DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Integrating TeaMPI with ULFM for Hard Failure Tolerance in Simulation Software

Alexander Hölzl

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Integrating TeaMPI with ULFM for Hard Failure Tolerance in Simulation Software

Integration von TeaMPI und ULFM zur Toleranz von abgestürzten Prozessen in Simulationsanwendungen

Author:	Alexander Hölzl
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M.Sc. Philipp Samfaß
Submission Date:	15.08.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2020 Alexander Hölzl

Acknowledgments

I would like to thank my supervisor, Philipp Samfaß for his great support and guidance. I a would also like to thank Robert Mijaković of the LRZ for setting up ULFM, without his help the experimental part of this thesis would not have been possible. Furthermore I want to thank all my friends and family for supporting me while writing this thesis and finding all the spelling mistakes that I overlooked.

Abstract

With ever growing system complexity the mean time between failures (MTBF) of HPC systems is getting shorter. The conventional solution to failures in those kind of systems is checkpoint restart, but checkpoint restart does not scale well with regards to an increase in system scale. The reason for this is that an increase in system size goes hand in hand with a decreasing MTBF and a potential increase in checkpoint cost. One of the solutions to this problem is replication — fault tolerance through redundant computation. When using replication the work is performed multiple times either by running the same process multiple times or by running the complete application multiple times. In the first case, process replication, extraordinary care needs to be taken, so that the replicas of processes stay consistent, so that they can be swapped out with each other, should one fail. In the second case, known as group replication, this is not the case but should a failure occur the whole group is affected by the failure. Transparent group replication with MPI has been implemented with teaMPI. TeaMPI alleviates some of the performance issues caused by replication by dividing the workload into different tasks and sharing the outcomes of the tasks between the groups/teams. Although teaMPI achieves replication it does not yet offer actual fault tolerance and will crash if a failure of a process occurs. In this thesis we introduce extensions to teaMPI that make teaMPI robust against hard failures. As standard MPI does not offer needed fault tolerance functionality a modified version of OpenMPI, ULFM is used. ULFM offers a set of features that allows the programmer to respond to failures of MPI processes.
The implementation is tested using a simple solver for shallow water equations, SWE, and compared against an implementation using traditional checkpoint/restart. The evaluation showed that it is possible to achieve faster wall clock times in a scenario with multiple failures than it is with traditional checkpoint restart, although an integration with task sharing is necessary to be able to benefit from a fault tolerant teaMPI.

Contents

1 Introduction

The need for more computational power in science and engineering applications is leading to much more complicated HPC systems being built. The current target for the next generation of supercomputers is breaking the exaflop mark. As the era of Moore's Law is slowly coming to its end, such machines will be of much larger scale than current systems. One of the major challenges such systems will face is a much lower mean time between failures (MTBF). Whereas the MTBF of petascale systems is measured in days, it might be measured in minutes for exascale systems. This means that techniques that can handle frequent failures will be of increasing importance [1].
The main way of dealing with failures currently is global checkpoint restart which works by periodically saving recovery information to stable storage and using that information to restart the job in case of failure. A major drawback with this solution is, that it does not scale very well and exascale systems may spend much more time creating checkpoints and recovering from failures than doing actual computational work.
Therefore other alternatives need to be considered with one contender being replication. Replication works by executing the same calculations multiple times, either on a per process or per program instance level [2]. Multiple attempts, such as redMPI and SDR-MPI have been made to realize replication in MPI environments. The latest implementation teaMPI tries to mitigate the severe performance impact of other solutions by giving up on the strict consistency requirements that have been enforced by older approaches and instead allows splitting the program execution into multiple tasks and changes the order in which tasks are executed for each replica. If one replica finishes a task, the result can be shared with other replicas. One of the main problems of teaMPI that has not yet been solved is the lack of fault tolerance mechanisms in MPI itself [3]. The current MPI standard does not specify how an MPI implementation has to react to failures and as such, a process failure in an MPI application will lead to the whole MPI job being aborted. To tackle this problem an extension to the MPI standard, ULFM, has been proposed and implemented into OpenMPI [4].
The goal of this thesis is to modify teaMPI in a way to handle failures and allow the application to restart work from a consistent state without using checkpoints and the file system.
To be able to test the performance of the implementation, SWE, a simple finite volume solver for shallow water equations for tsunami simulations will be adapted to use features provided by teaMPI. A simple global checkpoint restart mechanism will be implemented as well, so that a comparison against current fault tolerance techniques can be drawn.

Overview

This thesis is structured in the following way: Section 2 will give background information about fault tolerance and replication in general. In Section 3 related work such as different replication implementations, Fenix and LAIK will be presented. The tools and software that was used will be explained in section 4 and the implementation and evaluation of the implementation will be discussed in sections 6 and 7.

2 Background

2.1 Fault Tolerance in HPC

A fault tolerant system is a system that is able to perform its intended function in spite of faults. Faults are root causes for an error such as a bitflip in the CPU's cache. An error in turn is the part of the state that might lead to a failure, that is when the service provided by the system deviates from the expected service. The most important metric to measure a system's performance in regard to fault tolerance is the MTBF, the mean time between failures. As fault tolerance is a very important topic, especially in the HPC world, countless approaches to improve the fault tolerance have been developed [5].

There are two main types of fault tolerance techniques: proactive and reactive fault tolerance. Reactive solutions try to keep the system or application running, by implementing tools that allow the recovery from failures. Proactive solutions on the other hand try to anticipate and proactively react to failures, so that they can be tolerated without the need to recover. An example for a proactive solution would be preemptive migration, which works by migrating processes from nodes that are about to fail to new nodes [6].

The most widely used fault tolerance technique is checkpointing and rollback recovery, a reactive solution. It works by periodically saving the application state and using the saved data to restart the application should a failure occur. A problem that arises when checkpointing distributed software is that the checkpoint data of each process needs to be consistent with the rest. Multiple ways such as coordinated checkpointing or uncoordinated checkpointing with message logging exist to ensure consistent checkpoints [1]. Another problem that has been investigated is the choice of the correct checkpoint interval. If checkpoints are not written frequently enough a lot of work has to be redone in case of a failure. But if the application saves its checkpoints too often time will be wasted as the creation of checkpoints is usually a costly process. A simple approximation for an ideal checkpoint interval has been proposed by J. Young in 1974:

$$\tau_{opt} = \sqrt{2\delta M} \tag{2.1}$$

with τ_{opt} being the optimal checkpoint interval, δ the time needed to save the checkpoint and M the MTBF of the system [7].

Other approaches that are not within the scope of this thesis are application specific solutions, that can be used to significantly improve the performance of general purpose solutions using specific properties of the application or ways on how to deal with silent errors, which are errors that do not manifest themselves in a crash, but for example in data corruption or wrong results [1].

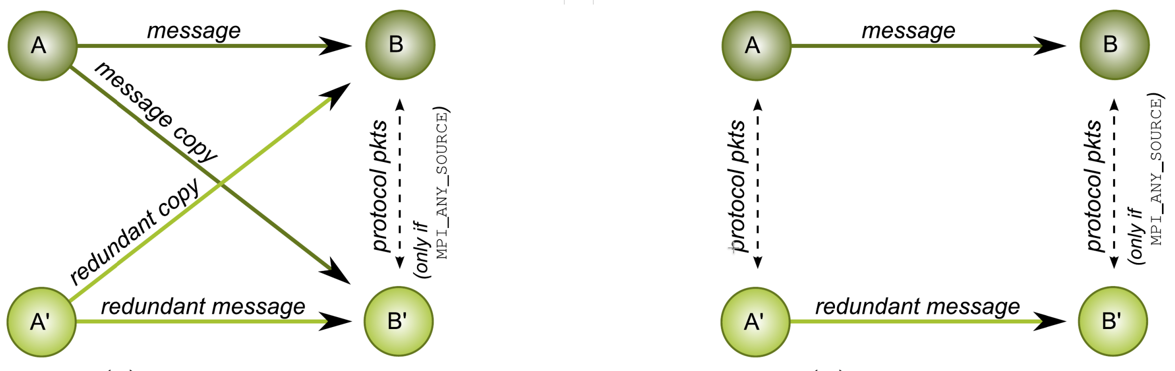
Another major fault tolerance technique that needs to be covered is replication.

2.2 Replication

A major problem that arises when using Checkpointing and Rollback Recovery is that the scalabilty in relation to decreasing MTBF is bad. The current way to improve the execution speed of parallel programs is to add more processors. But with a rising number of processors the probability that a failure occurs during program execution rises. That means that the MTBF gets smaller. To counteract this the checkpoint frequency needs to to raised. A higher checkpoint frequency means that the program has to waste more time writing checkpoints. All of this put together leads to the fact that the total expected program run-time will rise after a certain threshold regarding the number of processors has been exceeded, as the program will spend most of its time writing checkpoints and recovering from failures.
One possible solution for this problem is replication. When using replication the same computation is performed redundantly on different processors. Due to the redundancy the crash of a processor does not lead to failure of the complete application. At first glance the use of replication may seem counter-intuitive as it will waste at least half of the used resources, but replication drastically improves the parallel efficiency as each replica is run at a smaller scale, that means with fewer processors, and does not suffer from the aforementioned problem. The use of replication also provides some other advantages. As writing checkpoints is not as critical when using replication, the I/O system does not need to be as powerful which may lead to a lower overall system cost and a better energy efficiency. The result computed by different replicas can also be compared against each other so that silent errors can be detected.
It is important to note that replication is not a stand-alone solution and needs to be accompanied by some kind of checkpointing system, to recover from failures that are too severe to be handled by replication. But, as already stated, when using replication the MTBF of the system will be much higher, so that checkpointing is not as problematic as without replication [2, 8].

2.2.1 Process Replication

When using Process Replication only one instance of the application is launched, but each process of the application is replicated multiple times. The main challenge that needs to be solved is synchronization. It is necessary that all replicas of the same process ensure that they always keep their states synchronized, so that in case of failure a surviving replica can take the place of the failed process. Synchronization is not a trivial problem as it essentially needs to be made sure that every process receives the same messages as its replicas. In an MPI environment this is mostly a problem that stems from non-deterministic message passing using the `MPI_ANY_SOURCE` parameter in receive operations or other MPI functions such as `MPI_Wtime`, `MPI_Probe` and `MPI_Test`. To solve this problem two different synchronization modes have been proposed: the mirror and parallel protocol.
When using the mirror protocol each sender transmits its message to every replica of the receiver. The receiver requires only one message to arrive in order to progress. The parallel protocol on the other hand requires that every replicated rank has a single replica for every other rank with which it communicates. Should a failure happen, another replica has to take



(a) Mirror Protocol **(b)** Parallel Protocol

Figure 2.1: When using the Mirror Protocol every replica has to communicate with every other replica. The Parallel Protocol only requires communication between designated replicas. When using `MPI_ANY_SOURCE` additional synchronization is necessary. [8]

the part of the failed one [2, 8]. A program that uses process replication will only need to restart from a checkpoint if every process from the same replication set fails. This leads to a drastically increased MTBF, but the obvious disadvantage is that the synchronization of processes incurs a significant communication overhead.
Some ways to improve efficiency have been proposed such as passive replication where only so called master ranks communicate with each other. Each master rank has one or multiple replicas, so called slave ranks, and the result from MPI operations is broadcast from the master rank to the slave ranks [9]. Partial replication where only some ranks may have replicas has also been studied. But partial replication as well as passive replication did not yield the desired performance improvements [10]. There have been efforts, such as rMPI or MR-MPI, to realize transparent process replication in MPI that will be covered in more detail in chapter 3.

2.2.2 Group Replication

In contrast to Process Replication, Group Replication works by running the same application on different processor groups. The different groups work independent from each other and are not strongly synchronized.
Group Replication requires that the application is moldable, that means the workload can be split into different chunks. If one group finishes a chunk of work, it takes a checkpoint and distributes that to the other groups. Should one group suffer from a failure it can also reload a checkpoint made by another group. The other groups can then load this checkpoint, so that no unnecessary redundant computation is performed [2]. An implementation that realizes Group Replication in MPI is teaMPI, which will be explained in further detail below. With teaMPI a concept called task-shuffling and task-outcome sharing was introduced. This works by having each group executing the individual chunks of work, the so called tasks,

in a different order and share the outcome of those tasks with each other. This significantly reduces the overhead of group replication as the amount of redundant work is lowered drastically. But making use of this technique is only possible with workloads that allow the shuffling of tasks [3].
One major advantage of group replication is, that it can be added to existing applications very easily, as demonstrated by teaMPI. But from a theoretical standpoint group replication is outperformed by process replication as process replication much more drastically lowers the MTBF [2]. This is due to the fact that when using a group replication, a failure will disable a whole group, whereas with processes replication a failure will disable only a single process. Sadly there are no direct comparisons between different implementations, so no statement about performance differences, especially between teaMPI with task-shuffling and older process replication based implementations can be made.

3 Related Work

3.1 Replication in MPI

There have been multiple attempts to add replication functionality to MPI. The first such attempt has been rMPI [8] which was published in 2011. rMPI implements the parallel as well as the mirror protocol to ensure consistency between replicas but due to the way how replicas and masters are differentiated, only duplex replication, that means one leader and one replica per process, is possible. Furthermore rMPI is implemented using PMPI but also uses some MPICH internal functionality and can therefore not be used with other MPI implementations. rMPI's successors MR-MPI [11] and redMPI [12] also included partial redundancy and allowed the usage of replication factors higher than two. Further investigation of partial replication showed that it is not a useful feature as failures strike at unpredictable locations. Implementing passive replication within MPI has been the goal of the MPIEcho project. MPIEcho is implemented using the MPI Profiling Interface, which allows it to be used with all modern MPI implementations. MPIEcho uses passive replication which means that only master ranks communicate with each other and the results of those communication events are broadcast to the replicas. The problem with this approach is, that it results in higher latency [9].

