DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

# Integrating Task Sharing with Team Recovery for Hard Failure Tolerance in teaMPI and SWE

Simon Schuck

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor´s Thesis in Informatics

# Integrating Task Sharing with Team Recovery for Hard Failure Tolerance in teaMPI and SWE

# Integration von task-basierter Aufgabenteilung mit der Wiederherstellung von Teams zur Toleranz von abgestürzten Prozessen in teaMPI und SWE

| | |
|---|---|
| Author: | Simon Schuck |
| Supervisor: | Univ.-Prof. Dr. Michael Bader |
| Advisor: | M.Sc. Philipp Samfass |
| Submission Date: | 15.02.2021 |

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

Munich, 15.02.2021 Simon Schuck

# Acknowledgements

I want to deeply thank my advisor Philipp Samfaß for all his guidance during the past four months. He has been extremely helpful. My thanks also go to my girlfriend who's been a great help with proofreading and cheering me up throughout these stressful times.

# Abstract

Hard failure tolerance becomes ever more important as the scale of high performance computing systems increases and their mean time between failures grows smaller. Checkpoint/Restart has been the conventional way of recovering from hard failures, but becomes increasingly more expensive as the trend of upscaling through parallelization continues. This bachelor's thesis will explore an approach to combine replication with task sharing and reactive checkpoint/restart to create a cheap and performant way of dealing with hard failures. With replication, we avoid losing data in case of a failure. We can use a failed process's replica to create a checkpoint on disk and spawn a new process which loads that checkpoint and replaces the failed one. To counteract the grave performance impact of replication we employ task outcome sharing between the replicas. This results in a resilient yet performant approach, that can even keep up with more conservative proactive checkpoint/restart techniques and provides a promising alternative for future exascale scenarios.

# Contents

# 1 Introduction

With Moore's Law nearing its end and quantum computing technology still in its infancy, the ever growing demand of high performance computing (HPC) power can only be met by way of upscaling through parallelization. Each additional node on a computer cluster represents an additional point of possible failure, thus increasing the overall likelihood of a failure happening at any point in time. This means a reduction in the Mean Time Between Failures (MTBF) of the whole system. This trend is further promoted by hardware growing more fragile as component sizes decrease. The conventional way of mitigating such problems has been the use of checkpoint/restart techniques. As this comes at the cost of many, mostly unnecessary I/O operations, it is not an ideal solution. A different approach is replication, where each node has at least one twin doing the exact same computations. If one process fails, a replica can replace it. While replication has no extra cost in I/O operations, we are effectively cutting our available processing and memory resources at least in half when employing this approach, a major drawback. Also, if all copies of a process should fail, all work would be lost. Both these shortcomings have been solved individually in the past. The first by having the two nodes in a pair communicate with each other and exchange intermediate results so that not all computations are done twice [12]. The second by facilitating a reactive checkpoint/restart approach which creates checkpoints only when a failure actually occurs [7]. This thesis will first look at these two main techniques of dealing with hard failures in HPC and how User Level Failure Mitigation and the teaMPI library give us the tools necessary to combine checkpointing, replication, and task outcome sharing to create a possibly better one. To demonstrate this approach, it will be implemented in a simulation software for two-dimensional shallow water equations called SWE. This will be done in a chronological fashion. Beginning with an explanation of the base application, it will look at the implementation of reactive team recovery and how we integrate task sharing into it. It will then evaluate the results of benchmarks conducted on the Linux cluster of the Leibniz Supercomputing Centre and compare team recovery with task sharing to the previous approaches. Lastly, it will list the successes and shortcomings of our work as well as give an outlook upon future work to be done.

# 2 Background & Related Work

## 2.1 MPI - Message Passing Interface

**Definition** The MPI standard specifies an interface for communication between processes. Development started in 1992 and MPI version 1.0 was released in 1994. Since then, the MPI standard has been under ongoing development with the current version being MPI-3.1 released in 2015 [8]. The latest 2020 draft standard is a release candidate for version 4.0. The typical use case for MPI is Single Program Multiple Data, i.e. running multiple instances of the same application in parallel with each of them processing different data but working towards a common goal. MPI specializes in distributed memory environments, e.g. HPC clusters where multiple systems with their own processors and memory are connected together via high-bandwidth connections. That being said, most MPI implementations can also be used in shared memory environments without any problems which is very useful for rapid prototyping.

**Implementations** Because MPI itself is only a standard it needs to be actually implemented for it to be usable. The first implementation was MPICH developed at the Argonne National Laboratory to support the efforts of the standards process [10]. Today's most well known implementation is Open MPI. It has its roots in three different MPI implementations, which have been merged together to form Open MPI [11].

**Usage** Disclaimer: Because this paper uses the C++ language and a version of Open MPI in its implementation, usage examples and source code will be based on this configuration.

To use Open MPI, one needs to download the source code and build the library. During this process, the compiler wrapper `mpic++` and the `mpiexec` executable are built as well. The user now only needs to include the header file *mpi.h* and compile the program using the compiler wrapper instead of the normal compiler. The compiler wrapper calls the system's compiler with the necessary compilation flags to link the program against the MPI library. To run an MPI program, the users runs `mpiexec` with the compiled application as the argument. They can further specify how many parallel instances of the program should be run. `mpiexec` launches the application and assigns each instance a number, a so-called rank. In code the user can retrieve the rank with the function `MPI_Comm_rank()` and the number of ranks with `MPI_Comm_size()`. This is how each instance might determine, what part of the data it needs to process. The ranks are also used for addressing. They can be used in routines like `MPI_Send()` and

`MPI_Recv()` to specify a target or source for the communication.

## 2.2 Fault Tolerance Techniques

When running a highly parallelized application for long durations, it can be quite disappointing if a single rank experiences a hard failure (i.e. freezes or crashes) and many hours of work are lost. The MPI-3.1 standard itself does not specify fault tolerance, it reads:

> "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."

A lot of research work has been done to provide fault tolerance to MPI applications.

