DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Towards Soft Error Resilience in SWE with TeaMPI

Atamert Rahma

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Towards Soft Error Resilience in SWE with TeaMPI

Erforschung von Techniken gegen Datenkorruption in SWE mit TeaMPI

Author:	Atamert Rahma
Supervisor:	Prof. Dr. Michael Bader
Advisor:	M.Sc. Philipp Samfass
Submission Date:	15.08.2021

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

Munich, 15.08.2021 Atamert Rahma

Acknowledgments

I would like to thank Philipp Samfass for his guidance and support. He has been patiently listening, supporting and helping me figuring out all the problems that I encountered during my work.

Abstract

Increasing demand for HPC applications has resulted in large clusters with thousands of nodes that can suffer from various types of failures which are even expected to increase in future systems. Hence resilience must be taken into account while running such large applications. In this thesis we focus on soft errors which can corrupt the state of an SWE application without raising an error. We introduce and analyse different methods that can provide soft error resilience. While our first method can only provide soft error detection using hash-value comparison of the results, our second and third methods can additionally provide recovery within the application. Their recovery and detection mechanisms utilize process replication and admissibility validation of the results. We analyze the soft error outcomes of our methods in various situations using bit-flip injections, and discuss their advantages and drawbacks. We have found out that our second method which employs task sharing is the most efficient one in terms of performance. However our third method which depends on independent redundant computation can additionally correct some errors that otherwise would only be detectable using the other methods.

Contents

1 Introduction

Today, the increasing demand for high-performance computing (**HPC**) has resulted in large computer clusters with thousands of nodes which are designed to run large parallel jobs. As the number of system components such as CPU and memory increase, the mean time between failures (**MTBF**) will decrease, and the future systems will be more exposed to hardware errors than today [22, 20]. The exact cause of the failures can vary depending on the structure and the environment of the system, and their consequences can be fatal. In this thesis we focus on transient (soft) errors that may result in silent data corruptions (**SDCs**). SDCs materialize themselves as bit-flips that can be induced by particles such as neutrons from cosmic radiation or alpha particles emitted from packaging material [2]. The consequences can be crash, infinite loop, or undetected incorrect results. Studies have showed that simple Error Correction Codes (**ECC**) mechanisms cannot correct a large amount of DRAM errors [12], and SRAM soft error rates keep growing as memory chips become larger [15]. Therefore mechanisms for error resilience are needed especially for the applications that run in large computer clusters. Replication or Checkpoint/Restart (**CR**) procedures are the most well-known approaches for failure tolerance against physical (hard) errors such as process failures [10].

CR however cannot be used alone for soft error resilience if SDCs remain undetectable. We thus construct and analyze three different replication-based methods for soft error resilience in SWE applications that utilizes TeaMPI library in this thesis. In the next section we briefly look at other related works, in section 3 we give an overview of the fault-tolerance in HPC and give some background information about the tools and software that are used in our implementations. Later in section 4 we introduce our soft error resilient methods including their implementation decisions and the fault-tolerance they provide. We then evaluate our methods, compare their soft error outcome rates and performances, and discuss their advantages and drawbacks in section 5. Then in section 6 we give an outlook on the future work that can be done, and finally in section 7 we summarize our findings.

2 Related Work

2.1 RedMPI

RedMPI library [8] provides the capability of soft error resilience for MPI applications by comparing the communicated messages of replicated MPI processes. It uses the MPI profiling layer to intercept the MPI calls and makes the replicated process executions transparent to the application. The comparison is optimized by not comparing the full message contents, but only comparing their hash values on the receiver side. In double redundant mode RedMPI makes sure it can detect faulty messages because the replicas are run in a deterministic manner. Additionally in triple redundant mode a voting algorithm can be used to even correct faulty results. RedMPI assumes that an SDC that is not immediately communicated over MPI should eventually be detected as the corrupted memory may be operated on and finally be transmitted. It is important to note that RedMPI does not provide resilience for transmission errors. Results show that even under extreme scenarios RedMPI can prevent SDCs from propagating without being detected.

2.2 Soft Error Detection and Automatic Recovery

A recent study [13] has proposed a soft error resilience solution that combines replication and CR in order to recover from failures. The first feature it has, is SDC detection by employing thread-level replication. The second feature is recovery with double redundant mode by using either multiple system-level checkpoints or a single application-level checkpoint. Multiple checkpoints can be written in different time intervals to make a safe rollback possible in case there is no guarantee of a checkpoint to be consistent. For example if the latest checkpoint has already been corrupted by a previously occurred SDC, a prior checkpoint must be loaded. However this feature comes with the overhead of storing multiple checkpoints, and this still cannot guarantee the validity of the prior checkpoints. As an alternative a single safe application-level checkpoint can be written. Each thread calculates the hash value of its own checkpoint and compares it with its replica. If the checkpoints match, the prior checkpoint can be discarded. If not, then a rollback to the prior safe checkpoint has to be made.

3 Background

3.1 Message Passing Interface

The Message-Passing Interface (**MPI**) is a message-passing library interface specification that describes a parallel programming model. The model focuses on communication between processes with different address spaces through cooperative operations on each process. MPI is mainly used in distributed memory communication environments such as large clusters that are used in HPC. It is being developed since 1992, and at the time of writing the latest official version is MPI-4.0 [14].

MPI is not an implementation itself but a specification. It only contains MPI operations that are expressed as functions, subroutines, or methods. One of the most well-known implementations of MPI, Open MPI [9], is used in this thesis and as a consequence all the examples and provided source codes will be based on Open MPI.

3.2 Fault Tolerance Techniques in HPC

The rate and root cause of a failure in a machine can depend on its size, structure, and the intensity/type of the workload running on it [20]. In this thesis we focus on the failures of following types:

- **Hard Error** : Failures that cause a process to fail and stop execution.
- **Soft Error** : Failures that do not cause a permanent failure.

Unlike the hard errors, soft errors may remain "silent" and not be detected until the program's output is observed. They are triggered by a subatomic event, where energetic particles such as neutrons from cosmic rays or alpha particles that are emitted from packaging material hit the silicon device and cause a sufficient charge to get deposited around a transistor that inverts the state of a logic device such as an SRAM cell, latch or a gate [15].

Soft errors can cause an SDC if the faulty bit has no error protection and it affects the outcome of the program. Some examples of SDCs are [22]: undetected arithmetic computation error, undetected control error that results in a premature termination of an iterative loop, incorrect memory or network transfers that were not detected by the error protection mechanisms. Soft errors may eventually be detected in later cycles depending on their impact on the application, however they may also affect the results of an application but never raise an alert.

Hard and soft error rates are expected to increase in future systems and therefore lots of research has been done in the past years towards better failure mitigation techniques. In the following we look into two main fault-tolerance techniques in HPC.

3.2.1 Checkpoint/Restart

One of the most well-known techniques in HPC is the application level CR, where the state of an application is stored in a safe place (e.g. hard disk) to prevent any loss of useful work caused by failures. CR allows the application to be restarted at any time from its stored state, and it is widely used for hard error tolerance [22]. However using CR for soft error tolerance is not very effective since the stored checkpoint could include an SDC which would disable the restart mechanism.

One limitation of CR is that the overhead of saving a checkpoint increases with the number of cores [22]. A study shows that only 35% of the work is spent on actual computation for a 168 hour job on 100k nodes with a MTBF of 5 years while the remainder is spent on CR [7].

3.2.2 Replication

Replication in HPC means the duplication of the same application or task by a process or thread. As the overhead of CR increases with the increasing core count, replication can be a better alternative in large-scale systems where a replica can take over the execution of a failed process. For example the protocol rMPI provides dual-execution of MPI applications that can provide hard error resilience, and it can actually be cheaper than CR at large core counts where CR overhead exceeds 50% [7]. Redundancy can also be combined with CR using partial process replication and CR in non-replicated processes to allow a trade-off between additional resources and wall-clock time that can be tuned depending on the application and available resources [6].

Another major advantage of replication against CR is that it can provide soft error resilience. One example implementation is RedMPI which uses process replication to compare the outcomes of the replicated processes [8]. It optimizes the comparison by not sharing the full content but just comparing the hashes of the outcomes at the receiver side. Results show that it can provide soft error resilience even on systems with a high failure rate, and therefore can potentially be used on future systems. Unfortunately replication needs to be at least tripled to provide correction by employing a voting mechanism. To avoid triple redundancy CR can be added to replication to save a chain of system-level checkpoints that ensure recovery [13]. After an error detection, if the latest checkpoint has already been corrupted, a prior checkpoint must be used. This solution however has the overhead of storing multiple checkpoints which is not expected to be the best solution for recovery in future systems. Instead, the study [13] states that a single safe checkpoint can be written as the last stored checkpoint to reduce multi-checkpoint overhead. Replicas can validate the checkpoints by comparing their hash values and make the recovery from an SDC possible while only utilizing double redundancy – one replica per process.

One major disadvantage of replication is losing at least half of the available computation power. Team-based resilience using task sharing with TeaMPI can reduce this redundancy overhead [19]. It shuffles the task execution order among the replicated ranks that allows replicas to save some redundant computation. On the other hand without redundant computation providing soft error resilience becomes more challenging since e.g. we cannot utilize task validation using hashes.

3.3 User Level Failure Mitigation

One important feature that the current MPI version lacks is failure tolerance in terms of recovering from component failures such as corrupted transmission over a faulty network interface or process failures due to a process or node failure. MPI-4.0 addresses this as: *"MPI does not provide mechanisms for dealing with transmission failures in the communication system [..] MPI itself provides no mechanisms for handling MPI process failures, that is, when an MPI process unexpectedly and permanently stops communicating (e.g., a soft-ware or hardware crash results in an MPI process terminating unexpectedly)."* [14]. The User Level Failure Mitigation (**ULFM**) [3] was proposed as an extension in order to integrate fault tolerance into MPI. ULFM provides different error codes such as MPI_PROC_FAILED that can be returned to notify the application about a possible process failure. The error codes then can be used to handle process failures within the application.

3.4 SWE

We have tested all of our techniques that we present in section 4 in SWE which is a flexible simulation software that was developed for teaching parallel programming. It implements simple Finite Volume models that solve two-dimensional shallow water equations. We will consider the approach described in equation 3.1, where we formulate the changes from one simulation-time to the next.

$$\begin{aligned} Q_{i,j}^{n+1} = Q_{i,j}^{n} &- \frac{\Delta t}{\Delta x}(\mathcal{A}^{+}\Delta Q_{i-1/2,j} + \mathcal{A}^{-}\Delta Q_{i+1/2,j}^{n}) \\ &- \frac{\Delta t}{\Delta y}(\mathcal{B}^{+}\Delta Q_{i,j-1/2} + \mathcal{B}^{-}\Delta Q_{i,j+1/2}^{n}) \end{aligned} \tag{3.1}$$

The term $Q_{i,j}^{n} = [h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ is the vector of conserved quantities: water height (h), horizontal (hu) and vertical (hv) momentum at simulation-time $t^n \in \mathbb{R}_0^+$, Δt is the length of the time step between the simulation-times, Δx and Δy are the size of the grid cells. The term $\mathcal{A}^{\pm}\Delta Q_{i\mp 1/2,j}$ accumulates the solution to the Riemann problems on the left and right side, and $\mathcal{B}^{\pm}\Delta Q_{i,j\mp 1/2}^{n}$ on the top and bottom side of the cell $(i,j) \in \Omega \subset \mathbb{R}^2$ [4].

The main purpose of SWE is to introduce students different parallel programming models such as OpenMP, MPI, and CUDA. SWE can also be used to simulate tsunamis and wave propagation in various domains.

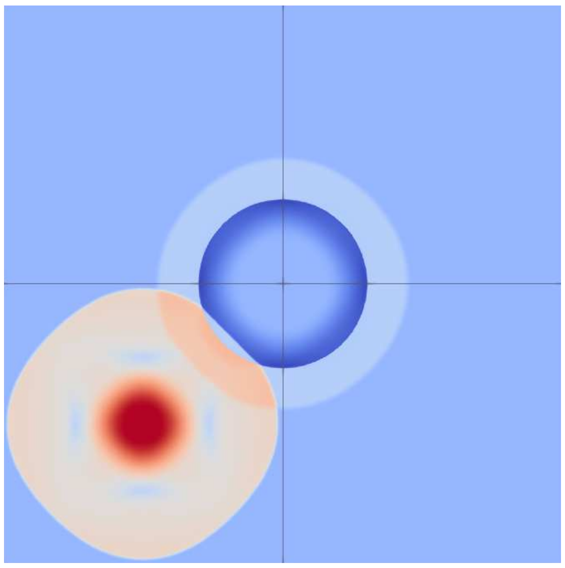

Figure 3.1: Wave propagation in *radialBathymetryDamBreak* scenario. Black lines are the boundaries of the divided domain for parallel processing. 4 MPI processes were used for the simulation.

The software architecture of SWE is kept simple for educational purposes but it can change a lot depending on the implementation and the methods being used. In the following we cover the sub-partitions of the main computational domain in SWE: *blocks*.