SDR-MPI (Send Determinism MPI) is another approach, but it differs from the previous

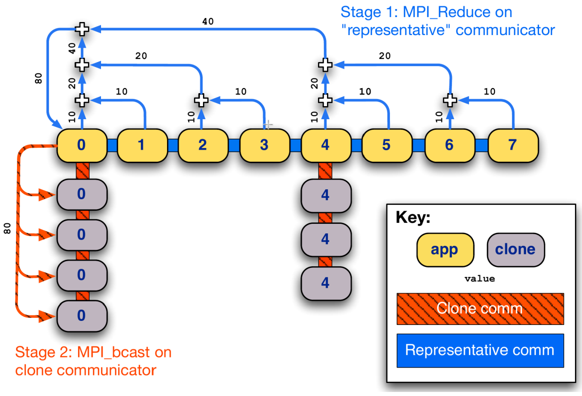

Stage 1: MPI_Reduce on "representative" communicator
40
20
20
10
10
10
10
80
0
1
2
3
4
5
6
7
0
0
0
0
4
4
4
Key:
app
clone
value
Clone comm
Representative comm
Stage 2: MPI_bcast on clone communicator

Figure 3.1: Passive Replication in MPIEcho. Only the master ranks participate in the actual reduce operation. The result is broadcasted to the existing clones. [9]

mentioned ones in that way that it uses a modified MPI implementation and is therefore not portable. With SDR-MPI work-sharing between replicas was introduced as well, but to make use of it applications had to be rewritten using a custom task API [13, 14][15].

3.2 LAIK

LAIK, standing for Leichtgewichtige Anwendungsintegrierte Datenerhaltungskomponente or in English Lightweight Application Integrated Fault-Tolerant Data Containers, is a library that allows to implement parallel applications. To be more specific LAIK offers data containers that allow programmers to access data using a memory model which resembles a software controlled non-uniform shared memory architecture on a distributed memory system. With LAIK there is no need to write communication code directly, communication is handled by LAIK. LAIK is able to use multiple backends such as TCP or MPI for communication. LAIK also offers transparent fault tolerance and also makes use of ULFM when using the MPI backend. To achieve fault tolerance LAIK offers in-memory checkpointing. To ensure tolerance of large scale failures LAIK stores two copies of the checkpoint on two different ranks. It is the job of the programmer to select the appropriate ranks so that the checkpoints are stored on different physical nodes. As LAIK manages the application data, it can automatically create checkpoints. In case of a failure, LAIK can remove failed ranks and redistribute data to surviving nodes. As the application programmer does not need to write communication code, but communication is carried out by LAIK, LAIK can transparently restore the environment. The application programer only needs to ensure that the program is able to continue execution at the same point where the checkpoint was created. This might be challenging as this might differ from the point where the application has called the LAIK restore routines. [16].

3.3 Fenix

Another approach to fault tolerant HPC application has been realized with Fenix. Compared to LAIK, Fenix is a more low-level approach which is also implemented using PMPI. This means that it will be possible to use Fenix with any MPI implementation should the features proposed in ULFM become part of the official MPI standard. In contrast to LAIK, the application programmer still has to write communication code, but in case of failures Fenix automatically restores the MPI communication capabilities using features provided by ULFM. To be able to successfully restore the application, Fenix relies on checkpoints that are stored in neighbor node's memory. Those checkpoints need to be created manually by a call to the `Fenix_Checkpoint_Allocate` function [17].

Functionality

If a failure is detected, the first step is to propagate the failure notification to all running processes. This is done by revoking all existing communicators. To be able to do that, the user has to register self created communicators using the `Fenix_Comm_Add` function. Next, the world communicator is shrunk to remove all failed processes and new processes are spawned and merged with the world communicator and assigned the correct ranks. If this has been done successfully, all surviving processes use a long jump to revert program execution to `Fenix_Init`. Newly spawned processes are already inside `Fenix_Init`. After that the checkpoint can be loaded and program execution can resume. Fenix is also able to

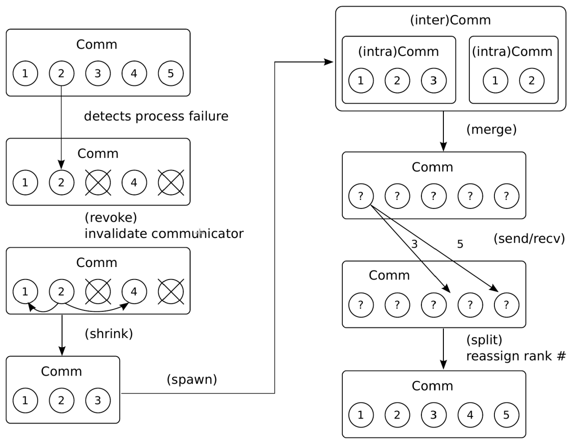


Figure 3.2: Fenix recovery strategy when using cold spares [17]

use warm spares, processes that are started during normal program startup but are not used for computation, instead of newly spawned cold spares.
The functionality and recovery strategy is very similar to the fault tolerant teaMPI that will be presented in this thesis. The main difference is that with teaMPI the checkpoint only has to be created in case of an actual failure [17].

4 Software Techniques for Fault Tolerance

4.1 MPI and ULFM

MPI, the Message Passing Interface, is a standard that describes a set of functions that can be used to implement communication via message passing between address spaces of different processes. There are multiple MPI implementations of this standard such as MPICH, Intel-MPI and OpenMPI. MPI is one of the most used APIs in the HPC world and provides a variety of functionality, such as point-to-point communication and collective operations. With the introduction of MPI 2 additional features such as parallel file I/O and dynamic process creation were also added.[18]
One of the major issues of the current MPI standard is that it does not require any fault tolerance functionality to be implemented. Although it is possible to implement error-handlers that are called when the MPI implementation detects an errror during a call to a MPI function, these can not be used to write fault tolerant code. Instead section 8.3 of the MPI standard states that:

> *"After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so"*

Meaning it is not possible to continue to use MPI for reliable communication after a failure has occurred. Additionally, the default error handler that is set when starting a MPI application is `MPI_ERRORS_ARE_FATAL`, which will simply abort the MPI job if a failure is detected.

4.1.1 ULFM

In order to alleviate this problem, an extension to the MPI standard has been proposed. ULFM, standing for User Level Fault Mitigation, introduces a set of new functionality that can be used to write fault tolerant MPI programs. It is important to note that ULFM does not specify a specific failure recovery model such as for example a checkpointing algorithm. Instead it provides tools that can be used to clean up the MPI state, so that communication is possible after failures occur.
In the following subsection a short overview over the additional features added by ULFM will be given[4].

Error Codes

ULFM introduces three new MPI error codes that can be returned and used to notify the program about failures. `MPI_ERR_PROC_FAILED` is used when a process failure prevents the completion of MPI operations. `MPI_ERR_PROC_FAILED_PENDING` is used in conjunction with non-blocking receives using the `MPI_ANY_SOURCE` parameter. This error is returned when a potential sender, that matches the posted receive has failed. Finally `MPI_ERR_REVOKED` is returned when a process has used the `MPI_Comm_Revoke` to mark a communicator as unfit for further usage.

Failure Reporting

In MPI failures are reported on a per communication basis. It may be possible that a communication object contains failed processes, but if not all processes communicate directly with each other some processes will not be notified of failures. Similarly a collective operation might finish successful on some processes and fail on others. A situation that also might arise is that some processes start a failure recovery procedure whereas others wait for blocking collective calls to finish resulting in a deadlock. To avoid this situation, the function `MPI_Comm_revoke` has been proposed. It is similar to `MPI_Abort` in its behavior, as it is a collective operation that does not require a symmetric call on all participating processes. `MPI_Comm_revoke` can be used to revoke a communicator. If a process tries to use a MPI operation on a revoked communicator, the call will return with the error `MPI_ERR_REVOKED`. This renders the communicator unusable. As all operations on this communicator will eventually fail, all processes can enter repair procedures without the danger of deadlocks arising.

Restoring Communication Capabilities

In order to restore the communication capabilities on a communicator basis, ULFM offers `MPI_Comm_shrink`. This function creates a new communicator from an old one containing failed processes. The new communicator will contain the same processes as the old one but without the failed ones. Should there be new failures discovered during the execution of the shrink operation, these failed processes will also be excluded from the new communicator.

Working with failed communicators

If using `MPI_Comm_Revoke` and `MPI_Comm_shrink` is not necessary, for example because the communication pattern of the application is structured in a way that deadlocks are not possible, ULFM also provides constructs to continue using a communicator containing failed processes for point-to-point operations. `MPI_Comm_failure_ack` acknowledges the group of failed processes and notifies the MPI library, that the application knows about failed processes. This means that the application itself is responsible to make sure that when using MPI wildcard receive operations with `MPI_ANY_SOURCE`, the receive is not matched by a failed process. `MPI_Comm_failure_get_acked` can be used to get the group of processes that are

locally known to have failed. Both of these functions can be used on revoked communicators as well.
Acknowledging that the application knows about failures is necessary, because when using `MPI_ANY_SOURCE` it might be possible that a surviving process is waiting for a message that is supposed to be sent by a failed processes which will lead to a deadlock. `MPI_Comm_failure_ack` essentially signals the MPI library that the processes has knowledge about failed processes and will ensure that no deadlocks happen on the application level. If this is not done MPI has to return `MPI_ERR_PROC_FAILED_PENDING` every time an operation using `MPI_ANY_SOURCE` is done, which makes the use of such operations impossible.

State Agreement

To be able to ensure a consistent state `MPI_Comm_agree` can be used. This function can be thought of as a fault tolerant `MPI_Allreduce` that computes a conjunction of boolean values provided by the living processes. Dead processes will participate with a default value of false. This operation will complete successfully even if a communicator contains failed processes or if failures happen during the agreement.

4.1.2 Possible Recovery Strategies

There are two main strategies to handle dropouts of processes due to failures with ULFM. Those are shrink and substitute. The substitute strategy uses either cold or warm spare processes that can replace failed ones. Warm spares are MPI ranks that are created when launching the MPI application, but do not perform real computation until they are needed to replace failed ones. Cold spares are processes that are spawned when needed using the dynamic process creation functionality added by MPI 2. The other way of dealing with failed processes is the shrink strategy, which works by simply removing failed processes from the communicators.
The substitute strategy has the advantage of being a more general purpose strategy as after a failure there are exactly as many working MPI ranks as the application has been launched with. There are some applications that impose certain restrictions on the needed number of ranks, e.g. a cube or square number of ranks is needed. This might make the use of the shrink strategy unsuitable for a number of applications. Studies of recovery comparing the shrink strategy and warm spares have shown that the use of spare processes can provide a performance advantage, but with growing number of processes this advantage decreases. The major disadvantage that arises when using spares is that it can disrupt the regular communication patterns of the application as the spares can be run on different nodes than the original ones. This might mitigate the performance benefits of spares if the application has high communication overhead.
To my knowledge there are no direct comparisons between cold and warm spares but it is to be expected that cold spares will suffer from higher overheads than warm spares as cold spares need to spawned and initialized during run-time [19].

4.1.3 MPI Profiling Interface

MPI declares all functions twice, each function is declared as `MPI_Function_name` as well as `PMPI_Function_name`. The `MPI_Function_name` functions are defined as weak symbols which allows programmers to overwrite the original functions with different implementations that alter the semantics of those functions. The original MPI functionality can be accessed when using the `PMPI_Function_name` versions. The main purpose of this interface is to write profiling code that can be used to do performance measurements, but it is also possible to implement completely new features such as replication using the profiling interface. This has the advantage that such code can be used with different MPI implementations.

4.2 teaMPI

TeaMPI enables transparent group replication for MPI applications. This is done by evenly dividing all ranks between multiple groups. Those groups are also called teams. When starting an application with teaMPI, each is rank is assigned to a special team communicator and every call to a MPI function made by the application using `MPI_COMM_WORLD` is automatically translated to a call to the same function but using the team communicator. This way each team only sees its own processes and behaves like it would have, had it been launched with fewer processes from the beginning. TeaMPI is implemented using the MPI Profiling Interface, that means existing applications can easily make use of the features added by teaMPI. It is important to note that not every available MPI feature has been implemented in teaMPI, but adding additional functionality can be done easily. For most operations the only necessary change is to map `MPI_COMM_WORLD` to the team communicator. The implemented functionality is sufficient to run production code such as ExaHyPe. Calls to functions that have not been implemented will be using the standard MPI implementation.

In contrast to other approaches teaMPI does not enforce a strict synchronization between

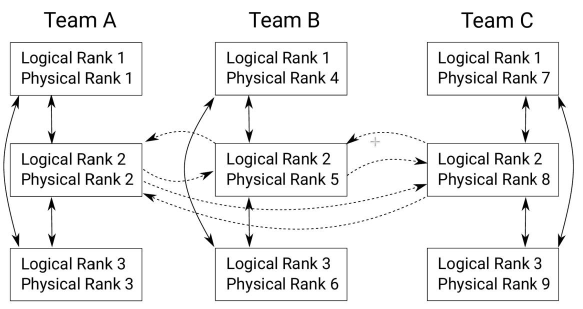


Figure 4.1: During normal operation the ranks only communicate in their respective teams. Heartbeats (dashed lines) are exchanged between the same logical ranks. [3]

teams, but it is possible to exchange heartbeats containing timestamps between corresponding ranks of different teams. Those heartbeats can be used to identify slower teams. The heartbeats can also carry hash values of results supplied by the application, in order to ensure data consistency. These heartbeats are carried out asynchronously and can be initiated by a call to `MPI_Sendrecv`. To be able to differentiate between a normal call to `MPI_Sendrecv` and a heartbeat, the heartbeats need to be called with `MPI_COMM_SELF` as the communicator. As a Send-Receive operation on this communicator is not a useful operation in normal code, this can be safely used to implement the heartbeat functionality.[15]

Another interesting feature that can be implemented using teaMPI is Task Sharing. To be able to use this feature, the workload has to be split into multiple independent tasks. Each team can then execute the tasks in a different order — the so called task shuffling. Outcomes of successfully finished tasks can then be shared with other teams, so that they do not need to redo already completed tasks. This greatly reduces the cost of replication as the degree of redundancy is much lower.

One of the major issues with teaMPI is that it has not been integrated with fault tolerance mechanisms yet. This leads to the fact that if a process failure happens, the whole MPI job will abort, even if the redundancy provided by different teams could be used to recover from failures [3].