### 2.2.1 Checkpoint/Restart

Checkpoint/Restart is the most conventional and widely used way of limiting the damage of a failing process. During execution, the application periodically outputs checkpoints. These files typically hold the current state of the program, including all information necessary to restart and continue from that point. The creation of checkpoints mostly requires output operations, which can be very slow, depending on the hardware used. The biggest downside of checkpoints is their inherently bad scalability in relation to the MTBF of the system. As the MTBF grows smaller, more checkpoints need to be created for them to be useful. With the current state of technology, the best way of increasing performance is throwing more computation power at the problem, i.e more processors. More machines running means more points of possible failure and a smaller MTBF. To address the issues with checkpointing, researchers try to make them more efficient. At Berkeley lab, a system-level checkpointing mechanism was developed for use on Linux clusters. [6] With access to system-level resources it is possible to measure temperatures and other metrics to predict when a failure might be about to happen. This allows the user to create checkpoints less frequently and rely on the prediction system to create a checkpoint just in time before it's too late. Berkeley's checkpointing implementation was later integrated into LAM/MPI, one of the predecessors to Open MPI. [13] This MPI checkpointing implementation is fully transparent to the application itself. The application code does not have to be modified and as such, involuntary creation of checkpoints is possible as well. This is useful in cases where a computing cluster has to undergo emergency maintenance for some reason. If all MPI applications used this framework, then the system could shut them all down and restore them after the maintenance was over. Another approach at improving checkpoint/restart is SCR (Scalable Checkpoint Restart for MPI) [9]. The focus of this project was to reduce the overhead caused by writing checkpoints. To do so, a multi-level checkpointing system was designed, that incorporates multiple types of checkpoints. Lightweight checkpoints are faster to create but can only handle recovery from more common and less severe faults. Heavier checkpoints, including copying to a parallel file system, take much longer but allow recovery from total system failure. These approaches offer great ways of generalized checkpointing, that work with most applications. The alternative to such an approach is application-specific checkpointing, where the checkpointing code is part of the application. This might allow the user to take shortcuts that are usually not possible with generalized checkpointing like omitting certain data from the checkpoints to make them more lightweight.

### 2.2.2 Replication

Replication means running two or more copies of the whole group of processes in parallel. As long as at least one of the replicated groups survives, nothing is lost, but the price for this is performance. The total available processing power is divided by the number of copies. Using only a single replica already halves the amount of available processing power, which goes against the central principle of high performance computing. As a result of this downside, replication-based resilience mechanisms tend to not get a lot of attention in the HPC community. These preconceived notions are challenged by a study conducted at the Sandia National Laboratories in 2011. They came to the conclusion, that replication can be more efficient than classic checkpoint/restart once the number of ranks is big enough, storage bandwith gets too limited, or MTBFs grow too small. All of these are conditions we are expecting to meet sooner or later in exascale computing. Additionally, replication can help with the detection and correction of soft faults (e.g. bit flips) [4]. Part of this study was *rMPI*, an MPI replication library which "replicates each MPI rank in an application and lets the replicas continue when an original rank fails. To ensure consistent replica state, rMPI implements protocols that ensure identical message ordering between replicas." In the following year, researchers from the North Carolina State University presented RedMPI, which followed similar principles as rMPI, but offered additional protection against silent errors while lowering replication overhead [5]. What these libraries (and others) have in common, is that they keep the replicas very consistent with each other. The additional checking of messages for soft errors adds synchronization and slows down the entire system. To "reduce the pain" of replication-based fault tolerance, a group at the Technical University of Munich took on the task of reducing the inherent major performance penalty of replication. The result of their work is teaMPI, a small MPI wrapper library in C++. At program startup teaMPI automatically divides the ranks into groups of replicas, so-called teams. The amount of teams can be freely chosen via the environment variable `TEAMS`, so both 2-fold and 3-fold replication are possible. The replication is fully transparent to the application itself. Each rank sees `MPI_COMM_WORLD` as a communicator for its own team only and doesn't know about the existence of other teams. TeaMPI achieves this separation by acting as a wrapper around MPI calls. A call to `MPI_Send()` for example will call teaMPI's definition of the function. TeaMPI will then use its knowledge of the teams to send the message to the correct recipient within the sender's team. To do

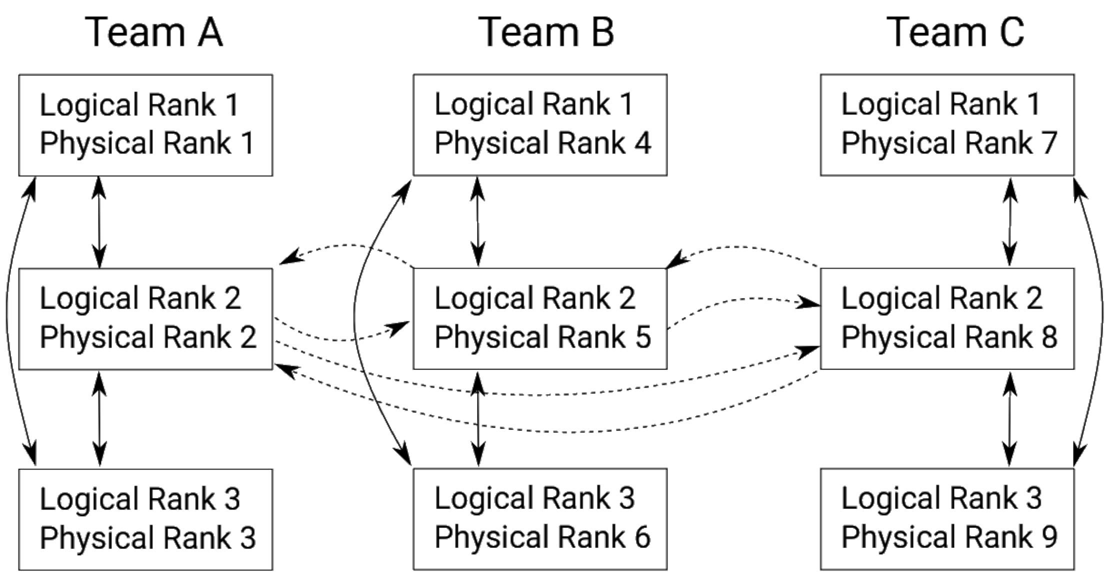

Figure 2.1: Example layout of a teaMPI application with 3 teams and 3 ranks per team. Communication can only occur orthogonally, i.e. within a team or between replicas. Inter-team communication arrows have only been drawn for rank 2 to avoid clutter [12]