3.4.1 Blocks in SWE

In SWE a single Cartesian grid block can be used to represent the computational domain. We will use the MPI implementation of SWE and divide the computational domain into multiple grid blocks which are then assigned to MPI processes for parallel processing. At the end of the decomposition each grid block will represent a single task that needs to be computed by a process. In SWE the blocks are typically derived from a base abstract class *SWE_Blocks*, which contains:

- 2D arrays for the bathymetry (*b*), water height (*h*), horizontal water momentum (*hu*) and vertical water momentum (*hv*). It is assumed that *b* stays constant during the simulation and the variables *h*, *hu* and *hv* are updated by the classes derived from *SWE_Blocks*.

- Virtual methods that define important components for the simulation to be implemented by the derived blocks. Two main components that are important for this thesis are *computeNumericalFluxes()* and *updateUnknowns()*. In our derived block implementation we use the first method to compute the net-updates from the arrays *h,hu,hv* and the maximum possible time step that we can update in our simulation. We use the second method to actually update the arrays *h,hu,hv* with the updates that were computed by the first method.

- Other information such as grid sizes, maximum possible time step etc.

Before running a simulation the user can typically specify the number of cells for the simulation grid in horizontal (x) and vertical (y) directions and the total simulation duration in simulated time. When using MPI the initialized simulation domain is divided and assigned to the ranks, and the simulation begins. In figure 3.2 in the next page we demonstrate a simple code structure along with its flow diagram in figure 3.3 where the main methods are executed and the simulation domain is updated. We will refer to this loop as the computation-loop in further sections. In our example we first exchange boundaries between the blocks. Then each rank can compute their tasks and globally agree on an admissible time step size. After the agreement the ranks can update their blocks' arrays and their current simulation time. The computation-loop is then iterated until the current simulation time reaches the simulation duration given by the user.

An example output can be visualized using a visualization application like *ParaView* [1]. Figure 3.1 in the previous page is an example visualization of a wave propagation in a hard-coded scenario (*RadialBathymetryDamBreak*) simulated using four MPI ranks.

```
float t = 0.f;
/* Computation−loop */
while ( t < simulationDuration ) {
  /* Exchange boundaries between blocks */
  [...]
  /* Compute the current net−updates */
  block.computeNumericalFluxes();

  /* Agree on an admissible time step size */
  float local_maxTimestep = block.maxTimestep;
  float agreed_maxTimestep;
  MPI_Allreduce(&local_maxTimestep,
    &agreed_maxTimestep, 1,
    MPI_FLOAT, MPI_MIN,
    MPI_COMM_WORLD);
  block.maxTimestep = agreed_maxTimestep;

  /* Update the arrays h,hu,hv */
  block.updateUnknowns(agreed_maxTimestep);

  t += agreed_maxTimestep;

  /* Output can be written here if desired */
}
```

Figure 3.2: Basic SWE code structure with MPI.

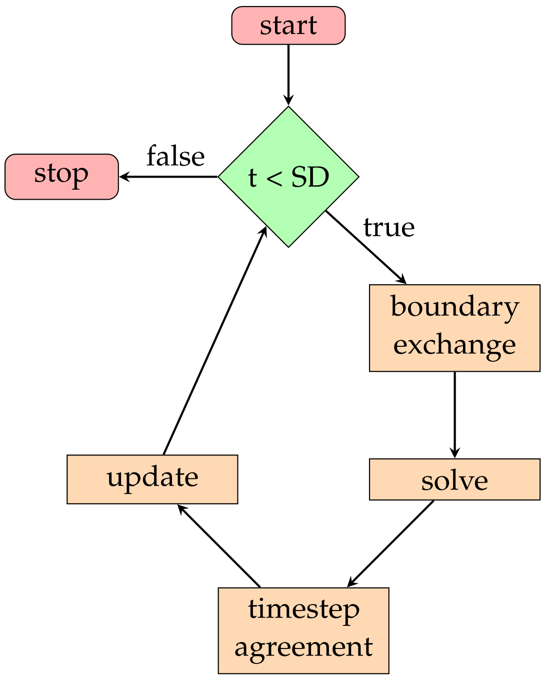

Figure 3.3: Basic SWE flow diagram with MPI. [t=time, SD=Simulation Duration]

3.5 TeaMPI

TeaMPI is a minimalist MPI wrapper implemented using the MPI Profiling Interface as a C++ library [19]. It enables replication for MPI applications by dividing all the processes into groups at the beginning of an application where each rank in a group has a replica in the other groups. This requires the number of total processes and the number of groups to be divisible with no remainder. Groups can be referred as teams which run the same application redundantly. Each process is assigned to its team communicator where the process can use the MPI world communicator (MPI_COMM_WORLD) as a communicator that includes its own team only. This makes TeaMPI transparent to the application, meaning that the application does not need to replicate any data structures or to be aware of the replication. We use TeaMPI wrapper in our methods that we will propose in section 4.

3.5.1 Heartbeats

Processes can exchange non-blocking messages between their replicas to identify any failing ranks. We call those messages *heartbeats*. Exchange of a heartbeats is done using inter-team communicators which contains all the replicas of a rank. Using the heartbeats ranks can spot failing or slowing ranks including themselves. Even with the inter-team communication TeaMPI requires no strict synchronization between the teams unlike other replication models [8].

The messages in the heartbeats do not necessarily have to contain any data to detect failing ranks. However the heartbeats can carry e.g. hash values of the results computed redundantly by the replicated ranks. This can be used as an early SDC detection mechanism that ensures data consistency.

3.5.2 Task Sharing

One of the concepts that can be used to reduce the overhead of replicated task computations is task sharing [19, 21]. A task can be defined as a combination of data which we continuously update in our computations like the arrays *h*, *hv* and *hu* in SWE. The main idea of task sharing is dividing the computation domain into multiple tasks that can be assigned to different processes which then can share those tasks among other replicated processes. This can only work if the tasks are shareable. We define a task as shareable if [19]:

- the task has a unique id and it is assignable to a single process,
- the task does not depend on the results of other tasks in the task computation,
- the task's results can be sent to other processes via MPI.

In addition to shareable tasks if the tasks require intensive calculations and their computation time is greater than the MPI communication, then task sharing can significantly reduce the overhead of redundant computations in replication. In order to achieve this, the replicated processes in replication can skip their calculations and use the shared tasks of their replicas. This can yet also make the application more sensitive to soft errors. A possible SDC in a single rank can be spread to its replicas and corrupt them as well.

3.5.3 ULFM Integration into TeaMPI

In his thesis Alexander Hölzl has integrated team recovery using ULFM (3.3) into TeaMPI [11]. If a slowing rank or process failure is detected using the heartbeats, his integration uses the features of ULFM to handle the failure. There are three strategies to mitigate a hard failure:

1. **KillTeam** : kills the team if one of its ranks fails. Remaining teams can continue their executions with a new inter-team communicator which does not include the team that has been killed.
2. **RespawnProc** : spawns new ranks to replace the failed ones.
3. **WarmSpare** : uses warm spares: Spare ranks that are launched together with the other ranks at the start of the application to replace the failed ones. Warm spares are interrupted during TeaMPI initialization to wait for such a replacement.

In order to use this integration in an SWE setting, one should:

- first set the desired error-handling strategy,
- create a callback function that can create a checkpoint,
- create a callback function that can recover from a checkpoint,
- and periodically check for failures using the heartbeats provided by TeaMPI.

The error handler can call the callback functions only if a rank fails which can reduce the total checkpointing time in the application. The major overhead of this reactive recovery mechanism comes from the redundant replication of processes. His analytical model showed that the redundant computation can become more efficient than CR with higher costs of writing checkpoints, however his tests on SWE reported that CR is faster than the reactive team recovery. In order to reduce the redundant computation overhead he mentioned task sharing as a possible solution.

3.6 Hard Failure Tolerance with Task Sharing in SWE with TeaMPI

Simon Schuck has integrated the mentioned task sharing candidate into the Hölzl's work in his thesis [21]. His task sharing mechanism is implemented between the replicated processes in each team in order to reduce the replication overhead mentioned in the previous section.

3.6.1 Task Sharing in SWE

In his method of task sharing Schuck first divides the domain in SWE into multiple blocks like mentioned in section 3.4.1 and defines each block as a single task which have the properties defined in section 3.5.2. After these blocks are assigned to the ranks, each rank has to compute and share its results in every iteration. Since the method requires task sharing between the replicated processes in each team and each replicated process must have the same blocks, it is essential for the method that each rank has at least one primary block which is computed and shared by the rank to all of its replicas, and one secondary block per replica which the rank tries to receive from its replicas. If a replica-rank that the rank tries to receive form runs into a failure, the failure is detected by the tools provided by ULFM and the rank just computes the secondary block by itself. The blocks are ordered using the algorithm in figure 3.4 below.

```
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: Algorithm to determine primary and secondary blocks [21].

Schuck has evaluated his task sharing mechanism with team recovery from Hölzl's work and reported a 35% performance improvement. Team recovery with task sharing could keep up with CR. Which one is better depends on the checkpoint interval in CR.

One major drawback of this task sharing method is the increased risk of SDCs. As already mentioned in section 3.5.2, task sharing can make the SDCs propagate into the other teams: If all the replicas of a rank use the shared corrupted solution of that rank, then there would be no team left with a correct solution.

3.6.2 Decomposition Factor

Dividing the computation domain into smaller domains by splitting it into smaller problems is also called domain decomposition. For the task sharing explained in the previous section each rank gets a fixed number of primary blocks that is equal to the decomposition factor **d**. For $d = 1$ each rank gets a single primary block and *numberOfTeams - 1* (also equals to number of its replicas) secondary blocks. Increasing *d* allows us to divide the domain even further [21]:

$$\boldsymbol{blocksPerRank} = numberOfTeams \cdot d \tag{3.2}$$

Above in equation 3.2 we see the general formula for the total number of blocks (primary + secondary) per rank where each rank has d primary blocks. Additionally each rank holds also d secondary blocks for each of its replica. We can then calculate the total number of blocks in a team as:

$$\boldsymbol{totalBlocks} = blocksPerRank \cdot ranksPerTeam \tag{3.3}$$

In this thesis we will always use the term decomposition factor in this context. Schuck's implementations already include the decomposition factor, however he did not evaluate its effect on the performance. For example he reported a 70% overhead of inter-node communication when the ranks of the teams are on the same node (we will also use: *teams on the same node* in this sense in future sections). His workaround for this was to map the replicas onto the same node, because there are a lot more MPI communication between the replicas than within the teams due to task sharing. But depending on the implementation, the decomposition factor may also increase performance if the application can do some useful computation while waiting for the non-blocking communication of task sharing to finish for the secondary blocks. Schuck also does not share the data arrays (*h*,*hu*,*hv*) in his task sharing, but he shares the net-update arrays which are a lot larger than the data arrays in total. This might also have an effect on the large overhead that he reported when teams were on the same node.

4 Integrating Soft Error Resilience Techniques in SWE with TeaMPI

In the following we introduce different techniques for providing soft error detection/recovery in SWE applications with TeaMPI using replication with two teams. Then we look at their implementation decisions and briefly preview their soft error tolerances. All the source code is written in C++ to be compatible with TeaMPI and SWE which are also based on C++.

4.1 Soft Error Detection Using Heartbeats with Hashes

We use software-level redundancy in this method, mentioned in the section 3.2.2, by redundantly running the application twice with MPI. We hash the tasks computed by the replicas of two teams and share the hashes across them in fixed simulation-time intervals. Any mismatching hash can be a sign of an SDC, and therefore this method is a very simple usage example of software-level redundancy that can provide an early SDC detection. We will refer to this method as **Hashes** in later sections.

4.1.1 Implementation

We use the C++ standard library function **std::hash** for better performance. Other hash functions like SHA-1 [5] could also be used but it is expected to be slower due to its greater length. In order to update the hash value of our important data arrays we have implemented the method **updateHash** which first converts the data arrays (*b,h,hu,hv*) and the agreed time step size to strings and then hash them individually. Then it combines their resulted 8-byte long hash values using *xor* operations.

In our implementation we issue a single-heartbeat that includes a hash value in regular heartbeat intervals in simulation time. Our heartbeat usage is different from the usage explained in section 3.5.1 where the heartbeats are used to detect any slowing or failing rank by comparing the wall-clock time differences of the heartbeats. In our case we do not have to periodically start a heartbeat before a heavy task computation and end it after with a total of two heartbeats. Therefore we have integrated a single-heartbeat mechanism into TeaMPI that can include messages. Additionally, since hard error tolerance is out of scope for this thesis we do not keep the wall-clock times of the single-heartbeats, but we compare the received hashes from the other replicas within the TeaMPI library. The actual sending, receiving and the comparison of the hashes are done transparently to the application and we use non-blocking send and receives to avoid additional synchronization between the replicas. The fixed heartbeat interval is determined by dividing the given total simulation duration to the total number of hashes that we want to send. The more hashes we send, the smaller the heartbeat-interval is and the sooner we can detect an SDC.

