligible overhead for fault tolerance is introduced and the drawback is decreasing as the number computing of units is increasing.

2.5 MPI

MPI is a message passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation (such as a message buffering and message delivery progress requirement) [31]. Currently it is the most applied technique to enable efficient communication in High-Performance-Computing applications. There are many implementations of MPI available, with the drawback that most of them provide no acceptable way for tolerating failures. This is why we analyze fault tolerance in MPI in Section 2.6. MPI is an easy to use technique and provides an abstraction for the developer from lower layers in the software stack like the network layer. It also provides a collection of efficient group communications which allow developers to implement complex computations with complex communications inside.

2.6 Fault Tolerance in MPI

Currently there are multiple implementations of the Message Passing Interface available with the same basic functionality. Mostly all of them share the lack for tolerating failures, also because the MPI standard does not include specifications about failures. Since fault tolerance is becoming a major issue in the High-Performance-Computing community with respect to exascale systems, several approaches tried to introduce fault tolerance into MPI. Because of the complexity, the limited applicability and the high diversity of recovery techniques, none of these approaches has been included into the MPI standard yet [11].

In this section we are introducing the most important examples for fault tolerant MPI implementations. An overview about the mentioned implementations can be seen in Figure 2.7

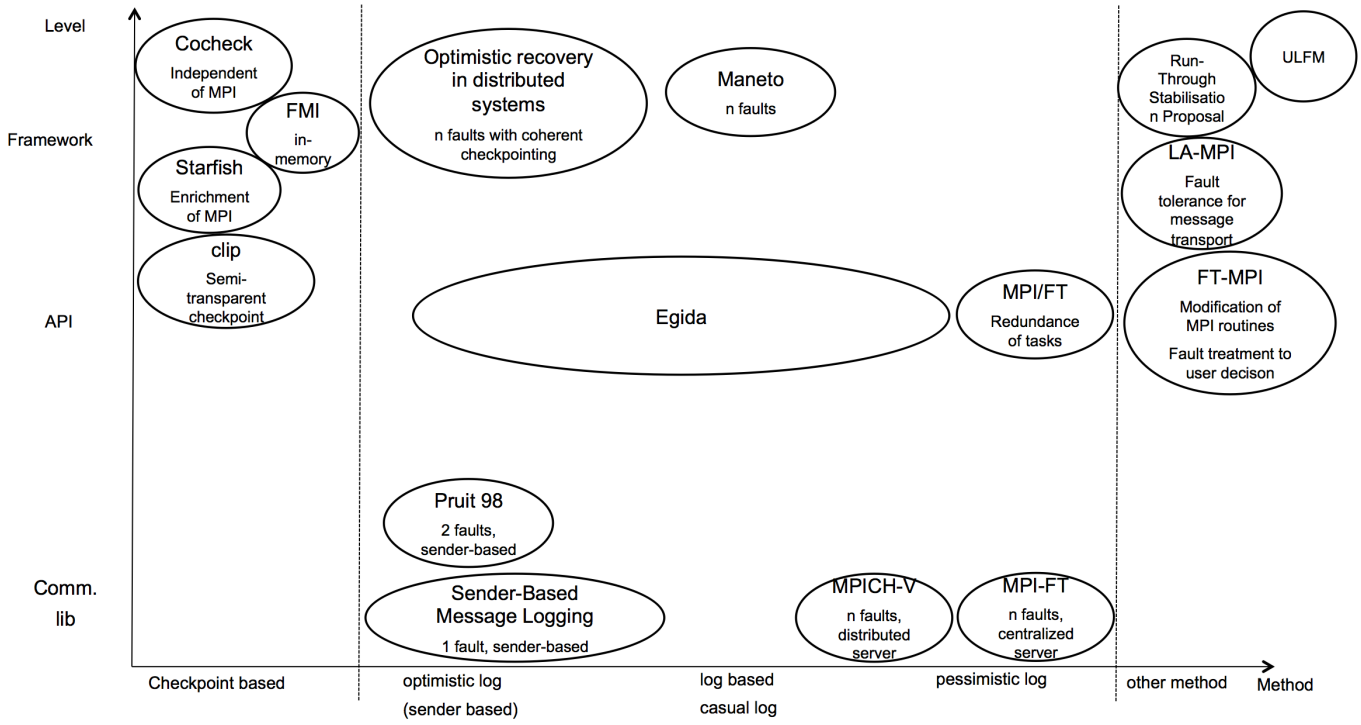

Figure 2.7: Fault Tolerant MPI Implementations [14]

One of the first fault tolerant MPI implementations was CoCheck [55]. The CoCheck environment provides fault tolerance by using checkpointing and migration techniques. It was implemented on top of an existing MPI implementation and therefore represented a library for MPI itself.

Another MPI implementation with the support for fault tolerance is Starfish. In Starfish fault tolerance is introduced by combining group communications with checkpoint restart techniques. In Starfish a Starfish daemon is started on every node in the system. These daemons are responsible for interacting with clients, spawn MPI programs, track and recover from failures [3].

In FT-MPI [25] the states of a communicator are extended from being invalid or valid to *FT_OK*, *FT_DETECTED*, *FT_RECOVER*, *FT_RECOVERED* and *FT_FAILED*. With the help of this extension FT-MPI allows the application to define its behavior in case of failures. It provides the possibility to shrink communicators, by excluding failed processes and offers therefore the support for ABFT techniques.

MPI-FT [42] uses an observer to detect failures and uses message logging to recover from them. In error cases the observer is spawning new processes and adds them to the communicator.

MPI/FT [7] is a middleware/tool for fault detection and recovery. It uses parallel self-checking threads to introduce robustness and supports user-coordinated recovery and checkpointing. It can detect erroneous messages by introducing a voting algorithm and can survive process-failures [26].

MPICH-V [14] and the successor MPICH-V2 [17] support a mix of uncoordinated checkpointing and distributed message logging.

LA-MPI (Los Alamos Message Passing Interface) [30] supports fault tolerance for message transportation, this means it is focusing on network fault tolerance. Fault tolerance for node failures is not supported.

The Fault Tolerant Messaging Interface (FMI) [50] is a framework which is focusing on fast failure recovery. The approach of FMI is to use a survivable communication runtime and recover failures with fast, in-memory checkpoint-restart techniques and dynamic node allocation.

The Run-Through Stabilization Proposal [34] is a first proposal about semantics in MPI which allows the application to continue on failures. The idea is to prevent the fail stop behavior of standard MPI applications which stop if a failure is recognized. This approach was proposed by the MPI Forum's Fault Tolerance Working Group which also presented the User Level Failure Mitigation Framework (ULFM).

The User Level Failure Mitigation Framework (ULFM) [10] is the latest proposal of the MPI Forum's Fault Tolerance Working Group. ULFM focuses on fail-stop process failures only and provides mechanisms for application-level failure detection [37]. In ULFM new semantics are added to MPI in order to make applications able to react to failures. Failure notification is performed on a per-operation basis and indicate whether an operation was successful or not [11]. It offers the following new constructs.

- ***MPI_COMM_REVOKE***: Revokes a communicator by invalidating it for further communications.
- ***MPI_ COMM_SHRINK***: Shrinks a communicator to a new stable one by excluding all failed processes.
- ***MPI_COMM_FAILURE_ACK***: Acknowledges the occurrence of a failure.
- ***MPI_COMM_FAILURE_GET_ACKED***: Returns the group of the failed processes. This group is determined by acknowledging the occurrence of failures.
- ***MPI_COMM_AGREE***: Agreement over the group of functioning processes. This function is useful to determine a consistent state of the application and also to identify failures.

The main benefit of the ULFM Framework is that in failure free cases the overhead introduced by the new semantics is insignificant [12]. Because of the flexibility, the insignificant overhead and the non-deadlock guarantee provided by the ULFM interface, this approach is used for the development of fault tolerance mechanisms for collective communications in this work.

3 Fault Tolerance in Collective Communication Algorithms

In Chapter 2 different techniques for resilience in distributed systems and possibilities to use these techniques in MPI are shown. The technique which is currently used by the majority of the High-Performance-Computing community is checkpointing. The main drawback of checkpointing is the overhead introduced by it. Checkpoints have to be taken periodically, stored efficiently and resistantly and computations have to be restarted. With the increasing size of distributed systems, checkpointing is becoming more costly. Therefore, other solutions, with regard to exascale systems, have to be developed.

In this chapter we are proposing a new approach to resilience by securing collective communications against the major threat in distributed systems, the failure of processes or even nodes. In the following we explain the general strategy of our approach for fault tolerance in collective communications, before we outline the fault tolerant collective communication algorithms, which we have designed and show their applicability in case of a distributed sorting algorithm. *Broadcast*, *Scatter*, *Gather*, *Reduce* and the *All-to-all* communication are the algorithms which we examine in this work. These collective communications form a basic subset of all the available collective communications and more complex collective communications can be performed by using them. For example the *Allgather* communication is a combination of a *Gather* communication followed by a *Broadcast* communication. Furthermore, the techniques used to design resilience in this work can be applied to any other collective communication.

3.1 Fault Tolerance Strategy

Our goal is to design a fault tolerance strategy which is efficient, correct and in general applicable to collective communications. We are not including checkpoint restart techniques in the algorithms but propose a run-through approach for fault tolerance. The User Level Failure Mitigation framework provides the semantics, which we use in this fault tolerance strategy. We understand under correctness that the algorithms are always terminating successfully, that no deadlocks occur and that the end result remains unchanged in case of failures. To identify failures and to restore an error-free state we are using the semantics introduced by the ULFM framework. The ULFM framework adds an insignificant amount of overhead [12] and is providing a non-deadlock property. This supports our aim to design efficient, resilient algorithms.

Figure 3.1 is sketching the steps of our fault tolerance strategy.

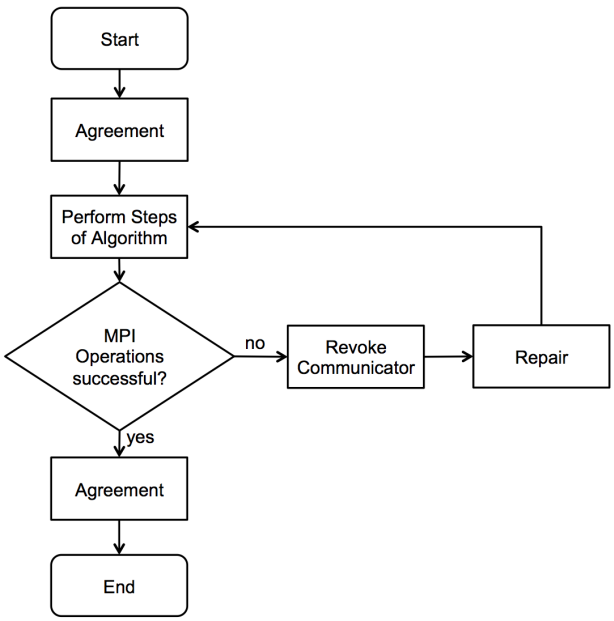

Figure 3.1: Fault Tolerance Strategy

In our strategy to include fault tolerance into collective communication algorithms we start every algorithm with an agreement, provided by the ULFM framework (*MPI_COMM_AGREE*), in order to identify errors which have happened before the execution of the communication starts. In case of occurred errors the algorithm performs the repair routine, which is explained in Section 3.1.1, and returns afterwards to the normal execution. Therefore, we guarantee the successful execution of an algorithm even in case of failures that happened at an earlier point in time and not during the collective communication. If no errors have happened, the algorithm starts the normal execution. If failures appear during the execution of the algorithm, the errors are noticed and the MPI communicator will be revoked. This means that no further communication can succeed on this communicator and all processes, even those which are not included in the failed communication, can notice these errors. If a process has recognized an error, it jumps into the repair routine. After the repair routine the normal execution can continue because a failure free state is restored. At the end of the algorithm a second agreement is performed, to verify that no failures happened between the last communication and the return of the result. Therefore, we guarantee the correct execution of the algorithm independent of the number and points in time of the occurred failures. Furthermore, we assure that no deadlocks and no unexpected terminations occur as long as at least one process is staying responsible.

The run-time complexity of the algorithms changes due to the overhead introduced by this strategy. In our analysis, we neglect the runtime complexity for transmitting a message. In case of failure free executions the added overhead is minimal. This is due to the agreements at the start and at the end of the algorithms. An overview about the run-time complexities and implementation details of the semantics of ULFM can be found in [12]. In a scenario with P processes, an agreement is performed in $O(\log(P))$. In the implementation of ULFM the agreement is performed as an all-reduce collective communication. First a reduction of input values to a coordinator is conducted before the coordinator makes a decision and broadcasts the output value. Therefore, the overhead of this strategy is also in $O(\log(P))$.

In case of errors more overhead is introduced. Revoking a communicator is performed by using the topology of a binomial graph [5]. The alternative is to use flooding, but this approach is not scaling [12]. In a communicator of size P the revoke algorithm is performed in $O(P \log(P))$, because at the reception of a revoke message, every process acts as new initiator and broadcasts the message in a binomial graph topology. Furthermore, overhead is added by the recovery routine.

3.1.1 Recovery

In the recovery routine the state of the algorithm is repaired. In case of failures certain processes are not responsive anymore and have to be excluded from the communication. Since these processes are performing a particular task which is necessary for the successful termination, these tasks can not be omitted.

Therefore, we have introduced a task model for the fault tolerant collective communications. Every process is responsible for performing certain tasks. If no failures occur, the processes are only responsible for their own task. If failures happen, the lost tasks are assigned to other processes, which will participate in the communication as the failed process.

In our strategy for repairing the state of the communication to a failure free state we have chosen to introduce process redundancy. Instead of performing the same operations on multiple processes, like in process duplication, we have chosen to not introduce redundancy before the occurrence of failures. This means that the task of a failed process is taken over by an existing process. How the new process is chosen is not relevant for the continuation of the communication but in order to keep the execution of the algorithm balanced we choose the new process in a fair way to avoid that one process is handling all the failed tasks and to retrieve a fair distribution of tasks.

But choosing a new process for handling a failed task is not sufficient in order to create a stable failure free state of the algorithm. The MPI environment has to be repaired too. To do so all the failed processes have to be identified first. For that purpose we are using the ULFM constructs *MPI_COMM_FAILURE_ACK* and *MPI_COMM_FAILURE_GET_ACKED*. *MPI_COMM_FAILURE_ACK* is called

before *MPI_COMM_FAILURE_GET_ACKED* is returning the group of the failed processes. After this group has been identified, the repair routine establishes new processes for the tasks of the failed ones. Afterwards the failed processes have to be removed from the communicator. This operation is performed by using the ULFM function *MPI_COMM_SHRINK*. This function is implemented analogous to the agreement function and has therefore also a run-time complexity of $O(\log(P))$ [12]. After executing this function a new repaired communicator without failed processes is available for the further communication of the algorithm.

Therefore, the total overhead in case of failures can be identified by the run-time complexity of $O(P \log(P))$, because two agreements in $O(\log(P))$, one revoke operation in $O(P \log(P))$ and one shrink operation in $O(\log(P))$ are performed. After the recovery has been performed, the algorithm continues its execution. In the normal execution only the communications which have not been performed successfully are repeated, in order to reduce the costs of failures. This means that the algorithm has not to be restarted completely from a certain point in time as in checkpointing techniques and the overhead of terminating the algorithm successfully is reduced in comparison to checkpoint-restart approaches.

Since our strategy is focusing on the applicability in the area of distributed database systems we are assuming that the data of a failed process is not lost but recoverable. How this recovery is performed is left to the implementer who is using the fault tolerant collective communication. This recovery can either be performed by introducing data redundancy which allows the quick recovery of the lost data or by using a resilient DBMS. In a resilient DBMS data is recoverable for example by recalculating it. This approach is used in resilient distributed datasets (RDDs) which are used for example by the framework Apache Spark [61]. An interface for restoring the lost data is given in the signature of the implementation of the proposed algorithms (see Chapter 4).

The discussed steps of the repair routine are depicted in Figure 3.2.

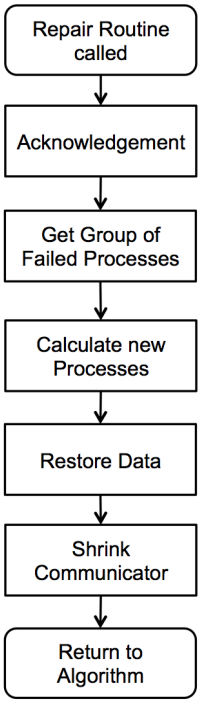

Figure 3.2: Repair Routine

3.1.2 Correctness of Fault Tolerance Strategy

In the following we argue shortly that the proposed fault tolerance strategy is correct. We assume that the semantics, provided by ULFM, are correct and the non-deadlock property holds. Furthermore we

assume that the collective communications and the proposed fault tolerance strategy are implemented correctly. We define correctness for fault tolerant collective communications similar to [52], based on the task model, which we have introduced.

Fault Tolerant Collective Communication: For any task the associated functioning process has received the corresponding message.

Assuming the proposed fault tolerance strategy is wrong. Then a task has to exist, which has not received the corresponding message. Because of the introduced task model, every process knows which tasks it has to execute and from which tasks it is receiving message. With that knowledge and the proposed recovery of our strategy, a process can always decide which message corresponds to which task. Therefore, if the collective communication and the proposed fault tolerance strategy are implemented correctly, the process can not receive a message which does not belong to any of its tasks. Thus, the process of the task, has not received the message and is still waiting to receive it. This scenario is only possible, if the process is waiting to receive a message from a non functioning process. The communication is stuck in a deadlock. This contradicts our assumption of the non-deadlock property and therefore, our proposed fault tolerance strategy is correct.

3.2 Broadcast

In the broadcast group communication a root process has a message for every other process which is participating in the communication. The message that is distributed to every process is identical. In our implementation we make use of a binomial graph in order to design this algorithm in an efficient and scalable way. A description about binomial graph topologies can be found in [5]. In a naive implementation the root process sends the same message to every other process. This means the same message is sent p-1 times, where p is the number of processes participating in the communication and therefore has a run-time complexity of $O(p)$.

In the binomial tree broadcast algorithm, the algorithm has a complexity of $O(\log(p))$. In every step of the algorithm the number of processes, to which the message has to be send to, is reduced to the half. The processes which have received the message are further participating in the communication by sending the message to continuing processes. This algorithm is shown in Figure 3.3. A binomial tree for 9 processes is shown, the arrows indicate the send operation of the message.