so, teaMPI uses MPI's profiling interface. The profiling interface is essentially a copy of MPI's standard interface, but uses the prefix `PMPI_` instead of `MPI_`. In the example above, teaMPI will call `PMPI_Send()` with the correct communicator and rank to reach the intended recipient. TeaMPI's specialty is the full asynchronisity it allows between the teams. As such the replication induces virtually no overhead beyond the default ~50% and ~66% for 2-fold and 3-fold replication respectively. To identify failing ranks, teaMPI utilizes so-called heartbeats. These are non-blocking messages carrying the current wall time. A process sends its heartbeats to its replicas. The heartbeats can be used to identify, when a rank drops out or fails completely. A rank can even identify its own failure, when it starts receiving heartbeats at a faster rate than it sends them. Once a failing rank has been identified, a recovery mechanism could be employed, but this was out of scope for that paper. TeaMPI also allows a process to directly communicate with its replicas, if it chooses to do so. TeaMPI has the necessary interface for processes to inquire about the number of teams and get access to an inter-team communicator which contains a process and its replicas. This made it possible for them to set up task shuffling and result sharing between replicas. One replica would execute one task while another would execute a different task and afterwards they would share the results of their tasks with each other. This system also allowed processes which were delayed, e.g. due to I/O operations, to catch up with their replicas by using the shared results. This allows teaMPI to perform much better than traditional replication, as long as the shared tasks are compute-heavy enough for sharing to be faster than computing [12].

## 2.3 User Level Failure Mitigation

At this point in time, neither the MPI specification nor the current version of Open MPI 4.1 have fault tolerance included. However, Open MPI was and still is a vehicle for research into fault tolerance. Over the time it supported and deprecated several mechanism including the already mentioned checkpoint/restart. Open MPI 5 will include User Level Failure Mitigation (ULFM), an MPI extension specification first released in May of 2012, which identifies failed processes and gives the user the means to act upon the failure in whatever way they desire. ULFM defines new error codes and operations on communicators to deal with failures. The error codes include for example `MPI_PROC_FAILED`, which is returned from MPI operations, which could not complete because a process has died. The communicator operations include for example `MPIX_Comm_shrink()`, which creates a new communicator from an old communicator, but with all the failed processes removed. With tools like these at their disposal, users can deal with failures by employing different fault mitigation techniques. [1]

### 2.3.1 Error Handling in teaMPI

Later in 2020, Alexander Hölzl added an error handling mechanism to teaMPI. To accomplish this, teaMPI was built against ULFM. When a failure is detected either through heartbeats or due to an MPI error, teaMPI will use the tools provided by ULFM to handle the failure. The user can choose the error handling strategy, that teaMPI will follow. The default strategy `KillTeam` stops the execution of a team if one of its ranks fails. The remaining teams shrink down the inter-team communicators and remove the failed team's ranks from them. Then they can resume normal operation. The alternative to this is to replace the failed process. Two more error handling strategies are based upon this concept. `RespawnProc` uses `PMPI_Comm_spawn()` to launch a new replica, which will take the place of the failed one. `WarmSpare` instead uses spare processes that have been launched together with the all normal processes. TeaMPI uses the environment variable `SPARES` to determine how many ranks to set aside for this purpose. These warm spares will be interrupted during MPI/teaMPI initialization to wait for a failure. To integrate the spares with the running processes, the user can register callback functions for writing and loading checkpoints with teaMPI. These checkpoints will only be created after a failure occurs and not during normal execution. Once a failure happens, teaMPI will first remove the failed processes from the world communicator. If the `RespawnProc` strategy is used, the new processes will now be spawned. In the meantime, teaMPI determines the rank numbers of the failed processes and the team which they were a part of. If no teams without failed processes exist at this point, teaMPI will abort execution as recovery is no longer possible in this state. If there is still at least one healthy team, teaMPI will determine, which of them will create the checkpoint. Now the spares will be assigned their new ranks and teams and teaMPI will tell them, which team they will get the checkpoint from. To rebuild all intra-team and inter-team communicators, teaMPI will first create one big communicator with all survivors and new spawns and create the necessary communicators from there. Lastly, the checkpoint creation and loading callbacks are called and the whole application is back in a working state. [7]

# 3 Integrating Task Sharing with Team Recovery

## 3.1 SWE

### 3.1.1 Introduction to SWE

SWE is a simulation software, which implements a finite volume model to solve two-dimensional shallow water equations. [3] Its main use is simulating the propagation of tsunamis and other waves in a given domain. The code (C++) is written to be education-oriented and the components are modular to allow for easy editing of parts of the program without affecting the rest. This in turn makes it much easier to experiment and try out concepts. For example there currently are eleven different Riemann problem solvers, all with different properties and capabilities.The code itself is written with education in mind. For students, SWE acts as a entry point to the world of scientific applications. It can be built with MPI support, thus also teaching the basics of parallel programming on HPC clusters.

### 3.1.2 Basic SWE walk-through

This section describes the inner workings of SWE without additional features like checkpoint/restart or task sharing.

SWE uses scenarios for its input data. A given scenario defines starting properties of the scenario domain such as:

- Location and type of scenario boundaries
- Bathymetry at any given point in the scenario domain
- Water height at any given point in the scenario domain
- Water velocity at any given point in the scenario domain

Scenarios can either be artificial and hardcoded or use measured real world data (e.g. from the 2011 Tohoku tsunami). Data files use the NetCDF format and follow the CF 1.5 conventions.

When SWE is run, the user defines properties of the simulation domain such as:

- Size of the simulation grid as number of cells in horizontal and vertical directions