```
float t = 0.f;
for (int i = 0; i < numberOfHashes; i++) {
  /* Computation−loop */
  while (t < sendHashAt[i]) {
    /* Exchange boundaries between blocks */
    [...]
    /* Compute the current net−updates */
    block.computeNumericalFluxes();

    /* Agree on an admissible time step size */
    float local_maxTimestep = block.maxTimestep;
    float agreed_maxTimestep;
    MPI_Allreduce(&local_maxTimestep,
      &agreed_maxTimestep, 1,
      MPI_FLOAT, MPI_MIN,
      MPI_COMM_WORLD);
    block.maxTimestep = agreed_maxTimestep;

    /* Update the arrays h,hu,hv */
    block.updateUnknowns(agreed_maxTimestep);

    t += agreed_maxTimestep;

    /* Output can be written here if desired */

    updateHash();
  }
  /* Send a single heartbeat with the hash value
   * to the replica rank */
  [...]
}
```

Figure 4.1: Computation-loop for method *Hashes*.

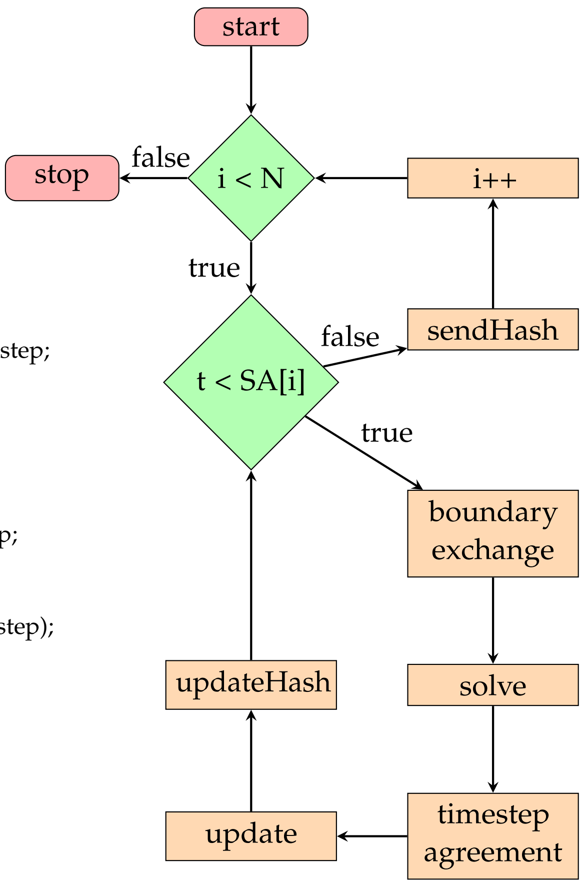

Figure 4.2: Flow diagram of method *Hashes*. [t=time, N=total hash sends, SA=equally distanced simulation-times for each hash to send]

In method *Hashes* we only compute one block per rank so we divide the domain in

SWE into *number_of_ranks_in_a_team* blocks. After we read important user parameters like the heartbeat-interval and initialize the blocks, we go into our computation-loop in figure 4.1 in the previous page. The flow diagram in figure 4.2 shows the general structure of the computation-loop which is very similar to the example 3.3 that we have discussed in the previous section except for loop conditions. Since we use the TeaMPI wrapper, the original MPI_Allreduce function is modified for the MPI_COMM_WORLD communicator to agree on an admissible time step size across all the ranks in the team. The ranks continue to compute their blocks and update their hashes as explained above until the current simulation time reaches the next simulation time to send the updated hash value. Then the ranks issue a single-heartbeat that includes the hash value to their replicas and go into the loop again until all the hashes are sent. The simulation time intervals between the heartbeats are equally distanced (*simulationDuration / numberOfHashes*), and the last heartbeat is always issued when the simulation-time reaches the simulation duration given by the user.

4.1.2 Soft Error Tolerance

We assume that any bit-flip that can occur in our data will eventually be detected by the hash comparisons. The hash value is updated with *xor* operations after every iteration with the hash values of the data arrays (*b,h,hu,hv*) and the current agreed time step size which should be same in all the replicas. This also allows us to detect any SDC that eventually affects these important data arrays. One downside of method *Hashes* is that it uses replication which loses half of our computation power, and even if an SDC is detected we must always restart the application unlike the other methods that we propose in the following sections.

4.2 Soft Error Resilience Using Verification and Task Sharing

Instead of comparing the results calculated by the replicated processes, we can adopt an *a-posteriori* approach where the validity of each computed task is checked using some predefined admissibility criteria in every iteration. With this approach we can assume that we have an SDC if an admissibility criterion gets violated, which can provide SDC detection without replication. In this method we use replication only for recovery of the corrupted ranks. One major advantage of this is that the replicated ranks are no longer needed for redundant computation, so we reduce their overhead by employing task sharing. We will refer to this method as **Sharing**, again for better readability. In the following we first introduce our predefined admissibility criteria, and then explain our task sharing mechanism. Finally we introduce our implementation for this method and give a preview of its soft error resilience.

4.2.1 Admissibility Criteria

Our admissibility criteria are very similar to the criteria defined in [17]. We use the following two main criteria:

1. **Physical Admissibility Criteria**: computations are within some certain physical constraints which are highly application specific. We use the following two constraints:
    - Bathymetry (b) must be constant.
    - Water height (h) cannot be negative.
2. **Numerical Admissibility Criteria**:
    - No floating point number errors (NaNs). This criterion is checked for all the data and update arrays.
    - Relaxed discrete maximum principle (**DMP**) in the sense of polynomials. We check this criterion only for the data arrays h, hu, hv.

$$\min_{y \in V_{i,j}} u(y, t^n) - \delta \;\leq\; u^*(x, t^{n+1}) \;\leq\; \max_{y \in V_{i,j}} u(y, t^n) + \delta \tag{4.1}$$

The term $u^*(x, t^{n+1})$ is a candidate solution for h, hu or hv at cell $x = (i, j)$ at time t^{n+1}. The candidate solution fulfills the DMP, if the relation in 4.1 above is fulfilled for all conserved variables and for all points x. $V_{i.j}$ is the set that contains the neighbouring cells of cell (i, j). The strict DMP ($\delta = 0$) may however be violated even without a soft error in some cases. Therefore a relaxation factor δ is used to relax the strict DMP for small surpasses for both directions. We have e.g. run the *seaAtRest* scenario with the strict way and it did not violate the DMP because all the arrays remain constant throughout the simulation, but for wave propagation scenarios the relaxation factor should not be set to zero. In the next sections we will always assume that a relaxation factor is set properly for the scenario so that the DMP does not give us false positives – always validates when there is no soft error. Additionally relaxed DMP cannot always detect soft errors – e.g. when the corrupted value still lies in the allowed interval.

We mostly use library functions like **std::min** or **std::isnan** for the implementations of the admissibility criteria to minimize their overhead. However relative to the other criteria, the DMP can still be very expensive to check because it additionally needs to access all the neighbouring indexes.

Only the predefined admissibility criteria explained above are used in the methods that we are going to present in the next sections. With these criteria and the assumption that we have made we can assume that a rank may have an SDC if any of the criteria fails for at least one of its blocks.

4.2.2 Task Sharing

Task sharing becomes available for method *Sharing* since soft error detection does not depend on redundant computation. Similar to Schuck's work explained in section 3.6 we need replication only for recovery in this method, but in our case we focus on soft error resilience. We use the same algorithm and techniques explained in sections 3.6.1 and 3.6.2 for dividing and ordering of the primary/secondary blocks in each rank. We however change the actual task sharing implementation by only sharing the updated data arrays (*h*,*hu*,*hv*). We do not share any bathymetry data since it should stay constant.

In the following we present the structure and implementation decisions of our task sharing. We assume that up to this point the application has already called *computeNumericalFluxes* and calculated the primary blocks' updates for the current iteration, however it did not call *updateUnknowns* to update their data arrays (*h*,*hu*,*hv*) yet. We will later look at the computation and validation of the primary blocks, as well as the complete algorithm of method *Sharing* in section 4.2.3.

```
/* Receive requests of the non-blocking receive calls */
std::vector<MPI_Request> recv_reqs(totalRecvReqs_taskSharing, MPI_REQUEST_NULL);
/* Iterate the secondary blocks, receive h,hu,hv per block */
for (int i = decompFactor; i < blocksPerRank; i++) {
  const int& currentBlockNr = myBlockOrder[i];
  auto& currentSecondaryBlock = *simulationBlocks[currentBlockNr];
  /* Save previous h,hu,hv locally for the DMP criterion */
  currentSecondaryBlock.savePreviousData();
  /* Size of h,hu,hv */
  const int dataArraySize = currentSecondaryBlock.dataArraySize;
  /* The source replica for the current secondary block */
  int source_replica = currentBlockNr % numTeams;

  MPI_Irecv(...); /* For h */
  MPI_Irecv(...); /* For hu */
  MPI_Irecv(...); /* For hv */
}
```

Figure 4.3: Non-blocking MPI receive posts for the task sharing.

Figure 4.3 above implements the very beginning of the task sharing, where each replica posts three non-blocking MPI receive calls (for the data arrays *h*,*hu*,*hv*) for all of its secondary blocks. The source replica that the rank is going to receive from is calculated using the ordering algorithm 3.4. Before posting the receives, current states of the secondary blocks are redundantly stored. This is necessary for the DMP criterion explained in the previous section.

```
/* Iterate the primary blocks, send h,hu,hv per block */
for (int i = 0; i < decompFactor; i++) {
  const int& currentBlockNr = myBlockOrder[i];
  auto& currentPrimaryBlock = *simulationBlocks[currentBlockNr];
  /* Save previous h,hu,hv locally for the DMP criterion */
  currentPrimaryBlock.savePreviousData();
  /* Update the data arrays */
  currentPrimaryBlock.updateUnknowns(timestep);
  /* Size of h,hu,hv */
  const int dataArraySize = currentPrimaryBlock.dataArraySize;

  /* Iterate all my replicas */
  for (int destTeam = 0; destTeam < numTeams; destTeam++)
    if(destTeam != myTeam) {
      MPI_Isend(...); /* For h */
      MPI_Isend(...); /* For hu */
      MPI_Isend(...); /* For hv */
    }
}
```

Figure 4.4: Non-blocking MPI send calls for the task sharing.

After the non-blocking receives, ranks go into the loop in figure 4.4 above and iterate their primary blocks which they need to update and share to all of their replicas. They first save their primary blocks' current state redundantly for the DMP criterion in the current iteration before updating them to the next time step. Then the blocks are updated by calling the *updateUnknowns*, and lastly their data arrays (*h*,*hu*,*hv*) are sent to all of their replicas by using non-blocking MPI send operations. At this point we have posted all the necessary non-blocking send/receive MPI calls for the task sharing.

In the end of our task sharing algorithm we need to wait for all the previously posted non-blocking MPI communications to be completed. Figure 4.5 in the next page is the algorithm that we use to achieve this. Ranks go into a loop where they check if a secondary block is received using three **MPI_Test** calls for the pending data arrays. If a secondary block is completely received, then we immediately validate the received data arrays using our predefined admissibility checks again, even it was already validated by the sender right after the *computeNumericalFluxes* call as we will see in the next section. If an admissibility criterion is violated here, we mark this secondary block as potentially corrupted by setting a flag. We will see what we do with these flags and how a potentially corruption can be handled in the next section. Ranks iterate this loop until all of their secondary blocks are received, checked and flagged accordingly.

```
std::vector<int> received(totalSecondaryBlocks, 0);
int pendingBlocks = totalSecondaryBlocks;
/* Loop until all the non−blocking receives are completed */
while (pendingBlocks) {
  /* Iterate the secondary blocks */
  for (int i = decompFactor; i < blocksPerRank; i++) {
    /* If the block is not received yet */
    if (!received[i−decompFactor]) {
      /* Check if the receive requests are completed */
      int h_received, hu_received, hv_received;
      MPI_Test(...); /* Check if h is received */
      MPI_Test(...); /* Check if hu is received */
      MPI_Test(...); /* Check if hv is received */
      if (h_received && hu_received && hv_received) {
        bool admissible;
        /* Check admissibility criteria for the
         * received block, set flag if violates */
        [...]
        received[i−decompFactor] = 1;
        pendingBlocks−−;
      }
    }
  }
}
```

Figure 4.5: Algorithm for checking the admissibility criteria on the received secondary blocks while waiting for the non-blocking MPI communications to be completed.

4.2.3 Implementation

We start with our method *Sharing* after the blocks are initialized and ordered. Figure 4.6 in the next page represents our main implementation and figure 4.7 depicts our algorithm for this method. We have included heartbeats that are bound to wall-clock time. They however do not contribute to soft error resilience. We start the heartbeat and iterate the inner loop until the heartbeat interval (in wall-clock time) is reached. Then we end the heartbeat and go into the outer loop again until the simulation is finished.

In the computation-loop after the boundary exchange ranks first iterate and compute their primary blocks. Each block is checked using our admissibility criteria predefined in section 4.2.1 right after the computation and flagged accordingly – e.g. as possibly-corrupted if any of the admissibility criteria fails. After all the primary blocks are computed and flagged accordingly, ranks follow the procedure defined in figure 4.8 in the second next page which represents our reporting and recovery mechanism for

```
float t = 0.f;
while (t < simulationDuration) {
  /* Start Heartbeat (HB) */
  [...]
  /* Computation-loop */
  while (timeSinceLast_HB < HB_Interval
    && t < simulationDuration) {
    /* Exchange boundaries between blocks */
    [...]
    /* Iterate the primary blocks */
    for (int i = 0; i < decompFactor; i++) {
      /* Compute the current net-updates */
      [...]
      /* Validate admissibility checks */
      [...]
    }

    /* Report to all my replicas */
    [...]
    /* Go into recovery mode if an SDC
     * is detected in my primary blocks,
     * or reported from a replica */
    [...]