5 Introduction to the SWE Code

5.1 SWE

The SWE framework [20] provides a finite volume solver for shallow water equations, mainly designed to simulate tsunamis. The focus of SWE is to teach parallel programming models, therefore the code is designed to be easily understandable for students inexperienced in parallel programming and allows the simple implementation of different programming models such as MPI, Cuda or OpenMP. Implementations using UCP++ and Charm++ have also been done [21]

5.2 Shallow Water Equations

The Shallow Water Equations itself are a set of nonlinear partial differential equations, that describe the flow of water if the horizontal dimension is predominant. This means that the wavelength of the modeled body of water is much larger than the its depth. The underlying principles of these equations are the conservation of mass which is used to derive the equation of the water height h and the conservation of linear momentum which is used to derive the water velocity in horizontal and vertical direction, hu and hv. The gravitational constant is denoted as g.

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t,x,y) \tag{5.1}$$

$S(t,x,y)$ is a source term that can describe the influence of the Coriolis effect, wind, friction and bathymetry. In the case of the SWE simulation, only the effect of the bathymetry is taken into account and is given by 5.2.

$$S(t,x,y) = \begin{bmatrix} 0 \\ -\frac{\partial}{\partial x}(ghb) \\ -\frac{\partial}{\partial y}(ghb) \end{bmatrix} \tag{5.2}$$

A Finite Volume approach is used to solve the equations. The changes from one time step to the next are described in 5.3.

$$\begin{aligned} Q_{i,j}^{n+1} = Q_{i,j}^{n} &- \frac{\Delta t}{\Delta x}\left(A^+\Delta Q_{i-1/2,j} + A^-\Delta Q_{i+1/2,j}^n\right) \\ &- \frac{\Delta t}{\Delta y}\left(B^+\Delta Q_{i,j-1/2} + B^-\Delta Q_{i,j+1/2}^n\right) \end{aligned} \tag{5.3}$$

With $Q^n_{i,j} = \left[h_{i,j}, (hu)_{i,j}, (hv)_{i,j}\right]$ being the vector of conserved quantities at t^n. Δt denotes the length of the time step and Δx and Δy the size of the grid cells. $A^{\pm}\Delta Q_{i\mp 1/2,j}$ is the solution to the Riemann problem on the left and right side and $B^{\pm}\Delta Q_{i,j\mp 1/2}$ the solution to the Riemann problem on the top and bottom side of the cell i, j [20, 21, 22].

5.3 Software Architecture

SWE_Block

SWE splits the computational domain between multiple cartesian grid blocks. These blocks are the core element of SWE's software architecture and are represented by the abstract `SWE_Block` class. This class provides 2D arrays that store water heights, momentum and bathymetry. It also provides virtual methods for setting boundary conditions and for simulating a time step. The data transfer between different blocks is implemented using ghost and copy-layers. Ghost-layers are used to implement boundary conditions and store data from adjacent blocks, whereas copy-layers are replicated to adjacent blocks. `SWE_Block` provides functions that allow the programmer to register copy-layers and get the ghost-layer. The data in the ghost-layer is the data contained in the copy layers of adjacent blocks and is sent between neighbouring blocks before the next time step can be calculated. The calling component is responsible for providing that data, so it is possible to implement the information exchange between blocks using different techniques. In the MPI version every rank is assigned one block and multiple calls to `MPI_Sendrecv` are used to exchange data between adjacent blocks.

Solver

The SWE-Framework implements a number of different solvers that vary in computational power and accuracy. The purpose of those solvers is to compute the one-dimensional Riemann-problem between two neighbouring cells of the grid blocks, given at initial time $t = 0$ as:

$$q(x,0) = \begin{cases} q_l, & \text{if } x < 0 \\ q_r, & \text{if } x > 0 \end{cases} \tag{5.4}$$

with

$$q_l = \begin{bmatrix} h_l \\ (hu)_l \\ b_l \end{bmatrix}, q_r = \begin{bmatrix} h_r \\ (hu)_r \\ b_r \end{bmatrix} \tag{5.5}$$

with q_l and q_r being the cell variables on the left and right side of the edge with the bathymetry values b_l and b_r. The implemented solvers are the *F-Wave Solver*, the *Augmented-Riemann Solver* and the *Hybrid Solver*.

SWE_Scenario

The `SWE_Scenario` interface is used to provide initial values that are needed to start the simulation. `SWE_Scenario` provides a number of functions such as `getWaterHeight(float` `x, float y)` or `getVeloc_u(float x, float y)` that will be called by a `SWE_Block` to fill the arrays storing the data with the necessary start values. These scenarios can be of analytical nature but with ASAGI real geo-data can also be used [23].

Output Files

Output produced by SWE is written using the NetCDF [24] format, although VTK files can be written if the NetCDF library is not available. When using the MPI version of SWE, every rank produces its own output file. The user can specify how many timesteps should be computed. The total duration of the simulation as given by the scenario is then split into as many intervals as desired and during simulation every time one of those intervals has passed an output file is written. Those output files contain the bathymetry, momentum in x and y direction and height values for each cell of the block.

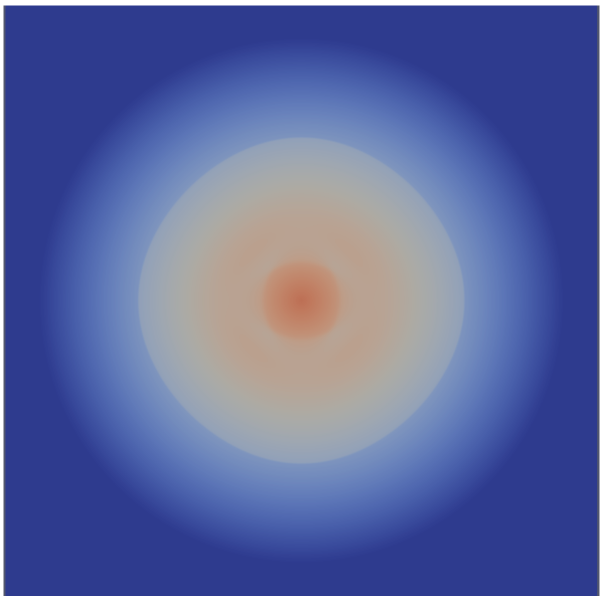


Figure 5.1: Output of SWE when viewed in ParaView. This output has been created using the splashing cone scenario

SWE_MPI

The first step when starting a simulation using MPI is to calculate how many cells each block has. Then depending on the chosen solver a suitable `SWE_Block` is created and initialized with the start values using the chosen scenario. Information is exchanged with the adjacent blocks to obtain the boundary values. The output file is then created and initialized with the non time-dependent values which are the bathymetry and the coordinates of the cells. Then the main computational loop is entered. As long as not all desired outputs have been written, the loop exchanges all ghost layers, computes the new values and determines the smallest global time step via a call to `MPI_Allreduce`[20].

6 Implementation

The main goal of this thesis is to add reactive team recovery to teaMPI and to evaluate the performance of the implementation using SWE. With reactive team recovery teaMPI will be able to tolerate process failures. To do this, two tasks need to be achieved. The first is implementing a simple checkpoint/restart mechanism in SWE so that a comparison to an implementation using teaMPI can be drawn. The second is changing teaMPI so that it is able to respond to failures of processes.
As already stated in section 4.1.2, there are three ways to implement fault tolerance with ULFM: warm spares, cold spares and shrinking the active set of processes. In this thesis warm and cold spares have been implemented. The reason for this is that spares allows to keep the size of teams constant which simplifies the changes that need to be done in teaMPI.

6.1 Checkpoints in SWE

As SWE is already able to write output files (see section 5.1) which already contain almost all information needed to restart program execution, only a small amount of changes had to be done to enable checkpoint/restart in SWE. First in addition to the already existing output-files, a small amount of meta data, that is necessary to reload the checkpoint, needs to be created. This meta data includes information like the end time of the simulations, and boundary conditions of the simulation domain.
In order to read the stored data, a new SWE-scenario was added that initializes the `SWE_Block` with the data from the output files. A limitation in the current implementation is that the simulation needs to be restarted with the same number of MPI ranks as it was originally launched. Furthermore SWE used to create the output files by having the user specify how many files should be written in total. The total simulation time was then split into as many checkpoints as needed, and every time a new interval has been simulated completely, a new output file was written. This behavior was changed so that the user can now enter a desired checkpoint interval directly, by supplying it as a startup parameter. Every time the main computational loop has been running as long as the checkpoint interval, a checkpoint is created. This is done so that the user has a more direct control over the time that passes between two IO operations which is more convenient when doing experiments regarding the performance of the checkpointing implementation.

6.2 Reactive Team Recovery for Fault Tolerance in teaMPI

To be able to choose how teaMPI behaves when a failure occurs, the programmer can choose the desired error handling strategy using `TMPI_SetErrorHandlingStrategy`. The available strategies are `TMPI_KillTeamErrorHandler`, `TMPI_RespawnProcErrorHandler` and `TMPI_WarmSpareErrorHandler`. `TMPI_KillTeamErrorHandler` will simply exit all surviving processes in a team, if one of the processes in the team encountered a failure. Strategy `TMPI_RespawnProcErrorHandler` uses cold spares and `TMPI_WarmSpareErrorHandler` uses warm spares, both of those strategies will be described in further detail below. By default teaMPI does not do error handling and will crash if a failure occurs.
As integrating spares into existing communication structures will lead to inconsistent states in the application software, teaMPI will call user-defined callbacks in the application that allow the user to restore a consistent state. The first callback is called by a team without failed processes and is used to create a checkpoint or to send data to damaged teams. The other callback is used to load or receive the checkpoint. The callbacks can be set with the `TMPI_SetCreateCheckpointCallback` and `TMPI_SetLoadCheckpointCallback` functions. When a checkpoint needs to be loaded, teaMPI passes the number of the team that can be used to load the data as a parameter to the callback. If a team has to create a checkpoint, it receives the failed teams as a parameter.
The main idea behind fault tolerant teaMPI is that teaMPI manages a number of communicators that are automatically rebuilt if a failure occurs. Those communicators encompass the ones created during teaMPI initialization, such as the team communicators, but also `TMPI_COMM_WORLD` which is a managed version of `MPI_COMM_WORLD`. Every time an error is detected in one of those communicators, failed processes are removed from the world communicator and all other communicators are rebuilt.
As errors are detected on a per-communicator-basis errors in different teams might go unnoticed. To counteract this, the application needs to periodically send out heartbeats to ensure that there are no failures in the world communicator.

6.2.1 Heartbeats

To be able to respond to failures every process needs to be able to recognize that a process has crashed. During normal operation there are no messages exchanged between processes of different teams. This will lead to the problem that a failure in a team will lead to the team starting the recovery process but other teams will simply proceed with normal operation. This is fixed by exchanging heartbeats. TeaMPI needs to guarantee that the only place where a failure affects an otherwise healthy team is when calling the heartbeat function. It also needs to be guaranteed that all processes in a team return with a consistent view of the overall health of the MPI application. That means that if one process is notified of a failure in another team, all processes will know about this failure by the end of the heartbeat operation. If this would not be the case deadlocks could arise, due to non uniform failure reporting. For example consider two processes in the same team communicating with blocking operations. If one is notified of the failure it will enter recovery procedures whereas the other one will

Listing 6.1: The heartbeat code used to detected failures in teams.

```c
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {

  int err;
  if(comm == MPI_COMM_SELF){
    int send = TMPI_UNFINISHED;
    int recv = 0;
    /*Set Errorhandler to MPI_ERRORS_RETURN so that
    *an error code is returned in case of error
    */
    PMPI_Comm_set_errhandler(getTeamComm(MPI_COMM_WORLD), MPI_ERRORS_RETURN);
    PMPI_Comm_set_errhandler(getLibComm(), MPI_ERRORS_RETURN);

    //Global operation on comm containing all processes
    err = PMPI_Allreduce(&send, &recv, 1, MPI_INT, MPI_MIN, getLibComm());
    //Check if error has been returned
    int flag = (err == MPI_SUCCESS);
    //Check that all processes in team have the same result
    PMPIX_Comm_agree(getTeamComm(MPI_COMM_WORLD), &flag);
    //If error has been detected enter recovery procedures
    if(!flag){
      (*getRecreateWorldFunction())(false);
    }
    PMPI_Comm_set_errhandler(getTeamComm(MPI_COMM_WORLD), *getTeamErrhandler());
    PMPI_Comm_set_errhandler(getLibComm(), *getTeamErrhandler());

  } else{
    err = PMPI_Allreduce(sendbuf, recvbuf, count,
                         datatype, op, getTeamComm(comm));
  }
  return err;
}
```

wait in the blocking call that will never be answered.
Another task performed by the heartbeat is to synchronize the call to `MPI_Finalize`. It may be possible that one team finishes its work before the other teams are able to finish their calculation. The already finished team still needs to take part in the heartbeats, for that reason the status of all teams is communicated via the heartbeats and only if all teams are finished `MPI_Finalize` is called. At the moment the heartbeat used to detect failures is not integrated

with the already existing heartbeats and works by hijacking a call to `MPI_Allreduce` with `MPI_COMM_SELF` as communicator. In contrast to the already existing heartbeats which are only exchanged between the corresponding replicated ranks, the fault tolerant heartbeats are a global operation on all processes. This makes the heartbeat costly and calling it to often should be avoided. On the other hand if the heartbeats are not executed frequently the response to failures might be delayed.

6.2.2 Fault Tolerance with Cold Spares in teaMPI

The main idea of cold spares is to use the dynamic process creation features added in version 2.0 of the MPI standard to spawn new processes that can replace failed ones. The changes that needed to be done to teaMPI in order to implement cold spares are the following: First the teaMPI initialization routine that is called when a process calls `MPI_Init` needed to be changed so that newly spawned processes are treated differently from the original ones. Second, a MPI error handler that starts the necessary steps to recover from the failure on surviving processes needs to be written. And third a function that recreates the communicators and restores normal program functionality needs to be implemented.

Changes to TeaMPI-Initialization

When a program calls `MPI_Init`, `initialiseTMPI` is called to initialize teaMPI. During normal operation, `initialiseTMPI` is the part of teaMPI where the MPI ranks are assigned to their corresponding team-communicators. Additional communicators that are used internally and for heartbeats between ranks are also created in this function.
Newly spawned processes need to be treated differently from the ones started during the normal startup. In order to differentiate between spawned and normal processes, `MPI_Comm_get_parent` can be used. It will return `MPI_COMM_NULL` for original processes; for newly spawned processes it will return the parent intercommunicator that can be used to communicate with the original processes. A newly spawned process will then call the `recreate_world` function in order to rebuild the communicators.