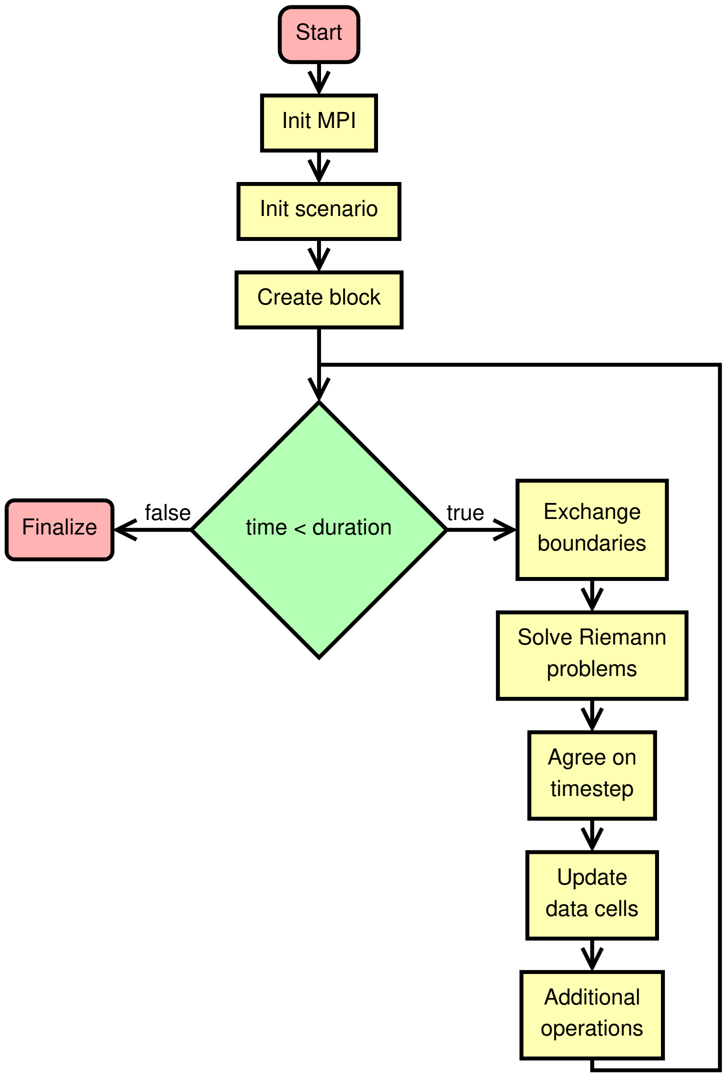

Figure 3.1: Typical program flow of basic SWE

- Duration in simulated time

In a typical MPI usage scenario, every rank of SWE will hold a part of the simulation domain, a so-called block. The simulation domain is divided between these blocks without overlap. Blocks can be of different sizes if the cells cannot be divided evenly between them. At startup, SWE will first initialize its MPI environment. With the amount of ranks known, each one can determine which part of the simulation domain it will hold. They will then get the data from the scenario and initialize the blocks to their starting states. Additional components (e.g. an output writer) are initialized as well. At this point, the simulation can begin. At the beginning of each timestep, neighbouring blocks must first exchange their border cells with each other. Next, all ranks will solve their local Riemann problems and store the results. Before the water heights and velocities can be updated, the ranks must agree on how large the current timestep will be (smallest calculated value across all cells). After all blocks are updated, the timestep is done. If wanted, this is the point at which an output snapshot of the simulation domain can be written. Once the user specified maximum simulated time has passed inside the simulation, all ranks will finish and shut down.

## 3.2 Team Recovery in SWE

Team recovery is a combination of replication and checkpoint/restart, made possible by teaMPI. When a process fails in one team, the application is able to create a checkpoint from the non-failed replicas of another team and restart the failed team's processes from those checkpoints. To achieve this, certain additions had to be made to SWE and teaMPI by A. Hölzl in his bachelor's thesis in 2020. [7]

### 3.2.1 Checkpoint/Restart in SWE

The output writer module, that is already part of SWE was extended by an interface that writes out a checkpoint. This writes the simulation data at the current point in time to the output file, then makes a copy of the output file, which acts as a checkpoint. While it would also be possible to only save the current timestep and not the whole output file as a checkpoint, we would then risk the failed process corrupting his output file and not repairing it during recovery. It should also be possible to use some form of checksum or hash to check whether the output file has been damaged in the crash and, based on that, create only a single timestep or full output checkpoint, but this is out of scope of this paper. Additionally, a text file containing metadata is created. This is necessary, because some data like the behaviour of the simulation boundaries is not saved in normal output files. A command line parameter was added, with which the user can specify a time interval in real time, after which a new checkpoint is created.
For restarting, a new command line parameter was added, with which the user can specify a checkpoint file. During initialization, SWE will read the checkpoint file and