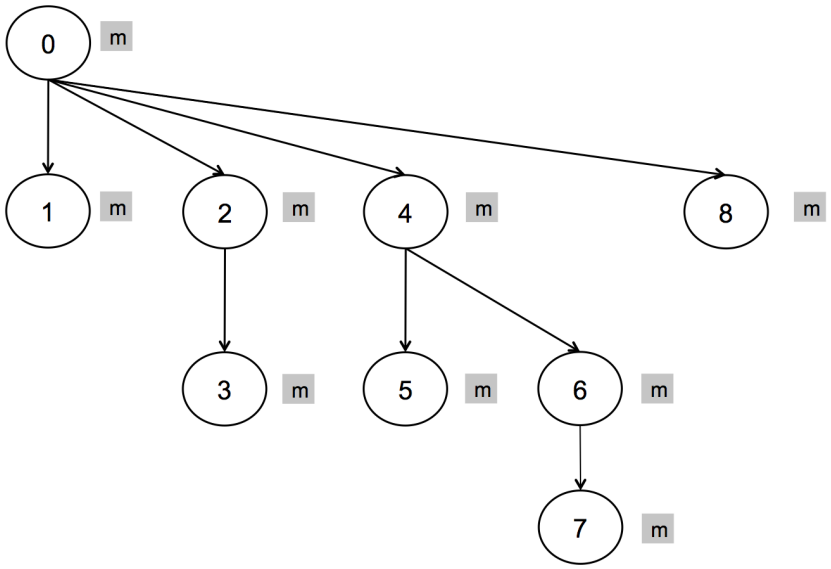

Figure 3.3: Binomial Tree Broadcast

In the first step of the algorithm the root process, in this case 0, has the message m which should be sent to every process and is sending the message to the Process 1, 2, 4 and 8. In the next steps the Process 2 is forwarding the message to 3, 4 is forwarding the message to 5 and 6, 6 is sending the message to 7. If more processes would be participating, the binomial tree would be built further accordingly to the same structure.

The pseudocode for the binomial tree broadcast algorithm with fault tolerance techniques is shown in Algorithm 1. The algorithm is shown for a number of processes p and the root process 0. It is to mention that the root process is not receiving any message (Line 2-3) and the last process with the ID p - 1 is not sending the message to other processes (Line 5-6).

Fault Tolerance: To introduce fault tolerance in this algorithm the fault tolerance strategy explained in Section 3.1 is used. Therefore, an agreement is performed at the beginning (Line 1) and at the end (Line 8) of this algorithm. If an error is occurring during this algorithm, the MPI communicator will be revoked and the repair strategy is applied. For example, if Process 4 is failing during the communication, Process 5 will take over the task of Process 4. Since the processes involved in the failed communication are Process 0, 4, 5 and 6, only this communication path has to be repeated. Therefore, process 0 will send the message m to Process 5 which will forward the message to Process 6. For reasons of simplicity we have not illustrated the task model in the pseudocode.

Algorithm 1 Binomial Tree Broadcast

```
1: Perform Agreement
2: if ProcessID > 0 then
3:     Receive message m
4: end if
5: for k := min{⌈log p⌉ , trailingZeroes(ProcessID)} − 1 downto 0 do
6:     Send m to process ProcessID + 2^k
7: end for
8: Perform Agreement
```

3.3 Scatter

The scatter communication is performed similar to the broadcast communication. One root process is sending a message to every other process, the difference is that every process receives an individual message, in broadcast every process is receiving the same message. We have used a binomial tree based implementation similar to the implementation of the broadcast communication. Therefore, the scatter communication has a run-time complexity of $O(\log(p))$.

The main difference between scatter and broadcast is the sending of different messages and different message sizes. We consider an example with 9 processes. In the first step of the algorithm the root process, here 0, is sending the message *m1* to Process 1. To Process 2 it is sending the messages *m2* and *m3*. To Process 4 the messages *m4*, *m5*, *m6* and *m7* are sent. To Process 8 only the message *m8* is forwarded because this is the last process in the communication. In the next step Process 1 is not sending any message, Process 2 is sending the message *m3*, which is intended for the Process 3, to Process 3. Process 4 is forwarding the messages *m5* to Process 5 and *m6* to Process 6. In the last step Process 6 is forwarding the message *m7* to Process 7. This procedure is visualized in Figure 3.4.

The pseudocode for the implemented scatter algorithm is shown in Algorithm 2. It is to mention, that the number of the messages which are retrieved and sent is determined via the process id (Line 6).

Fault Tolerance: The fault tolerance strategy of Chapter 3.1 is applied to the scatter algorithm. At the start (Line 1) and at the end (Line 9) of the communication an agreement is performed to identify failed processes. In case of failures the state of the algorithm will be repaired and only the unsuccessful communications are repeated. The task model is not illustrated in the code.

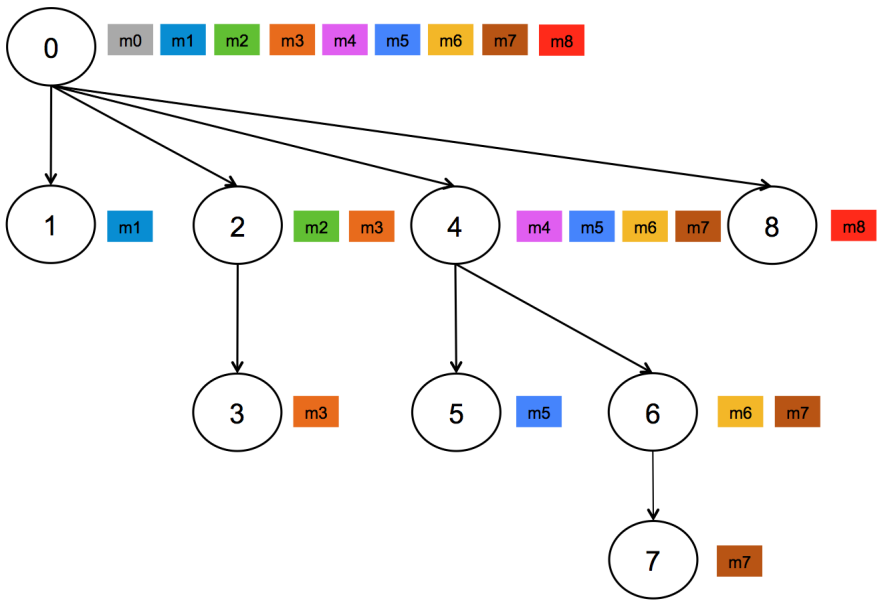

Figure 3.4: Binomial Tree Scatter

Algorithm 2 Binomial Tree Scatter

1: Perform Agreement
2: **if** ProcessID $>$ 0 **then**
3: Receive (ProcessID - SenderID) different message
4: **end if**
5: **for** k $:=$ min$\{\lceil \log p \rceil$, trailingZeroes(ProcessID)$\} - 1$ **downto** 0 **do**
6: Send (ProcessID + 2^k - ProcessID) different messages to process ProcessID + 2^k
7: **end for**
8: Perform Agreement

3.4 Gather

The purpose of the gather group communication is to gather information at a specified process. Therefore, many processes are sending different messages to one root process. To design this communication we have chosen to use the same approach as in the scatter communication, a binomial tree. The runtime complexity is in $O(\log(p))$, because the number of participating processes is halved in each step and thus a binomial tree of height $\log(p)$ is created. The communication is the exact opposite of the scatter communication which is also highlighted by the pseudocode for this algorithm, in Algorithm 3.

Every process is calculating the amount of different messages it has to send (Line 3) and the amount of messages it has to receive (Line 6) with its own process id and the id of the process it is communicating with.

Fault Tolerance: Because of the algorithmic structure we apply the fault tolerance strategy as we do for broadcast and scatter. The algorithm starts by performing an agreement (Line 1) and calls the repair routine, if that agreement was not successful. Afterwards the algorithm is executed. If any MPI communication during the algorithm is not successful, the communicator is revoked and the repair routine is called.The other processes are noticing that the communicator is revoked and call the repair routine too. These steps and the task model are not included in the pseudocode. At the end of the algorithm a second agreement is performed (Line 8) to guarantee the correctness of the result.

In Figure 3.5 an example for a gather operation with 9 processes and the root process 0 is shown.

Algorithm 3 Binomial Tree Gather

1: Perform Agreement
2: **if** ProcessID > 0 **then**
3: Send (ProcessID + 2^k - ProcessID) different messages to process ProcessID + 2^k
4: **end if**
5: **for** k := min{$\lceil \log p \rceil$, trailingZeroes(ProcessID)} $- 1$ **downto** 0 **do**
6: Receive (ProcessID - SenderID) different messages
7: **end for**
8: Perform Agreement

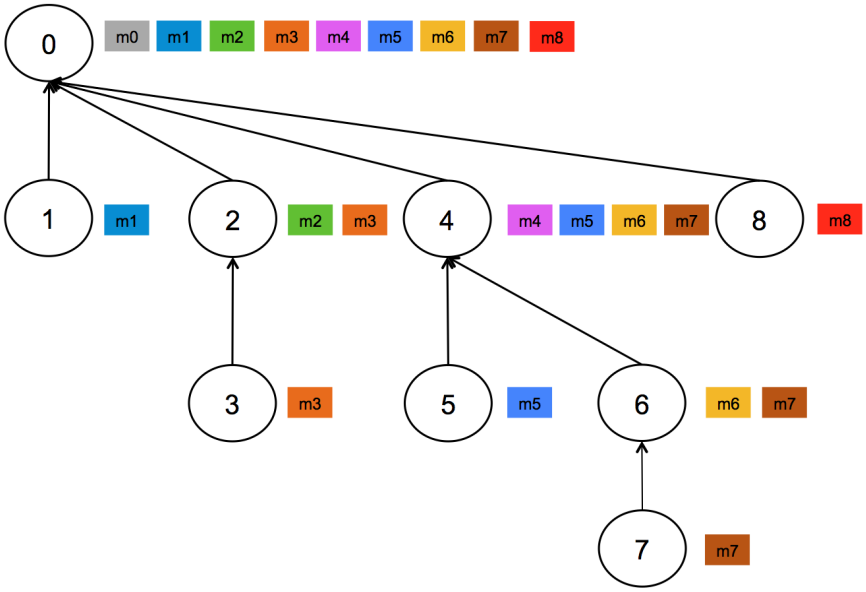

Figure 3.5: Binomial Tree Gather

In the first step of this algorithm Process 7 is sending the message *m7* to Process 6. In the next step Process 3 is forwarding *m3* to Process 2, Process 5 is forwarding *m5* to 4 and Process 6 is forwarding the messages *m6* and *m7* to Process 4. In the last step Process 1 is sending *m1* to Process 0, 2 is sending *m2* and *m3* to 0, Process 4 is forwarding the messages *m4*, *m5*, *m6* and *m7* to 0 and 8 is sending *m8* to 0. After this step, all the information is gathered at 0.

3.5 Reduce

The structure of the reduce communication is slightly different, compared to the structure of the previously shown communications. The idea is to reduce a set of messages into a smaller set of messages. This means that all processes have at least one message which should be sent to the root process. The difference to the gather communication is that the root process is not gathering all the messages but applying a certain operation to reduce this list of data into a smaller list. For example every process can hold a number and the root process is reducing this list of numbers to the sum of it. Other often used operators are the product of numbers and the maximum or minimum. Furthermore, it is possible to use self defined operations in the reduce communication.

We have designed the reduce algorithm accordingly to the broadcast, scatter and gather algorithm by using a binomial tree. Therefore, the runtime complexity is in $O(\log(p))$. The binomial tree for this algorithm is built different compared to the binomial tree in the previous algorithms because the reduce operation can already be performed after every communication step. An example for the binomial tree of a reduction is shown in Figure 3.6.

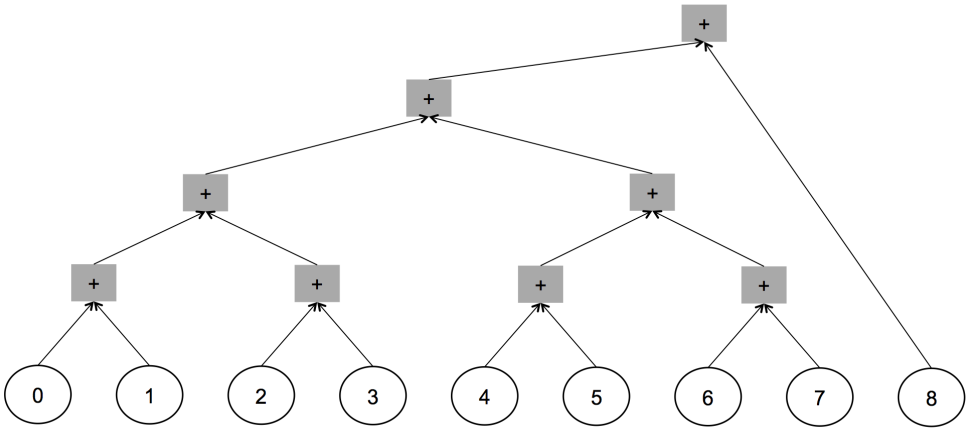

Figure 3.6: Binomial Tree Reduce

In this example 9 processes are participating in the communication and are calculating the sum of numbers by performing a reduce communication. The communication is split into three steps. In the first step Process 1 is sending its number to Process 0, Process 0 is adding the number to its own and is holding the temporary result. The same action is taking place for Process 2, 3, 4, 5, 6 and 7. At the end of this iteration the temporary results for this step are stored at Process 0, 2, 4 and 6. In the next step Process 2 is sending its temporary result to Process 0. Process 0 is adding this number to its own and storing the temporary result for this step. The same is done for Process 4 and Process 6. At the end of this step the temporary results are at Process 0 and Process 4. After this, Process 4 is sending the temporary result to Process 0 and Process 0 is performing the operation. Process 0 is holding the temporary result. In the last step Process 8 is sending its number to Process 0. Process 0 is adding the number to the temporary result of the previous step. After that, Process 0 is holding the final result.

Fault Tolerance: Since the algorithmic structure of this communication is different to the previous communications, the fault tolerance strategy is also performed in a different way. Again the basic idea is to finish the communication with the remaining processes and to guarantee the termination of the communication and the correctness of the result. This means that in case of a failure a new process is chosen fairly to take over the task of the failed process. The iteration in which the error occurred is repeated and only the necessary communications are performed. Because of the computations that are performed in every step, we have introduced a backup strategy. In every step of the algorithm the number of active processes is halved. Therefore, the processes which will not participate in the communication are serving as backup nodes. After the computation has been performed, the result is sent back to the process which has sent the original message. For example, in the first step Process 0 is receiving the message of Process 1 and performing the operation. After the operation has ended successfully, Process 0 is sending the result back to Process 1. Process 1 is not participating in the remaining communication and can therefore be used as backup node. If Process 0 fails in the next iteration, Process 1 will take over the task of Process 0 and continue with the backup of the temporary result. Therefore, no communication and no additional operation has to be performed. If a backup node fails, no communication has to be repeated too. If a process fails and no backup is available, another process has to take over the task of this process and has to restore the lost message. All the communications and operations which are necessary to recreate the lost result have to be performed again. If a backup result of a previous iteration is available, this backup will be used to save the costs for the communications and operations.

This backup strategy is adding twice the amount of needed communications as overhead. But in case of failures, especially for failures near to the end of the communication, this strategy can save a lot of time and computational power. Also, for complex operations this backup strategy is saving a lot of resources in case of failures.

The pseudocode for the reduce algorithm including the backup strategy is shown in Algorithm 4. The task model is not visualized.

Algorithm 4 Binomial Tree Reduce

```
 1: Perform Agreement
 2: active := 1;
 3: i := ProcessID;
 4: for k := 0 to ⌈log p⌉ do
 5:    if active then
 6:       if bit k of ProcessID then
 7:          Send message x_i to i - 2^k;
 8:          active := 0;
 9:          //optional
10:          Receive backup x_{i-2^k} from i - 2^k;
11:       else
12:          Receive message x_{i+2^k} from i + 2^k;
13:          Perform operation x_i := x_i ⊕ x_{i+2^k};
14:          //optional
15:          Send backup x_i to i + 2^k;
16:       end if
17:    end if
18: end for
19: Perform Agreement
```

The backup strategy is included in the pseudocode but marked as optional (Line 11 and Line 16). Otherwise, the same steps for fault tolerance are included. At the beginning of the algorithm an agreement is performed (Line 1) to check for errors which have happened previously. During the algorithm failures will be recognized and the repair routine will be called. At the end an agreement is performed (Line 20) to verify the correctness of the result.

In case of distributed database systems a cost-based optimizer can be used. This optimizer can determine whether it is useful to do backups based on the costs of the computations, that might be lost in a failure case.

3.6 All-to-all

The purpose of the all-to-all communication is to exchange messages with every other process. This means in a communication with p processes every process is holding messages m_0, ..., m_{p-1} for the corresponding processes and wants to exchange these messages with them. Therefore, the communication in this pattern is always two-sided. We have designed this algorithm by following the algorithmic structure of the 1-factor algorithm [48]. In every iteration of the algorithm it is estimated which process pair is communicating with each other. If the number of processes p in the communication is odd, one process is always idle in every iteration. The maximal number of communication steps that are needed to conclude the communication is p - 1 steps. So the runtime complexity of this algorithm is in *O(p)*. In Figure 3.7 an example for the 1-factor algorithm is illustrated.

In total 5 iterations are needed. For every iteration different communication pairs are estimated. For example in iteration 0 Process 0 and 5, 1 and 4, 2 and 3 are communicating with each other. This means that in every communication every process is sending and receiving a message. After iteration 4 every process has communicated with every other process and the communication is finished. The behavior of this algorithm is shown as pseudocode in Algorithm 5. The task model is not included.

Fault Tolerance: The fault tolerance strategy follows the structure accordingly to Section 3.1. Before the execution of this communication, an agreement is performed (Line 1) to identify errors which

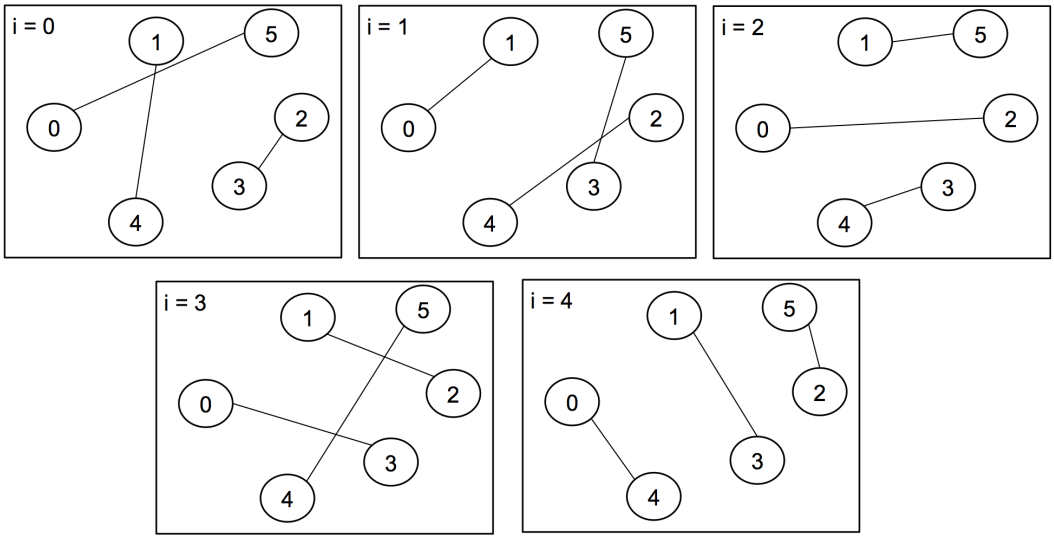

Figure 3.7: 1-Factor All-to-all

happened before the communication. Then the algorithm is performed until one process is noticing the failure of another process. If a failure is recognized the communicator is revoked and every communication afterwards is not able to complete successfully. Therefore, every process will switch to the repair routine in which another process is estimated to take over the task of the failed process. This new process is restoring the data of the failed node and the algorithm will restart in iteration 0. The communications which have not succeeded are being repeated. At the end of the algorithm another agreement is performed (Line 20) to guarantee that the algorithm has finished without a failure after the last communication.