The Error Handler

In MPI error handlers can be set on a communicator. Those error handlers are called when a MPI-operation on that communicator fails. By default MPI implementations are required to provide two such error handlers `MPI_ERRORS_ARE_FATAL` which will abort the job in case of a failure and `MPI_ERRORS_RETURN` which will return the corresponding error code to the user. It is also possible to implement user-specific error handlers.
If a failure occurs in fault tolerant teaMPI, the first thing that will happen is that the processes directly affected by the failure will enter a user-specific error handler. The main purpose of this error handler is failure reporting, so that processes that are not directly communicating with the failed process or processes will also be notified of said failure. This is done by having the error handler revoke the team communicator, the teaMPI internal world communicator

TMPI_COMM_WORLD, the teaMPI internal library communicator and the inter team communicator used for heartbeats. Should a process, unaware of the failure, use one of these communicators the operation will fail with a revoked communicator error. That means it will also enter the error handler. After those four communicators have been revoked the recreate_world function will be called. This is illustrated in listing 6.2.

Listing 6.2: The error handler when using cold spares

```
void respawn_proc_errh(MPI_Comm *pcomm, int *perr, ...)
{
    int err = *perr;
    MPI_Comm comm = *pcomm;
    int eclass, rank_team, team;

    PMPI_Error_class(err, &eclass);

    //Make sure error is due to process failure, not missuse of MPI calls etc
    if (MPIX_ERR_PROC_FAILED != eclass && MPIX_ERR_REVOKED != eclass)
    {
        PMPI_Abort(comm, err);
    }

    //Revoke all communicators
    PMPIX_Comm_revoke(getWorldComm());
    PMPIX_Comm_revoke(getLibComm());
    PMPIX_Comm_revoke(getTeamComm(MPI_COMM_WORLD));
    PMPIX_Comm_revoke(getTeamInterComm());

    //Enter recreate world function
    respawn_proc_recreate_world(false);
}
```

Rebuilding the Communicators

The core part of the fault tolerance code is the recreate_world function, which performs two main tasks: Respawning as many processes as needed and integrating those new processes into the communication structure of teaMPI. It can be divided into three pieces: code that is executed by surviving processes, code that is executed by newly spawned processes and code that is executed by both.
The first operation surviving processes need to carry out is to create a new communicator without failed processes using MPI_Comm_shrink. Using this new communicator and MPI group operations, it is possible to identify which processes have failed and how many failures have occurred. Next as many processes as have failed need to be spawned using

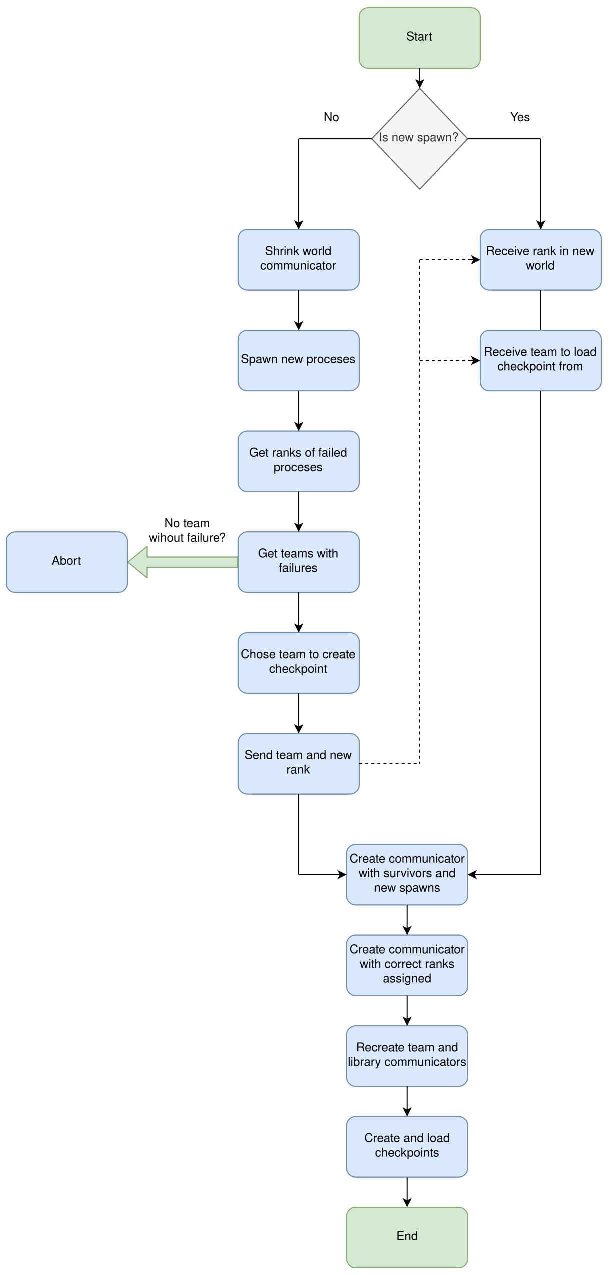

Start
Is new spawn?
No
Yes
Shrink world communicator
Spawn new proceses
Get ranks of failed proceses
Get teams with failures
No team wihout failure?
Abort
Chose team to create checkpoint
Send team and new rank
Receive rank in new world
Receive team to load checkpoint from
Create communicator with survivors and new spawns
Create communicator with correct ranks assigned
Recreate team and library communicators
Create and load checkpoints
End

Figure 6.1: Flowchart of the `recreate_world` function when using cold spares.

`MPI_Comm_spawn`. The failed teams are also calculated and it is checked if a team exists that does not contain failures. If no such team exists, application execution is aborted. Next the ranks of the failed processes in the old world communicator are calculated using `MPI_Group_translate_ranks` and are sent to the newly spawned processes. The number of the first team that did not suffer from any failures will also be sent.
A newly spawned process will just receive said rank, as well as the team number.

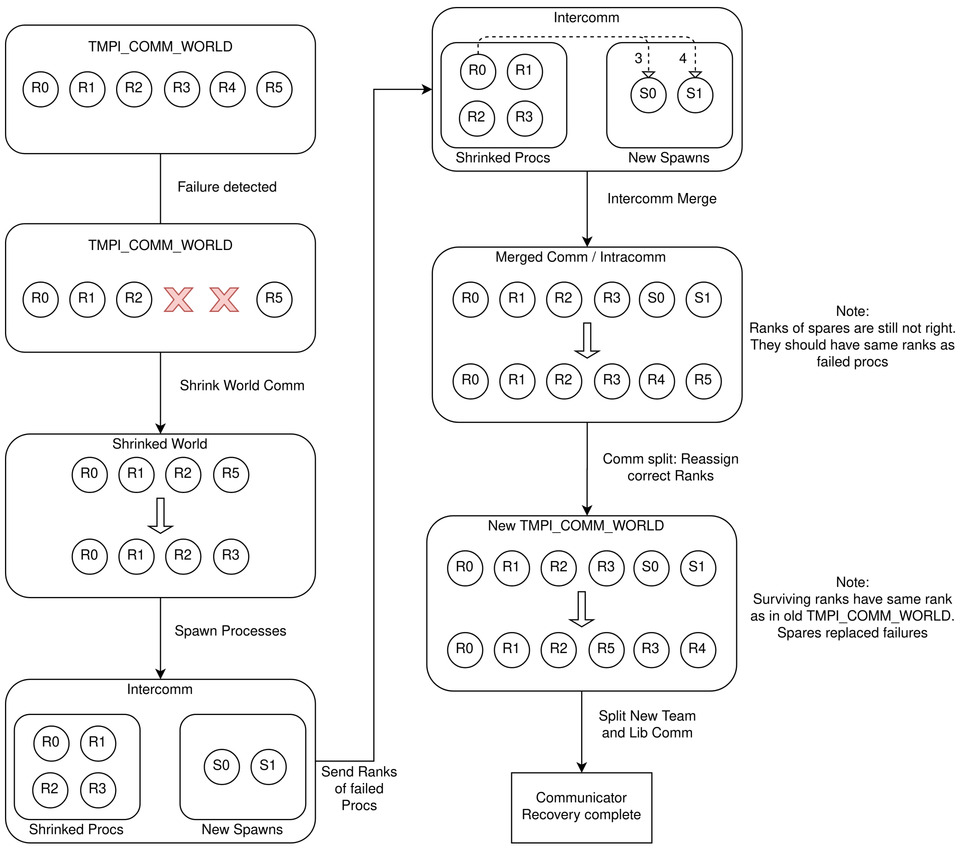


Figure 6.2: Flowchart of used communicators during a call to `recreate_world` when using cold spares

Next, a new communicator that contains surviving processes as well as the newly spawned ones is constructed by creating a new intracommunicator from the intercommunicator that is created when spawning new processes using `MPI_Intercomm_merge`. Due to the way how MPI orders processes when using this operation, the processes will not have the right ranks. To fix this, another new communicator is created using `MPI_Comm_split`. The key that is used to assign the rank in this new communicator is the rank in the old world communicator for surviving processes, and the rank that has been received for newly spawned ranks. This new communicator will then become the new `TMPI_COMM_WORLD`. A new team and library communicator will also be split from this communicator, just like it would happen during `initialiseTMPI`. As a last step, a team that did not contain failed processes will call the create-checkpoint callback, and failed teams will call the load-checkpoint callback that have been set by the application. The generell functionality of this process is shown in figure 6.1 and the used communicators are shown in more detail in figure 6.2.

6.2.3 Fault Tolerance with Warm Spares in teaMPI

The main concept of fault tolerance with warm spares is the same as when using cold spares: replace failed processes with new ones. But whereas with cold spares new processes are spawned, warm spares are created during normal program startup. They do not perform any meaningful computation but wait until they are needed to replace failed ones.
To tell teaMPI how many of the initial MPI ranks should be set aside as spares, a new environment variable, `SPARES`, has been introduced. If it is not set, a default value of zero spares will be used.

Changes to TeaMPI-Initialization

The initialization differs from normal teaMPI in that way, that the spare processes need to be assigned to their own team. All ranks which have a rank that is equal or higher than `total_number_ranks - SPARES` will be used as spare rank, the rest as normal ranks. All spare ranks are placed into their own team, which is the team with the highest team number. After the teaMPI-initialization has been finished, all spares will wait until a failure occurs, and not participate in normal program execution.

The Error Handler

When using warm spares, the error handler works exactly as the one used in conjunction with cold spares. It revokes all communicators and enters the `recreate_world` function that rebuilds the communicators after a failure has occurred.

Rebuilding the Communicators

Just like when using cold spares, the core fault tolerance functionality is provided by the `recreate_world` function. Its purpose is to create new communicators where failed ranks have been replaced by spares.
After a process has entered this function, the first step is to shrink the world communicator, in order to create a new one without failed processes. Using MPI group operations, the failed ranks are identified and the teams they belonged to are calculated. A team that does not contain failures will be selected, so that it can create the checkpoint later on. The types of failed processes, meaning if the process was a spare or a normal process, are also identified. If there are not enough spares available to replace all failed normal ranks, the program execution is aborted. Next a new communicator is created using `MPI_Comm_split`. In this new communicator all failed normal processes will be replaced by spares and all surviving processes will retain their original rank. Based on this communicator, a new team and a new library communicator are created as well. All teams that contained failed processes will now call the load checkpoint callback and the team designated to create the checkpoint will call the create checkpoint callback. This is shown in figure 6.3.

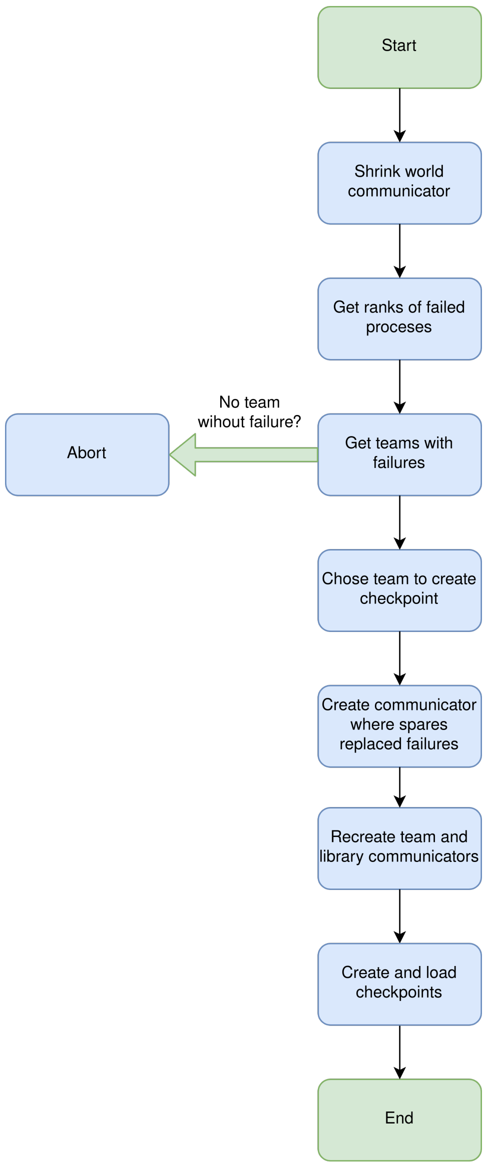

Start
Shrink world communicator
Get ranks of failed proceses
Get teams with failures
No team wihout failure?
Abort
Chose team to create checkpoint
Create communicator where spares replaced failures
Recreate team and library communicators
Create and load checkpoints
End

Figure 6.3: Flowchart of the `recreate_world` function when using warm spares.

6.3 Integrating SWE and Fault Tolerant teaMPI

In order to integrate SWE with teaMPI essentially four things need to be done.