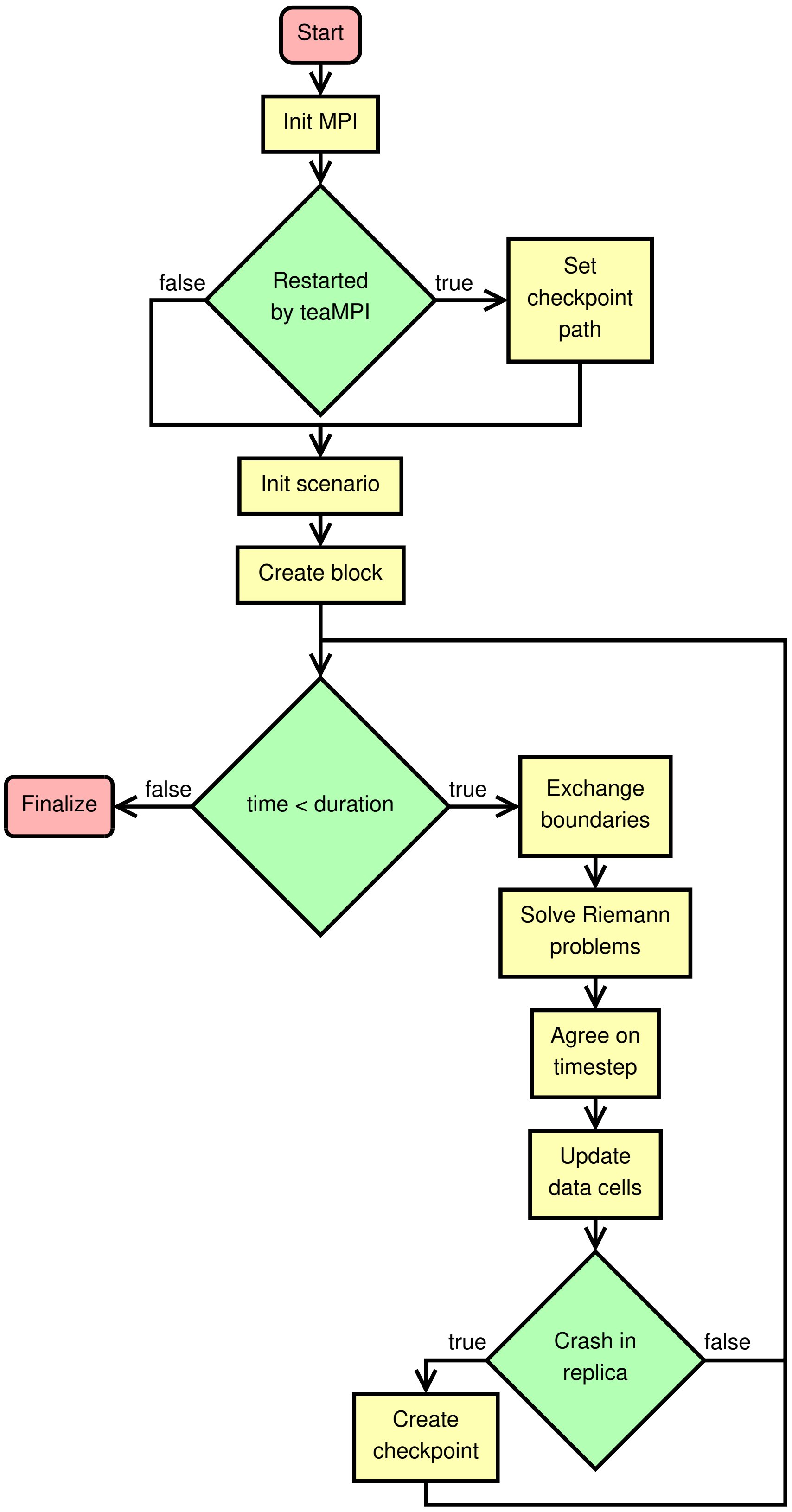

Figure 3.2: Typical program flow of SWE with team recovery

corresponding metadata file and create a new scenario from that data instead of using the default one. SWE then continues like normal, only the starting data is different.

### 3.2.2 Integration with teaMPI

After adding error handling to teaMPI as described in subsection 2.3.1, SWE now only needs to link against the teaMPI library and set up the callbacks. The callback for creating checkpoints calls the writer's checkpoint interface, then notifies the process, which the failed one was replaced with, that checkpoint creation has finished. The checkpoint loading callback modifies the variable containing the path to the restart file, thus triggering the creation of the checkpoint scenario instead of the default one. The command line parameter for a checkpoint interval can be ignored now. With these additions, SWE now has the capability to create checkpoints exactly when they are needed. This advantage comes with the major drawback of needing at least double the amount of processes to be run. Hölzl's test results showed, that as long as checkpoint/restart and team recovery get the same amount of processing power, checkpoint/restart only needs between 55 and 80 percent of the execution time compared to team recovery. These tests assumed, that the user chose a good checkpoint interval for the checkpoint/restart method. In extreme cases, where the number of failures was high and the checkpoint interval was not small enough, checkpoint/restart took up to 20 percent more time to complete than team recovery.

## 3.3 Task Sharing

### 3.3.1 Concept

First, we define tasks as a combination of data (e.g. an array of values) and a function that is applied to that data (e.g. calculating the sum of all values in the array). Tasks must additionally fulfill the following requirements:

1. Each task must be assignable to a single process. This makes shuffling the task order and sharing the results a much easier problem to solve.

2. Tasks must not depend on the results of other tasks. The output of one task cannot be used as the input of another task. This is required for the shuffling the task order.

3. The number of tasks available to each instance of the program should be greater than or equal to the amount of teams/replications. If there are less tasks than teams, it is impossible to shuffle the tasks, so that every team gets a different first task that it can share with the others.

With these requirements in place, the following theoretical task sharing system becomes possible: Let there be two teams named A and B consisting of one process each. The

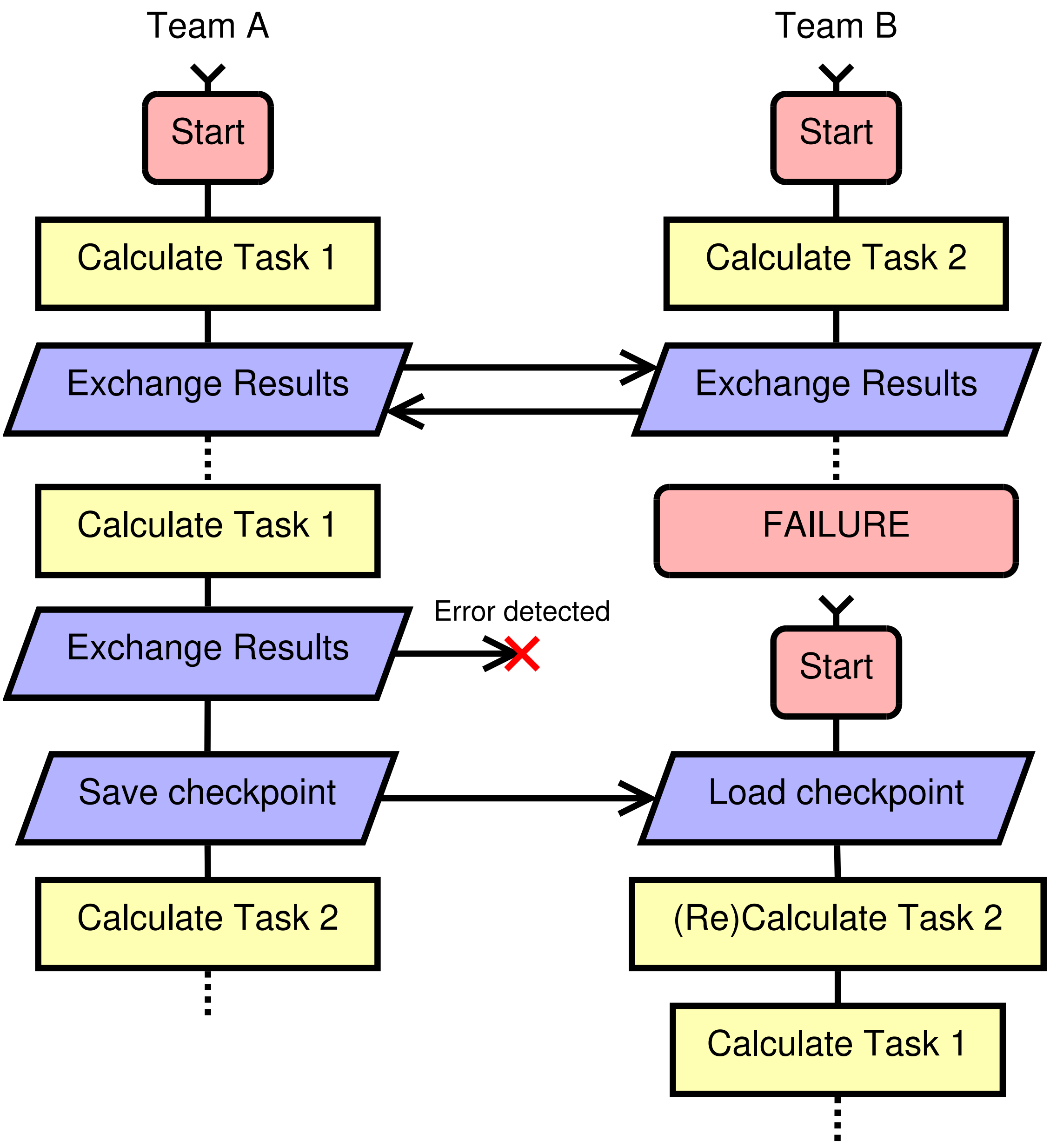

Figure 3.3: Example program flow of 2 replica processes with task sharing and team recovery