    /* Agree on an admissible time step size */
    [...]

    /* Update primary blocks and do
     * task sharing */
    [...]

    /* Report to all my replicas */
    [...]
    /* Go into recover mode if an SDC
     * is detected in my secondary blocks,
     * or reported from a replica */
    [...]

    t += agreed_maxTimestep;
    /* Output can be written here if desired */
  }
  /* End Heartbeat */
  [...]
}
```

Figure 4.6: Computation-loop for method *Sharing*.

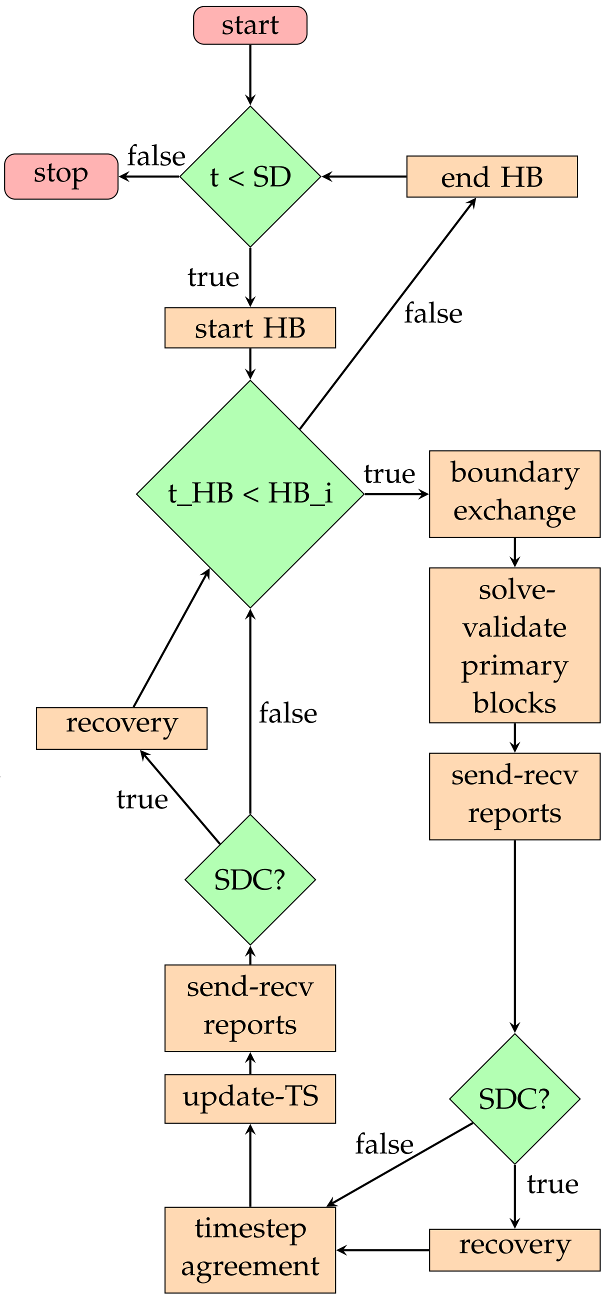

Figure 4.7: Flow diagram of method *Sharing*. [t=time, SD=simulation duration, t_HB=time since last heartbeat, HB_i=heartbeat interval, TS=task sharing]

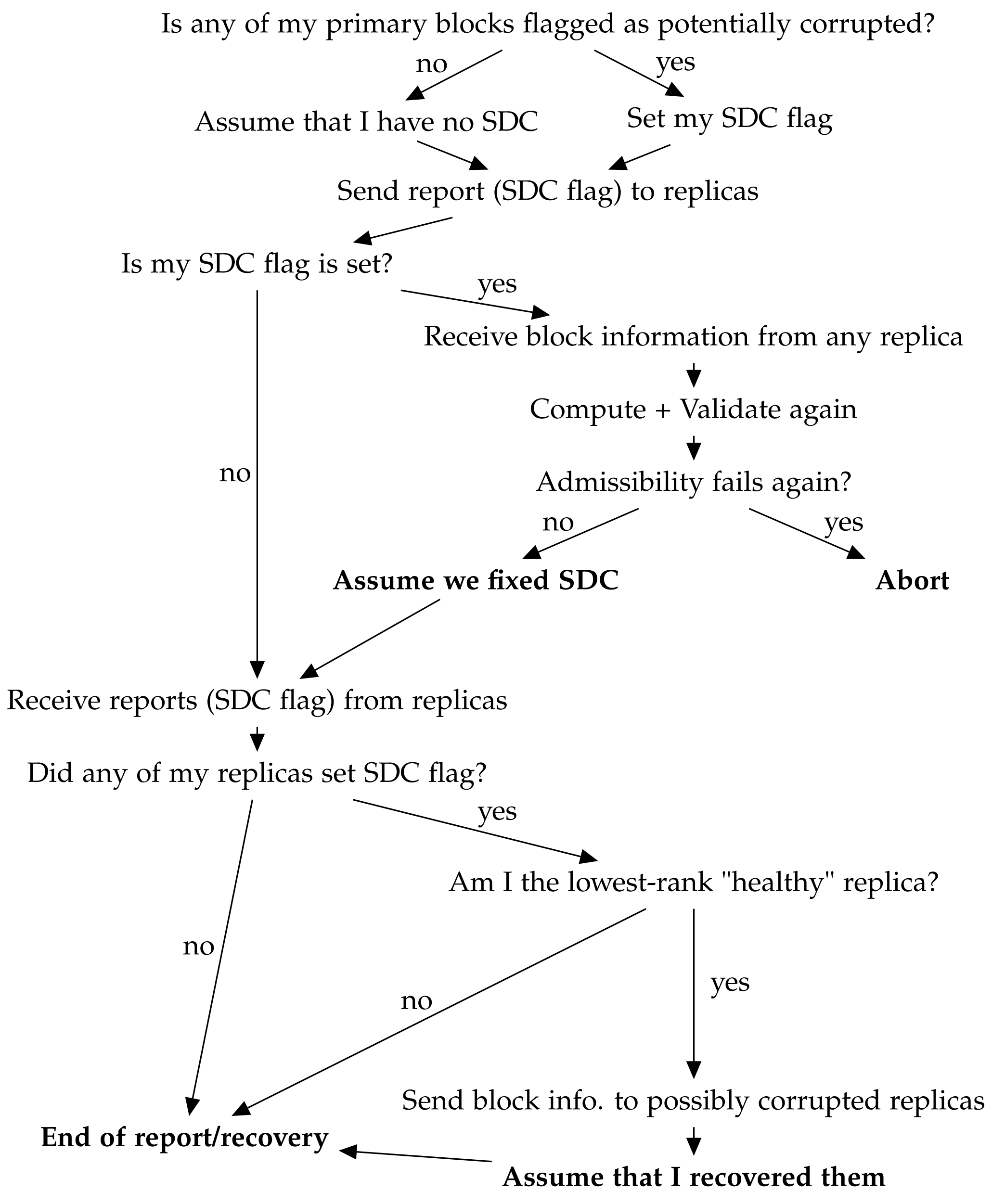
Is any of my primary blocks flagged as potentially corrupted?
no
yes
Assume that I have no SDC
Set my SDC flag
Send report (SDC flag) to replicas
Is my SDC flag is set?
yes
Receive block information from any replica
Compute + Validate again
no
Admissibility fails again?
no
yes
Assume we fixed SDC
Abort
Receive reports (SDC flag) from replicas
Did any of my replicas set SDC flag?
yes
Am I the lowest-rank "healthy" replica?
no
no
yes
Send block info. to possibly corrupted replicas
End of report/recovery
Assume that I recovered them

Figure 4.8: Reports and recovery after primary block validation.

their primary blocks. If any of the primary blocks of a rank is flagged as potentially corrupted after the admissibility checks then the rank sets its SDC flag and assumes that it has detected a possible SDC, but if all of its blocks are admissible then the rank assumes that it has no SDC in its primary blocks. Then the ranks send/receive a single MPI message that contains their SDC flag to/from each of their replicas. The purpose of this is to report their block status to each other and activate recovery mode in case they detect a possible SDC together with their replicas that hold the ranks' primary blocks as their secondary blocks. In the recovery mode – If a rank reports a possible SDC to its replicas – the possibly-corrupted rank waits for a respond from any of its replicas, and its other replicas that did not report a possible SDC (possibly healthy) decides who has the lowest rank and the lowest rank sends its blocks data arrays (*b*,*h*,*hu*,*hv*) to the possibly-corrupted rank and assumes that it has recovered the possibly-corrupted rank. The possibly-corrupted rank receives and immediately checks the received data arrays for admissibility. If any of the admissibility criteria still fails for any received block from the recovery then the rank assumes that its replica also holds a possibly-corrupted block – e.g. when a previously occurred SDC could not be detected and has propagated to the other ranks due to task sharing – and aborts. It is important to note here that we do not handle the situation where every replica reports a possible SDC in their primary blocks at the same time and everyone waits. This scenario can happen when e.g. multiple soft errors corrupt the application at all the replicas of a rank at the same time, or when a single soft error propagates to other teams corrupting their state because of task sharing and is detected later on. In this thesis we will only experiment with single soft errors per run, and if an error propagates due to task sharing and later be detected by all the replicas in their primary blocks causing the deadlock explained above, then a recovery would not be possible anyways because after the recovery the admissibility of the recovered block would again fail since all the replicas would also be corrupted with the exact same blocks that were shared in the previous iteration to those replicas. If the rank does not abort and accepts the solution of its replica because it is admissible, then the application continues and the ranks assume that they have managed to recover from an SDC.

After the primary block reports and recovery mode, the ranks agree on an admissible time step size and do the task sharing with secondary block validation using the admissibility criteria as explained in the previous section. After this, all the secondary blocks that have violated an admissibility criterion are flagged as possibly-corrupted, and ranks again follow a similar reporting and recovery mechanism depicted in figure 4.9 in the next page for their secondary blocks. At this point in the algorithm we actually assume that our primary blocks (and therefore all the main blocks) are valid and healthy because we have already checked them for soft errors, but this additional check on the secondary blocks may expose some soft errors that can occur after our first

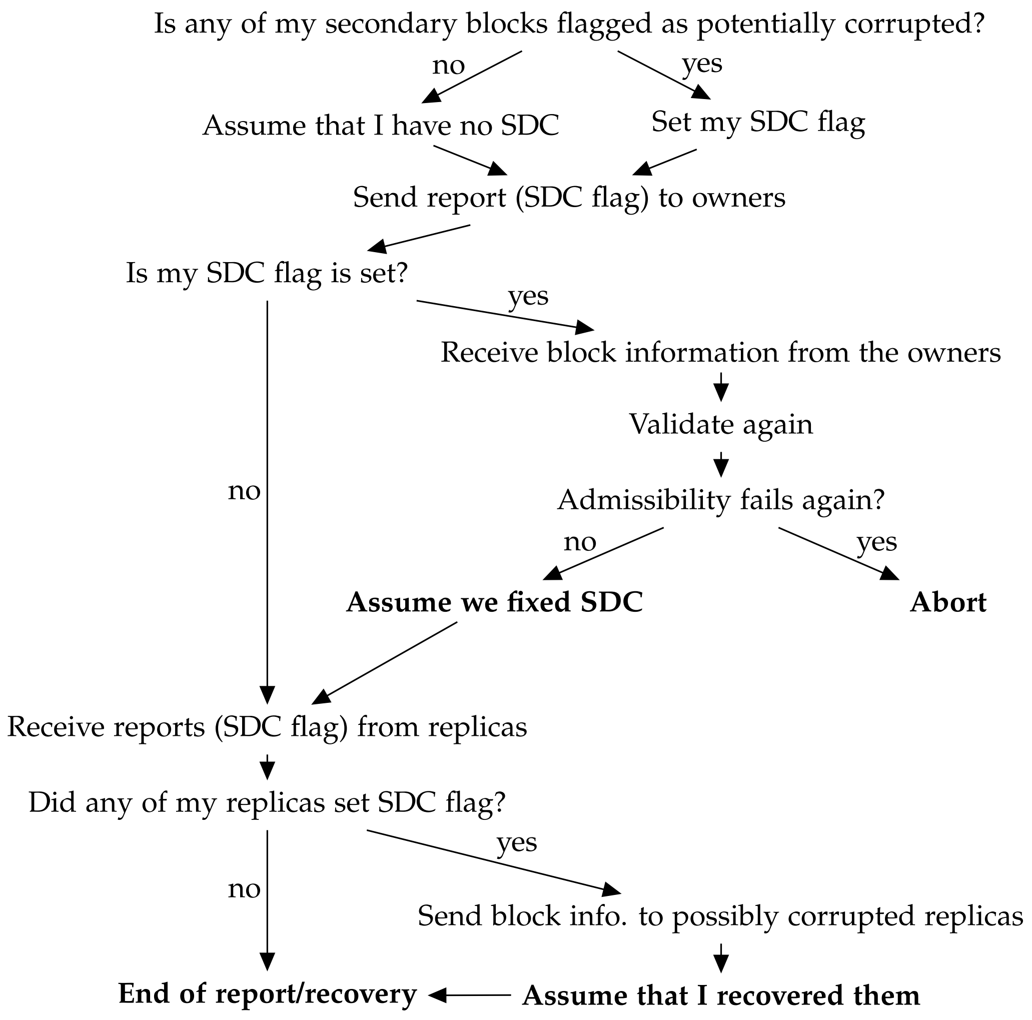
Is any of my secondary blocks flagged as potentially corrupted?
no
yes
Assume that I have no SDC
Set my SDC flag
Send report (SDC flag) to owners
Is my SDC flag is set?
yes
Receive block information from the owners
Validate again
no
Admissibility fails again?
no
yes
Assume we fixed SDC
Abort
Receive reports (SDC flag) from replicas
Did any of my replicas set SDC flag?
yes
no
Send block info. to possibly corrupted replicas
End of report/recovery
Assume that I recovered them

Figure 4.9: Reports and recovery after secondary block validation.

validation or during task sharing communication before actually writing any output. Its only difference to our previous recovery mechanism is that the possibly-corrupted blocks are tried to be recovered using their owners (same ranks that has sent the blocks in the task sharing) which are the replica ranks that hold the possibly-corrupted secondary blocks in the possibly-corrupted rank as their primary blocks, which means that they were already been validated and flagged as healthy in their computations in the current iteration. If any of the admissibility criteria still fails for any of the received blocks after the recovery then the rank aborts. This can happen e.g. when a soft error corrupts the primary block of a rank after its validation and causes the admissibility criteria to fail. We however did not implement a recovery system for this case. One could maybe try to first store the received blocks from the task sharing without overwriting the secondary blocks' current state and try to recover using them in case a possible SDC is detected. Our recovery for the secondary blocks can however recover from corrupted transmissions in the task sharing that directly violates our admissibility checks. We also do not handle some other cases where e.g. there is a transmission error during the recovery. If after the recovery the blocks validate all the admissibility checks without any problem, then the rank assumes that there was a faulty transmission but it has managed to recover its blocks and can continue its execution.

After the task sharing and secondary block reports/recovery, the current simulation time can finally be updated with the agreed time step size and the ranks can continue iterating the computation-loop.

4.2.4 Soft Error Tolerance

Since this and the next method that we will present rely on the admissibility checks to validate tasks, a possible SDC detection depends on the quality and sensitivity of the predefined admissibility criteria. If the relaxation factor is set properly, DMP can be a very sensitive admissibility criterion – we assume that it is high enough to give us no false positives when there is no soft error. However even with the DMP criterion we cannot guarantee SDC detection or recovery in this method. We can only assume that a block is corrupted if it violates an admissibility criterion and vice versa. For this method it is also very important when we detect the SDC because the method tries to save redundancy overhead by employing task sharing, but task sharing can also make the SDCs propagate to other teams like explained in the previous section. This makes our method even more sensible to SDCs because even if an SDC is detected in the later simulation times, the replica-ranks would already have been corrupted due to task sharing, and a recovery using a replica would not be possible. Additionally we have not implemented any recovery mechanisms for some cases that we are aware of –

especially for the cases where there can be more than one soft error at the same time, or for the soft errors that can occur in the primary blocks between the primary block validation and task sharing. In the following we propose a method that may recover even from such cases.

4.3 Soft Error Resilience Using Verification and Redundant Computation

Our last method is similar to our previous method *Sharing*. We again use our predefined admissibility criteria to validate the blocks but we disable the task sharing, and as a consequence ranks do not have any secondary blocks. Instead, every replicated process in a team keeps its blocks for itself and only send/receive blocks in the recovery. We will refer to this method as **Redundant** in later sections.

4.3.1 Implementation

Since we do not have task sharing we do not use any block-ordering, so the ranks only have primary blocks.

$$\boldsymbol{blocksPerRank} = d \tag{4.2}$$

In equation 4.2 above each rank has *d=decompositionFactor* number of primary blocks. We can again calculate the total number of blocks in a team using the equation 3.3.

After ranks initialize their blocks, they follow the computation-loop in figure 4.10 in the next page. We again include heartbeats that are bound to wall-clock time but they do not contribute to soft error resilience. The structure is very similar to our previous method *Sharing* without the task sharing, secondary block receive and validation procedure and secondary block reporting and recovery mechanism. After the ranks exchange their boundaries, they iterate their blocks where they compute the updates for the current iteration and validate their blocks using our predefined admissibility criteria. The ranks make the same assumptions that they did after their primary block validation in method *Sharing* – if any primary block of a rank is not admissible then the rank assumes that it has a possible SDC and it is possibly corrupted. Then the ranks follow the same reporting and recovery procedure defined in 4.8 which is already explained in 4.2.3 in detail. In short, ranks send/receive a potential SDC report to/from their replicas. If a rank is possibly-corrupted, a replica sends its data arrays to the rank. We again do not handle the case where all the replicas report a possible SDC in their blocks at the same time. In this method this can only happen when all the replicas suffer from an SDC at the same time since we run the teams redundantly without task sharing and a possible SDC in a team cannot propagate to