1. Setting the desired teaMPI error-handling strategy as described above
2. Creating a callback that is able to create a checkpoint
3. Creating a callback that is able to recover from a checkpoint
4. Periodically checking for failures using heartbeats

Listing 6.3: Extra steps necessary to initialize fault tolerant teaMPI

```
#ifdef TEAMPI
  //! TeaMPI team number
  int l_teamNumber;
  //Paramter: vector of failed teams
  std::function<void(std::vector<int>)> create(createCheckpointDisk);
  //Parameter: number of team used to load checkpoint
  std::function<void(int)> load(loadCheckpointDisk);
  TMPI_SetCreateCheckpointCallback(&create);
  TMPI_SetLoadCheckpointCallback(&load);
  TMPI_SetErrorHandlingStrategy(TMPI_WarmSpareErrorHandler);
#endif
```

The checkpoint can be loaded using the features described in 6.1 or via MPI.
When loading the checkpoint, a problem that will arise is that spares or newly spawned processes will be in a different state of process execution than surviving processes. This can be solved by having all processes restart execution from the same place. To achieve this `setjmp` and `longjmp` can be used. These two functions allow the implementation of control flow between function calls that deviates from the usual control flow consisting of calls and returns. With `setjmp` the calling environment can be saved in a buffer and `longjmp` can restore the saved data from the buffer. Using `setjmp` and `longjmp`, it is essentially possible to jump from called functions to calling functions. In the case of SWE those functions are used to jump to the part of the main function where `MPI_Init` is called. The heartbeats are carried out at the same place where a checkpoint would have been written in the checkpoint/restart version. In my experiments a heartbeat interval of 5 seconds was used.

Listing 6.4: Use of longjmp to restart program execution

```
void loadCheckpoint(int reloadTeam){
    //Load Checkpoint from disk or receive data via MPI...

    //Jump to predefined location, setjmp will return 1
    longjmp(jumpBuffer, 1);
}

int main( int argc, char** argv ) {

    //...

    /*setjmp returns 0 if called directly. When using longjump
    * to jump to saved states setjmp will return the value defines in
    * the call to longjmp
    */
    if(setjmp(jumpBuffer) == 0){
        /*When called after loading a checkpoint MPI_Init is not called
        * again as calling MPI_Init multiple times is illegal
        */
        if ( MPI_Init(&argc,&argv) != MPI_SUCCESS ) {
            std::cerr << "MPI_Init failed." << std::endl;
        }
    }

    //...

}
```

7 Results

7.1 Theoretical Model

A problem encountered in the practical part of the evaluation is, that the memory footprint and therefore the time needed to save a checkpoint of SWE is not particularly huge. For example in SWE writing a checkpoint with 16 processes on 4 nodes only takes a fraction of a second. Other applications might need tens of minutes to checkpoint their entire state. Fault tolerance by replication is targeted more towards larger scale systems, like upcoming exascale systems. This is the reason why the benefits gained by combining SWE and teaMPI are not as large as could be expected, as can be seen in section 7.2.4. A theoretical model that shows how reactive team recovery with teaMPI might behave in face of shorter MTBFs and higher checkpointing costs will be presented, and used to compare the performance of checkpointing against reactive team recovery with teaMPI.

7.1.1 Analytical Model

The analytical model assumes that the program has to do a certain amount of work before it is finished. This is work is represented as cpu time of meaningful work that needs to be done, this total work will be denoted as w_t. During program execution failures will strike exactly every time the mean time between failure t_{mtbf} has passed. To counteract the failure the checkpoint based implementation writes checkpoints that take t_{cp} of time units to write every time a checkpoint interval of t_i time units has passed. The total run-time of the program can than be represented as a sequence of intervals of length t_{mtbf}. During each of these intervals a certain amount of work is done, whereas the rest of this interval is wasted due to checkpoints being written or work being lost due to it not being saved in a checkpoint. The meaningful progress during one such interval is denoted as w_{prog}.

$$w_{prog} = k_{mtbf} * t_i \tag{7.1}$$

With k_{mtbf} being the number of not interrupted checkpoints that have been written until a failure strikes. For each uninterrupted checkpoint t_i time units of work have been done.

$$k_{mtbf} = \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor \tag{7.2}$$

The number of completely finished MTBF intervals needed to complete the given amount of work is then given by k_{prog}:

$$k_{prog} = \left\lfloor \frac{w_t}{w_{prog}} \right\rfloor = \left\lfloor \frac{w_t}{t_i * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor} \right\rfloor \tag{7.3}$$

The biggest part of the overall execution time T is therefore given by:

$$k_{prog} * t_{mtbf} = \left\lfloor \frac{w_t}{t_i * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor} \right\rfloor * t_{mtbf} \tag{7.4}$$

This equation does not account for the work that might be done in the final interval where no failure occurs. This final interval takes t_{final} time to complete.

$$t_{final} = w_{rem} + ncp(w_{rem}) * t_{cp} \tag{7.5}$$

where w_{rem} is the remaining work that has not been finished yet and $ncp(w_{rem})$ the number of checkpoints written while doing the remaining work. It holds that:

$$w_{rem} = w_t - k_{prog} * w_{prog} = w_t - \left\lfloor \frac{w_t}{t_i * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor} \right\rfloor * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor * t_i \tag{7.6}$$

The number of checkpoints written during that time is given by:

$$\mathrm{ncp}(w_{rem}) = \left\lfloor \frac{w_{rem}}{t_i} \right\rfloor = \left\lfloor \frac{w_t - \left\lfloor \frac{w_t}{t_i * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor} \right\rfloor * \left\lfloor \frac{t_{mtbf}}{t_i + t_{cp}} \right\rfloor * t_i}{t_i} \right\rfloor \tag{7.7}$$

The total time T needed for w_t work units with a MTBF of t_{mtbf} checkpointing costs of t_{cp} and a checkpoint interval of t_i is therefore:

$$T = k_{prog} * t_{mtbf} + w_{rem} + \mathrm{ncp}(w_{rem}) * t_{cp} \tag{7.8}$$

For teaMPI the formula is easier. In the approach teaMPI suffers from an overhead o that is caused by the fact that with replication more resources are needed. The overhead is multiplicated with the work time w_t. It is assumed that teaMPI works until a failure occurs after an interval of t_{mtbf} after which a checkpoint needs to be written which takes t_{cp} time units. Just like with checkpointing a correction needs to be added that considers the final interval where no failure occurs. The total execution with teaMPI is therefore:

$$\left\lfloor \frac{o * w_t}{t_{mtbf}} \right\rfloor (t_{mtbf} + t_{cp}) + ((o * w_t) \bmod t_{mtbf}) \tag{7.9}$$

It needs to be said that this approach is not ideal. As failures always strike at the same time it becomes trivial to guess a correct checkpoint interval, which is able to take exactly one checkpoint right before the failure strikes. This would allow checkpoint restart to perform more efficiently than teaMPI, as it provides the benefit of taking a checkpoint only in case of a failure but without any overhead. Another problem is that very bad combinations of MTBFs and checkpointing costs will let the formula for checkpoint diverge to infinity as failures always strike before a checkpoint is written. Also restarting the work after a failure does not have cost with checkpointing and teaMPI and in the model it is not possible for a failure to strike during the teaMPI recovery.

7.1.2 Simulation

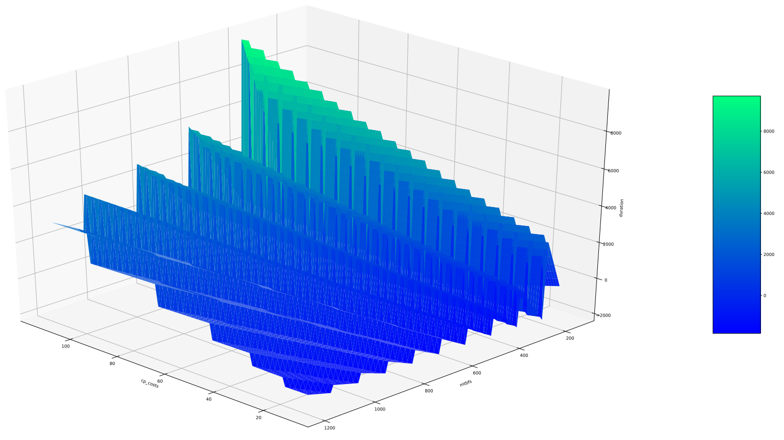
In order to evaluate the performance under a more realistic scenario, a simulation has been written. This simulation still works under the assumptions mentioned above, which includes the fact that there is no cost for restart and failures can not strike during teaMPI recovery but failures are exponentially distributed. This means that for checkpointing Young's formula mentioned in section 2.1 can be used to estimate the ideal checkpoint interval.
In the case of checkpointing the simulation works by keeping track of the total elapsed time T and the amount of work that is left to be done, w_l, which in the beginning is the overall work that needs to be done w_t. The algorithm works iteratively: First an interval of random but exponentially distributed length is generated. This can be done by generating a random value r between 0.0 and 1.0, an exponentially distributed random time interval I can than be obtained with:

$$I = \frac{-\ln r}{\lambda} \tag{7.10}$$

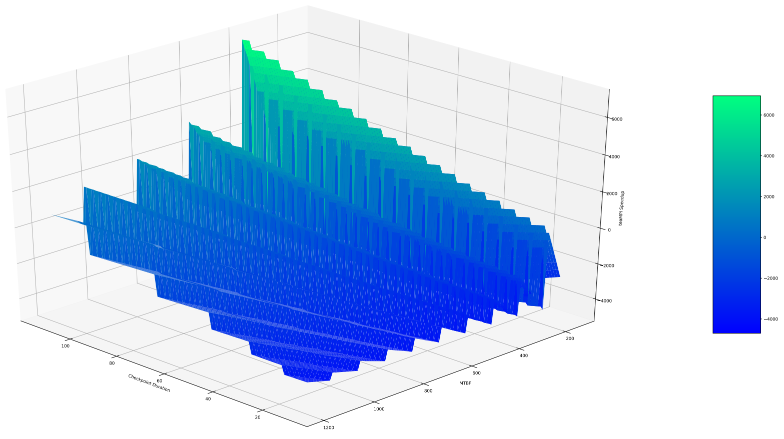
where λ is $\frac{1}{MTBF}$. Next it is checked how many sequences of work and complete checkpoints can be written in I. The sum of work done, that is saved by a checkpoint is then subtracted from w_l and I is added to the total elapsed time T. This processes is repeated until w_l is zero.
For teaMPI the simulation works almost the same. Just like in the aforementioned approach a overhead o is assumed for teaMPI which simply manifests itself as linear factor for the overall work w_t. Just like in the checkpointing simulation a exponentially distributed interval I is generated. The simulation then assumes that work is done until the failure strikes. After the failure occurs a checkpoint needs to be written that means the checkpoint costs needs to be added to T. After that the processes can start from the beginning and needs to be repeated until all work is done.

7.1.3 Results

The results have been calculated using a total work time of 4320 minutes. In case of the formula a constant checkpoint interval of 200 minutes has been used and for the simulation the checkpoints interval is has been calculated using Young's formula. The teaMPI overhead was set to 1.5 and 2.0.

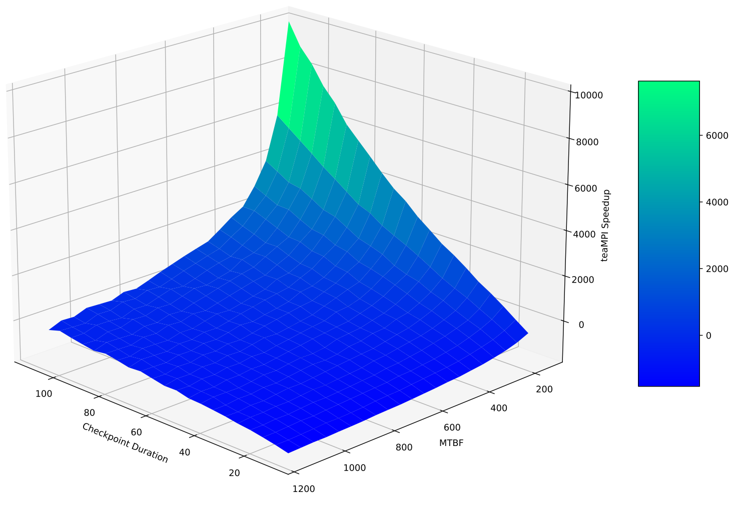


(a) Overhead of 1.5

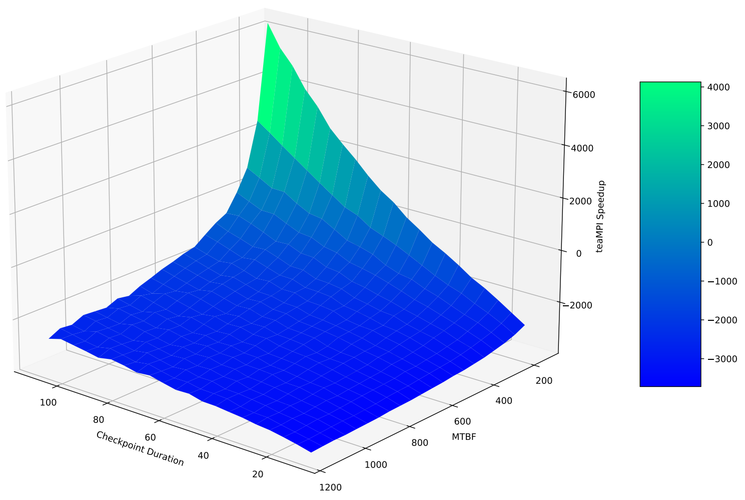


(b) Overhead of 2.0

Figure 7.1: Difference in execution time between teaMPI and checkpointing calculated using the formula from 7.1.1. TeaMPI is faster for positive values.



(a) Overhead of 1.5



(b) Overhead of 2.0

Figure 7.2: Difference in execution time between teaMPI and checkpointing calculated using the simulation from 7.1.2. TeaMPI is faster for positive values.

As can be seen in figures 7.1 and 7.2 teaMPI can be substantially faster than checkpoint restart when the MTBF becomes small or checkpointing costs become large. It needs to be noted that for the formula, steps occur in the output. The reason for that is that for some combinations of checkpoint interval and MTBF the amount of work that is lost due to a failure is minimized. This happens due to the assumption that failures always occur deterministically, after the MTBF has passed.
All in all it can be said that under the right circumstances teaMPI is faster than checkpointing even if teaMPI has an overhead of 2.