processes are replicas of each other. For any given program iteration, both processes have two tasks named 1 and 2 to complete. The process of team A can always choose to do task 1 first, while the process of team B can always choose task 2. As long as the partner process is still alive at this point, they can now send their own results to their partner process and in turn receive the partner's results. In case of a failure in one of the processes, the other can just complete the missing task by itself. In this case, the failed process has to be replaced and load from a checkpoint which the healthy process can create. Depending on the implementation, the healthy process could send the results to the recovered process or the recovered process takes the time to calculate them again. At this point, both teams have the results for both tasks and can advance to the next program iteration. Ideally, all tasks will only need to be completed once. In theory, a system like this should result in performance similar to or even better than the checkpoint/restart method, while being as highly scalable as the team recovery method.

### 3.3.2 Implementation in SWE

As a starting point for integrating task sharing into SWE, we will use the SWE fork of A. Hölzl, which contains the previously mentioned additions for team recovery. Solving all the Riemann problems of a block will constitute one full task. For this to be possible, each instance of SWE must be able to hold more than just a single block. For two teams with one process each, at least two blocks per rank are necessary for task sharing to be useful. As mentioned previously, SWE is highly modular. There exists a block module, which supports communicating with other instances of itself within a single SWE process and across processes. We need to use this module in our implementation, because of the mechanics for exchanging data between block boundaries. Now that there can be multiple blocks within a single SWE instance, not all boundary exchanges use MPI messaging. Blocks, which are part of the same SWE instance, can directly access each other's memory without the need for more overhead. Some changes need to be made to the original application to integrate this new type of block. Instead of a single output/checkpoint writer per rank, each block gets its own writer and an interface to trigger the writing operations. When a rank writes a checkpoint, it creates one checkpoint data file and one metadata file for each of its blocks. When a process is restarted, these checkpoints becomes the bases for new scenarios, which will provide the data to the blocks of the restarted rank. The total amount of blocks can be controlled in several ways. Each rank gets at least one block. Another block will be added for each replica, e.g. if we divide six ranks into three teams of two ranks, then each of these ranks will hold three blocks. Additionally, a command line parameter can be used to multiply the number of blocks per rank by a positive integer, called the decomposition factor. This is 1 by default. The total number of blocks can be expressed as the following formula:

$$blocksPerRank = numberOfTeams * ranksPerTeam * decompositionFactor$$

Depending on the specific implementation, more smaller tasks can potentially lead to better performance than a less big tasks. [2] After all blocks are initialized, the SWE rank determines the order in which it will work on the blocks in each iteration. Each rank will have a set of primary blocks, which it will always compute by itself first, then attempt to share the results with the ranks in the other teams. The number of primary blocks is equal to the decomposition factor. All other blocks are considered secondary blocks. The rank will first attempt to receive the results of a secondary block from the team, for which it is a primary block. If the rank, that we try to receive from fails before or during the communication, the `MPI_Waitall` operation will return with an error. In such a case, the rank computes the block by itself. This also triggers teaMPI's error handler and the currently active error handling strategy (i.e. in our case WarmSpare) will be executed. An iteration in SWE now runs as follows. Like in the base

```
std::vector<int> myBlockOrder{};
// Primary blocks
for (int i{myTeamNumber}; i < blocksPerRank; i += numberOfTeams)
{
    myBlockOrder.push_back(i);
}
// Secondary blocks
for (int i{0}; i < blocksPerRank; i++)
{
    if (i % numberOfTeams != myTeamNumber)
    {
        myBlockOrder.push_back(i);
    }
}
```

Figure 3.4: Code to determine in which order to compute blocks

version, the first step is to exchange boundary information between blocks. For Inter-rank boundaries this is accomplished via MPI messaging, whereas blocks at intra-rank boundaries lie in the same process's memory and can access the needed data directly. Before beginning computation, `MPI_Barrier` is called on the inter team communicator to ensure, that no data from a previous iteration still waits to be sent(Figure 3.5). Otherwise the computation in the current iteration could modify the data before it is sent. Each rank then iterates through its blocks as determined by its block order. For the primary blocks, the rank will immediately solve the Riemann problems in the block, then use non-blocking send operations to share its results with its replicas in the other teams. The send requests are immediately freed, because the `MPI_Barrier` guarantees, that overwriting the send buffers is not a problem. Then, for the secondary blocks, the process can use the number of the block to determine which team it will receive the results from and post according non-blocking receive operations. It will then wait on

```
// Code for task completion and sharing of results
// Barrier to avoid overwriting send buffer
MPI_Barrier(interTeamComm);
for (int i{0}; i < blocksPerRank; i++) {
  auto& currentBlock = *simulationBlocks[myBlockOrder[i]];
  if (i < decompFactor) {
    // Compute primary block and send out results
    currentBlock.computeNumericalFluxes();
    for (int destTeam{0}; destTeam < numberOfTeams; destTeam++) {
      // Do not send data to myself
      if (destTeam != myTeamNumber) {
        // Send all relevant data fields of the current block
        // to destTeam over interTeamComm and free the requests
        MPI_Isend(...);
        MPI_Request_free(...);
        MPI_Isend(...);
        MPI_Request_free(...);
        MPI_Isend(...);
        MPI_Request_free(...);
        [...]
      }
    }
  } else {
    // Determine source team from block number and post receive
    // requests for all relevant data fields of the block
    int sourceTeam{currentBlockNumber % numberOfTeams};
    std::vector<MPI_Request> requests(9, MPI_REQUEST_NULL);
    MPI_Irecv(...);
    MPI_Irecv(...);
    MPI_Irecv(...);
    [...]
  // Complete the receives
  int error{MPI_Waitall(9, requests.data(), MPI_STATUSES_IGNORE)};
  if (error != MPI_SUCCESS) {
    // Error detected, complete the task by myself
    currentBlock.computeNumericalFluxes();
  }
}
```

Figure 3.5: Code for task completion and sharing of results

all of these requests using `MPI_Waitall`. If no error is returned by `MPI_Waitall`, the transaction was a success and the execution can move on to the next block. If an error was returned, the block must first be computed by this process. After all blocks have been handled, all teams will have the same data across all their processes. Now as before, the processes within each team can agree on a timestep and update the cells accordingly. Since each time has the same data and uses the same algorithms, they will choose the same timestep and stay in sync data-wise. This marks the end of the iteration. At this point, output files can be written or other additional operations can be performed.