```
float t = 0.f;
while (t < simulationDuration) {
  /* Start Heartbeat (HB) */
  [...]
  /* Computation-loop */
  while (timeSinceLast_HB < HB_Interval
    && t < simulationDuration) {
    /* Exchange boundaries between blocks */
    [...]
    /* Iterate the blocks */
    for (int i = 0; i < decompFactor; i++) {
      /* Compute the current net-updates */
      [...]
      /* Validate admissibility checks */
      [...]
    }

    /* Report to all my replicas */
    [...]
    /* Go into recovery mode if an SDC
     * is detected in my blocks,
     * or reported from a replica */
    [...]

    /* Agree on an admissible time step size */
    [...]

    /* Update the blocks */
    [...]

    t += agreed_maxTimestep;

    /* Output can be written here if desired */
  }
  /* End Heartbeat */
  [...]
}
```

Figure 4.10: Computation-loop for method *Redundant*.

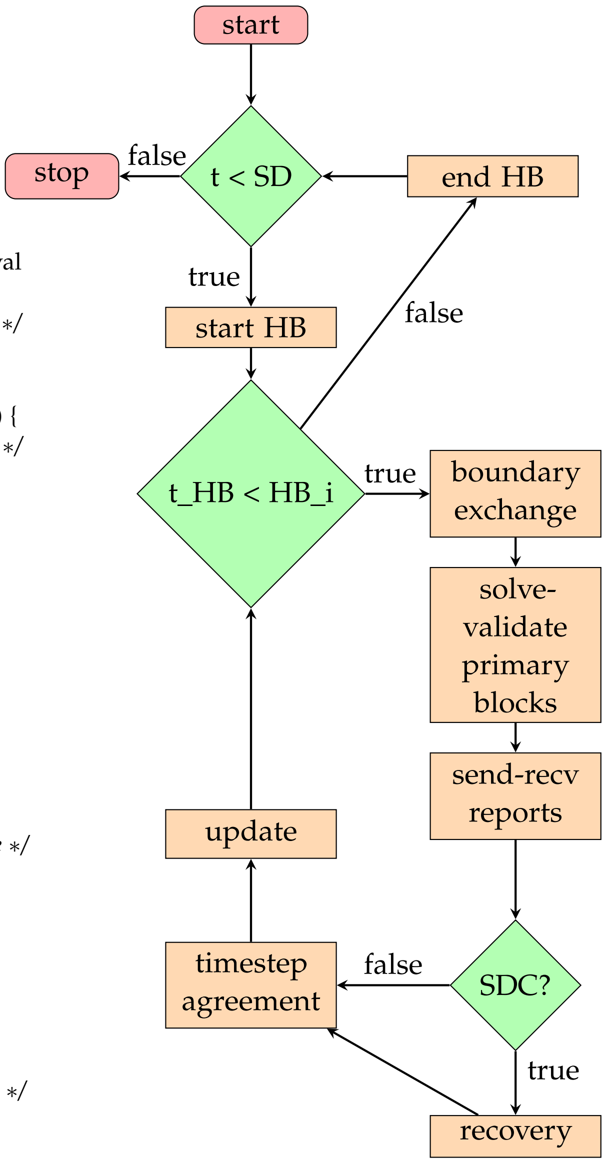

Figure 4.11: Flow diagram of method *Redundant*. [t=time, SD=simulation duration, t_HB=time since last heartbeat, HB_i=heartbeat interval, TS=task sharing]

other teams or replicas. This case would again require multiple soft errors at the same time. Unfortunately we forgot to include full recovery by recovering all the ranks in the possibly corrupted team in our recovery mode, and we have only implemented the recovery of the possibly corrupted ranks. This however does not guarantee us a complete healthy recovery since the error might have propagated to other ranks in the corrupted team, and all the corrupted ranks might not have detected the error. It is also unfortunate that we did not have the time to add this but this will not affect our results in this thesis since we will only analyze single soft errors and we have the option of relying on other teams' solution if the corrupted team detects an error. Nevertheless this is a very important future to have for the recovery especially for this method *Redundant*. After our primary block reporting and recovery mechanism, the replicas have redundantly computed, validated and reported all of their blocks. Similarly to our previous method a rank can still be corrupted but has appeared healthy in the admissibility checks. But in this method teams can detect a previously occurred SDC in later iterations and a recovery from such a situation would theoretically still possible (we again neglect multiple soft error situations) since an error cannot propagate to other teams. After the time step agreement, the ranks finally update their blocks and repeat this loop until the simulation is finished. The flow diagram in figure 4.11 in the previous page represents this method *Redundant*.

4.3.2 Soft Error Tolerance

Similar to the previous method, this method also relies on the same admissibility checks to validate tasks. So we cannot guarantee any SDC detection. We again assume that a rank is corrupted if any of our admissibility criteria gets violated for any of its blocks. We only use the admissibility checks once in the algorithm – we do not separate primary and secondary block validations since ranks only have primary blocks – and an SDC in a block can only occur before or/and after the validation of the block. SDCs that can occur before the block validation, and that are detectable by our admissibility checks directlytrigger the recovery mode, and SDCs that can occur after the block validation or that are not immediately detected by the admissibility checks can be spread only across the corrupted team. A recovery is theoretically possible in both of these cases if the other teams do not also get corrupted by other SDCs. Therefore we can guarantee to have at least one healthy team if we only consider single soft error situations with this method because teams do not share any data and run redundantly, which allows us to recover even in later iterations if the error can eventually be detected by our admissibility checks. In such a case we cannot prevent the corrupted team from writing faulty outputs in previous iterations. But if a team finally detects an SDC, the user can rely on the other teams' outputs. This is also our workaround in the absence of full recovery explained in the previous section. For single soft error scenarios we therefore expect to have at least one healthy solution under the following two conditions: (1) if the error does not naturally cause the application to abort, and (2) if the error can eventually be detected by our admissibility checks at any time until the simulation time reaches the simulation duration. Since we have disabled task sharing we can prevent SDCs from propagating but we may also suffer from redundant computation overhead.

5 Evaluation

In the following section we analyze our methods' soft error resilience by injecting random bit-flips into different data structures in SWE applications that utilize our methods, and then their SDC outcome rates (only detectable, or detectable and correctable) are compared and different situations are discussed. Later in section 5.2 we compare the performance of our methods on single- and multi-node environments, and discuss how different soft error resilience techniques can affect the performance.

5.1 Soft Error Outcome Rates

In our bit-flip injector we focus on the largest data structures that we have: data arrays containing water height (*h*), water momentum in both directions (*hu,hv*) and bathymetry (*b*); and last but not least the 8 net-update arrays for the computation. All the methods are run using two teams, and for each injection the error is injected into the first rank of the first team by flipping a single randomly selected bit from a randomly selected floating-point number right after the main computation function call *computeNumericalFluxes* which accesses all the data structure listed above. We do not analyze the errors that can occur in different code places or the transmission errors that can occur during MPI communications in this thesis. We analyze only our methods *Sharing* and *Redundant* explained in sections 4.2 and 4.3 respectively for the SDC outcome rates since method *Hashes* explained in section 4.1 relies on direct hash value comparison and cannot recover from failures. We will analyze *Hashes* only in performance section 5.2.

It is important to note that random bit-flip injections may not have an affect on the application if they are too small. We have prepared a script to extract different outcome rates of the injected SDCs using the stdout (standard output) of the runs with the methods and the procedure defined in figure 5.1 in the next page which helps us to decide if the injected error actually causes an SDC or the error is negligible (too small), detected or detected + correctable. Before applying this procedure we have run a reference solution to compare it with the NetCDF [18] outputs (actual results) of the runs where we actually inject the errors in order to see if they have correct results at the end. If an error can only be detected and the application aborts, we count it as a detectable but uncorrectable error (**DUE**). We do not count the injected SDC as

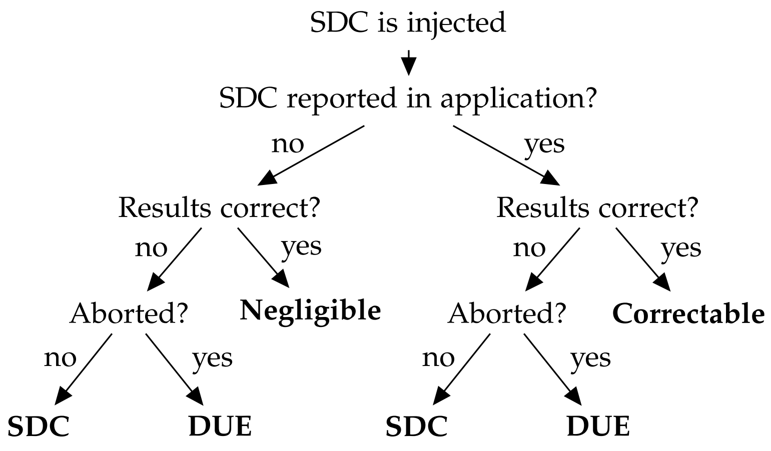

Figure 5.1: SDC Injection Outcome Analysis Procedure.

corrected if it is not reported in our application – e.g. in method *Redundant* an SDC may not be reported by the corrupted rank, but the other team may still have a correct solution at hand. But we count this case as an SDC because no team reports an error and we do not know which result is correct. If the corrupted team reports an error, it might have written faulty results before it manages to detect and recover its state. In this case we say the error is correctable if the other healthy teams' results are correct and the error is reported by the corrupted team. This assumption is also important because we did not implement full recovery as explained in section 4.3.1 and in some cases the corrupted team cannot be recovered completely after an error detection, but we rely on the healthy team's outputs if the error can at least be detected and reported as mentioned above. After this procedure we divide the total number of error injections to the number of total DUE and Correctable outcomes in order to get soft error detection (DUE) and correction (detectable + correctable) rates. We will refer to these rates as DUE and Correctable outcome rates in later sections.