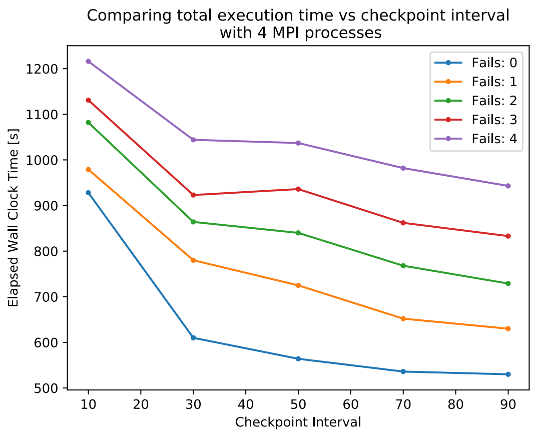
7.2 Performance Benchmark

7.2.1 Testing Environment and Methods

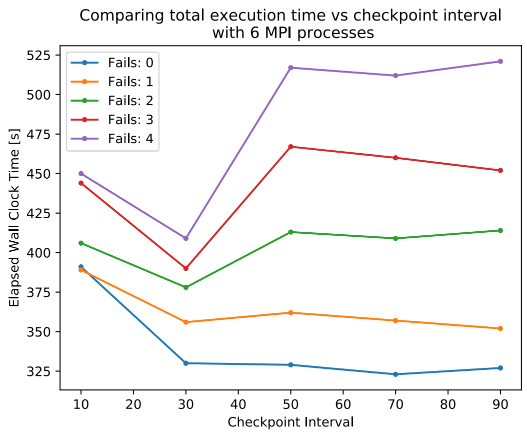
All tests were performed on the CoolMUC-2 Linux Cluster of the LRZ, with ulfm2 version 4.0.2u1 as the used MPI implementation. CoolMUC-2 provides a total of 812 x86 nodes using Intel Xeon processors with 28 cores per node, with the maximum size of a single job being 64 nodes. The code was compiled using gcc and the tests carried out use the SWE `SWE_RadialDamBreakScenario`, which simulates an elevated water cone in the center of the domain, on four nodes with a varying number of MPI processes. Every MPI process was given 7 OpenMP threads, that means that a maximum of 4 MPI processes per node are possible.
In order to be able to compare the performance of different team sizes and checkpointing intervals the input size of the SWE calculations was fixed to 3500 by 3500 grid cells. This size leads to a total execution between 130 and 500 seconds for a failure-free run depending on the number of used MPI processes. To be able to test how the implementation reacts to failures an artificial failure was injected by sending the `SIGKILL` signal to one of the running MPI processes. The failure was injected after waiting for 50 seconds after program startup or the recovery from a previous failure. Checkpoints were written to the SCRATCH file system provided by the linux cluster.
All tests using teaMPI were carried out using two teams. The tests could only be carried out using warm spares as an error in the MPI configuration in the Linux Cluster prevents the dynamic process creation features of MPI from working correctly. When trying to communicate with newly spawned processes ULFM fails with an internal error, stating that it is unable to reach them. The support has already been contacted and similar errors have also been found in the other MPI implementations that are available on the CoolMUC Linux Cluster, as well as the SuperMUC supercomputer.
When using teaMPI 16 MPI processes were allocated, but up to 8 of those processes were used as spares.

7.2.2 Evaluation of Checkpoint Restart in SWE

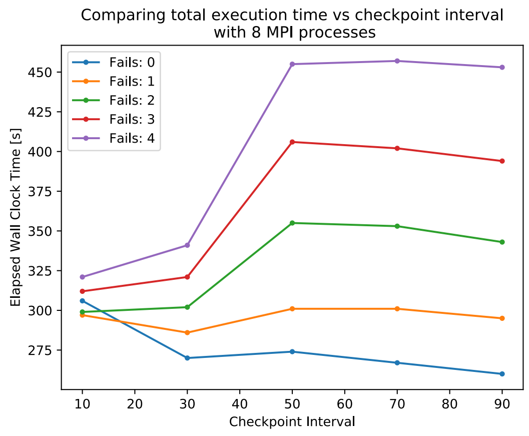
In order to be able to see the performance benefits of the reactive team recovery in teaMPI, the results need to be compared to the classical approach using checkpoint restart. The first



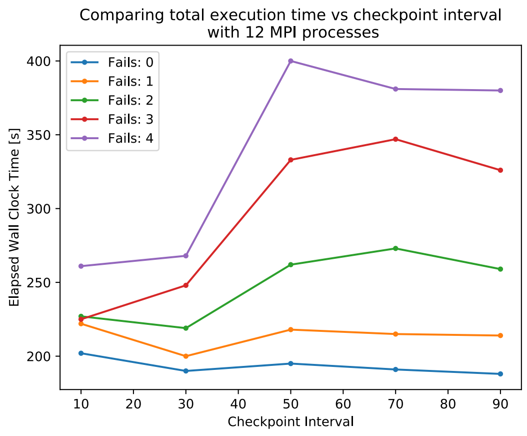
(a) 4 MPI processes



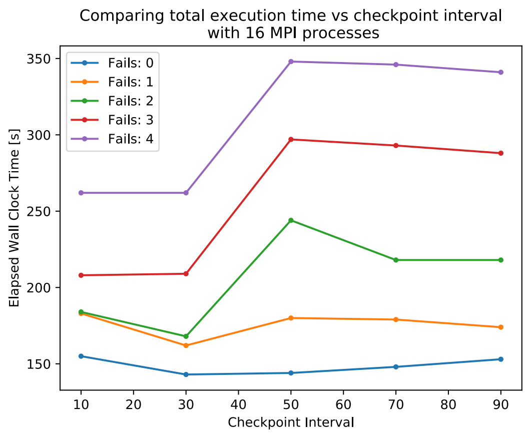
(b) 6 MPI processes



(c) 8 MPI processes

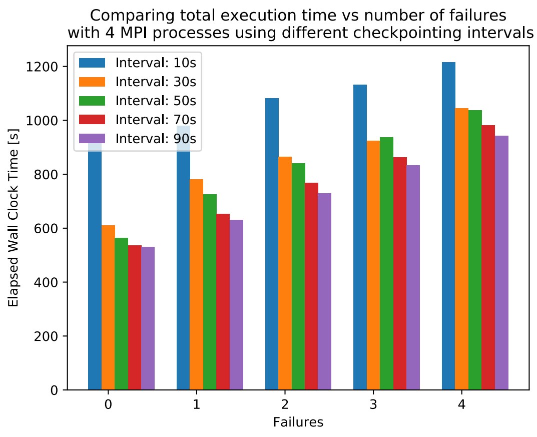


(d) 12 MPI processes

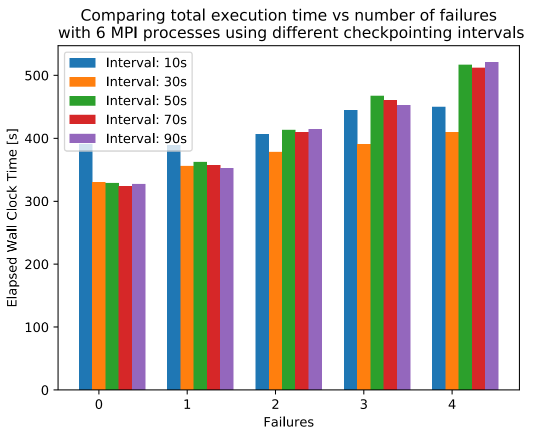


(e) 16 MPI processes

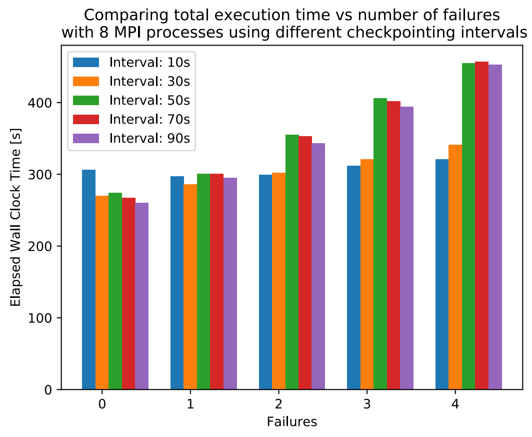
Figure 7.3: Total duration of a simulation run plotted against different checkpoint intervals for a varying number of processes and injected failures



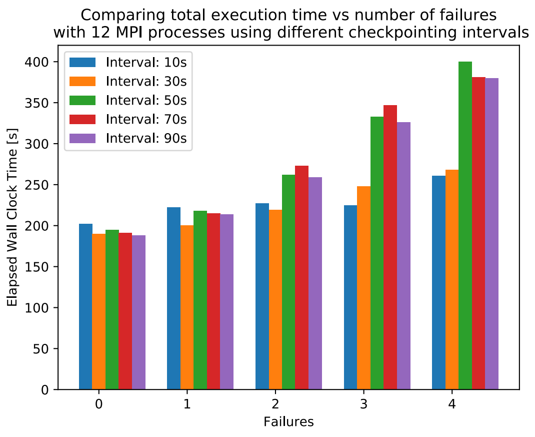
(a) 4 MPI processes



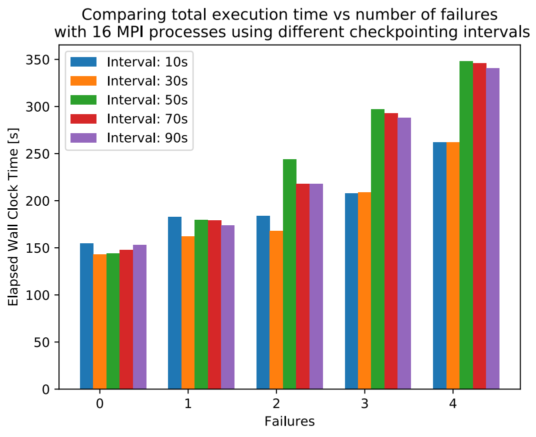
(b) 6 MPI processes



(c) 8 MPI processes

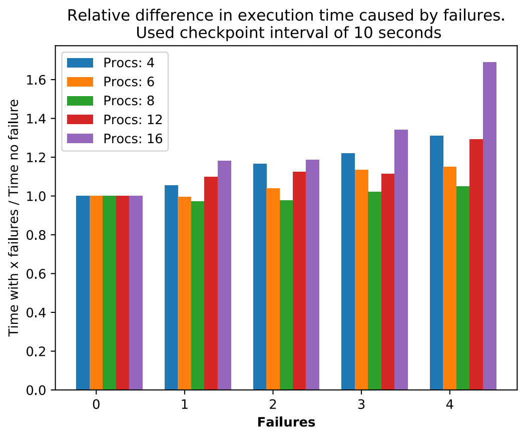


(d) 12 MPI processes

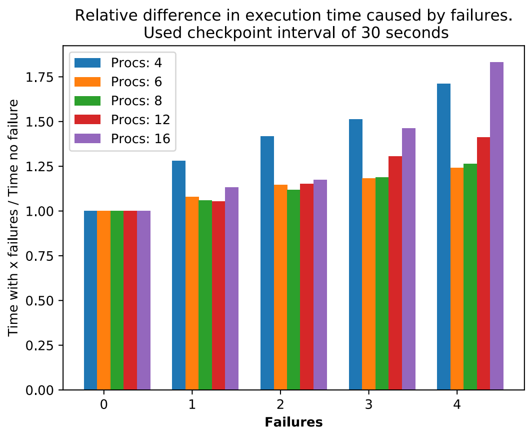


(e) 16 MPI processes

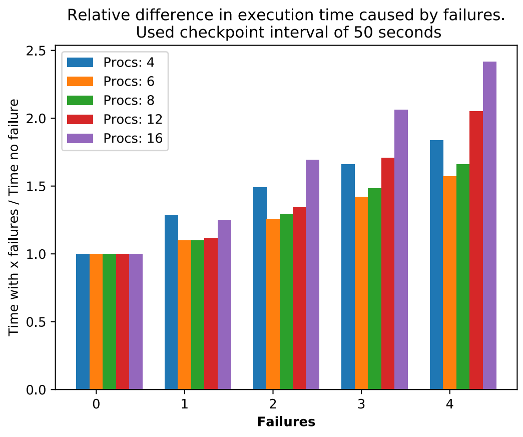
Figure 7.4: Total duration of a simulation run plotted against the number of injected failures for varying number of processes and checkpoint intervals



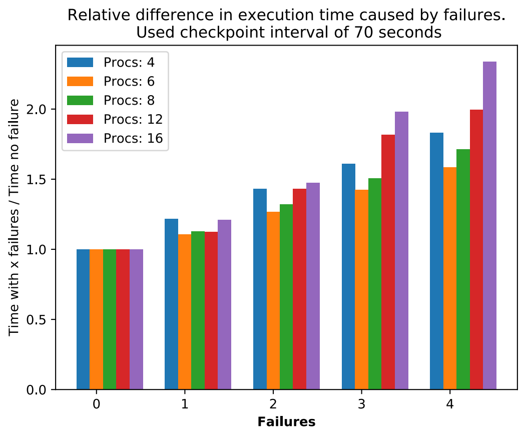
(a) 10 seconds



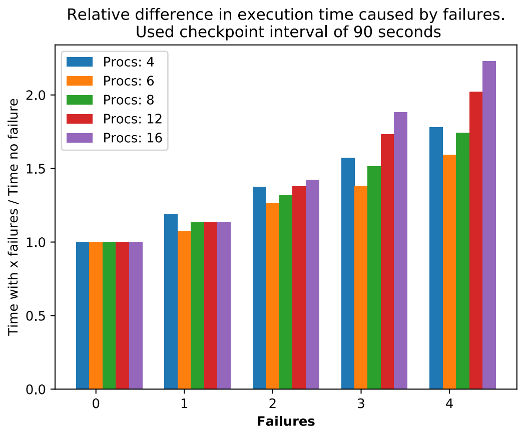
(b) 30 seconds



(c) 50 seconds

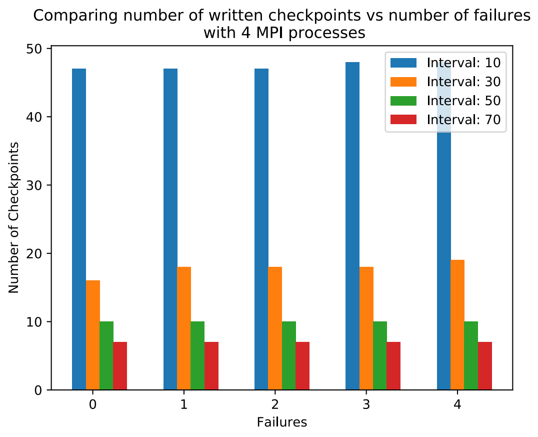


(d) 70 seconds

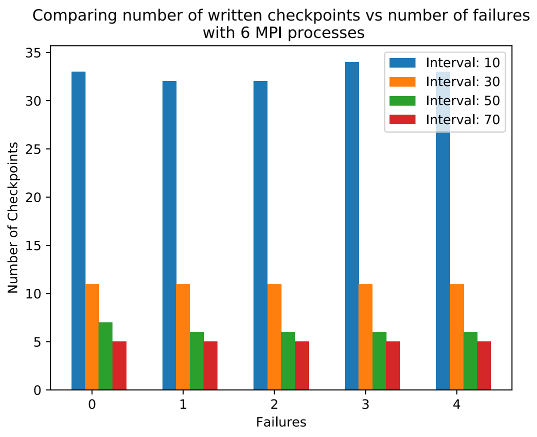


(e) 90 seconds

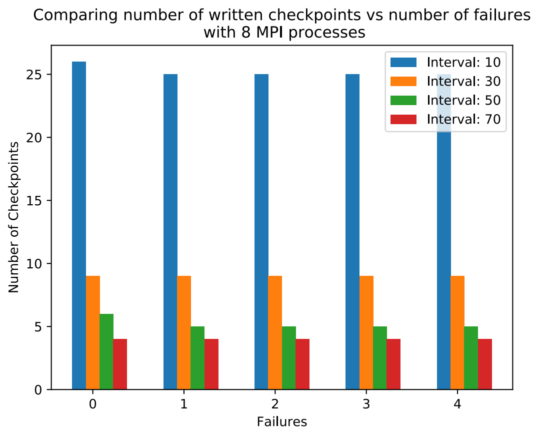
Figure 7.5: Relative difference in execution time between failure and failure free scenario for varying number of failures.