# 4 Performance Benchmarks

## 4.1 Test Setup

### 4.1.1 Hardware

All tests were conducted on the CoolMUC-2 Linux cluster at the Leinbniz Supercomputing Centre. The cluster runs 812 Haswell-based nodes, each of which run 28 cores at a nominal frequency of 2.6 GHz. Each node has access to 64 GB of RAM. At its peak, the system can output 1400 TeraFLOPS. All test runs were done on 2 nodes, 8 cores per node for a total of 16 processes. Disclaimer: Because teaMPI's cold spares are still not fully implemented, we need to allocate extra cores for warm spares when testing with failures. We've chosen to not count these cores as they do not contribute any extra performance until a failure happens, at which point the core of the failed process no longer contributes and the total amount of contributing cores stays the same throughout the test.

### 4.1.2 Software

The compiler used was the Intel C++ Compiler version 19.0.5.281. The compiler wrapper used is from User Level Failure Mitigation 2 version 4.0.2u1. Input/Output of NetCDF data was handled by the NetCDF library version 4.7.

### 4.1.3 Scenario

All test cases used the artificial `RadialBathymetryDamBreak` scenario. The scenario domain is a 2 by 2 kilometre square. This scenario simulates a 25 metres tall, circular dam breaking in the lower left quadrant of the simulation domain. In the center of the domain lies a circular 10 metres tall elevation at the bottom of the virtual ocean, which pushes up the water above it. Water spreads out from the dam and from the elevated area. The simulation area was chosen to be 3000 by 3000 cells, which results in an area per cell of around 0.44 square metres. The simulation duration was set to 30 seconds.

### 4.1.4 Results

**Baseline** We first start with a baseline case. This is to give us a theoretical maximum performance we can hope to achieve with the chosen parameters. The baseline case uses none of the fault tolerance mechanics introduced in this paper. It's a single team of

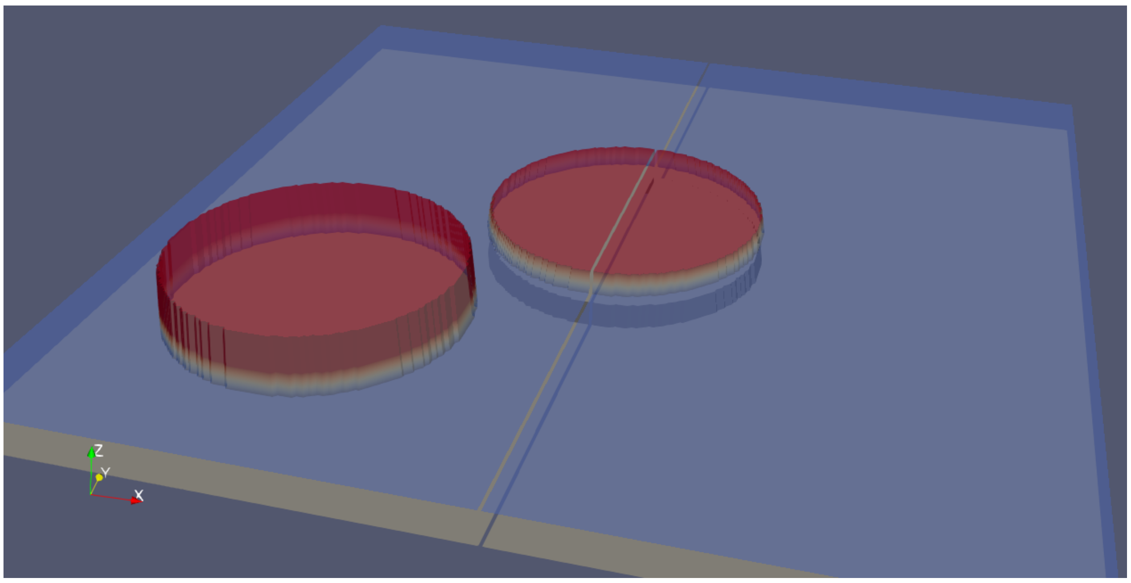

Figure 4.1: Visual representation of the RadialBathymetryDamBreak scenario made in ParaView

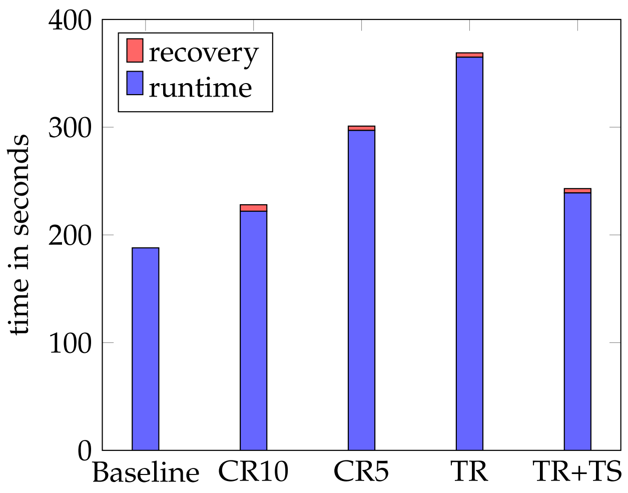

Figure 4.2: Runtime comparison of different fault tolerance configurations, red sections show time needed to recover from a failure

16 processes without replication running the simulation as fast as it can. The baseline result shows us, that the fastest possible execution time is 188 seconds.