In the following we first inject bit-flips randomly into each of the largest data structures to see the results of the DUE and Correctable outcome rates of our methods, and then in section 5.1.2 we focus on relatively larger bit-flip injections and discuss the parameters that can affect our DUE and Correctable outcome rates.

5.1.1 Random Injection

In order to get accurate outcome rates, we inject errors into the data structures of our application separately. The bathymetry array (*b*) is checked with 1000 error injections and it has a correction rate of 100%. This was expected because we store its initial value redundantly and compare it with its current value in every iteration with the admissibility checks. So, we only consider the other three data arrays (*h*,*hu*,*hv*) and the remaining 8 update arrays.

For our analysis we use the *radialBathymetryDamBreak* scenario, where a 25 meters tall dam breaks in the lower left side of the 2 by 2 km square domain, and we use 200 by 200 cells to simulate the domain. We will later see how different scenarios can affect the resilience in section 5.1.2. If not stated otherwise, we inject bit-flips at simulation-time 50 with a total simulation duration of 100, and relaxation factor set to 100. These parameters will also be changed to analyze their affects on resilience in the later section.

Table 5.1: Outcomes of 10000 random bit-flip injections into each listed array of method *Sharing*. Last random injection covers all the arrays and it is evaluated with 30000 injections.

outcome\array	*h*	*hu*	*hv*	updates	random
Correctable	18.55%	13.83%	14.47%	0%	15.22%
DUE	0.25%	0.13%	0.04%	7.32%	4.36%
SDC	81.21%	86.03%	85.49%	92.68%	80.41%
Negligible	2.95%	3.26%	3.31%	28.17%	19.84%

Table 5.2: Outcomes of 10000 random bit-flip injections into each listed array of method *Redundant*. Last random injection covers all the arrays and it is evaluated with 30000 injections.

outcome\array	*h*	*hu*	*hv*	updates	random
Correctable	18.6%	13.63%	15%	6.09%	19.06%
DUE	0.03%	0.1%	0.04%	0.96%	0.43%
SDC	81.37%	86.26%	84.96%	92.95%	80.52%
Negligible	2.91%	3.32%	2.95%	28.09%	19.56%

In table 5.1 above we see the evaluation of our method *Sharing*. The update arrays could not be corrected right after the error because we could not include DMP criterion for them, so the method could not detect and correct the error right after the injection. Moreover our method *Sharing* uses task sharing, and as we have discussed in sections 4.2.4 and 4.3.2 task sharing can make SDCs propagate and we believe because of the task sharing our method *Sharing* could only detect 7.32% of the errors injected into update arrays but could not correct any of them. In fact, our method *Sharing* could never correct the errors injected into the update arrays as we will also see in the next sections. On the other hand in table 5.2 in the previous page we see that our method *Redundant* could detect and correct 6.09% of the errors injected into update arrays because it employs redundant computation and avoids task sharing between the teams – if an error is detected in later iterations, then it can propagate but still stays in the same team so it can still be corrected using the other healthy team. Generally, method *Redundant* seems to have a higher Correctable but a lower DUE outcome rate than *Sharing*. We believe that this shows how redundant computation can help us to additionally correct some of the injected soft errors that our method *Sharing* could only detect. We also believe that the DUE outcomes in method *Redundant* are the cases where the injected bit-flip leads to an error in the application – e.g. division by zero – causing the corrupted team to halt the program execution abruptly which affects both of the teams.

Since both methods use the same admissibility checks, method *Redundant* should have better Correctable outcome rates because it additionally has redundant and independent computation (no task sharing) and in our bit-flip injection results in this section method *Redundant* has actually a slightly better Correctable outcome rate. The difference between the Correctable outcome rates can also depend on the application and some parameters that we will see in the next section. If we look at the sum of DUE and Correctable outcome rates, both methods have about 20% soft error resilience in this case.

5.1.2 Parameters that can Affect the Soft Error Outcome Rates

In random bit-flip injections in the previous section we did not restrict ourselves while injecting bit-flips – our injections covered all the floating-point number bits, even some rightmost fraction bits that can have a relatively small impact on the simulation. In the following sections we only focus on the leftmost 10 bits of the floating-point numbers in the error injections. This will help us analyze how different parameters such as the relaxation factor, total simulation duration and type of the scenario can affect the SDC outcome rates of our methods. We still run two teams and an error is still injected by flipping a single randomly selected bit per run. While analyzing a parameter other parameters are kept same as our previous set-up in the previous section.

Relaxation Factor

In section 4.2.1 we have introduced the relaxation factor in the relaxed DMP criterion. Table 5.3 in the next page shows how the Correctable outcome rates of our methods

Table 5.3: Correctable Outcome Rates of 2000 bit-flip injections into the leftmost 10 bits of randomly selected floating-point numbers, that are separately injected into the listed arrays with different relaxation factors (r).

	Sharing				*Redundant*			
$r \backslash array$	h	hu	hv	updates	h	hu	hv	updates
80	59.44%	43.37%	50.44%	0%	60.34%	45.31%	50.7%	15.76%
100	59.95%	43.47%	47.07%	0%	58.97%	43.91%	47.74%	15.34%
10000	39.69%	29.91%	29.24%	0%	39.96%	28.1%	30.62%	11.21%
1000000	34.9%	22.07%	25.32%	0%	37.68%	24%	25.46%	10.16%

can change as we increase the relaxation factor. The lower we set the relaxation factor, the higher the sensibility of the relaxed DMP criterion is, and therefore the higher the Correctable outcome rates are. However as explained in section 4.2.1 the DMP may give us false positives if we set the relaxation factor too small. In this particular scenario (*radialBathymetryDamBreak*) we could for example not set the relaxation factor to 50 because the simulation violated the DMP even without an error.

Simulation Duration

Another important parameter that can affect the SDC outcome rates of our methods is how long the scenario has to be simulated. In tables 5.4/5.5 below and in the next page we see how the DUE and Correctable outcome rates of our methods changes as we increase the total simulation duration. It is important to note that we do not change the simulation time in which we inject the error.

Table 5.4: DUE outcome Rates of 5000 bit-flip injections at simulation time 50 into the leftmost 10 bits of randomly selected floating-point numbers, that are separately injected into the listed arrays with different total simulation durations (t).

	Sharing				*Redundant*			
$t \backslash array$	h	hu	hv	updates	h	hu	hv	updates
50	0.06%	0.25%	0.17%	1.52%	0%	0.27%	0.16%	1.57%
100	1.35%	0.24%	0.2%	17.63%	0.01%	0.24%	0.18%	1.52%
200	1.13%	0.31%	0.14%	16.06%	0.02%	0.35%	0.31%	1.64%
400	1.24%	0.41%	0.14%	15.93%	0%	0.21%	0.18%	2.02%

Table 5.5: Correctable outcome Rates of 5000 bit-flip injections at simulation time 50 into the leftmost 10 bits of randomly selected floating-point numbers, that are separately injected into the listed arrays with different total simulation durations (*t*).