Algorithm 5 1-Factor Algorithm: Alltoall

```
 1: Perform Agreement
 2: if p mod 2 then
 3:    for k := 0 to p do
 4:       Exchange message with process ((k - ProcessID) mod p)
 5:    end for
 6: else
 7:    for k := 0 to p-1 do
 8:       idle := ((ProcessID / 2) * k) mod (p -1 )
 9:       if ProcessID == p - 1 then
10:          Exchange message with idle
11:       else
12:          if ProcessID == idle then
13:             Exchange message with process p - 1
14:          else
15:             Exchange message with process ((k - ProcessID) mod p - 1
16:          end if
17:       end if
18:    end for
19: end if
20: Perform Agreement
```

3.7 Sample Sort

To illustrate the applicability of the proposed fault tolerant collective communications for distributed database systems, we have implemented a distributed sorting algorithm. The distributed sorting algorithm of our choice is the sample sort algorithm [28]. We decided to implement this algorithm, because it is known as the best practical comparison based sorting algorithm for distributed memory parallel computers and is a generalization of the quicksort algorithm [49].

The sample sort algorithm is a sorting algorithm for distributed systems with distributed memory. The goal is to sort n distributed elements in an environment of p processes. The amount of elements at a specific process can vary. Assuming a process p_i has N_i elements, the first element x_0 has to be greater than the last element of process p_{i-1} and smaller than the first element of process p_{i+1}. This means that the result of the sample sort algorithm is correct, if for a process p_i with final sorted data sequence $\{x_{p,0}, ..., x_{p,N_i-1}\}$ the following condition is true.

$$x_{p_{i-1},N_i-1} < x_{p_i,0} \text{ and } x_{p_i,N_i-1} < x_{p_{i+1},0}$$

To achieve this goal, the sample sort algorithm determines partitions in the sequence, that is to be sorted, and assigns these to the participating processes. The processes determine which elements belong to which process and exchange them via communications. In Figure 3.8 the execution of the sample sort algorithm is shown in a flow graph diagram.

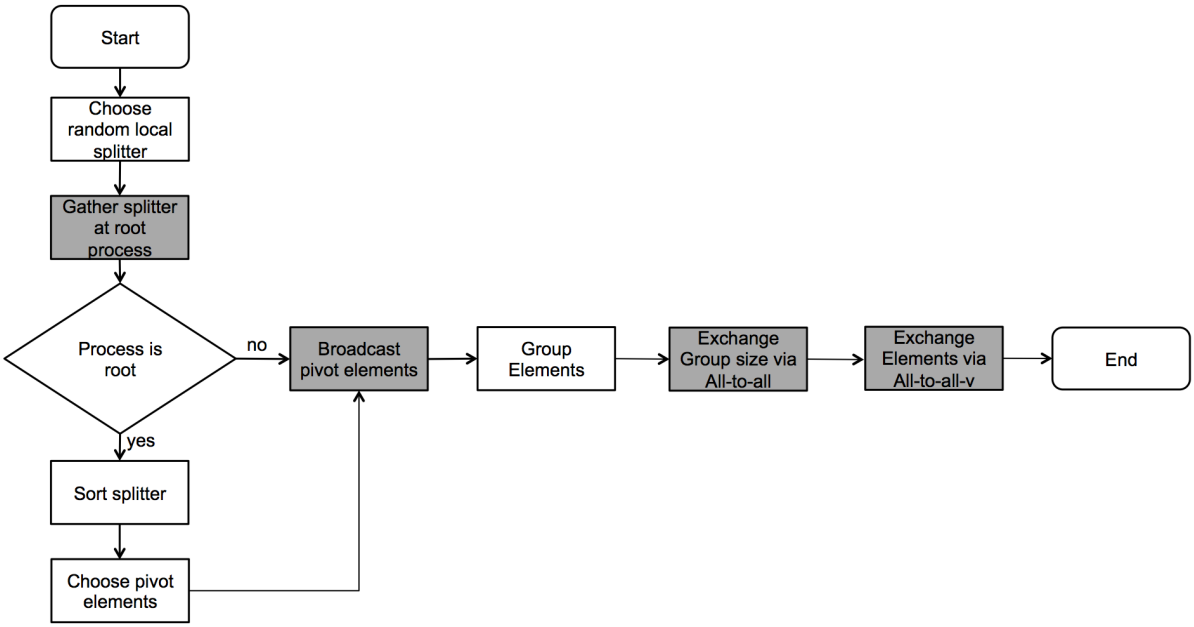

Figure 3.8: Sample Sort Algorithm

At the start of the algorithm the data is distributed over the set of processes. Every process chooses p random splitters out of this data and is performing a gather communication to store the splitters at the root process. The root process is receiving the splitters of every process and is sorting them. Out of the sorting set of splitters the root process is choosing correspondent pivot elements, which represent the limits for each process. These limits are broadcasted to every process. Afterwards every process knows which of its data belongs to which process and can group the data into these ranges. The processes exchange how many entries they are holding for which process via an all-to-all communication and after that they can exchange the elements by using an all-to-all communication for variable message sizes, an all-to-all-v communication. At the end every process holds only the data it is responsible for and can sort this result to know the final result.

Figure 3.9 illustrates an example of the sample sort algorithm. In this scenario 3 processes are available and holding a distributed set of numbers. In the first step every process is choosing randomly local splitters which are gathered at the root. In this case the local splitters for Process 0 are 7, 13 and 25, Process 1 is choosing 6, 17 and 10 and Process 2 has 20, 18 and 21. These splitters are gathered at the root process which is sorting them and choosing the limits for each process. In this example the limits for Process 0 are from $-\infty$ to 10, for Process 1 from 10 to 18 and for Process 2 from 18 to ∞. Each process can now group the data into the corresponding ranges and exchange it. At the end each process holds the sorted data for its range and the algorithm has succeeded.

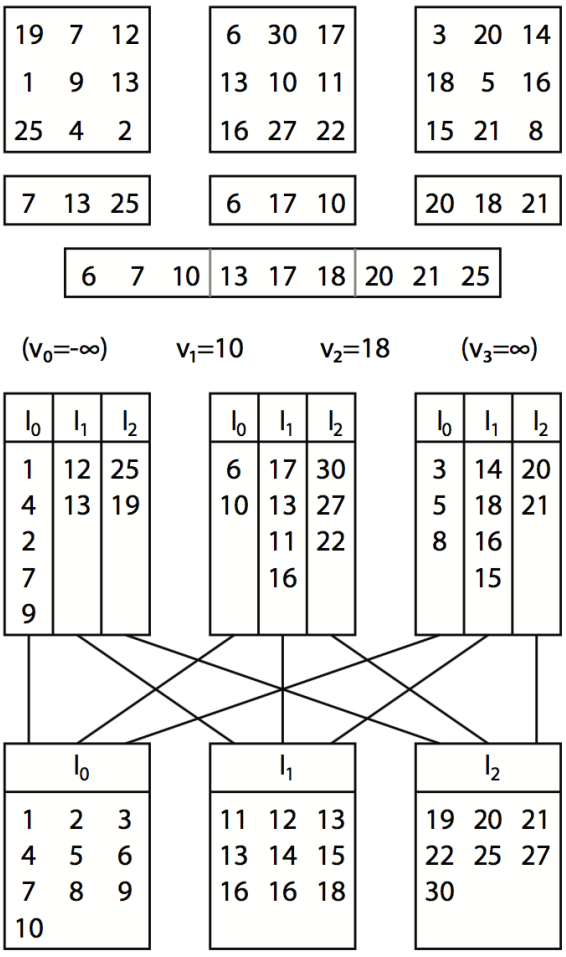

Figure 3.9: Example of the Sample Sort Algorithm [45]

As Figure 3.8 highlights, the workflow of the algorithm includes 4 collective communications, a gather, a broadcast, an all-to-all and an all-to-all-v communication. In order to provide fault tolerance in the sample sort algorithm, we have used for these 4 communications the fault tolerant communications, which we have designed in Section 3. If no fault tolerant collective communications are used, any process failure that occurs at any point in time of the algorithm leads to a non successful termination of the complete execution.

But the usage of the fault tolerant approach requires a functionality to restore lost data. In our implementation we are generating pseudo random numbers and therefore a lost partition can be regenerated by using the ID of the task of the failed process. This data generation can be replaced by different methods or even with replication.

In the execution flow of the sample sort algorithm we have identified 4 different sections in which a strike of failures is handled. In the following we explain which steps are taken in case of failures in these sections and how the data recovery is handled.

The first phase in which errors can occur starts at the beginning of the algorithm and lasts until the end of the gather communication, which is gathering the local splitters at the root process. Any failure that occurs in that period is recognized in the fault tolerant gather communication. The gather communication switches to the recovery routine as described in Section 3 and removes all failed processes. The data is recovered by simply restoring the lost data partition, which can be realized by calculating pseudo random numbers. The regenerated data is stored at the process that takes over the failed task. No further steps have to be performed to recover all the needed data to conclude the gather communication.

The second phase begins after the gather communication and reaches until the end of the broadcast communication. All errors which happen in this period are handled by the recovery routine of the fault tolerant broadcast communication. In any case of failure the lost partition has to be recovered. If the root process fails, all the gathered splitters are lost too. Therefore, the steps of choosing random splitters and gathering them at the new root process have to be repeated. After recovering the gathered splitters, the broadcast communication can continue and complete successfully.

Failures that happen during the third phase are handled by the recovery routine of the all-to-all communication. Therefore, this phase handles failures that happen after the broadcast communication until the end of the all-to-all communication. In case of process failures data is lost and has to be regenerated. Since the last collective communication is a broadcast operation, every process knows the result of it and no communication has to be repeated. Therefore, the process that is taking over a lost task is recovering the lost data partition and is grouping the elements according to the previously received pivot elements, that represent the limits. After this step the process knows the size of the groups and is using these values as input for the all-to-all communication.

The last phase covers the steps after the all-to-all communication until the end of the algorithm. Errors that occur in this phase are handled by the recovery routine of the all-to-all-v communication. In order to recreate a consistent state of the algorithm the lost data partition has to be restored and the steps after the broadcast communication have to be repeated. This means the restored data is grouped accordingly to the limits which have been exchanged by the broadcast communication. The size of these groups was exchanged in a previous step via an all-to-all communication. Therefore, every process knows the sizes of every other process and this step does not have to be performed again. So the newly generated data can be exchanged by using the all-to-all-v communication. After this communication every process has its final result and the algorithm finishes successfully.

4 Enhancing MPI Interfaces for Fault Tolerance

To increase the usability of the implemented fault tolerant collective communications we have adapted the interfaces of the communications slightly in comparison to the functions provided by the MPI implementation. In this section we are introducing the new interfaces and compare them to the interfaces of the original (not fault tolerant) MPI functions. We are explaining the usage of each parameter and highlight the differences between the original and the fault tolerant functions.

4.1 MPI_Bcast

In Table 4.1 the interface for the original MPI broadcast function is shown in comparison to the interface of the fault tolerant alternative.

MPI_Bcast	MPI_Bcast_ft
• **void** *buffer • **int** count • **MPI_Datatype** datatype • **int** root • **MPI_Comm** comm	• **void** *buffer • **int** count • **MPI_Datatype** datatype • **int** root • **map<int, int>** get_rank • **void** (*create_data)(int, int, int, void*, map<int, int>) • **MPI_Comm** comm

Table 4.1: Broadcast Parameter List

The first parameter of the interfaces, *buffer*, is a void pointer. It is pointing to the address at which the data that is to be sent is stored. The parameter is for both variants equal, because of the fact that one process is sending to every other process and it is always known which process is the root process. The second parameter is *count*. It is an integer value which is representing the number of messages that are sent. The messages which are sent are all of the type specified by *datatype*, which is stated by the third parameter. The parameter *root* is an integer value which represents the ID of the root process that is broadcasting all the messages. The last parameter of both interfaces is a MPI Communicator *comm*. It is representing the context in which the communication is taking place.

The difference between the original and the fault tolerant variant are the parameters *get_rank* and *create_data*. The parameter *get_rank* is a map of integer to integer. In this map the process IDs are stored by mapping the corresponding tasks to it, thus the task model of Section 3.1.1 is implemented by using this map. This is needed to provide the non-stop failure recovery strategy introduced by Section 3. At first every task ID is mapped to the same process ID. But this changes in case of errors. For example, we consider that the process with ID 1 is failing and the process with ID 0 is taking the task of the failed process. At first the entries in the map are the following, $get\_rank[0] = 0$ and $get\_rank[1] = 1$, but after the recovery of the error in process 1, the entries are $get\_rank[0] = 0$ and $get\_rank[1] = 0$. Therefore, the communication can always be implemented by sending to the process which is responsible to the corresponding task ID.

The second parameter which is different to the original MPI function is *create_data*. It is representing a pointer to a function with 5 parameters. This callback-function provides the possibility to restore the

original data. If the root process is failing, the data is lost and has to be restored. If the data is re-calculable, the computation to restore the data is performed in this function. If the data was replicated, the replicas can replace the failed data in this function. We have designed this callback-function with an interface of 5 parameters, to provide developers full flexibility to react to failures.

The first parameter of the callback-function is an integer which is representing the ID of the task which is calling the function. The second parameter is an integer that is representing the ID of the task of the failed process and the third parameter is an integer which specifies the length of the data to be restored. The void pointer is pointing to the address at which the lost data is restored and the last parameter is a mapping from task IDs to process IDs. These parameters provide developers enough flexibility to restore lost data.

4.2 MPI_Scatter

The interfaces of the original MPI scatter function and the fault tolerant alternative are shown next to each other in Table 4.2.

MPI_Scatter	**MPI_Scatter_ft**
• **void** *sendbuf • **int** sendcount • **MPI_Datatype** sendtype • **void** *recvbuf • **int** recvcount • **MPI_Datatype** recvtype • **int** root • **MPI_Comm** comm	• **void** *sendbuf • **int** sendcount • **MPI_Datatype** sendtype • **map<int, void*>** *recvbuf • **int** recvcount • **MPI_Datatype** recvtype • **int** root • **map<int, int>** get_rank • **void** (*create_data)(int, int, int, void*, map<int, int>) • **MPI_Comm** comm

Table 4.2: Scatter Parameter List

The purpose of the scatter communication is to send different messages from a root process to other processes. Therefore, the scatter function needs a buffer for the messages to be distributed which is given by the first parameter of the interface, *sendbuf*. This parameter is a void pointer which points to the address at which the messages are stored. Since this pointer has no declared type, it is possible to send messages of different types.

The parameter *sendcount* specifies the number of messages which are sent to each process and the parameter *sendtype* is declaring the type of the messages. Unlike the broadcast communication, in the scatter communication a receiving buffer is necessary to store the incoming messages. This buffer is provided by the parameter *recvbuf* and is also represented as a void pointer to store the data without a specific type. The amount of messages to be received is declared in *recvcount* and the datatype of these messages is given by *recvtype*. The parameter *root* is an integer value and is representing the process ID

of the process which is sending the messages. In *comm* the MPI communicator is given, in which the communication is performed.

The interface of the fault tolerant alternative is extended by two parameters. These parameters have the same functionality as the parameters of the broadcast communication. The first parameter is a map, *get_rank*, in which the mapping of tasks to corresponding process IDs is saved. The second parameter is a function, *create_data*. This function is represented as a callback function and is called when process errors are occurring. The function is called by every surviving process when the root process fails. If any other process is failing the data is not lost and does not have to be restored.

In case of a process fault another estimated process takes the place of the failed process and distributes the data in the binomial tree of the gather algorithm (see Section 3). To distinguish which messages belong to which task it is not sufficient to keep the messages in one receiving buffer. Therefore, we have extended the interface of the fault tolerant scatter with a map which is mapping integers to void pointers. The purpose of this mapping is to map process IDs to receive buffers. A process always knows for which task it is sending and receiving messages, thus it can insert the messages accordingly.

We consider as an example a scatter communication in which Process 3 fails and Process 2 is responsible for the task of Process 3. Process 2 will receive messages from Process 0 for the Task 2 and 3. Process 2 can distinguish which messages belong to which task and can insert the messages for Task 2 into the map with key 2 and the messages for Task 3 into the map with key 3.

4.3 MPI_Gather

In the gather communication messages from all processes are gathered at a specified root process. The interfaces for the original and for the fault tolerant gather implementation are compared in Table 4.3.

MPI_Gather	**MPI_Gather_ft**
• **void** *sendbuf • **int** sendcount • **MPI_Datatype** sendtype • **void** *recvbuf • **int** recvcount • **MPI_Datatype** recvtype • **int** root • **MPI_Comm** comm	• **map<int, void*>** sendbuf • **int** sendcount • **MPI_Datatype** sendtype • **void** *recvbuf • **int** recvcount • **MPI_Datatype** recvtype • **int** root • **map<int, int>** get_rank • **void** (*create_data)(int, int, int, void*, map<int, int>) • **MPI_Comm** comm

Table 4.3: Gather Parameter List

Since every process is going to send a message which is dedicated to the root process a buffer for the messages that are sent is necessary.

The parameter *sendbuf* represents this buffer and is implemented as a void pointer in order to allow this function to be used with different datatypes. The integer *sendcount* stands for the amount of messages which are sent and the parameter *sendtype* represents the datatype of these messages.

To receive messages a buffer is needed too. This buffer is represented by *recvbuf* and is implemented as a void pointer that points to the address at which the received messages are stored. This buffer is only filled by the root process since this process is gathering all messages. The number *recvcount* is the amount of messages which are received by each process and *recvtype* is the datatype of the messages that are received.

The parameter *root* is indicating which process is acting as the root process and therefore which process is storing the messages. The parameter *comm* is the MPI communicator in which the communication takes place.

The difference between the original and the fault tolerant variant are the parameters *sendbuf*, *get_rank* and *create_data*. In the map *get_rank* a mapping of task ID to process ID is stored which allows processes to find the process responsible for a task. The callback function *create_data* is the function which is called by every process in case of failures. Every process has individual messages for the root process and therefore in every case of failures this function is called. The parameters of this function have the same functionality as explained in Section 4.1.

Since every process has specific messages for the root process, every task has different messages. Therefore, a mapping of task ID to the buffer of messages to be sent is needed. At first every process is storing the messages in the map with its own process ID as key. If a process is taking the responsibility of another task it will restore the lost data and insert this data into the map with the task ID as key. Therefore, it is possible to distinguish which messages are sent for which task.

The buffer of the received messages does not need a mapping since this data is only stored at the process responsible for the root task.

4.4 MPI_Reduce

The interfaces of the reduce function provided by MPI and the fault tolerant implementation are compared in table 4.4.