**Checkpoint/Restart** Next we have the cases for proactive checkpoint/restart. The runtime here is highly dependent on the chosen checkpoint interval. A lower interval means more checkpoints have to be written, which takes more time. The time cost of writing each checkpoint is also highly dependent on the application at hand. The bulk of SWE's checkpoint data is made up of four floating point number arrays. In most cases these aren't very big unless extremely high grid sizes are chosen. Compare this to other simulations with potentially millions of objects, each with some amount of associated data. A checkpoint for such a simulation could be many orders of magnitude larger than one in SWE. This means, that the following numbers should be taken with a grain of salt, as they are very dependent on the kind of application at hand. The run with a checkpoint interval of 10 seconds took 222 seconds to complete, which is just 18% above the baseline. The time lost to a failure recovery can wildly differ. A failure could happen just after or just before writing a checkpoint. In the former case, one only pays the price of having to restart SWE which doesn't take more than a second. In the latter case, you lose the whole duration of a checkpoint interval, in this case 10 seconds. For these cases we have assumed, that a failure has the same chance of occurring at any point in the program. As such the time lost to recover from a failure is on average 5 seconds plus 1 more second for restarting SWE. When the checkpoint interval is lowered to 5 seconds, one can expect roughly double the amount of checkpoints to be written with a corresponding increase to the total runtime. This run took 297 seconds, about 58% longer than the baseline. As a positive contrast, lowering the checkpoint interval also has the effect of lowering the maximum possible time one can lose to a failure. In this case the maximum possible time loss to recovery from failure is 6 seconds. For our data, we again choose the expected average of 3.5 seconds additional recovery time for a run with a failure.

**Team Recovery** Next we have team recovery without task sharing. We divide our 16 processes into two teams of 8 processes each. Both these teams only have half the processing power of the first three cases. As expected, with only half the processing time available to each team, they took almost double the baseline time to complete their runs at 365 seconds, which is 94% above the baseline. This result falls perfectly in line with the results Hölzl got in his tests. He too reported checkpoint/restart with a 10 second interval being 40% faster than team recovery. Recovering from a failure with team recovery took only 4 seconds. In most cases, pure team recovery will lose to checkpoint/restart in terms of runtime. Checkpoints would need to be extremely costly for checkpoint/restart to take more time than team recovery.

**Team Recovery + Task Sharing** When task sharing is added to team recovery, the runtime improves considerably to 239 seconds, which is 27% above the baseline. As task sharing has no influence on the actual recovery part of team recovery, it still costs 4 seconds to recover from a failure. This is a 35% performance improvement compared to team recovery without sharing. As the chart shows, team recovery with task sharing is able to keep up with checkpoint/restart. Which of the two is better, depends on the checkpoint interval used for checkpoint/restart. Considering, that checkpoint intervals will continue to grow smaller as parallelisation, we hypothesize, that team recovery with task sharing will take the lead in most cases, as long as tasks stay compute-heavy enough for sharing to be faster than computing. This needs to be tested in further, larger-scale benchmarks.

# 5 Conclusion & Further Work

Integrating task sharing with team recovery provides an exciting alternative to the classic checkpoint/restart mechanic. Especially considering a future, where parallelisation becomes ever more important, a corresponding decrease in mean time between failures seems very likely. Checkpoint/Restart suffers greatly from a reduction in checkpoint intervals, a disadvantage not shared by team recovery. Even then, team recovery with task sharing and especially the teaMPI library are still in an early stage of developement. Samfaß et al. noted a lack of a mature communication performance model for task sharing as well as an increased risk of data corruption by soft failures (e.g. bit flips) due to task sharing [12]. These examples already show, that there's is still a great amount of further work to be done. For SWE specifically, we noticed the following problems, which haven't been solved at the time of writing:

- When we tested team recovery without task sharing, we noticed that the teams drifted away from each other after a recovery. The teams agreed on different timesteps internally, which leads us to believe, that either the writing or reading of checkpoints is faulty. Multiple blocks per rank each with their own checkpoints did increase the complexity of that system to a degree. Blocks should be deeply analysed before and after writing/reading a checkpoint to hopefully find the source for this faulty behaviour.
- When we first tested team recovery with task sharing we were surprised to see a runtime of 804 seconds. For comparison we also ran the test on a single node with 16 cores and in fact received a time 3 times faster. Our workaround to this was always binding a pair of replicas to the same cluster node.This is achieved by passing the argument `--map-by node` to `mpiexec`. With this setup, we then got the test results we reported in chapter 4. With task sharing, there is much more communication between the teams than within the teams, but this should not result in such a large bottleneck in inter-node communication. This matter should also be investigated further in the future.
- While not exactly a problem at this point, our implementation of task sharing is very rigid and inflexible. In its current form, the number of tasks in each iteration must be exactly the same. This implementation also requires a certain amount of synchronization between the teams, which works against teaMPI's goal of having teams runs as asynchronously as possible. The task sharing implementation of the team around Samfaß in the ExaHyPE engine is of a much more dynamic nature. Instead of expecting a task result to be sent to them at fixed points, the ranks keep

a database of task results that have been sent to them. Before computing a task, they check to see if they already have a corresponding result in the database. While this exact implementation brings with it some overhead and is only suited for compute-heavy tasks, we should still explore a fully asynchronous implementation for SWE to see if performance can be improved that way.

# List of Figures

# Bibliography

[1] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. “Post-failure recovery of MPI communication capability: Design and rationale.” In: *The International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254.

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

[3] 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.

[4] 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.

[5] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. “Detection and correction of silent data corruption for large-scale high-performance computing.” In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–12.

[6] P. H. Hargrove and J. C. Duell. “Berkeley lab checkpoint/restart (BLCR) for Linux clusters.” In: *Journal of Physics: Conference Series* 46 (Sept. 2006), pp. 494–499.

[7] A. Hölzl. “Integrating TeaMPI with ULFM for Hard Failure Tolerance in Simulation Software.” Bachelorarbeit. Technical University of Munich, Aug. 2020.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. 2015. URL: `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System.” In: IEEE Computer Society, 2010.

[10] MPICH Group. *MPICH Overview*. URL: `https://www.mpich.org/about/overview/`.

[11] Open MPI Team. *General information about the Open MPI Project*. URL: `https://www.open-mpi.org/faq/?category=general#what`.

[12] P. Samfass, T. Weinzierl, B. Hazelwood, and M. Bader. “TeaMPI - Replication-Based Resilience Without the (Performance) Pain.” In: *High Performance Computing* (2020), pp. 455–473.

[13] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. "The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing." In: *Int. J. High Perform. Comput. Appl.* 19.4 (2005), pp. 479–493.