	Sharing				*Redundant*			
t\array	*h*	*hu*	*hv*	updates	*h*	*hu*	*hv*	updates
50	60.39%	43.97%	47.38%	0%	58.72%	43.25%	46.98%	0%
100	59.95%	43.47%	47.07%	0%	58.97%	43.91%	47.74%	15.34%
200	58.37%	43.78%	47.97%	0%	59.99%	44.16%	48.29%	15.18%
400	58.1%	44.06%	47.95%	0%	60.31%	44.7%	48.34%	14.89%

One can see that if the simulation stops right after the injection at total simulation time 50, then even method *Redundant* cannot help us correct any errors injected into the update arrays because the error cannot be detected and recovered in later iterations – the error is injected in the last iteration. In this case we get similar SDC outcome rates for both of our methods. We however see that method *Redundant* starts correcting errors injected into the update arrays – this again means that the detected error stays in the corrupted team and is correctable using the other healthy team – and method *Sharing* increases its DUE outcome rate of the injected errors as the simulation duration exceeds the simulation time in which the error is injected. The DUE outcome rates of the water momentum arrays (*hu*,*hv*) are quite similar which may indicate that these outcomes were the natural detections of the application causing a halt and the errors could not be detected by the admissibility checks, whereas the DUE outcome rates of the water height array (*h*) and the update arrays are quite different which might be because that the propagated error could be detected in later simulation times by the admissibility checks however could not be corrected in *Sharing* and has resulted in DUE and it could be recovered in *Redundant* using the healthy team and resulted in Correctable outcome. The increasing Correctable outcome rates of *Redundant* also supports this idea but we need more evaluations to get more accurate rates. This may indicate that soft error resilience of our methods can even depend on the different data structures of our application.

Scenario

The last parameter that we analyze is the type of scenario that we simulate. In table 5.6 in the next page we compare our standard scenario *radialBathymetryDamBreak* (***A***) with the scenarios *splashingPool* (***B***) and *seaAtRest* (***C***). In contrast to the setup in *A*, the other scenarios have relatively little wave propagation. In fact, scenario *C* does not have any water momentum at all. The water is at rest position where there are no waves. Similar to *C*, scenario *B* reaches the rest position at the end of the simulation and the water height changes relatively less than scenario *A* during the simulation. We have calculated the mean and the standard deviation of the water height array *h* in both *A* and *B*. The standard deviation is around 5.5 in *B* whereas it is around 19.9 in *A*. The mean value of the water height is around 15.1 meters in scenario *A* and 245 meters in scenario *B*, and the mean value of the water momentum in both directions is about 6.5 in *A* and 171.5 in *B*. Water height is always 10 meters in scenario *C* and there is no water momentum.

Table 5.6: Correctable outcome rates of 2000 bit-flip injections into the leftmost 10 bits of randomly selected floating-point numbers, that are separately injected into the listed arrays in 3 different scenarios: A = *radialBathymetryDamBreak*, B = *splashingPool*, C = *seaAtRest*.

	Sharing				*Redundant*			
$t\backslash array$	*h*	*hu*	*hv*	updates	*h*	*hu*	*hv*	updates
A	59.95%	43.47%	47.07%	0%	58.97%	43.91%	47.74%	15.34%
B	89.77%	89.39%	90.28%	0%	99.95%	90.1%	90.5%	15.99%
C	58.25%	0%	0%	0%	61.65%	0%	0%	0%
C (r=0)	100%	100%	100%	0%	100%	100%	100%	99.8%

We first compare scenarios *A* and *B*, and see that the errors injected into scenario *B* could be corrected with a much greater rate. Since the values in the arrays in scenario *B* are bigger than the values in the arrays in scenario *A*, the injected errors into the leftmost 10 bits of the selected floating-point numbers that includes mostly the exponent bits have a larger impact and therefore make the DMP criterion more sensible when using the same relaxation factor (*r = 100*). Additionally, having a less standard deviation in the array *h* can also affect the sensibility of the DMP because the previous values in the arrays can be close to the current computed values. Therefore we get an almost 100% correction rate in method *Redundant* for the *h* array in scenario *B*. In scenario *C* almost all the error injections into momentum arrays (*hu*,*hv*) are negligible except for the leftmost second bit which changes the value from 0 to 2. We also count the sign-bit flip as negligible in this case because it changes the value from 0 to -0. If we use the same relaxation as we did for the other scenarios, we cannot detect the errors that are injected into the momentum arrays (*hu*,*hv*) that changes the value from 0 to 2. And since the water height in scenario *C* is similar to the water height in *A*, they have similar error correction rates while using the same relaxation factor (*r = 100*). But scenario *C* can also be run using the strict DMP criterion (*r = 0*) because the arrays remain constant, and with the strict DMP we can also detect and correct the other errors that could not be detected with a relaxed DMP. This is not possible with scenario *A* or *B* where we have wave propagation.

5.1.3 Limitations of our Resilience Methods

We have seen the effect of task sharing on our soft error correction rates, especially for the errors that were injected into the update arrays. One should keep this in mind while using our method *Sharing*. The injected errors that could not be immediately detected after the injection may eventually be detected in later iterations. Using the task sharing mechanism however limits our ability to recover our data structures in this case. In *Redundant* we try to overcome this situation by using redundant computation by running the teams independently. Therefore it has higher Correctable and lower DUE outcome rates overall.

The relaxation factor in our relaxed DMP criterion cannot be set to lower values that are violating the DMP criterion even without an error injection. In our case we use wave propagation simulation and the minimum relaxation factor that can be set depends on the changes in the water height (*h*) and momentum (*hu,hv*) arrays from one time step to another, which are highly scenario dependent.

5.2 Performance Comparison

Our code was compiled using the gcc version 8.4.0 and the MPI compiler wrapper from ULFM version 4.0.2u1. We use the same *radialBathymetryDamBreak* scenario explained in the previous section with 3000 by 3000 cells and a total simulation duration of 30 seconds. The decomposition factor is set to one if not stated otherwise. All the performance tests in this section are evaluated on the Linux cluster segment CoolMUC-2 at Leibniz Supercomputing Centre (**LRZ**). The cluster provides 812 Haswell-based nodes that each run 28 cores at the nominal frequency of 2.6 GHz and with 64 GB of memory.

In the following we compare the performance of our soft error resilience methods explained in section 4 with a baseline model that has no integrated error resilience. We will refer to the baseline model as **NoRes** which uses the computation-loop defined in figure 3.2 without the TeaMPI library, where ranks divide the simulation domain into blocks and each rank holds a single domain block. We focus on single-node and multi-node performances in the following sections 5.2.1 and 5.2.2 respectively.

5.2.1 Single-Node

We use 16 processes of a single node and run the baseline model and our methods to get their wall-clock times. Additionally we profile them using Hotspots Analysis of the Intel VTune Profiler 2020 Update 3 to list the CPU-time percentages of the most important function calls in our application and resilience methods to get the results depicted in figure 5.2 below.

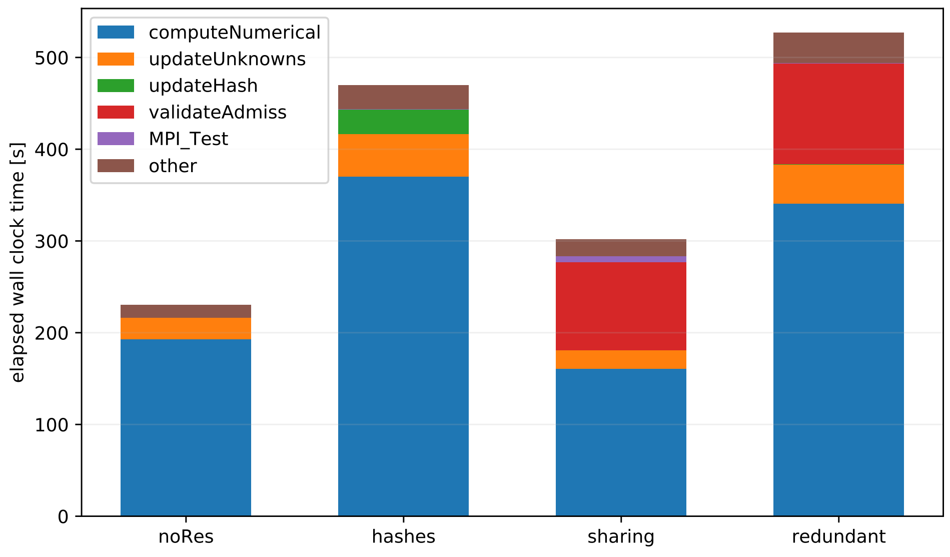

Figure 5.2: Profiling on a single node.