MPI_Reduce	**MPI_Reduce_ft**
• **void** *sendbuf • **void** *recvbuf • **int** count • **MPI_Datatype** datatype • **MPI_Op** op • **int** root • **MPI_Comm** comm	• **map<int, void*>** sendbuf • **void** *recvbuf • **int** count • **MPI_Datatype** datatype • **MPI_Op** op • **int** root • **map<int, int>** get_rank • **void** (*create_data)(int, int, int, void*, map<int, int>) • **MPI_Comm** comm

Table 4.4: Reduce Parameter List

In the reduce operation a set of data is reduced into a smaller set of data. Therefore, every process has messages to send to a root process. These messages are stored in a buffer, the parameter *sendbuf* in the interface of the original MPI function. The result of the reduction is stored in the buffer *recvbuf*. The parameter *count* indicates how many messages are sent and how many messages are stored in the reduced result. The messages and the result are of the datatype *datatype* and the operation which is applied is specified in the parameter *op*. The communicator for the communication is given by *comm*.

The difference between the original and the interface for the fault tolerant implementation are the parameters *sendbuf*, *get_rank* and *create_data*. For every process the messages are individual and therefore different for every task that is executed. If a process fails and the task is taken over by another process, this process has to be able to distinguish which messages belong to which task and therefore we have extended the interface with a map that maps task IDs to corresponding messages. A mapping is provided by *get_rank* which is mapping tasks to processes and a function is given to restore lost data. The parameters for *create_data* are the ID of the task which is calling the function, the ID of the failed task, the amount of the messages to be restored, a buffer to store the messages and a mapping of task IDs to process IDs.

4.5 MPI_Alltoall

The all-to-all communication is behaving in a different way than the communications discussed before. In the all-to-all communication every participant has messages for every other participant and has to exchange these messages. Therefore, no root process is existing which is going to send or to receive all the messages. This difference is also visible in the interface of the original and of the fault tolerant implementation which are compared in Table 4.5.

MPI_Alltoall	MPI_Alltoall_ft
• **void** *sendbuf	• **map<int, void*>** sendbuf
• **int** sendcount	• **int** sendcount
• **MPI_Datatype** sendtype	• **MPI_Datatype** sendtype
• **void** *recvbuf	• **map<int, void*>** *recvbuf
• **int** recvcount	• **int** recvcount
• **MPI_Datatype** recvtype	• **MPI_Datatype** recvtype
• **MPI_Comm** comm	• **map<int, int>** get_rank
	• **void** (*create_data)(int, int, int, void*, map<int, int>)
	• **MPI_Comm** comm

Table 4.5: Alltoall Parameter List

A buffer for the send messages is needed, which is represented by the void pointer *sendbuf* that is pointing to the address in the memory of the process, at which the messages are stored. To know how many messages have to be sent to the other processes the parameter *sendcount* is necessary. It indicates how many messages are going to be sent to a single process. The datatype of these messages is passed by the parameter *sendtype*.

For receiving messages a buffer is needed too. This buffer is specified by the parameter *recvbuf*. It is a void pointer, which is pointing to the address in the memory, where the messages are stored. The parameter *recvcount* indicates how many messages are received by a single process and the parameter *recvtype* is specifying the type of the messages. The communication is taking place in the communicator *comm*.

If a process fails, the messages that this process is going to send and the messages that this process has received are lost. Therefore, an overtaking process needs to know which messages belong to which task it is executing. For this reason we have introduced a map in which the buffers for the sending messages are accessed by their corresponding task ID. The same approach is applied for the buffers for the receiving messages. A mapping from task IDs to process IDs is needed for our approach and given by the parameter *get_rank*, as well as the callback-function *create_data* which is specifying how the lost data can be restored.

In order to provide better flexibility we have also implemented an all-to-all communication for variable message sizes, we call it for the rest of this work all-to-all-v. This function is using the same algorithm as the previously introduced all-to-all communication for fixed message sizes, but the interface of the function is varied. In MPI the function *MPI_Alltoall_v* is providing this functionality and is adapting the interface of *MPI_Alltoall*. In Table 4.6 we highlight the changed parameters.

MPI_Alltoall_v	**MPI_Alltoall_v_ft**
• **int*** sendcounts	• **std::map<int, int*>** sendcounts
• **int*** senddispls	• **std::map<int, int*>** senddispls
• **int*** recvcounts	• **std::map<int, int*>** recvcounts
• **int*** recvdispls	• **std::map<int, int*>** recvdispls

Table 4.6: Alltoall_v Parameter List

The parameter *sendcounts* is representing a list of numbers which define the size of the message that is sent to a process. Corresponding to that, the parameter *recvcounts* is a list of numbers that define the size of the message that is received by a certain process. The parameters *senddispls* and *recvdispls* define a displacement for the messages. Therefore, they represent the starting index of the message that is about to be sent (or received) in the buffer for the messages. In the fault tolerant implementation these parameters are all provided as a map. This is due to the task model which we use for providing fault tolerance. Processes and tasks are handled separately and therefore the parameters have to be mapped to the corresponding tasks. The remaining parameters have not changed in comparison to the all-to-all communication.

5 Evaluation

In this section the proposed concept of fault tolerant collective communications of Chapter 3 is evaluated. The goal of the evaluation is to show the overhead that is introduced by using fault tolerant communications in comparison to the original collective communications provided by the MPI implementation. Furthermore, we evaluate the benefit of our approach in scenarios of process failures in a TPC-H Benchmark query and show that our approach is to be preferred to restarting the computation on the occurrence of errors.

5.1 Setup

To compare the overhead that is introduced by the fault tolerant collective communications, we measure the individual runtime of these communications and compare it to the runtime of the collective communications provided by MPI. To evaluate the additional cost in relation to the number of processes that are participating in the communication, we use a weak scaling approach. In the collective communications the processes use messages of a constant size (e.g. 4 byte for 1 integer). Therefore, the size of the messages that are sent is independent of the number of processes which are participating in the communication. Furthermore, we show that the introduced overhead is independent of the message size by using a strong scaling approach. In this approach the number of participating processes is fixed and the message size is varying.

In order to measure the running time of collective communications reliable, we first create a consistent state of the program by using a barrier. After the barrier every process is taking the current walltime and is then executing the communication. After the communication the processes wait in a barrier for every process to finish the communication and then take the walltime. The first walltime is subtracted from the second walltime. The result is representing the total runtime of the process. In total 1000 iterations are performed and the average of the measured time is taken as the final result. The variances for these tests can be found in Appendix A. This procedure is visualized in Figure 5.1.

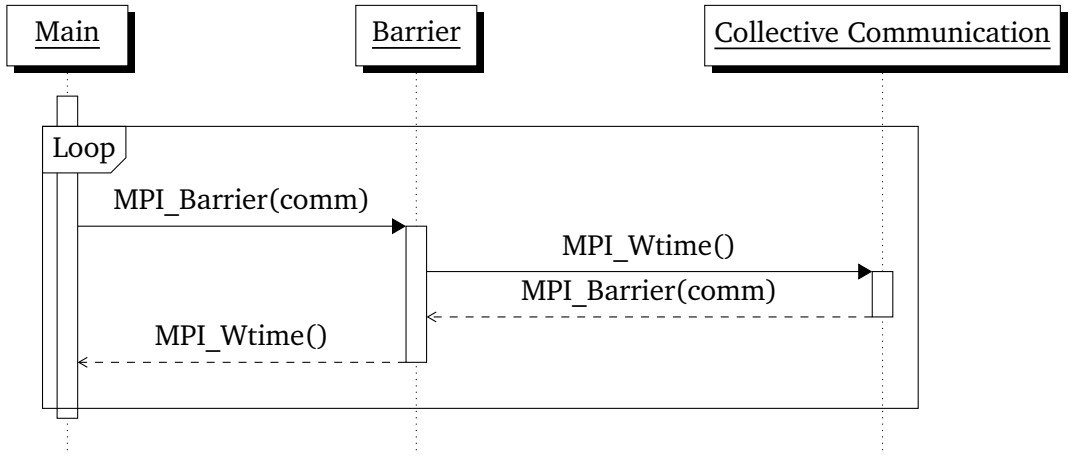

Figure 5.1: Measuring Runtime Performance

To show the benefit of our proposed approach we evaluate the runtime performance of the TPC-H benchmark query 3 (see Section 5.3.1) by using fault tolerant communications in comparison to non fault tolerant communications. We inject random process failures to demonstrate how both approaches react to failures and which approach is to be preferred. We will use a weak scaling approach for this evaluation in which every process is handling 1GB of data. Therefore, with increasing size of processes, the size of data handled by a single process is not increasing.

The algorithms are implemented in C++ using the basic functionalities for sending and receiving messages, which are provided by the MPI implementation. As MPI implementation, we have used the ULFM framework version 1.1. This implementation is based on OpenMPI version 1.7.1 and is extending it with

the fault tolerant semantics of ULFM. Calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt. This cluster consists of the 3 sections MPI, MEM and ACC. For our evaluation we are using the MPI section. The cluster was build in 2 phases, the first phase consists of 780 computing nodes, the second phase of 632. Our calculations were performed on a MPI island (out of 19) of phase 1 that posseses 162 computing nodes, 2952 cores and 5184 GB main memory. This means that each computing node has 16 cores and 32 GB RAM. The network connection is realized with one FDR-10 InfiniBand and one 1GBit-Ethernet connection.

5.2 Evaluation of Fault Tolerant Collective Communication Algorithms

This section evaluates the overhead of the proposed fault tolerant collective communications. To identify the overhead the runtime of the algorithms was measured. Furthermore, the correctness of the algorithms is tested.

5.2.1 Correctness

To verify the correctness of the implemented communications we have developed a test framework which is performing unit tests. To implement the unit tests we have used the Google C++ Testing Framework Google Test (GTest). Before the start of the tests, we are generating different test cases in which process failures are specified. The point in time at which these failures strike is dependent of the algorithmic structure of the communication under test. The injection of failures is performed accordingly to [57]. A process failure is emulated by a kill system call. We have chosen to emulate process failures with this call because the behavior is comparable to the behavior of real process failures. The process stops and can not be repaired anymore. In every test case the according process is stopped by raising a *SIGKILL* signal. A *SIGKILL* signal is a signal that terminates a process immediately and is therefore a suitable fit for the simulation of real process failures. The test always executes the communication under verification first without failures and stores the correct result. Afterwards the test executes the algorithm for every generated test case and compares the result with the original one. If the results are equal the test will run the next test case otherwise it will abort and indicate that the test has not finished successfully.

In the reduce algorithm the communication can be divided into $\lceil log(p) \rceil$ iterations of communications. Therefore, the failures are injected at the beginning of an iteration. For every iteration possible process failures are generated but at least one process is not failing to guarantee the correct execution. Furthermore, possible combinations of failures in different iterations are created. An example for a possible unit test scenario is shown in Table 5.1.

ProcessID	Iteration
0	0
1	1
2	1
3	2

Table 5.1: Unit Test Case Scenario

In this scenario we assume a total number of processes of $p = 5$. In the first iteration Process 0 stops because of an error. In iteration 1 Process 1 and 2 are struck by failures and in the last iteration Process 3 is stopping. After these failures no further failure is happening because only Process 4 is still functioning correctly. If every process would abort, the execution would terminate. Therefore, we can only guarantee the correctness of the algorithm as long as still one process is staying responsible.

The all-to-all communication can be tested in a similar way. The communication is divided into iterations of at most p - 1 steps. The test case scenarios are generated similar to the reduce communication. Possible combinations of process failures and point in times for these failures are generated. The failures are injected at the beginning of every iteration.

The algorithmic structure of the broadcast, scatter and gather communication is slightly different compared to the structure of the reduce and all-to-all communication. In these algorithms every process is determining the processes it is communicating with and then performs the communication. In reduce and all-to-all different iterations of communications have to be performed. Therefore, in broadcast, scatter and gather, all possible combinations of process failures are generated and the failures are injected at the beginning of the execution of the algorithm.

This test framework allows us to test the correctness of the proposed communication with a high test coverage. It is not possible to achieve a test coverage of 100% since the point in times at which process failures can occur are unlimited.

The process of performing the tests is shown in Figure 5.2. At first the test framework calls the test case generation. After the test case generation succeeded, the test cases are stored in a configuration file. The test framework then reads every generated test case of the configuration file and starts the testing. Parameterized tests are not possible, because a process is lost after a failure and the program has to be restarted.

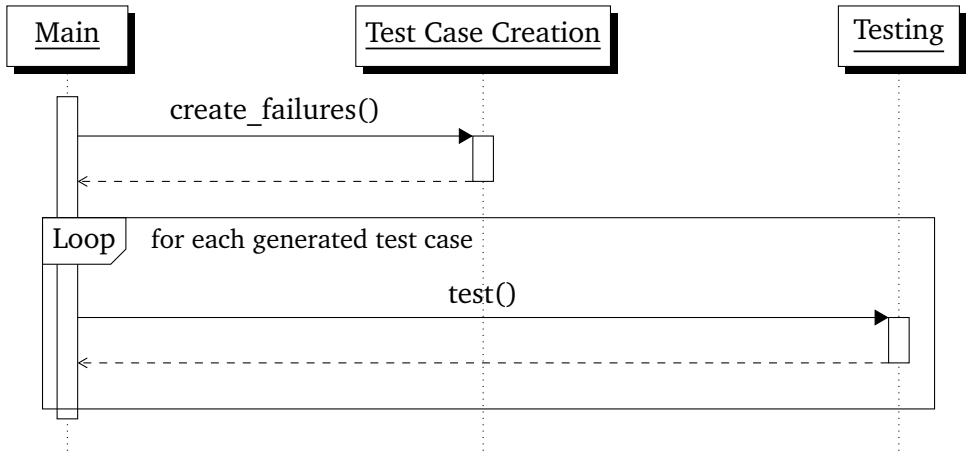

Figure 5.2: Sequence Diagramm for Testing Framework

5.2.2 Broadcast

In Figure 5.3 the analyzed runtime of the broadcast communication is shown.

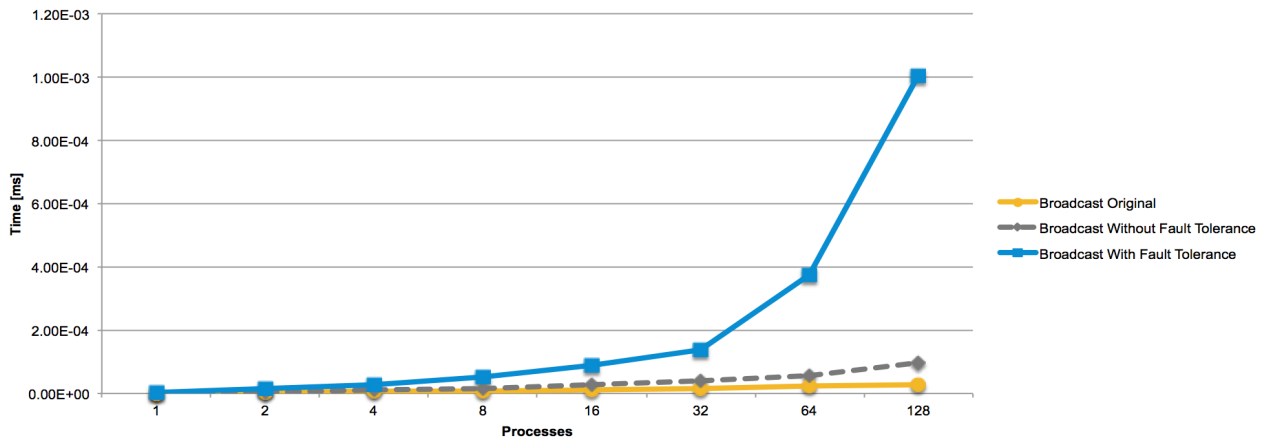

Figure 5.3: Broadcast Runtime

This result visualizes the runtime of the original broadcast communication, provided by the MPI implementation, in comparison to the fault tolerant alternative. The communication is performed by

broadcasting messages of a constant size from the root process to the other processes. The amount of participating processes was varied. It is obvious that the fault tolerant reimplementation is introducing overhead. To analyze the introduced overhead we have reimplemented the broadcast communication without using fault tolerance techniques. It is noticeable that some overhead is already introduced by reimplementing the broadcast communication. This is due to a numerous amount of optimizations and implementations provided by the MPI implementation. There are many different strategies implemented and MPI is choosing the best algorithm with regard to the current system landscape. Most of the overhead is introduced by using the fault tolerance strategy, because the agreement function itself is more costly than broadcasting a single message of 4 bytes.

The overhead visualized as a factor can be seen in Figure 5.4.

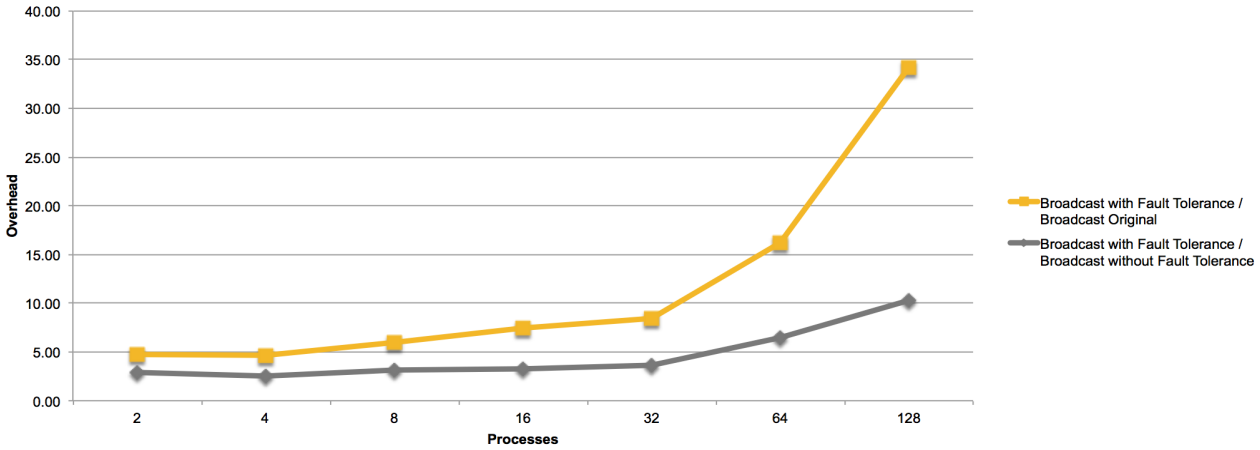

Figure 5.4: Broadcast Overhead

The overhead is calculated as the factor of the runtime of the fault tolerant implementation divided by the runtime of the original MPI function. The average of the total overhead is 11.69. The average overhead that is introduced by using fault tolerance techniques is 4.60. Therefore, it can be seen that some overhead is already introduced by reimplementing the broadcast strategy. With more optimizations, like MPI is using in its implementation, the overhead of using the fault tolerance strategy can be reduced.

In Figure 5.5 the runtime of the broadcast communication for a fixed amount of participating processes is shown.