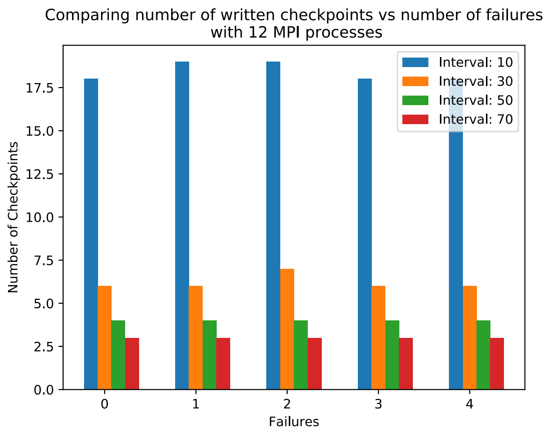
(a) 4 MPI processes



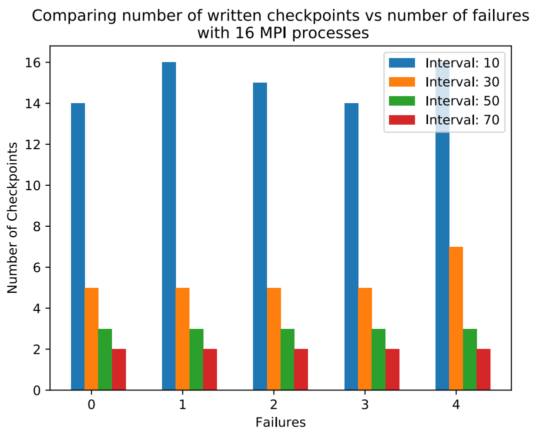
(b) 6 MPI processes



(c) 8 MPI processes



(d) 12 MPI processes



(e) 16 MPI processes

Figure 7.6: Number of checkpoints written plotted against the number of injected failures for varying number of processes and checkpoint intervals

takeaway from the experiments is the influence of the checkpoint interval on the execution for a different number of processes and failures as shown in figure 7.3. When comparing those graphs with each other it is obvious that using 4 processes, as shown in subfigure 7.3a, results in a significant outlier regarding the performance. When using 4 processes access to the file system becomes so slow that even with failures, longer checkpointing intervals are always performing better than writing checkpoints more frequently. When using 6 or more MPI processes most other runs seem to perform best when using a checkpoint interval of 30 seconds. When writing checkpoints more often the overhead of writing to the file system increases drastically and when writing less checkpoints more rework is required in case of a failure. It can also be seen, that as expected injecting more failures leads to a longer elapsed wall clock time. The reason that an interval of 30 seconds is the best performing interval is that failures are injected 50 seconds after starting or restarting the simulation. That means that with a shorter interval unnecessary checkpoints are written and with longer intervals no checkpoint is written before the failure occurs, leading to more wasted work.

The response to the injected failures can be seen in figures 7.4 and 7.5. It can be seen that with the best performing checkpoint interval the overhead caused by the failures ranges between 1.1 and 1.75 depending on the number of used MPI processes. The overhead is calculated by comparing the elapsed wall time without failures to the wall time with failures. Therfore the overhead contains the time needed for restarting as well as writing checkpoints and work that is lost due to the failure. It is interesting to note that the highest relative overhead that occurs is observed when using 16 MPI processes. Comparing that to the absolute duration, using 16 processes still yields the fastest execution time.
As can be seen in figure 7.6, the number of checkpoints written stays relatively constant and does not change much when injecting more failures. The reason for this is that the checkpoint interval is measured in respect to actual being done and not overall elapsed time.

7.2.3 Evaluation with teaMPI redundancy

The experiments with redundant teaMPI have been carried out using a total of 16 processes with either 4, 6 or 8 spares, which means either 12, 10 or 8 processes participating in normal computation. The size of a single team is then 6, 5 or 4 processes. The recovery works as described in section 6.2.3, in case of a failure the failed processes is replaced by a spare and the callbacks set in SWE are called. The surviving team then creates a checkpoint which is loaded by the failed team, and the failed team then resumes operation using the data loaded from the chceckpoint. The absolute run times in a failure free scenario can be seen in figure 7.7. As expected using more spares results in a significant performance overhead, as processes are started that do not take part in the actual computation. The absolute and relative overhead caused by injecting up to four failures can be seen in figure 7.8 and 7.9. When using 6 or 4 spares the overhead caused by failures is very low, but with 8 spares the overhead rises much stronger. The reason for this is, that when using 8 spares the size of one team is 4 processes and as already observed in section 7.2.2 writing checkpoints with few processes is much more costly.

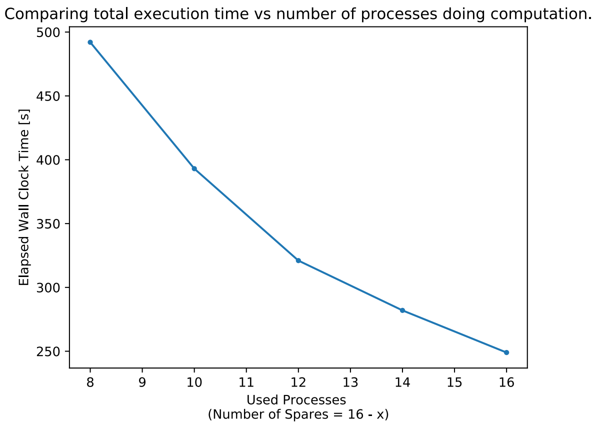


Figure 7.7: Execution time in a failure free scenario with differing number of spares.

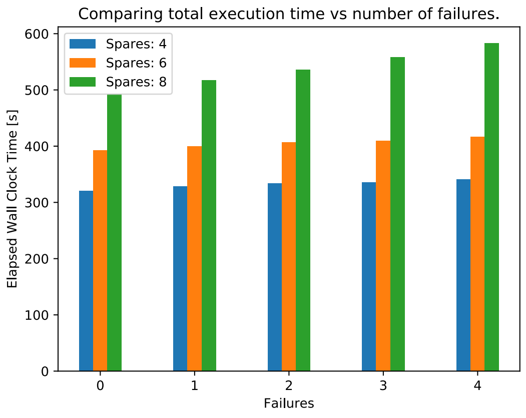


Figure 7.8: Duration of total execution time with failures.

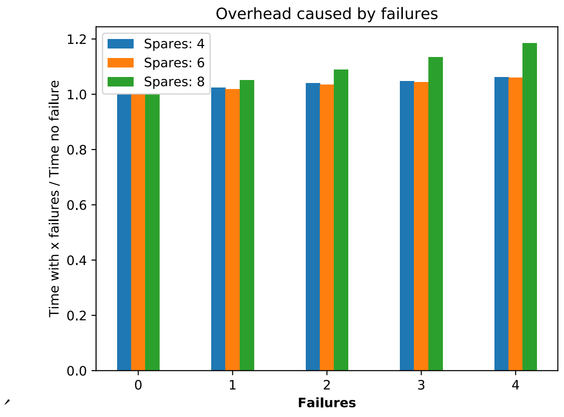
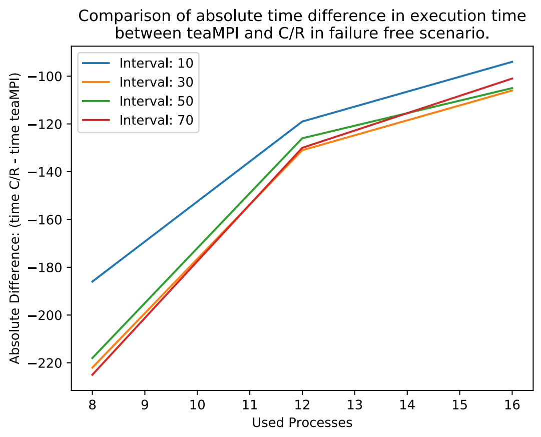


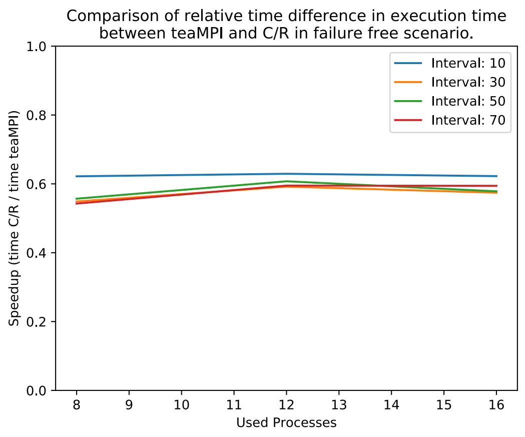
Figure 7.9: Relative overhead caused by failures

7.2.4 Comparison of CR and teaMPI

Before comparing teaMPI and Checkpoint/Restart it is important to note that when using teaMPI the number of processes needed to achieve the same speed as without it, is at least doubled, depending on the number of used teams. When using spares additional processes need to be set aside and can not participate in normal program execution. The doubling of needed resources might be alleviated to a certain extend by using task sharing as described in [3], but this was out of scope for this thesis. All of this means there are two ways to compare the performance of teaMPI and Checkpoint/Restart.
The first comparison that needs to be drawn is to compare the performance when looking at the total number of used processes. For a run with teaMPI this is total number of MPI processes minus the number of warm spares and for C/R it is the total number of used processes. In the experiments performed with teaMPI the number of processes was always set to 16, with a varying number of spares. As can be seen in figures 7.10 and 7.11, in this case C/R always performs faster as long as a good checkpoint interval is used. Only when using 12 processes and a unsuitable checkpoint interval teaMPI outperforms C/R when 3 or 4 failures are injected. The reason for this is that C/R can use double the processes for computation as teaMPI is performing redundant calculations without task sharing.



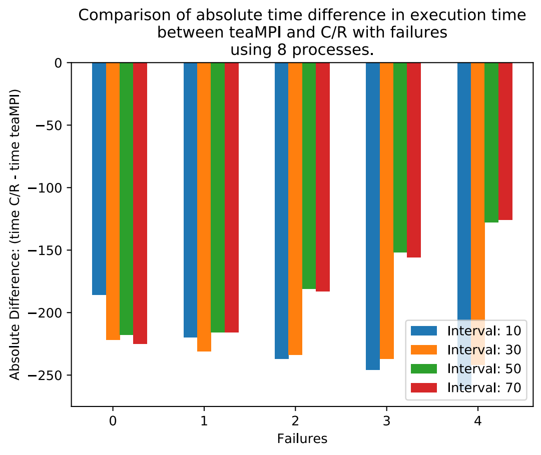
(a) Absolute difference



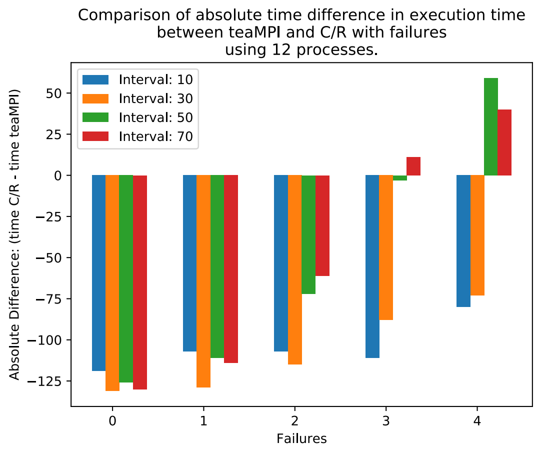
(b) Relative difference

Figure 7.10: Absolute difference in execution time in failure free scenario using different number of processes
Team size is half the number of used processes

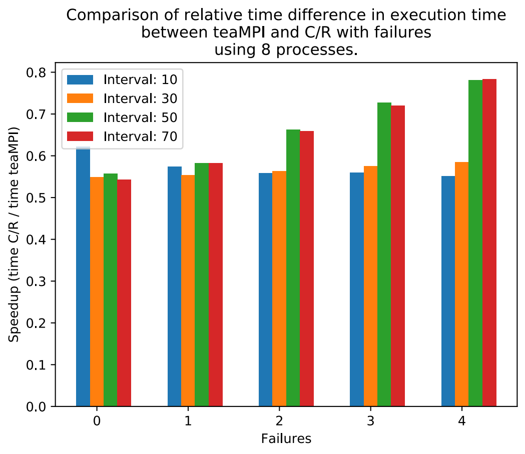
The next comparison that can be drawn is looking at how C/R performs compared to teaMPI when C/R uses exactly as many processes as a single team in teaMPI does. The results of this comparison is shown in figures 7.12 and 7.13. In this case the advantage is clearly on the side of the teaMPI implementation. TeaMPI holds a significant advantage to C/R using only four processes which is due to the fact that checkpointing with this number of processes is much more expensive then with more. Even without failures teaMPI