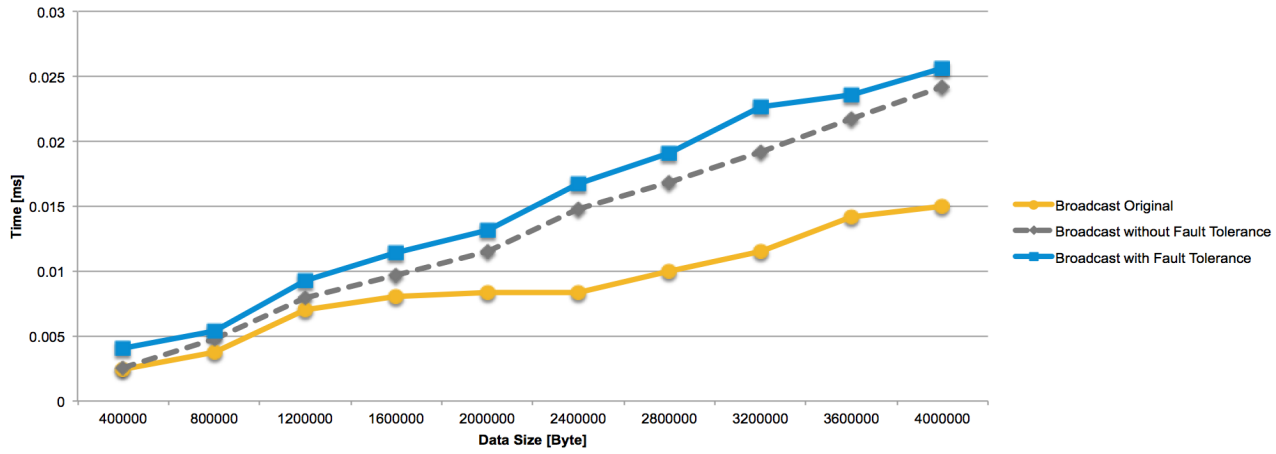

Figure 5.5: Broadcast Overhead (Variable Message Size)

The size of the message is varying. The runtime of the fault tolerant implementation is not as good as the runtime of the original MPI implementation. The runtime of the not fault tolerant reimplementation

is also not as good but the overhead that is introduced by the fault tolerance strategy is remaining almost constant. Therefore, we can argue that our developed fault tolerance strategy is not dependent on the message sizes. But the overhead of the fault tolerant implementation in comparison to the original MPI function is not remaining constant. This is due to the optimizations that are provided by MPI.

5.2.3 Scatter

The results of the scatter communication are illustrated in Figure 5.6. It shows the runtime of the scatter communication with respect to the number of participating processes. The runtime of the original scatter function of MPI is compared to the proposed fault tolerant implementation. The runtime of the reimplementation without using fault tolerance techniques is also shown to highlight how much overhead is introduced by applying the developed fault tolerance strategy.

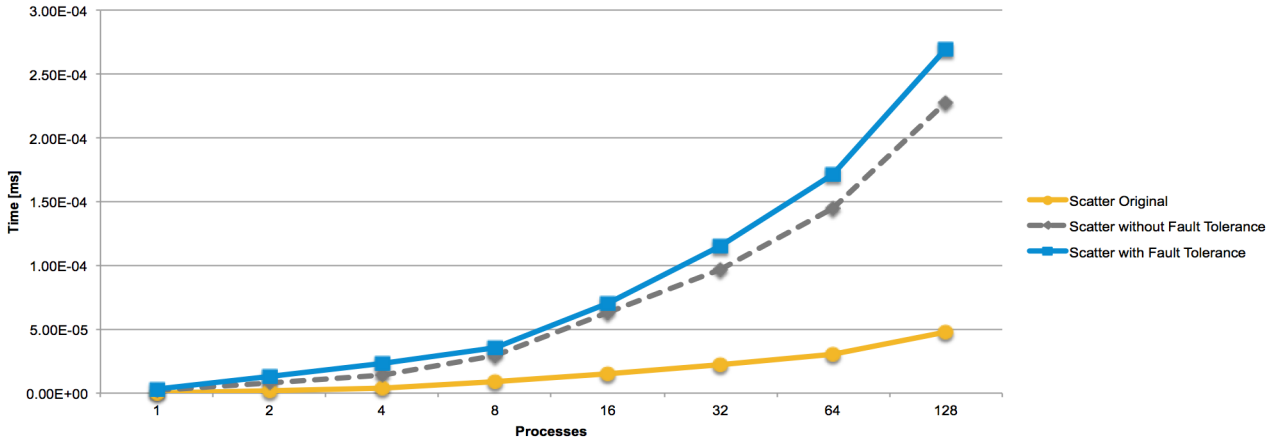

Figure 5.6: Scatter Runtime

The scatter function provided by MPI is clearly performing better than the fault tolerant scatter implementation. To analyze the overhead that is introduced by using the fault tolerant scatter communication, we have removed the fault tolerance techniques of the scatter reimplementation and analyzed its runtime. It is obvious that only reimplementing the scatter function is introducing a certain overhead. The analysis of the overhead is illustrated in Figure 5.7. It is showing the factor of the runtime of the fault tolerant approach divided by the runtime of the original scatter function.

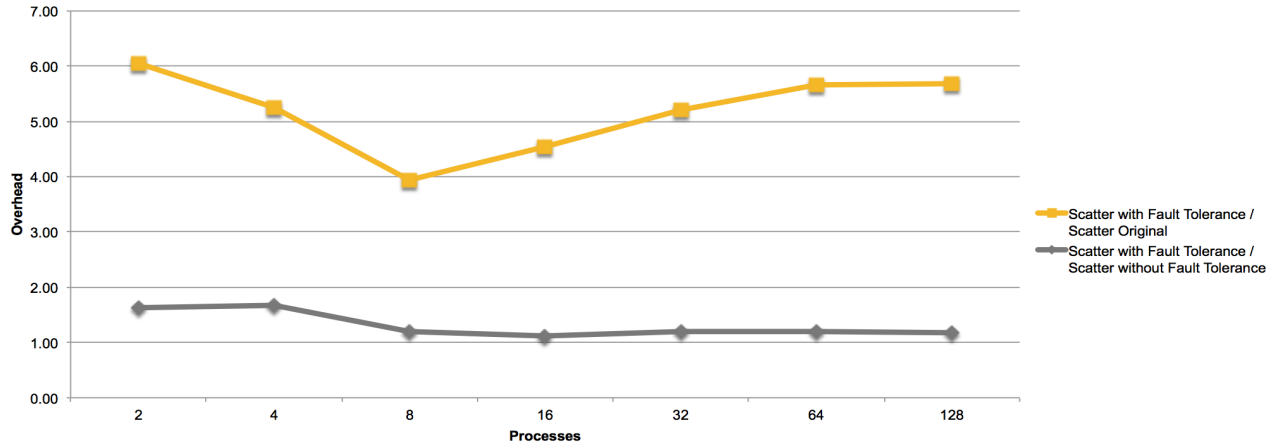

Figure 5.7: Scatter Overhead

The average of the total overhead is 4.90. The overhead of the reimplementation of the scatter communication without fault tolerance techniques is in average 1.31. Therefore, the additional overhead

that is introduced only by using the fault tolerance strategy is not as high as the total overhead. This means that most of the overhead is introduced by using the reimplementation and not by using the fault tolerance strategy. This is because MPI has implemented a lot of different strategies for the scatter communication depending on different algorithms. MPI is choosing which implementation to use depending on different factors about the system. Therefore, MPI is optimizing the scatter communication a lot which is resulting in the overhead of the reimplementation.

5.2.4 Gather

The results for the gather communication are similar to the scatter communication. The measured runtime in relation to the amount of participating processes is shown in Figure 5.8. The runtime of the original gather function provided by the MPI implementation is shown in comparison to the fault tolerant gather implementation. For further visualization the reimplemented gather communication without fault tolerance techniques is also shown.

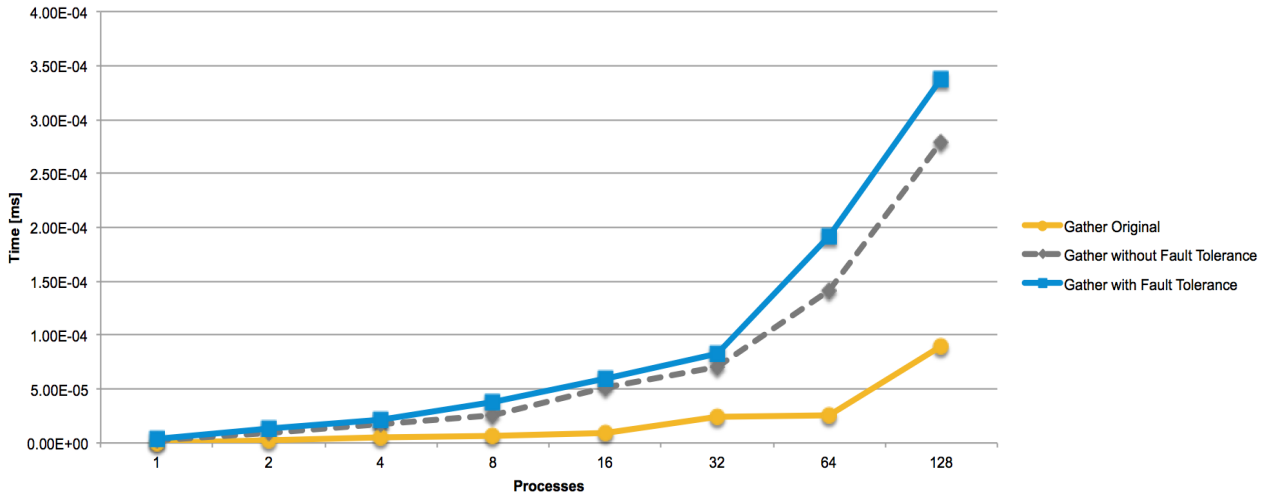

Figure 5.8: Gather Runtime

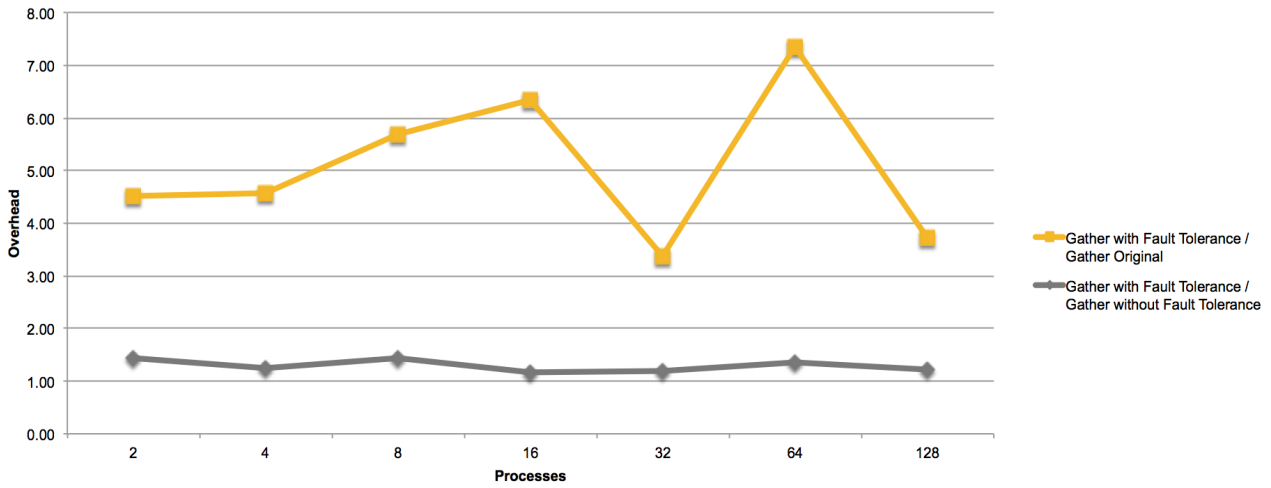

Figure 5.9: Gather Overhead

It is obvious that the fault tolerant gather communication is introducing overhead in comparison to the function of MPI. Seeing the runtime of the reimplementation without fault tolerance it is recognizable

that the reimplementation is already introducing a non negligible overhead. Applying the fault tolerance strategy to the reimplementation of gather is adding overhead too.

The overhead as a factor of the runtime of the fault tolerant implementation by the MPI function is shown in Figure 5.9. In average the factor of the total overhead is around 5.09. The average overhead of the reimplementation is 1.29. Therefore, the total overhead that is introduced only by applying our fault tolerance strategy is not as high as the total overhead.

It is obvious that the most overhead is introduced by reimplementing the gather communication without using fault tolerant semantics. The reason for this are the various optimizations that are included in the MPI implementation. Therefore, it can be seen that the overhead of our proposed fault tolerance strategy is acceptable but more optimizations have to be performed for the reimplementation of the gather communication.

5.2.5 Reduce

The fault tolerance strategy for the reduce communication is slightly different compared to the previous communications. The results for the runtime measurements are visualized in Figure 5.10.

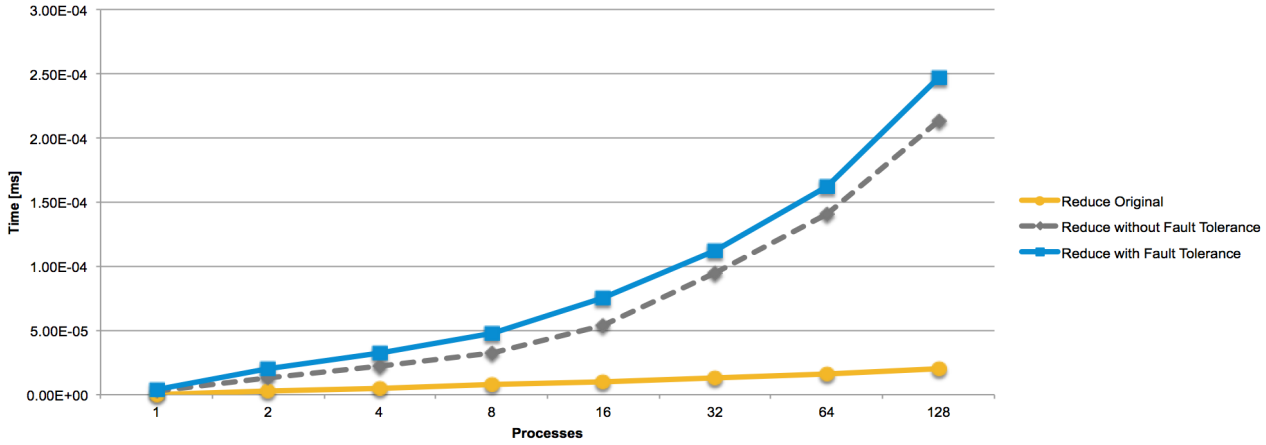

Figure 5.10: Reduce Runtime

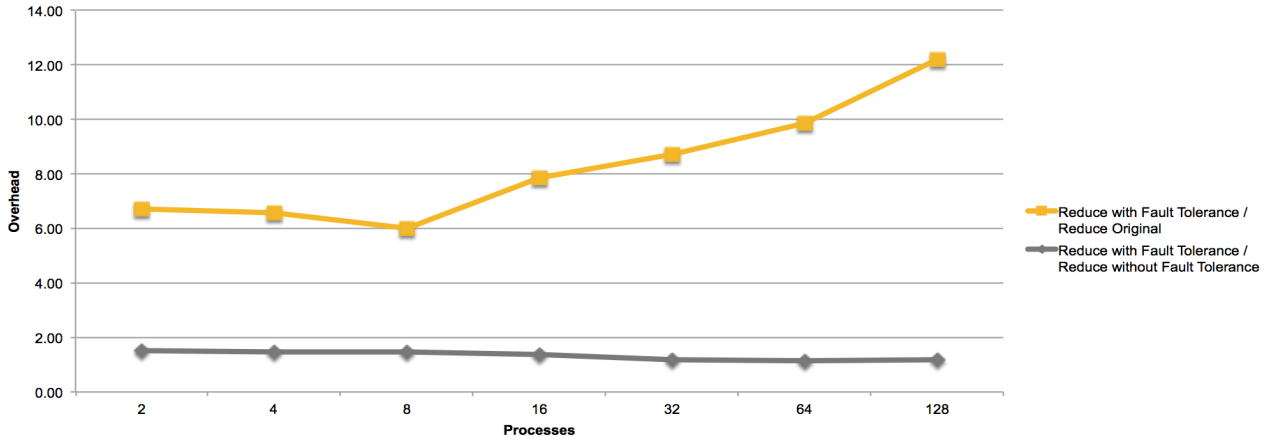

Figure 5.11: Reduce Overhead

In Figure 5.10 the runtime measurements of the original MPI function, the reimplementation without fault tolerance techniques and the fault tolerant implementation are visualized. It is obvious that the runtime of the original MPI function is not increasing as much as the runtime of the reimplementation.

This is due to the changed runtime complexity and the added overhead of the fault tolerance strategy. Furthermore, it can be seen that the overhead that is added to the non fault tolerant reimplementation is much less than the overhead added by reimplementing the reduce communication. This means that the optimizations provided by the MPI implementation are improving the reduce functionality.

Looking at the overhead the same observations can be made. Figure 5.11 is showing the overhead as factor of the runtime of the fault tolerant (and the reimplementation of the) reduce communication divided by the runtime of the reduce communication of the MPI implementation.

The average total overhead is 8.28. The average of using the fault tolerance strategy in the reimplementation is only 1.34. Therefore, the overhead that is introduced by the fault tolerance strategy is not as high as the overhead of reimplementing the communication. Thus, more optimizations should be implemented, like in the MPI implementation.

5.2.6 All-to-all

The results of the all-to-all communication are visualized in Figure 5.12 and 5.13. The measured runtime of the all-to-all function provided by the MPI implementation is compared to the reimplementation and the fault tolerant implementation.

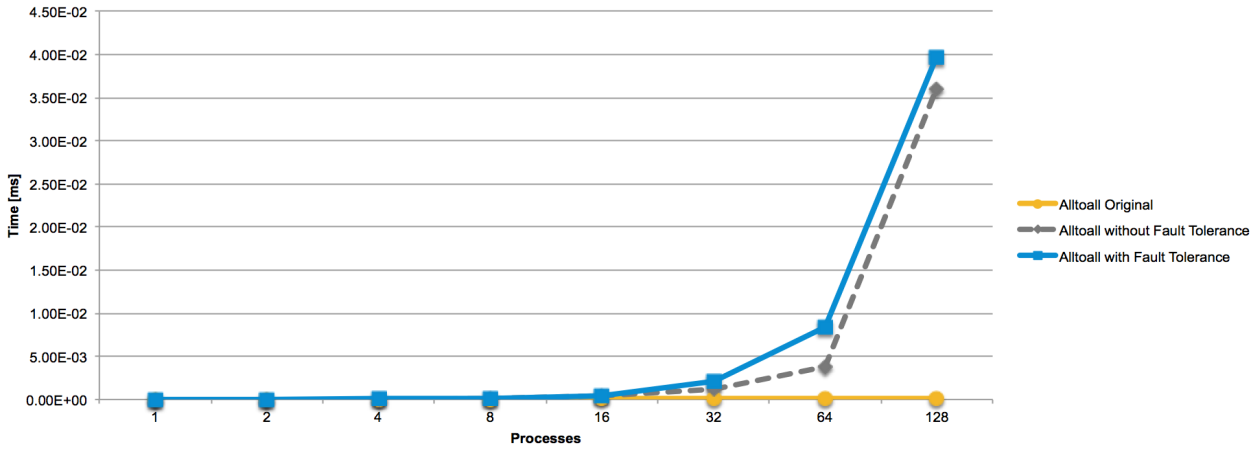

Figure 5.12: All-to-all Runtime

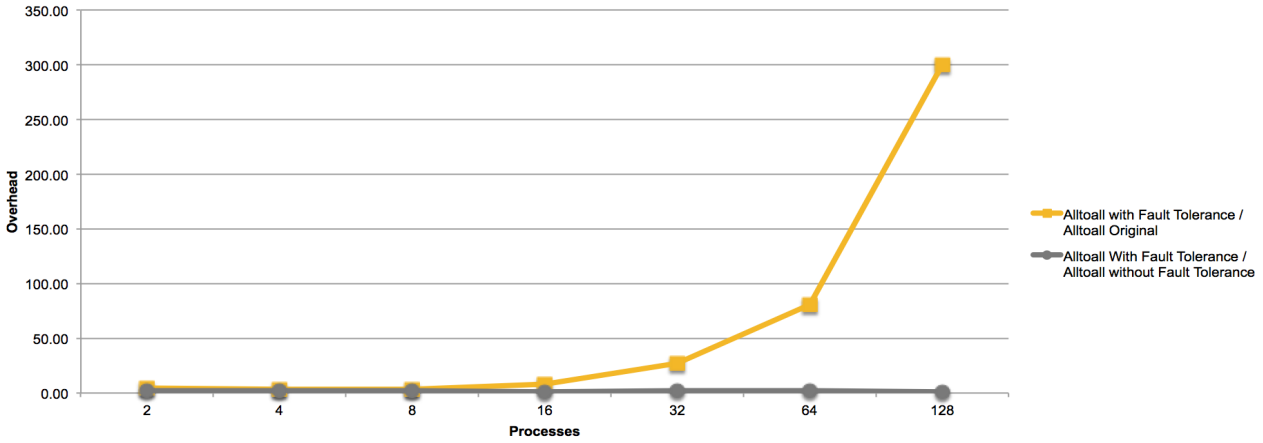

Figure 5.13: All-to-all Overhead

The runtime of the MPI function is clearly better than the runtime of the other implementations. This is due to several reasons. The first reason is the optimization performed by the MPI implementation.

The second is reason is the choice of the algorithm. We have chosen to use the 1-factor algorithm [48]. This algorithm is especially performing well in scenarios where huge messages are sent but the test was performed with a message size of 4 bytes. Figure 5.14 shows that for a message size of 4MB per process the runtime performance is already slightly better than the MPI implementation.

The overhead is showing a similar trend. The overhead introduced by applying the fault tolerance strategy is not as high as the overhead introduced by reimplementing the all-to-all communication.

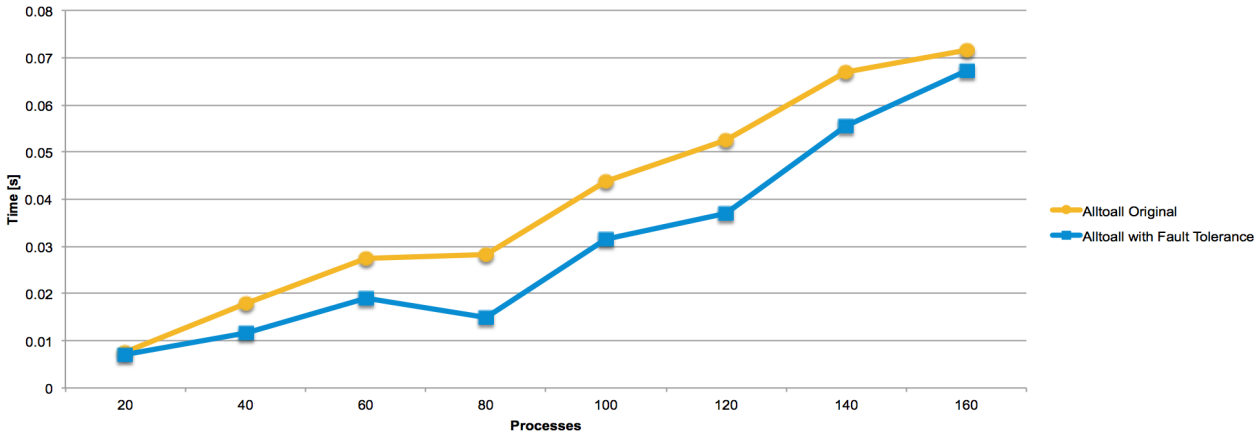

Figure 5.14: All-to-all Runtime (4MB per Node)

5.3 Evaluation of TPC-H Benchmark Query

To analyze the benefit that is introduced by using fault tolerant collective communications and to argue whether the overhead is acceptable, we have compared our approach with the alternative, to restart the computation. Therefore, we have used a scenario for distributed database systems, the TPC-H benchmark query 3, and have injected process failures. We have analyzed the behavior of the TPC-H query for using fault tolerant collective communications and for using the non fault tolerant communications provided by MPI. In case of failures our approach continues the execution, in the non fault tolerant approach the execution of the computation is terminated by the cluster. To measure the complete runtime even if the execution is terminated because of failures we take the runtime that is provided by the SLURM system that is running on the Lichtenberg cluster.

5.3.1 TPC-H Benchmark

In order to evaluate the benefit of fault tolerant collective communications, we have implemented a database query, which is described in TPC-H. TPC-H is a decision support benchmark for databases and is defined by the Transaction Processing Performance Council (TPC). It is defining a set of database queries and data which has relevance for many industrial scenarios [1].

In TPC-H 22 queries are defined which answer special business related questions and are used to evaluate the performance of database management systems. TPC-H is also defining which data is generated and stored in the database. The minimum size of the data is around 1GB but it can also be used for larger database populations. In Figure 5.15 the database structure of TPC-H is illustrated.

The database that is used in the TPC-H benchmark queries consists of 8 base tables. The size of these tables depends on the used scale factor for generating the database. A scale factor of 1 corresponds to 1GB and accordingly a scale factor of 100 corresponds to 100GB. The database model is represented as a snowflake model [9]. The snowflake schema is an acyclic database schema, which consists of a central fact table and a set of constituent dimension tables [38]. Therefore, the tables are in relation to each other. For example a *lineitem* is holding a key for the corresponding order. An order has a key for the customer that placed the order and so on.

In our implementation we use a modified version of the database generator, which is provided by TPC-H, to create the database with the corresponding entries. The modified generator is generating the database distributed into the main memory of the processes in the distributed database system. The original database generator generates the database into files on the filesystem. Because of the distributed data generation the relations between the tables have to be categorized in local and remote join paths. This means that not all the data is available at every node. A local join path is available from *region* to *nation*, from *nation* to *supplier* and from *nation* to *customer*. Other local join paths are also from *part* to *partsupp* and from *order* to *lineitem*. On the other hand remote join paths are from *customer* to *orders*, from *partsupp* to *lineitem* and from *supplier* to *partsupp*. Remote data access is performed by using some sort of communication, therefore we can use the implemented fault tolerant communication algorithms to provide fault tolerance in TPC-H queries.

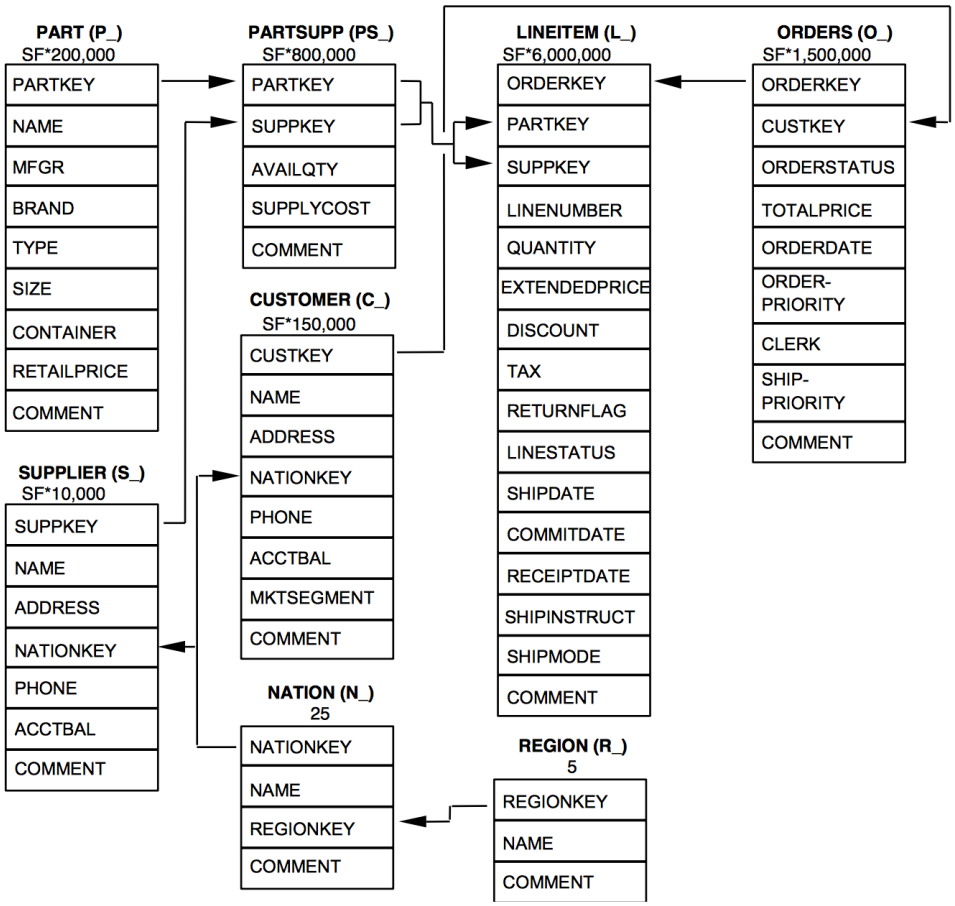

Figure 5.15: TPC-H Database Schema [1]

We have chosen to implement the TPC-H Benchmark query 3, because a gather, a broadcast, an all-to-all-v and a reduce communication are used. In this query the revenue of the orders that have not been shipped is calculated and the key of the order, the revenue, defined as the sum of $l\_extendedprice * (1 - l\_discount)$, the shipping priority and the order date is returned. The result is ordered in decreasing order of the revenue and only the top ten results are returned.

The SQL code for this business question is shown in Listing 5.1. Line 1-5 show the select statement. The key of the order, the revenue, the order date and the shipping priority are retrieved. In line 6-9 the from statement of the query is shown. It can be seen that only the tables *customer*, *orders* and *lineitem* are relevant for this query. Line 10-15 highlights the where clause. The customer entries are first sorted for customers of a given market segment, which is given by the parameter SEGMENT. A join path from

customer to *order* is existent which can be seen in line 12. This is relevant because from *customers* to *orders* no local join path is present. This means that the data is stored distributed and communications are necessary to realize this join. Another join from *lineitem* to *orders* is performed. For this join local data access can be used. In Line 14 and 15 the orders are filtered for orders where the order date is before a given date but the shipping date of the lineitems is after the given date. The date is specified by the parameter DATE. Therefore, only orders that have been ordered but not shipped yet are taken into consideration. At the end the result is grouped, in line 16-19, and ordered in descending order of revenues, in line 20-22.

```
1  select
2    l_orderkey,
3    sum(l_extendedprice*(1-l_discount)) as revenue,
4    o_orderdate,
5    o_shippriority
6  from
7    customer,
8    orders,
9    lineitem
10 where
11   c_mktsegment = '[SEGMENT]'
12   and c_custkey = o_custkey
13   and l_orderkey = o_orderkey
14   and o_orderdate < date '[DATE]'
15   and l_shipdate > date '[DATE]'
16 group by
17   l_orderkey,
18   o_orderdate,
19   o_shippriority
20 order by
21   revenue desc,
22   o_orderdate;
```

Listing 5.1: TPC-H Benchmark Query 3 [1]

To implement this query in a distributed database system local and remote operations have to be performed. After the data generation, every process has in its main memory a part of the generated data. Therefore, each table is split into p partitions, where p is the number of processes performing the query. For this query only the tables *customer*, *lineitem* and *orders* are relevant. At start of the query every process has a partition of *customer*, *lineitem* and *orders*, and all potential customers have to be calculated. The first step to perform the query is to filter the local customers for customers of the given market segment SEGMENT. After this step every process is holding only customer entries that are relevant for the execution of this query.

In order to calculate the revenue of the order hold at a process, the process has to know all the relevant customer keys. Since there is no local data access from *customer* to *order*, it is possible that the process is holding orders from customers which entries are stored at another process. Therefore, the processes have to exchange the customer keys. To do that the size of each customer table has to be estimated first and then exchanged. This means that a gather communication is performed, in which the processes are gathering the size of the *customer* table at a root process.

After this step the root process knows the size of every *customer* table and is broadcasting this to every other process. Then every process knows the exact amount of entries in the *customer* tables of the other processes. Now the processes can exchange the customer keys by using an all-to-all-v communication (all-to-all communication for variable message sizes). Therefore, a process can send and receive messages of different sizes from different processes and thus the exchange of the customer keys is performed by using this communication.

After this communication every process knows all the customer keys that are existent for the given market segment. Therefore, it is possible to calculate the revenue without further communications.

Because of the local data access from *orders* to *lineitem* every process knows for every lineitem entry it holds the corresponding order entry. The processes can now iterate over the lineitem entries, which they are holding and check whether the order date of the corresponding order is before the given date DATE and whether the shipping date of the lineitem is after the date. In that case, the process can calculate the revenue of this lineitem entry and add it to the revenue of the corresponding order.

After this calculation, every process is holding a local result of the revenues of the orders. The processes are sorting this result in descending order and are keeping only the top ten results. In order to create the final result further communications have to be performed. There are two possibilities to retrieve the final result. One option is that all the processes are gathering their results at the root process and the root process is then sorting these results and only keeping the top ten of it. The alternative is to perform a customized reduce communication in which the top ten result is already estimated in the reduce step. The benefit is that the root process does not have to store all the local results of the other processes. The choice of the communication does not affect the end result which is stored at the root process.

In Figure 5.16 the steps for executing the TPC-H benchmark query 3 in distributed database systems are visualized.

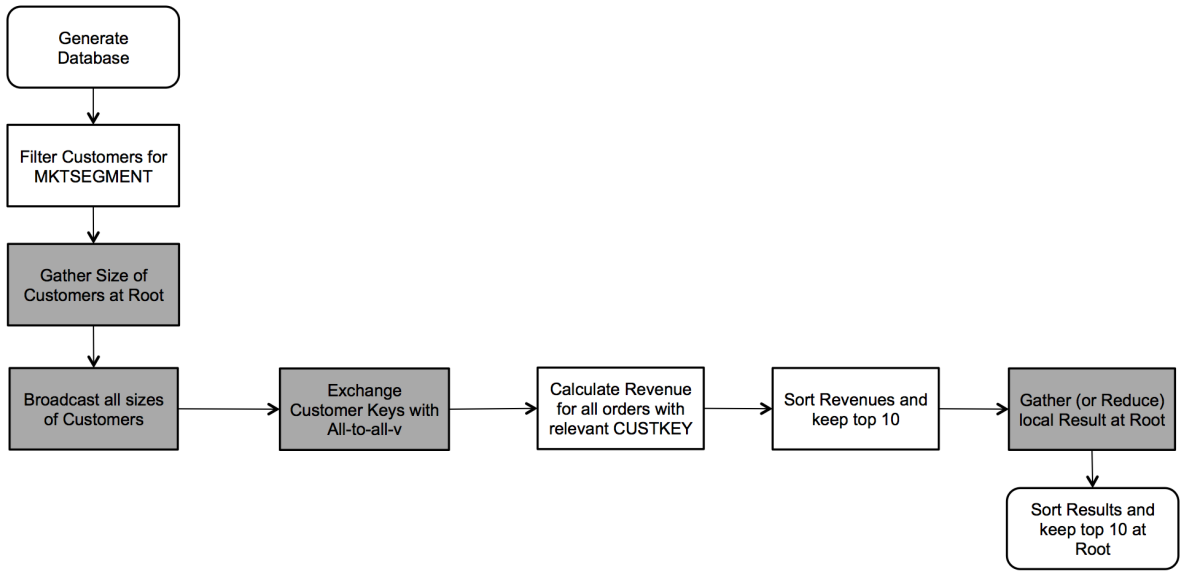

Figure 5.16: TPC-H Query 3

To provide fault tolerance for the TPC-H query, we have enabled the communication by using the fault tolerant collective communications, introduced in Chapter 3. Additionally, a possibility to restore lost data is needed. The data generator provided by TPC-H can perform this action.

The execution of the TPC-H query can be divided into 4 parts. Failures that occur in one of these parts are handled by the recovery routine of the fault tolerant communication in the corresponding part. The first part starts with the execution of the query and lasts until the end of the gather communication for gathering the size of the customer tables at the root process. Any failure that occurs during that part of the execution is handled by the recovery routine of the gather communication. In case of process failures the generated partition of the database for the failed process is lost. To continue the execution of the TPC-H query the lost data has to be recovered. In the recovery routine a process is estimated that takes over the execution of the failed process. This process will use the data generator of TPC-H to restore the lost partition. It will filter the newly restored customer table for the given market segment and will use the size of the remaining entries to continue the gather communication.

The second part starts after the gather communication and ends after the broadcast communication. Failures occurring during this period are handled in the recovery routine of the broadcast function. The lost data partition is restored by the overtaking process and the restored customer table is filtered for the

given market segment. If the failed process was not the root process of the broadcast communication, no further steps are necessary. If the failed process was the root process, the sizes of all the customer tables have to be restored. In order to do so, all the sizes of the customer tables have to be gathered at the new process before this process can use the data as input for the broadcast communication.

The third part of the execution reaches from the end of the broadcast communication to the end of the all-to-all-v communication in which the customer keys are exchanged. The recovery routine of the all-to-all-v communication is handling all failures in this part. For a lost process the lost data partition has to be restored and the customer table has to be filtered for the market segment. Because of the broadcast communication in the previous part, every process knows all the sizes of the customer tables. Therefore, no communication has to be repeated. It is sufficient if the newly created customer table is used as input data after it has been filtered.

The fourth part describes the last part of the execution. It starts after the all-to-all-v communication and lasts until the end of the execution of the TPC-H query. All failures that occur in this period are handled by the recovery routine of the last communication in this block. The communication can either be a gather or a customized reduce communication. Independent of the choice of communication, the recovery routine will perform the same steps to restore a consistent state in which the query can successfully be executed. The partition held by the failed process has to be restored with the help of the data generator of TPC-H. Since the communication of the last block is an all-to-all-v communication, the overtaking process knows the result of it and the collective communication has not to be repeated. Therefore, the overtaking process knows all the relevant customer keys and can iterate over the lineitem entries and calculate the local revenue results of the failed process, which are used as input for the last communication that will produce the final result.