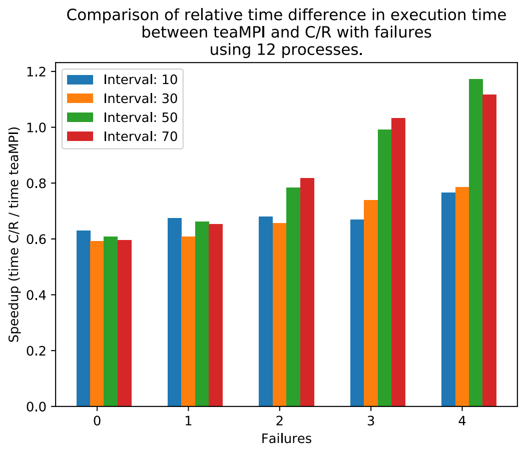
(a) Absolute difference using 8 processes
Team size of 4



(b) Absolute difference using 12 processes
Team size of 6

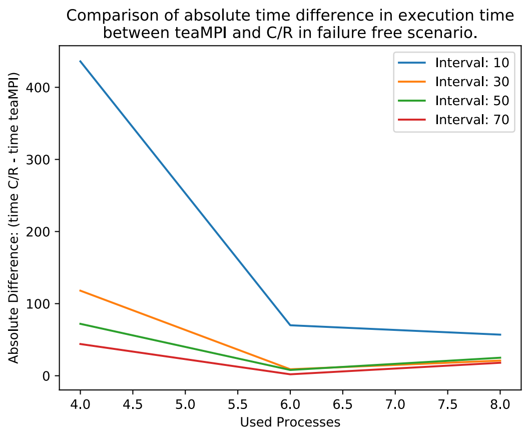


(c) Relative difference using 8 processes
Team size of 4

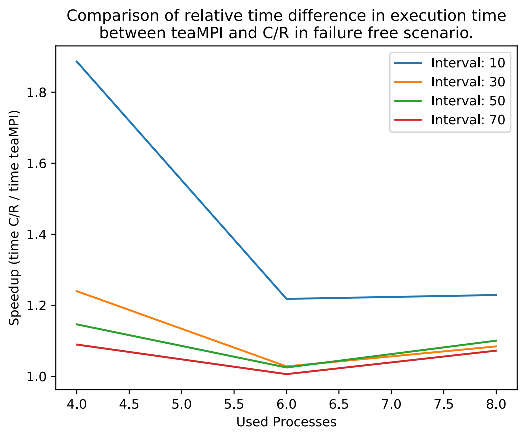


(d) Relative difference using 12 processes
Team size of 6

Figure 7.11: Comparing difference in execution time between teaMPI and C/R. The absolute number of processes excluding spares is used as the baseline for comparison



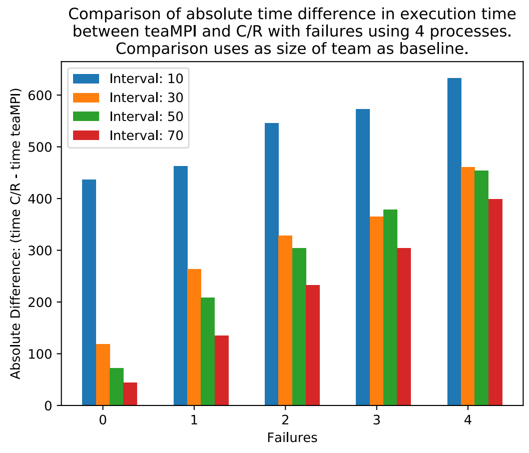
(a) Absolute difference



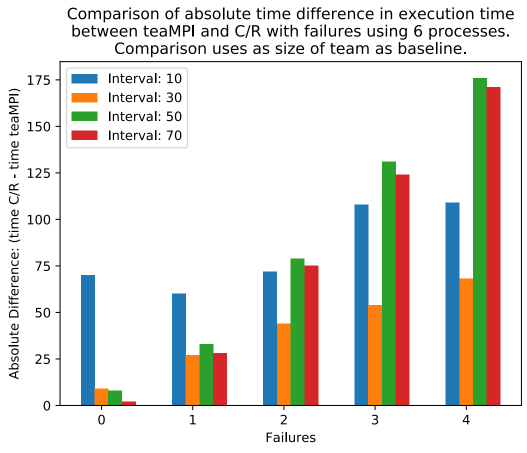
(b) Relative difference

Figure 7.12: Difference in execution time in failure free scenario using different number of processes
Team size is equal to the number of used processes

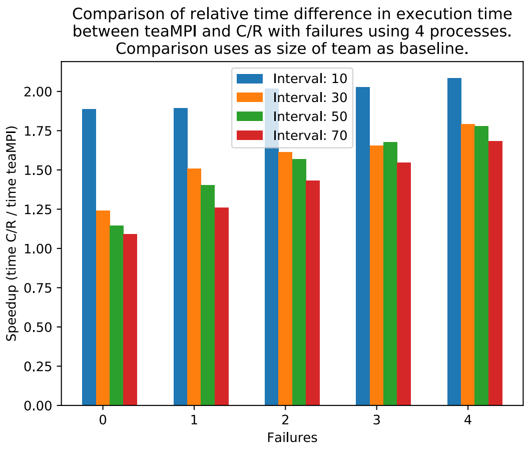
is about 90 seconds faster than C/R. When using 6 processes this effect is not so extreme as checkpointing becomes much cheaper. Still teaMPI is about a minute faster in the four failure case than the best performing checkpoint interval. In a failure free case teaMPI also holds a slight advantage over C/R as no time is wasted writing checkpoints. As with 6 processes checkpointing is not that expensive, and the effect is much less pronounced than when using only four processes.
All in all it can be said that when the performance penalty of using teaMPI can be reduced, which can be done by using task sharing, the fault tolerant teaMPI offers the possibility to create fault tolerant software without checkpoints or writing much less frequent checkpoints.



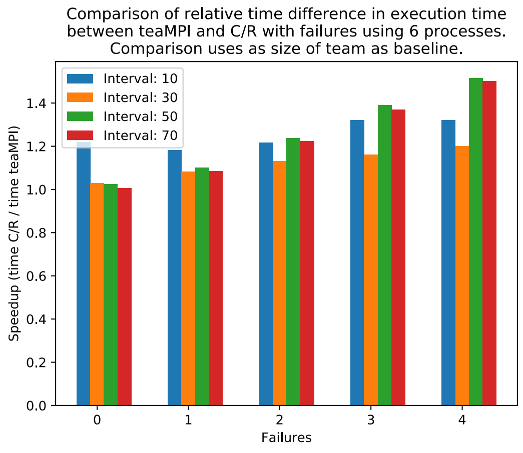
(a) Absolute difference using 4 processes
Team size of 4



(b) Absolute difference using 6 processes
Team size of 6



(c) Relative difference using 4 processes
Team size of 4



(d) Relative difference using 6 processes
Team size of 6

Figure 7.13: Comparing difference in execution time between teaMPI and C/R. The number of processes in a team is used as the baseline for comparison

8 Conclusion and Further Work

The evaluation of teaMPI showed that it might be an efficient tool to combat failures in an HPC environment. In order to be able to conclusively compare the performance of the implementation further work needs to be done. As SWE suffered from the problem that checkpoints were comparatively cheap to write, the overhead suffered by the checkpoint/restart implementation was negligible with more than four processes. Therefore teaMPI needs to be tested on a larger scale with production-level code like ExaHyPE [25]. Software like this will have a larger memory footprint and saving a checkpoint will become much more expensive. As shown in the theoretical model the teaMPI solution becomes more efficient than checkpointing with an increasing cost of writing a checkpoint. Testing on a larger scale with more realistic failure scenarios and MTBFs will also be necessary, as the problem of frequent failures is one that is much more likely to occur in very large scale computations, making use of hundreds or thousands of nodes. As shown when comparing the teaMPI solution against checkpointing, teaMPI holds an advantage when its overhead, caused by the redundant computation is ignored. For that reason fault tolerant teaMPI needs to be integrated with task sharing. This is not trivial as this removes the redundancy between teams that is used to restart in case of a failure, but it could be used to improve the performance and reduce the overhead of teaMPI compared to checkpointing.

Another task that needs to be done is testing the cold spares implementation, which could not yet be carried out due to an error in the MPI setup of the CoolMUC2 Linux-Cluster. Further work that can be done in regards to teaMPI also includes improving the performance of the fault tolerant heartbeats by rewriting them using non-blocking MPI constructs. In addition to the current warm and cold spares error handling strategies a shrink strategy could be implemented as well. Some use cases might profit from such a way of dealing with failures and it would allow to launch teaMPI without wasting processing resources for spares. TeaMPI reduces the number of failures that are fatal for the program execution but larger scale failures, for example ones affecting multiple teams can not easily be handled by teaMPI. Therefore the combination of fault tolerant teaMPI with other fault tolerance techniques such as regular checkpointing or application specific techniques might also be worth a more detailed theoretical analysis.

List of Figures

Bibliography

[1] J. Dongarra, T. Herault, and Y. Robert. “Fault Tolerance Techniques for High-Performance Computing”. In: *Fault Tolerance Techniques for High-Performance Computing*. Ed. by T. Herault and Y. Robert. Springer International Publishing, Apr. 2015. Chap. 1, pp. 3–85.

[2] H. Casanova, F. Vivien, and D. Zaidouni. “Using Replication for Resilience on Exascale Systems”. In: *Fault-Tolerance Techniques for High-Performance Computing*. Ed. by T. Herault and Y. Robert. Springer International Publishing, 2015, pp. 229–278.

[3] P. Samfass, T. Weinzierl, B. Hazelwood, and M. Bader. “TeaMPI - Replication-based Resilience without the (Performance) Pain”. en. In: *ISC High Performance 2020*. Frankfurt, 2020.

[4] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. “Post-failure recovery of MPI communication capability Design and rationale”. In: *International Journal of High Performance Computing Applications* 27 (Aug. 2013), pp. 244–254. DOI: 10.1177/1094342013488238.

[5] A. Gainaru and F. Capello. “Errors and Faults”. In: *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing, Apr. 2015, pp. 89–144.

[6] C. Engelmann, G. Vallee, T. Naughton, and S. Scott. “Proactive Fault Tolerance Using Preemptive Migration”. In: Jan. 2009, pp. 252–257. DOI: 10.1109/PDP.2009.31.

[7] J. W. Young. “A First Order Approximation to the Optimum Checkpoint Interval”. In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531. ISSN: 0001-0782. DOI: 10.1145/361147.361115. URL: https://doi.org/10.1145/361147.361115.

[8] 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: *SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12.

[9] G. Cobb, B. Rountree, H. M. Tufo, and M. Schulz. “MPIEcho: A Framework for Transparent MPI Task Replication ; CU-CS-1082-11”. In: 2011.

[10] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. “Combining Partial Redundancy and Checkpointing for HPC”. In: *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 2012, pp. 615–626.

[11] C. Engelmann and S. Böhm. “Redundant Execution of HPC Applications with MR-MPI”. In: *Parallel and distributed computing and networks* (2011).

[12] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. "Combining Partial Redundancy and Checkpointing for HPC". In: *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 2012, pp. 615–626.

[13] A. Lefray, T. Ropars, and A. Schiper. "Replication for Send-Deterministic MPI HPC Applications". In: *Proceedings of the 3rd Workshop on Fault-Tolerance for HPC at Extreme Scale*. FTXS '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 33–40. ISBN: 9781450319836. DOI: `10.1145/2465813.2465819`. URL: `https://doi.org/10.1145/2465813.2465819`.

[14] T. Ropars, A. Lefray, D. Kim, and A. Schiper. "Efficient Process Replication for MPI Applications: Sharing Work between Replicas". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 645–654.

[15] B. Hazelwood. "Asynchronous Teams and Tasks in a Message Passing Environment". MA thesis. Durham University, 2019. URL: `http://etheses.dur.ac.uk/13019/`.

[16] V. Bode. "Application-Integrated Fault Tolerance in High Performance Computing". Masterarbeit. Technical University of Munich, Nov. 2019.

[17] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 895–906.

[18] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. "MPI-2: Extending the message-passing interface". In: *Euro-Par'96 Parallel Processing*. Ed. by L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert. Springer Berlin Heidelberg, 1996, pp. 128–135.

[19] R. Ashraf, S. Hukerikar, and C. Engelmann. "Shrink or Substitute: Handling Process Failures in HPC Systems Using In-Situ Recovery". In: Mar. 2018, pp. 178–185. DOI: `10.1109/PDP2018.2018.00032`.

[20] A. Breuer and M. Bader. "Teaching Parallel Programming Models on a Shallow-Water Code". In: *2012 11th International Symposium on Parallel and Distributed Computing* (2012), pp. 301–308.

[21] M. Bogusz. "Exploring Modern Runtime Systems for the SWE Framework". Bachelorarbeit. Technical University of Munich, Sept. 2019.

[22] J. M. F. Marqués. *Introduction to the Finite Volumes Method. Application to the Shallow Water Equations.* URL: `caminos.udc.es/info/asignaturas/201/Finite%20Volumes%20IMWE.pdf`.

[23] S. Rettenberger, O. Meister, M. Bader, and A.-A. Gabriel. "ASAGI: A Parallel Server for Adaptive Geoinformation". In: *Proceedings of the Exascale Applications and Software Conference 2016*. EASC '16. Stockholm, Sweden: Association for Computing Machinery, 2016. ISBN: 9781450341226. DOI: `10.1145/2938615.2938618`. URL: `https://doi.org/10.1145/2938615.2938618`.

[24] R. Rew and G. Davis. "NetCDF: an interface for scientific data access". In: *IEEE Computer Graphics and Applications* 10.4 (1990), pp. 76–82.

[25] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems". In: *Computer Physics Communications* 254 (2020), p. 107251.