5.3.2 Comparison to Restart

In Section 5.3.1 we have analyzed four different parts of the query execution in which failures are handled differently. We inject process failures at these four different parts and perform measurements of the runtime. For simulating process failures we chose a random process and terminate it, by raising a *SIGKILL* signal.

In Figure 5.17 the runtime of both approaches without failures are shown. We are using a weak scaling approach for the measurements in which every process is handling 1GB of data.

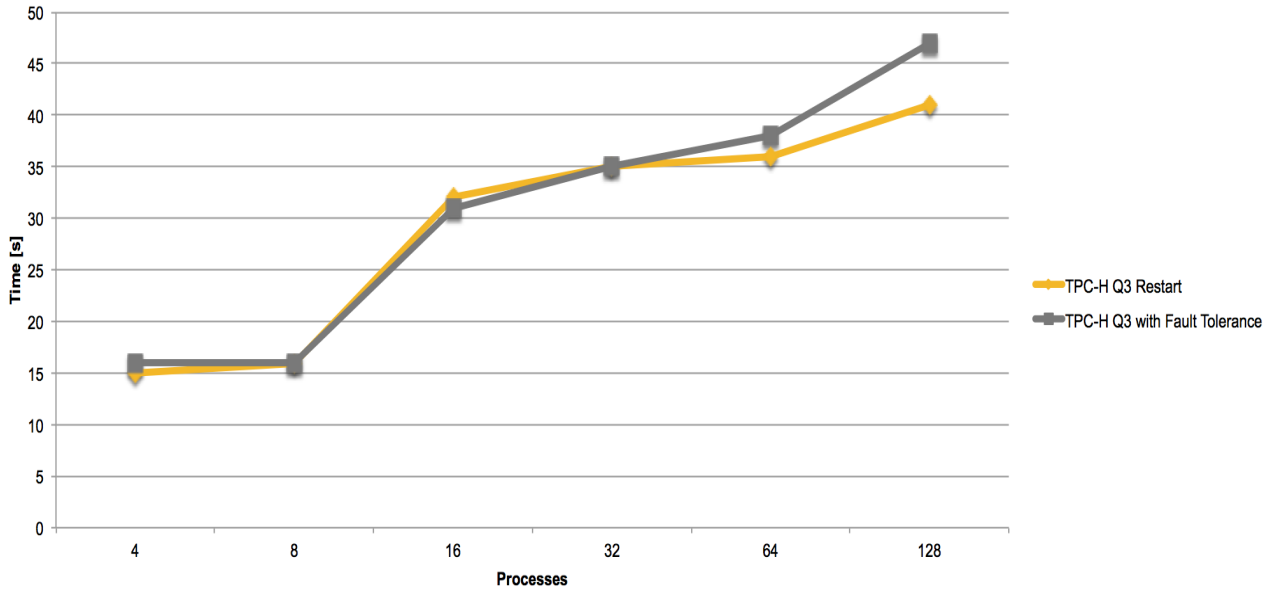

Figure 5.17: Runtime of TPC-H Benchmark Query 3 (1GB Data per Process)

It is obvious that the runtime of the fault tolerant approach is longer than the runtime of the non fault tolerant approach, for an increasing amount of computing processes. This is because of the introduced overhead of the fault tolerant collective communications. In order to discuss whether the introduced overhead is acceptable we have to evaluate the runtime under the occurrence of failures.

Figure 5.18 is showing the runtime for both approaches with a single process failure during the first phase of the query execution, until the end of the gather communication. The process failure is simulated by raising a SIGKILL signal on a randomly chosen process.

The non fault tolerant approach terminates the execution after the occurrence of the failure. Therefore, the computation has to be restarted from the beginning. The time until the termination of the computation is measured and added to the measured time of the successful computation. The fault tolerant approach is not terminating on failures but continuing after the recovery of the failure.

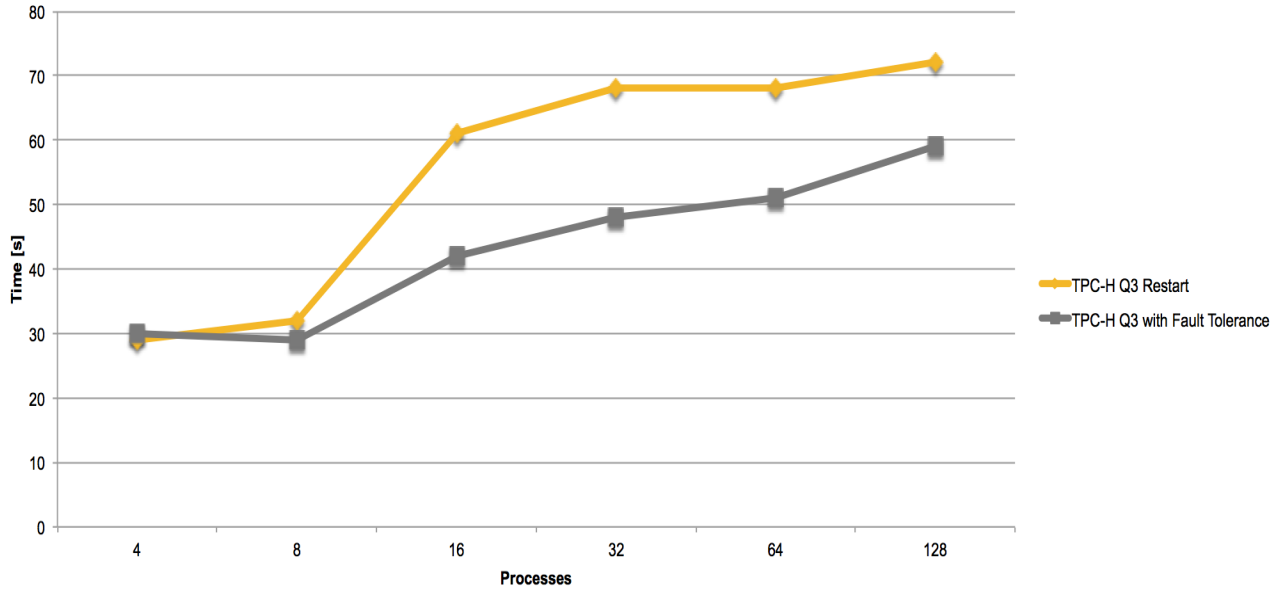

Figure 5.18: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 1(1GB Data per Process)

It can be seen that the runtime of the fault tolerant approach is performing better than the runtime of the restarted computation. In the fault tolerant approach only the lost data has to be restored while in the non fault tolerant approach every step is repeated, also the data generation for every process. Therefore, our approach is to be preferred even if the failure is happening early in the execution.

The measurements of runtime for introduced failures during the second phase of the query execution, which starts after the gather communication and lasts until the end of the broadcast communication, is visualized in Figure 5.19.

For both approaches a single process failure is simulated. It is obvious that the fault tolerant approach is better than the traditional approach. To repair the state of the execution the lost data has to be restored, but additional steps are only necessary if the root process failed. Restarting the execution on the cluster will result in repeating every step of the execution. Because of this the fault tolerant approach is performing better and is to be preferred.

In Figure 5.20 the results of the runtime measurements of both approaches are shown for an occurred error in phase 3 of the execution, the phase that starts after the broadcast communication and lasts until the end of the all-to-all-v communication.

The error recovery is handled differently to the error recovery of the previous phases. Since every process after the broadcast communication knows the same result of the communication no further steps have to be taken to create a consistent state. Only the failed data partition has to be restored.

This behavior can also been seen in the results of the measurements. The fault tolerant approach is performing better than the traditional approach in which the computation is restarted.

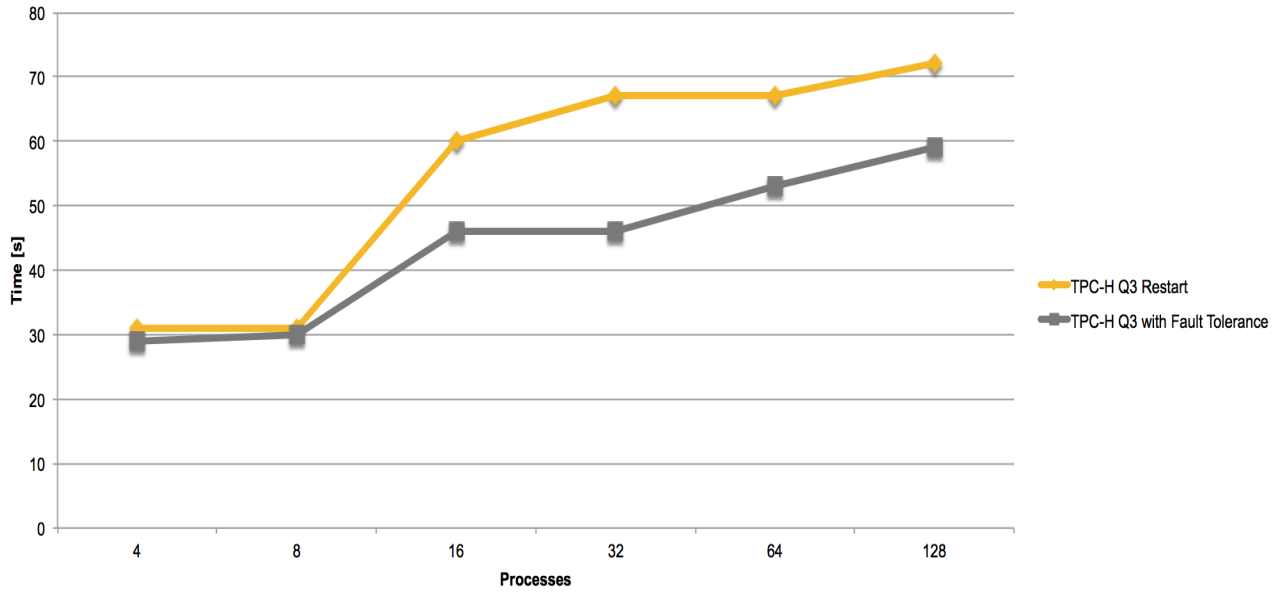

Figure 5.19: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 2 (1GB Data per Process)

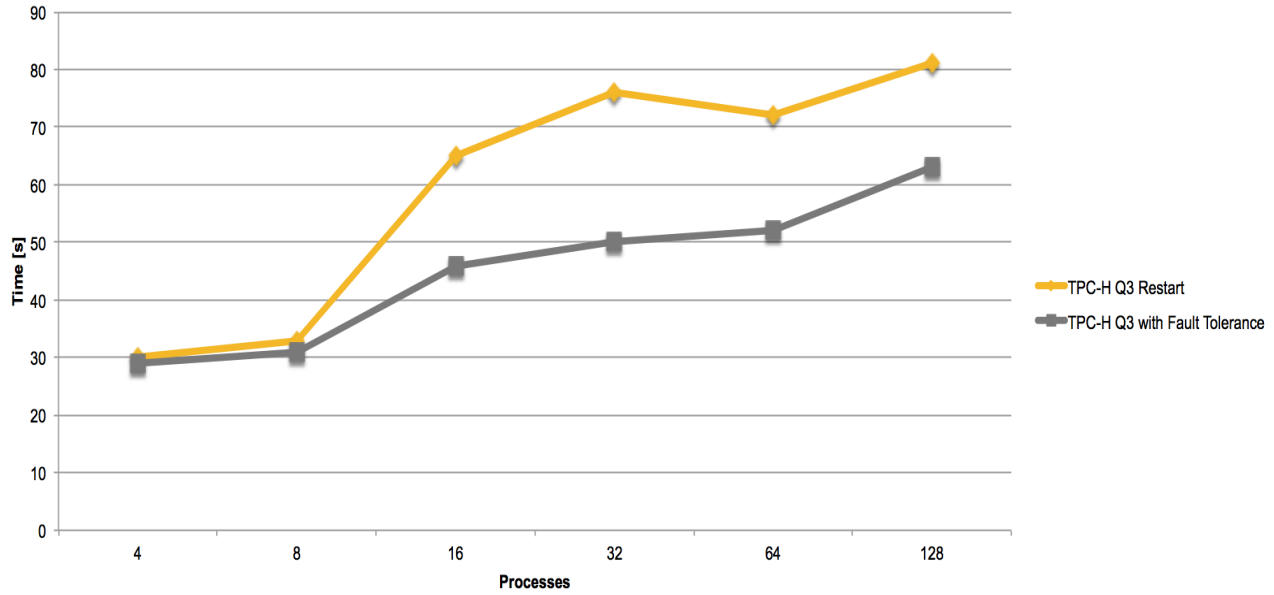

Figure 5.20: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 3 (1GB Data per Process)

The results of the measurements for failures during the last phase of the execution, which starts after the all-to-all-v communication and lasts until the end of the execution, are visualized in Figure 5.21. Since the last successful communication was an all-to-all-v communication every process knows the result of this communication and only the lost data partition has to be restored. No further communication steps are necessary to recreate a consistent state.

The runtime of the fault tolerant approach is better than the runtime of the non fault tolerant approach. The later failures occur during the execution, the more steps have to be repeated and therefore restarting

the computation results in overhead. Thus, the difference between the restarting approach and the fault tolerant approach increases for later point in times of the execution.

The results of the comparison between the fault tolerant approach and the non fault tolerant approach show that fault tolerance is adding overhead to the original communications. On the other hand restarting the computation in case of failures is introducing a serious amount of overhead too. Comparing these two approaches shows, in any case of failure it is preferable to use the fault tolerant approach. Even in case of failures that occur at the beginning of the computation the performance of the fault tolerant approach is noticeable better. In later failure cases the benefit is becoming even more. The only case in which the non fault tolerant approach is performing better is, if no failures are happening. If failures are occurring, our approach of fault tolerant collective communications is performing better and is to be preferred.

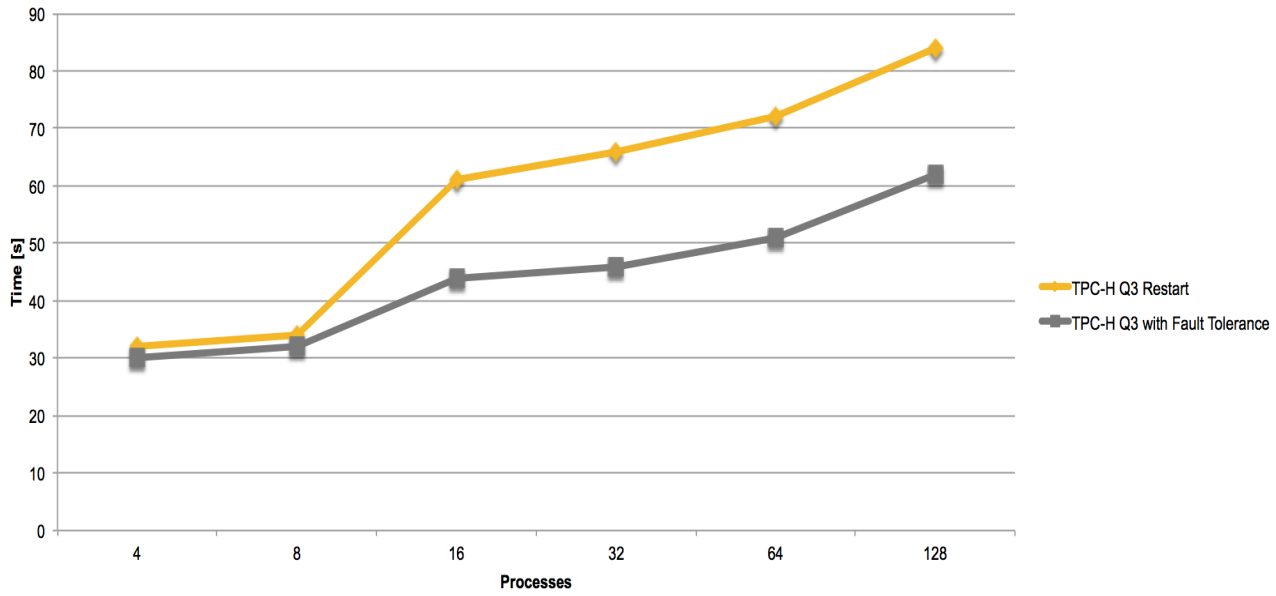

Figure 5.21: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 4 (1GB Data per Process)

5.4 Optimization

To reduce the introduced overhead of the fault tolerant collective communications we have implemented an optimization. The steps of the optimization are depicted in Figure 5.22.

In the optimization failures are identified and correspondingly the appropriate communication is chosen. If no failures have occurred before the communication the original MPI implementation with all its optimizations is used, otherwise the fault tolerant alternative is used. This is because MPI is not offering any possibility to conclude a collective communication with fewer processes than it is intended to. This means if a process is failing but the communication is continuing, a process has to replace the failed process and handle the tasks of the failed one. To use the original MPI communication in such a scenario, a process has to be spawned and to replace the failed process. If failures occurred before the communication, the fault tolerant implementation is used which is offering the possibility of a process to handle more than one task.

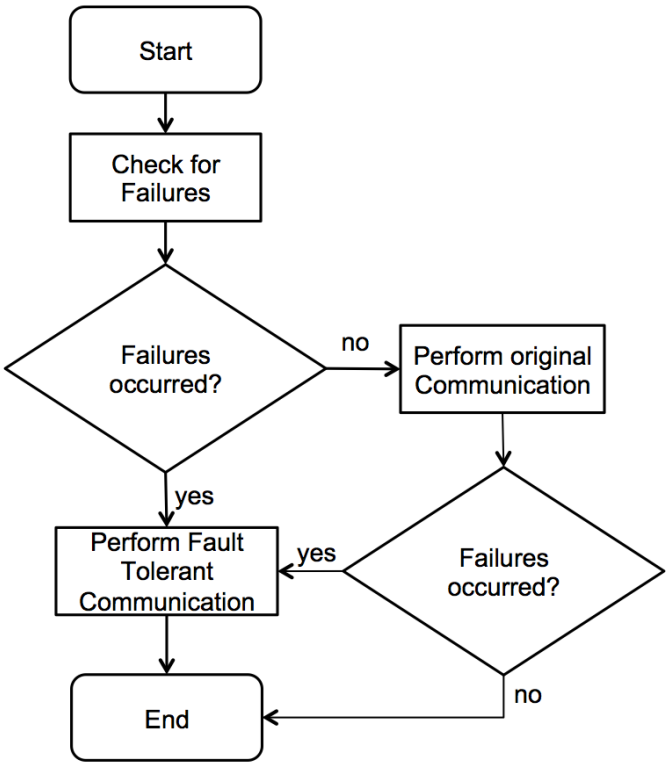

Figure 5.22: Optimization Strategy

If no failures occurred and the collective communication finished, an agreement is performed. This agreement is used to guarantee that no failure happened during the collective communication. Therefore, we could reduce the overhead to a minimal overhead in case of no failures. Only one agreement has to be performed to identify the successful termination of the communication.

These results can also been seen in the analysis of the runtime of the optimization strategy. The runtime for the reduce communication is shown in Figure 5.23, the results of the other communications behave similar.

It is obvious that the implementation of the optimization strategy is performing better than the fault tolerant implementation. The optimized implementation is also performing better than the reimplementation of the reduce communication without fault tolerance techniques. This is because the optimized variant is taking use of all the strategies and optimizations that are provided by the MPI implementation. Only a minimal overhead of one agreement is necessary.

The results of the other communications are similar. To evaluate the benefit of the optimization strategy we have used the optimized implementations while evaluating the TPC-H query 3. In Figure 5.24 the results for the runtime of the TPC-H query with a random failure during phase 4 are shown. The runtime of the TPC-H query which is using the original MPI functions is compared to the runtime of the TPC-H query that is using the fault tolerant implementation and to the TPC-H query that is using the optimized fault tolerant communications.

Since the implementation that is using the original MPI functions has to be restarted after the occurrence of failures, it is performing worse than the other implementations. The fault tolerant implementation does not have to restart the computation and therefore is performing better than the implementation with the original MPI functions. The optimized implementation is using the original MPI functions until an error occurs, therefore until phase 4 of the TPC-H query. This results in a clearly better performance than the implementation with the original MPI functions and the fault tolerant implementation.

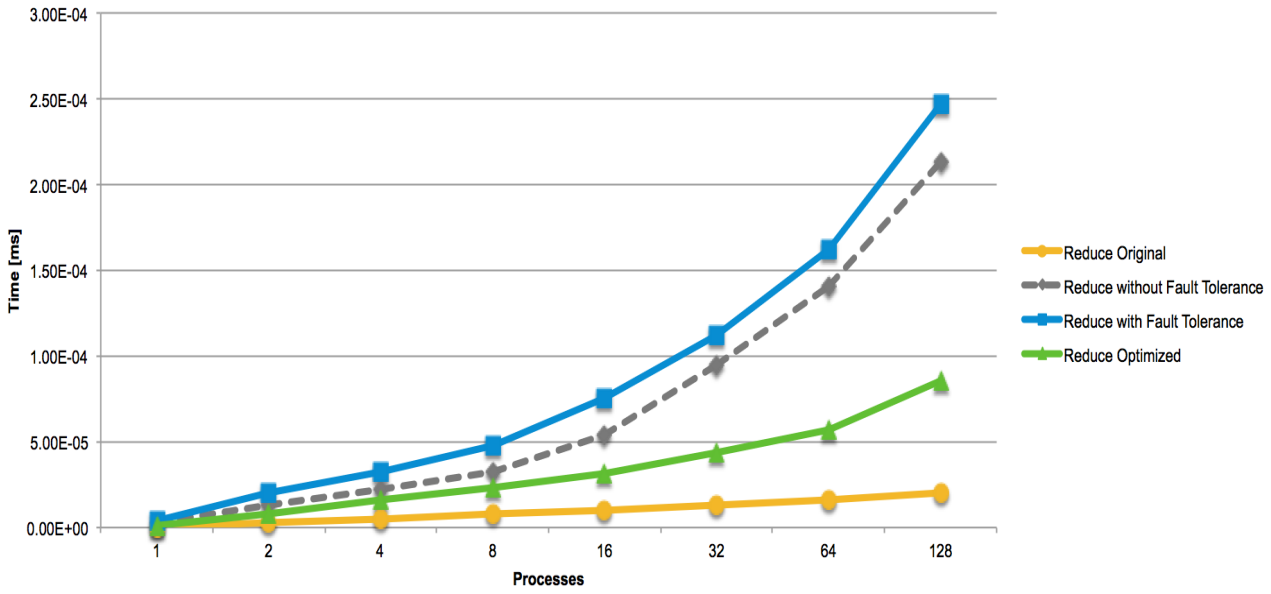

Figure 5.23: Optimization: Reduce Runtime

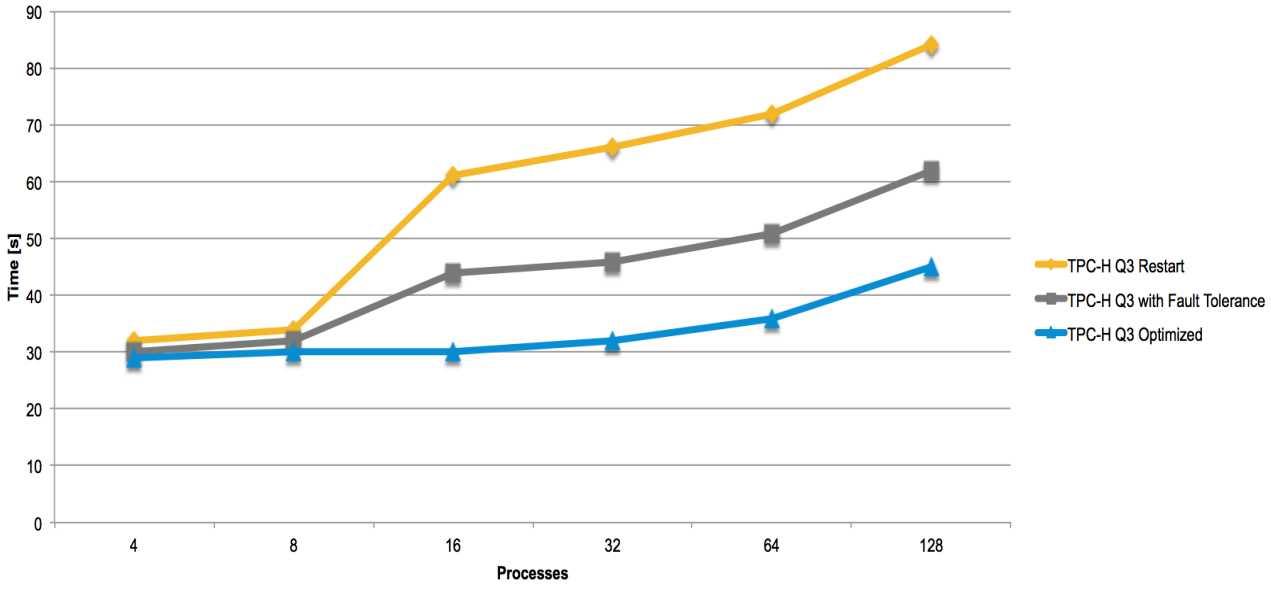

Figure 5.24: Optimization: Runtime of TPC-H Benchmark Query 3 with Failure in Phase 4 (1GB Data per Process)

6 Conclusion and Future Work

6.1 Conclusion

In this thesis we have proposed an approach for fault tolerance in distributed database systems by securing collective communications against failures.

At first, we have reviewed the background of this work by introducing the system model. Therefore, we have discussed parallel and distributed database management systems. Furthermore, we have revised the current state of the art of fault tolerance techniques for high performance computing systems before we introduced MPI and discussed approaches of applying fault tolerance techniques in MPI.

We designed a fault tolerance strategy that can be applied to collective communications and we developed five basic collective communications that can resist failures. Furthermore, we have shown which steps are necessary to recreate a stable state of the execution in order to continue without restarting the complete computation.

We tested the correctness of the algorithms under failures and guarantee that the algorithms will finish successfully as long as at least one process is surviving in the failed environment. In order to provide flexibility for implementers to react to failures, we have adapted the interfaces of the collective communications and compared them to the interfaces of the original MPI functions. Moreover, to test and show the applicability of our proposed fault tolerant communications for distributed database systems, we have implemented a distributed sorting algorithm and a TPC-H benchmark query while using the proposed fault tolerant collective communications.

To analyze the performance of our approach we have evaluated the introduced overhead by comparing the implemented fault tolerant collective communications with the original collective communications of MPI. We showed that applying our fault tolerance strategy is introducing overhead but the most overhead is introduced by reimplementing the MPI functions.

To show that the introduced overhead is acceptable we have evaluated our approach in a distributed database scenario, a TPC-H benchmark query, while simulating process failures. We have compared our approach with the alternative, non fault tolerant approach in which the execution has to be restarted and showed that in any case of failure our approach is performing better. We were also able to reduce the overhead of the fault tolerant approach by using a simple optimization strategy.

6.2 Future Work

Our implementation of fault tolerant collective communications for distributed database systems is performing better than the alternative of restarting the computation, but the overhead introduced by reimplementing the communications is still high and should be reduced. MPI is using multiple strategies and optimizations which choose the perfect algorithm for the current environment of the communication. Therefore, multiple algorithms that enable the collective communications can be implemented which will perform better under certain circumstances. Moreover, optimizations are needed to achieve the level of performance that MPI is providing.

The fault tolerance strategy could be introduced in the implementations of the MPI functions to reduce the overhead by using the existing implementations and optimizations.

We have implemented five basic collective communication algorithms that allow the implementation of more complex communications. Still, more algorithms can be implemented. For example an *Allgather* communication is a combination of a gather and a broadcast communication in which the result of the gathering is kept at every process that participates in the communication. But there are more efficient ways possible to realize this communication than simply using a combination of gather and broadcast.

Another important part can be to implement an optimizer that optimizes the program execution. Scenarios are possible in which processes are not having enough main memory to take over the tasks of failed processes or too many processes failed at some point in the execution where re-spawning processes is performing better than finishing the execution with the left amount of processes. We have not

implemented a strategy in which new processes are spawned for failed ones because the spawning function provided by MPI is exhibiting poor performance and scalability [12] and the resource management should be left to the implementer.

Our approach is focusing on the most common failure in distributed systems, the process failure. More failure types like silent errors or network errors are possible too and should be considered. Therefore, our fault tolerance strategy can be extended to provide resilience in more failure cases.

References

[1] TPC Benchmark H v.2.17.1. Technical Report, TPC, 2014. [Online] Published at http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf. Accessed: 2017-03-29.

[2] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[3] 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, 1999.

[4] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[5] Thara Angskun, George Bosilca, and Jack Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 471–482. Springer, 2007.

[6] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[7] R. Batchu, J. P. Neelamegam, Zhenqian Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. Mpi/fttm: architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 26–33, 2001.

[8] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 32. ACM, 2011.

[9] Pedro Bizarro and Henrique Madeira. The dimension-join: A new index for data warehouses. In *SBBD*, pages 259–273, 2001.

[10] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for user-level failure mitigation in the mpi-3 standard. *Department of Electrical Engineering and Computer Science, University of Tennessee*, 2012.

[11] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[12] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An evaluation of user-level failure mitigation support in mpi. In *European MPI Users' Group Meeting*, pages 193–203. Springer, 2012.

[13] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. In *European Conference on Parallel Processing*, pages 477–488. Springer, 2012.

[14] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, et al. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29. IEEE, 2002.

[15] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–9, Aug 2009.

[16] Aurelien Bouteiller, Thomas Herault, George Bosilca, Peng Du, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Transactions on Parallel Computing*, 1(2):10, 2015.

[17] Bouteiller Bouteiller, Franck Cappello, Thomas Herault, Krawezik Krawezik, Pierre Lemarinier, and Magniette Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 25–25. IEEE, 2003.

[18] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina. Compiler-enhanced incremental checkpointing for openmp applications. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[19] Guohong Cao and Mukesh Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, 1998.

[20] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127. ACM, 2006.

[21] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, pages 162–171. ACM, 2011.

[22] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer International Publishing, Cham, 2015.

[23] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[24] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[25] Graham E Fagg and Jack 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.

[26] Graham E Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J Dongarra. Process fault tolerance: Semantics, design and applications for high performance computing. *The International Journal of High Performance Computing Applications*, 19(4):465–477, 2005.

[27] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[28] W Donald Frazer and AC McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

[29] Ana Gainaru and Franck Cappello. Errors and faults. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 89–144. Springer International Publishing, 2015.

[30] Richard L Graham, Sung-Eun Choi, David J Daniel, Nehal N Desai, Ronald G Minnich, Craig E Rasmussen, L Dean Risinger, and Mitchel W Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *Proceedings of the 16th international conference on Supercomputing*, pages 77–83. ACM, 2002.

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

[32] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000. IEEE, 2011.

[33] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[34] Joshua Hursey, Richard L Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G Solt. Run-through stabilization: An mpi proposal for process fault tolerance. In *European MPI Users' Group Meeting*, pages 329–332. Springer, 2011.

[35] Joshua Hursey, Jeffrey M Squyres, Timothy I Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[36] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proceedings of IEEE 13th Symposium on Reliable Distributed Systems*, pages 42–51, Oct 1994.

[37] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, and Bronis R de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 57. ACM, 2014.

[38] Mark Levene and George Loizou. Why is the snowflake schema a good data warehouse design? *Information Systems*, 28(3):225–240, 2003.

[39] C-CJ Li and W Kent Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81. IEEE, 1990.

[40] Chung-Chi Jim Li, Elliot M Stewart, and W Kent Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, 24(10):871–886, 1994.

[41] Junsheng Long, W Kent Fuchs, and Jacob A Abraham. Compiler-assisted static checkpoint insertion. *Proc. 22*, 1:58–65, 1991.

[42] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, and Paraskevas Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. *Parallel Processing Letters*, 10(04):371–382, 2000.

[43] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. In *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, pages 65–73. IEEE, 2000.

[44] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[45] Tim Kieritz Michael Ikkert and Peter Sanders. Parallele algorithmen, 2009. [Online] Published at http://algo2.iti.kit.edu/sanders/courses/paralg16/skript.pdf. Accessed: 2017-03-29.

[46] James S Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[47] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2000.

[48] Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In *European Conference on Parallel Processing*, pages 799–803. Springer, 2002.

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

[50] Kento Sato, Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R de Supinski, Naoya Maruyama, and Satoshi Matsuoka. Fmi: Fault tolerant messaging interface for fast and transparent recovery. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1225–1234. IEEE, 2014.

[51] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[52] Fred B Schneider, David Gries, and Richard D Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, 1984.

[53] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, page 1094342014522573, 2014.

[54] Jon Stearley. Defining and measuring supercomputer reliability, availability, and serviceability (ras). In *Proceedings of the Linux clusters institute conference*, 2005.

[55] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 526–531. IEEE, 1996.

[56] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[57] Keita Teranishi and Michael A Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 51. ACM, 2014.

[58] Standards Coordinating Committee 10 (Terms and Definitions) Jane Radatz (Chair). The IEEE standard dictionary of electrical and electronics terms, 1996.

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

[60] L. Wang, Karthik Pattabiraman, Z. Kalbarczyk, R. K. Iyer, L. Votta, C. Vick, and A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821, June 2005.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[62] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.

A Appendix

Processes	Broadcast Original	Broadcast without FT	Broadcast with FT
1	$8.07949 * 10^{-14}$	$4.81312 * 10^{-13}$	$9.53187 * 10^{-12}$
2	$4.28954 * 10^{-13}$	$7.25704 * 10^{-13}$	$4.9268 * 10^{-11}$
4	$3.60234 * 10^{-13}$	$1.00503 * 10^{-12}$	$6.53325 * 10^{-11}$
8	$2.78381 * 10^{-11}$	$1.43798 * 10^{-11}$	$1.44501 * 10^{-7}$
16	$1.22126 * 10^{-11}$	$1.63266 * 10^{-10}$	$2.17102 * 10^{-7}$
32	$3.22958 * 10^{-11}$	$1.68528 * 10^{-11}$	$1.81405 * 10^{-7}$
64	$3.84919 * 10^{-11}$	$6.60447 * 10^{-11}$	$2.24603 * 10^{-7}$
128	$1.21863 * 10^{-10}$	$4.06838 * 10^{-11}$	$3.66932 * 10^{-7}$

Table A.1: Variance of Broadcast Evaluation

Processes	Scatter Original	Scatter without FT	Scatter with FT
1	$1.27706 * 10^{-13}$	$1.47393 * 10^{-12}$	$3.35007 * 10^{-12}$
2	$4.28502 * 10^{-13}$	$2.85875 * 10^{-11}$	$3.97101 * 10^{-11}$
4	$1.34222 * 10^{-7}$	$3.61308 * 10^{-11}$	$4.52074 * 10^{-11}$
8	$7.18118 * 10^{-7}$	$1.52189 * 10^{-10}$	$1.85132 * 10^{-10}$
16	$3.83015 * 10^{-11}$	$2.22667 * 10^{-9}$	$2.32426 * 10^{-9}$
32	$1.2133 * 10^{-10}$	$1.45226 * 10^{-9}$	$1.58083 * 10^{-9}$
64	$1.17046 * 10^{-10}$	$8.81638 * 10^{-10}$	$3.56842 * 10^{-9}$
128	$3.74709 * 10^{-8}$	$1.63075 * 10^{-8}$	$2.6367 * 10^{-8}$

Table A.2: Variance of Scatter Evaluation

Processes	Gather Original	Gather without FT	Gather with FT
1	$1.28119 * 10^{-13}$	$1.45501 * 10^{-12}$	$1.64142 * 10^{-12}$
2	$3.43515 * 10^{-13}$	$1.10688 * 10^{-11}$	$2.15008 * 10^{-12}$
4	$1.08101 * 10^{-12}$	$1.90991 * 10^{-11}$	$3.08081 * 10^{-11}$
8	$2.06188 * 10^{-12}$	$1.50023 * 10^{-10}$	$3.25892 * 10^{-10}$
16	$1.9568 * 10^{-12}$	$1.49157 * 10^{-9}$	$1.17343 * 10^{-9}$
32	$1.34733 * 10^{-10}$	$6.27033 * 10^{-10}$	$8.68472 * 10^{-10}$
64	$5.90226 * 10^{-10}$	$3.94966 * 10^{-9}$	$7.26951 * 10^{-9}$
128	$4.24123 * 10^{-8}$	$1.40926 * 10^{-7}$	$1.71948 * 10^{-7}$

Table A.3: Variance of Gather Evaluation

Processes	Reduce Original	Reduce without FT	Reduce with FT
1	$2.09398 * 10^{-13}$	$8.4191 * 10^{-12}$	$1.368 * 10^{-5}$
2	$6.54175 * 10^{-13}$	$3.13484 * 10^{-11}$	0.000308653
4	$2.51099 * 10^{-10}$	$4.29712 * 10^{-11}$	0.000741269
8	$2.47348 * 10^{-12}$	$1.17375 * 10^{-10}$	0.00154916
16	$2.77184 * 10^{-12}$	$4.25027 * 10^{-10}$	0.00384541
32	$5.15766 * 10^{-12}$	$2.19889 * 10^{-10}$	0.00929597
64	$7.71839 * 10^{-11}$	$7.9115 * 10^{-9}$	0.0366432
128	$1.57817 * 10^{-9}$	$7.00988 * 10^{-8}$	0.590208

Table A.4: Variance of Reduce Evaluation

Processes	All-to-all Original	All-to-all without FT	All-to-all with FT
1	$6.4313 * 10^{-6}$	$2.22208 * 10^{-12}$	$7.03189 * 10^{-12}$
2	0.000197712	$2.27275 * 10^{-11}$	$4.72488 * 10^{-11}$
4	0.000770725	$1.76681 * 10^{-11}$	$1.24099 * 10^{-10}$
8	0.00493304	$1.73208 * 10^{-10}$	$1.27859 * 10^{-9}$
16	0.0583225	$4.7608 * 10^{-6}$	$1.12147 * 10^{-7}$
32	0.730667	$7.31833 * 10^{-5}$	$1.95251 * 10^{-7}$
64	11.5723	0.00230769	$1.26828 * 10^{-6}$
128	104.266	0.0181848	$1.37167 * 10^{-6}$

Table A.5: Variance of All-to-all Evaluation