NoRes has the fastest possible execution time with 230.5 seconds. Our method *Hashes* uses replication and redundant computation and therefore it has taken twice the runtime of *NoRes* for the compute and update function calls that are used to compute and write results for the next simulation time. Additionally it has the overhead of its soft error detection mechanism that needs to update the hash values. In total it has a runtime of 469.5 seconds. Similarly, our method *Redundant* has a runtime of 527 seconds because it makes use of replication and redundant computation, and uses admissibility checks to validate the blocks' states. About 61% of the overhead of our admissibility criteria comes from the relaxed DMP criterion, and we see that hashing can be executed a lot faster than our admissibility checks. Our method *Sharing* improves our *Redundant* method's runtime by about 43% with a runtime of 301.6 seconds which is only about 31% above the method *NoRes*. Our task sharing mechanism manages to reduce the

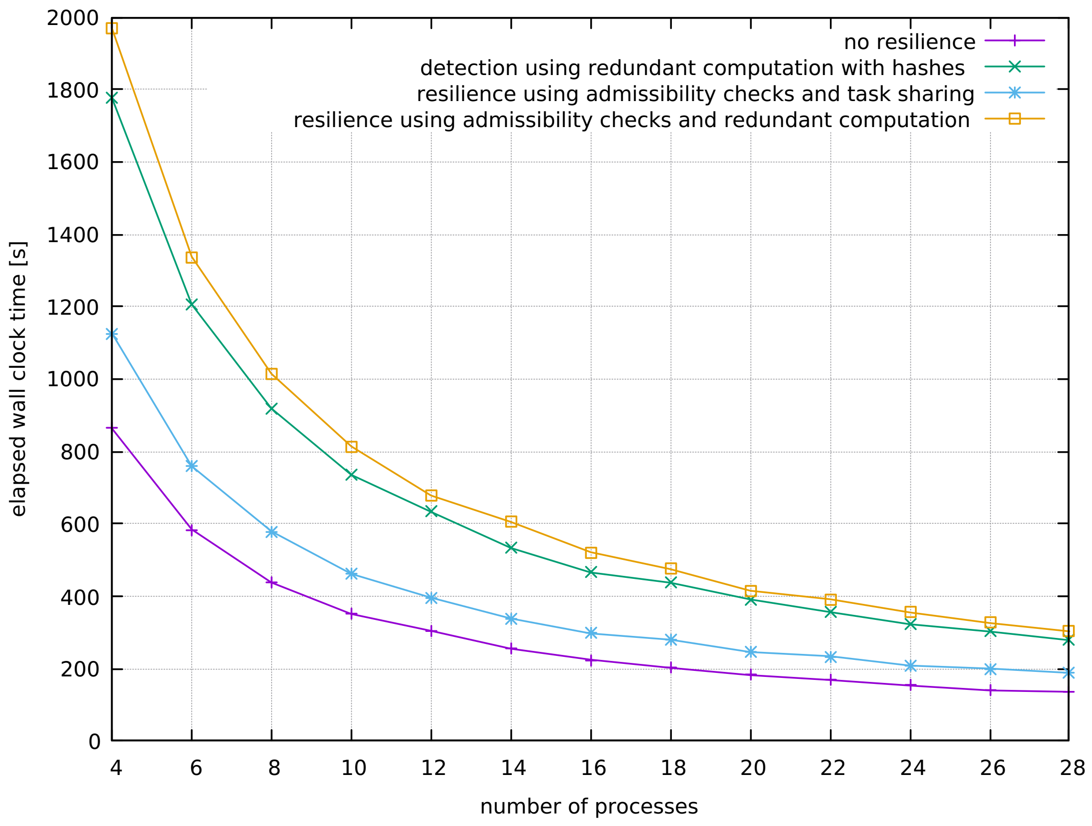

Figure 5.3: Strong scaling on a single node.

redundant computation overhead while still utilizing the replication which is needed for the recovery of the corrupted ranks. It only brings the additional overhead of sharing the tasks which is almost negligible in this single-node case.

Strong Scaling

We have also evaluated strong scaling comparison on a single-node environment to see our resilience methods scaling in figure 5.3 above from 4 up to 28 MPI processes. In the following section we will experiment with multiple nodes and compare its results with this single-node strong scaling.

5.2.2 Multi-Node

Similar to what we have done in our single-node performance analysis, we first analyze the methods' runtimes in a multi-node environment using 2 nodes and 8 processes per node with a total 16 MPI processes.

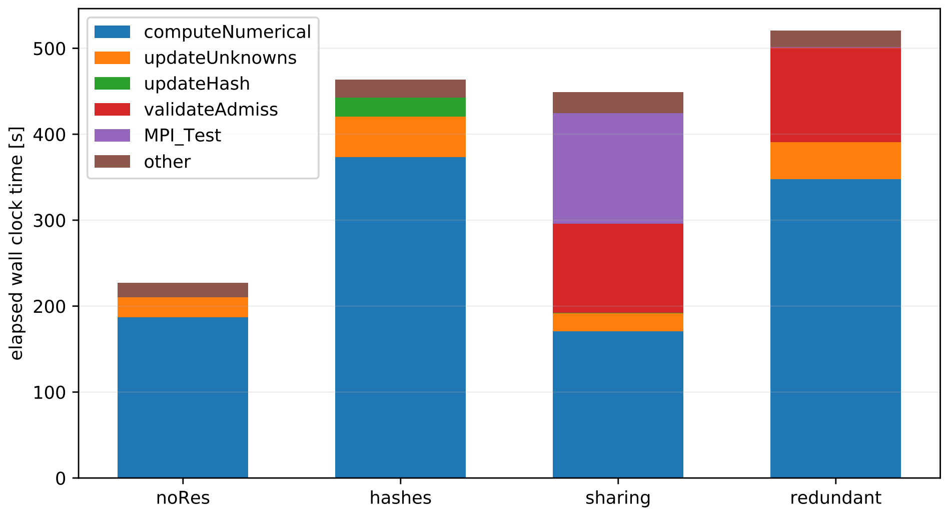

Figure 5.4: Profiling on two nodes with teams on the same node.

For the analysis we have two options while using multiple nodes. We can either map the ranks of the same team onto the same node, or map the replicated ranks onto the same node. We first look into the case where the teams are mapped onto the same node in figure 5.4 above. All the methods except our method *Sharing* seem unaffected, but our method Sharing now has a runtime of 448.7 seconds which lies slightly under the runtime of our method *Hashes*. The reason for this is the increased *MPI_Test* execution time which indicates our task sharing communication overhead. Ranks call *MPI_Test* while waiting for their ongoing MPI communication between their replicas to be completed (4.2.2). And since teams are one the same node and replicas are on different nodes, and the inter-node communication is a lot slower than the intra-node communication we get a much higher execution time compared to the single-node case.

We can reduce this overhead by mapping the replicated ranks onto the same node and as a consequence use task sharing (heavy MPI communication) on this node only, which is evaluated and plotted in figure 5.5 in the next page. In this case we get a very

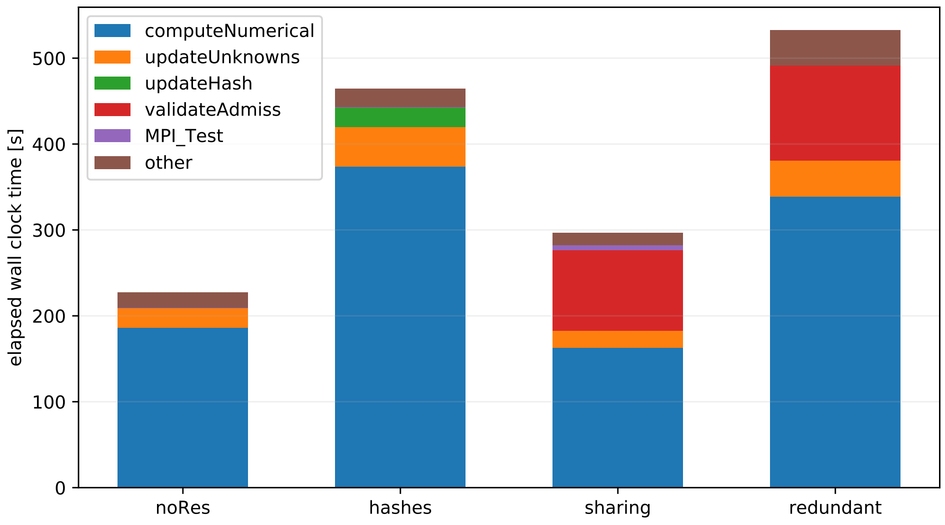

Figure 5.5: Profiling on two nodes with replicas on the same node.

similar result as the single-node run because the MPI communication between replicas is much more expensive than the MPI communication within teams if task sharing is used and by mapping the replicas onto the same node we can avoid expensive inter-node MPI communication.

Strong Scaling

In our multi-node strong scaling analysis we test with 2 nodes and from 2 processes per node up to 16 processes per node with a total of 32 MPI processes. For the strong scaling comparison on multiple nodes we again have the option of mapping the teams or replicas onto the same node. We first look into the expensive case for our method *Sharing* where teams are on the same node in figure 5.6 in the next page.

As we have already seen in our previous analysis, our method *Sharing* which employs our task sharing mechanism suffers from this option. The cyan line that represents our method *Sharing* loses its performance advantage that comes from its task sharing mechanism after about 8 processes per node, and becomes even less efficient than our redundant solutions in terms of performance. This decline in performance is because of the increased communication overhead. We have noted in section 3.5.2 that the tasks need to be compute-heavy in order for the task sharing to compensate for the

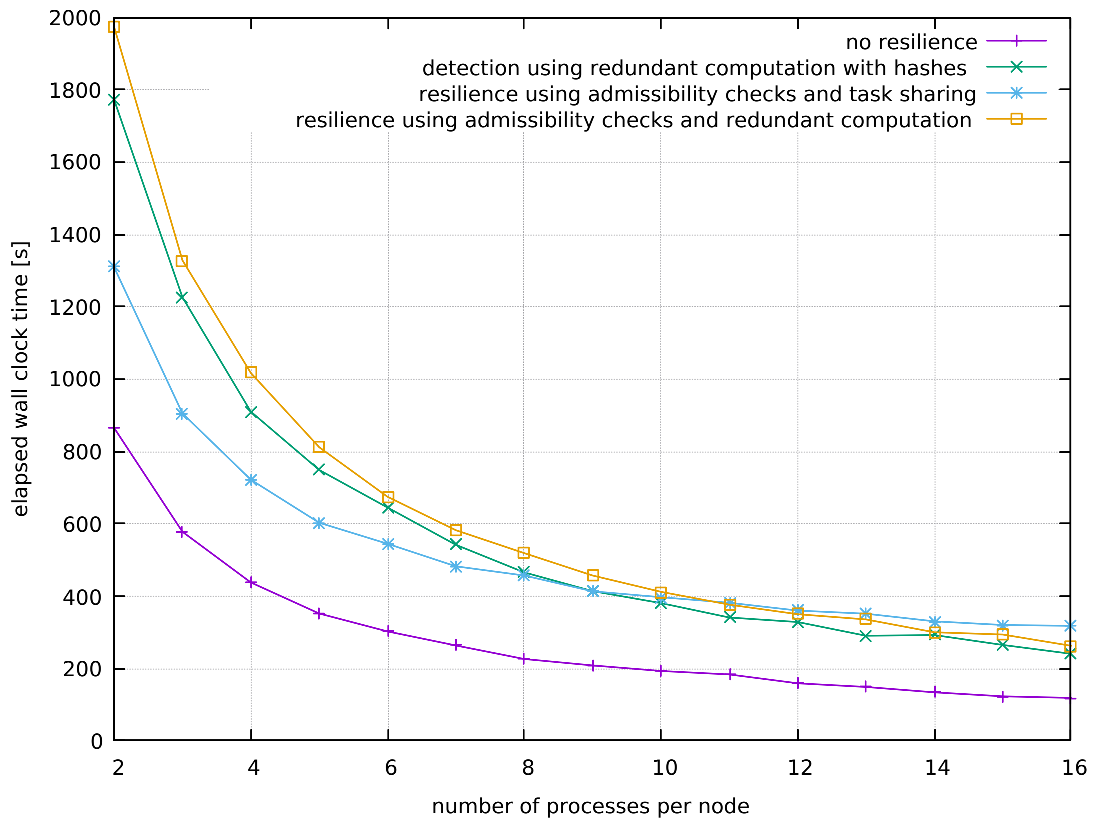

Figure 5.6: Strong scaling on two nodes with teams on the same node.

additional MPI communication overhead that it brings. In strong scaling the more processes we conduct in the experiment, the more the domain gets divided per process, and because the domain gets smaller per rank, the actual computation time of a task gets also smaller per rank which makes the tasks not compute-heavy enough. Instead of waiting for the MPI communication of the task sharing to be completed, computing the tasks redundantly becomes more efficient in this case.

We can again mitigate the performance problem explained above by mapping the replicas on the same node and use the intra-node communication for the more expensive MPI communications. The plot of this case is depicted in figure 5.7 in the next page. We have only included even numbers for the number of processes per node because we could not map all the replicas onto the same node otherwise. We again get a result that is very similar to the single-node case. Other methods than *Sharing* seem to be unaffected from the rank mapping options in this setup as well.

In multi-node performance evaluations we have solved our performance problems

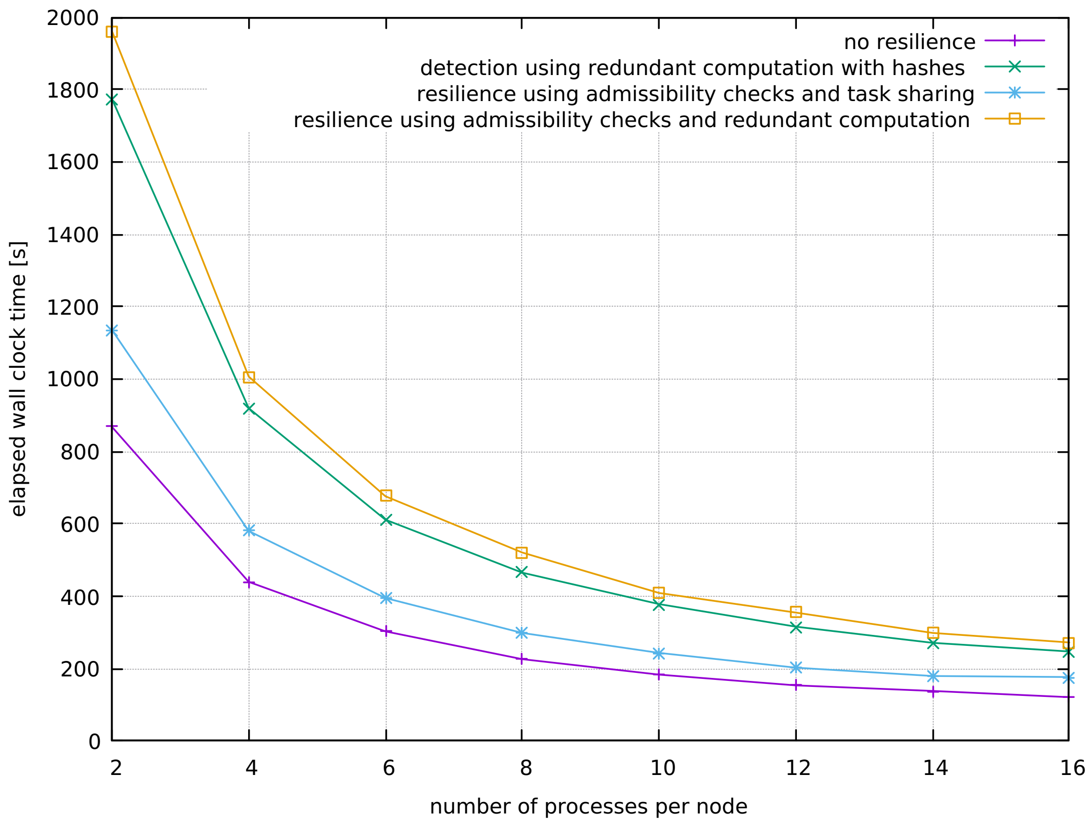

Figure 5.7: Strong scaling on two nodes with replicas on the same node.

for our method *Sharing* by employing a specific rank-to-node mapping. As we have mentioned in section 3.6.2, there is also another possible approach of increasing the decomposition factor which may help us improve the performance depending on the implementation of our methods.

5.2.3 Decomposition Factor Scaling

In this section we only focus on our method *Sharing* and the option of mapping teams onto the same node. We see its runtime in figure 5.8 in the next page, evaluated with different decomposition factors further dividing the domain and creating more primary and secondary blocks. We do not notice a significant change in the runtime. If we look into our task sharing implementation explained in section 4.2.2, we notice that the only useful computation that we have while waiting for the ongoing task sharing MPI communication to be completed is checking the admissibility of the secondary

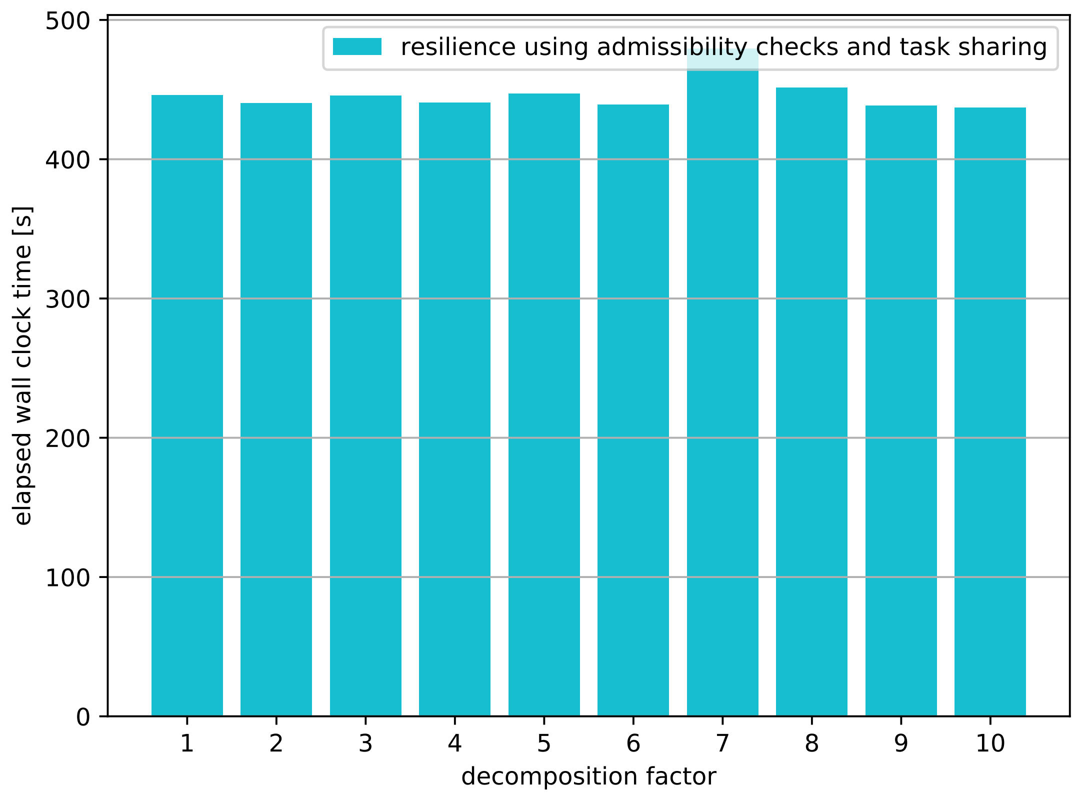

Figure 5.8: Decomposition factor scaling using 3000 by 3000 cells.

blocks after receiving (4.5). One thing we could not do e.g. was posting the task sharing receive calls and then computing, updating and sending the primary blocks iteratively while waiting for the posted receives to be completed, because we have to first agree on an admissible time step size and therefore we must first compute all of our primary blocks before updating them. If we increase the number of cells in x and y directions up to 6000 then we get the result in figure 5.9 in the next page. Here we notice that the only performance improvement comes from adding an extra decomposition factor, however the rest looks similar to our previous result. We believe that this performance improvement is related to the send buffer sizes and the protocols that are used within the MPI. For smaller message a different protocol is used and therefore we can see its difference only in the first additional decomposition factor. If we further increase the decomposition factor we still cannot see that our computations during the ongoing MPI communication can improve the runtime with our current implementation scheme.

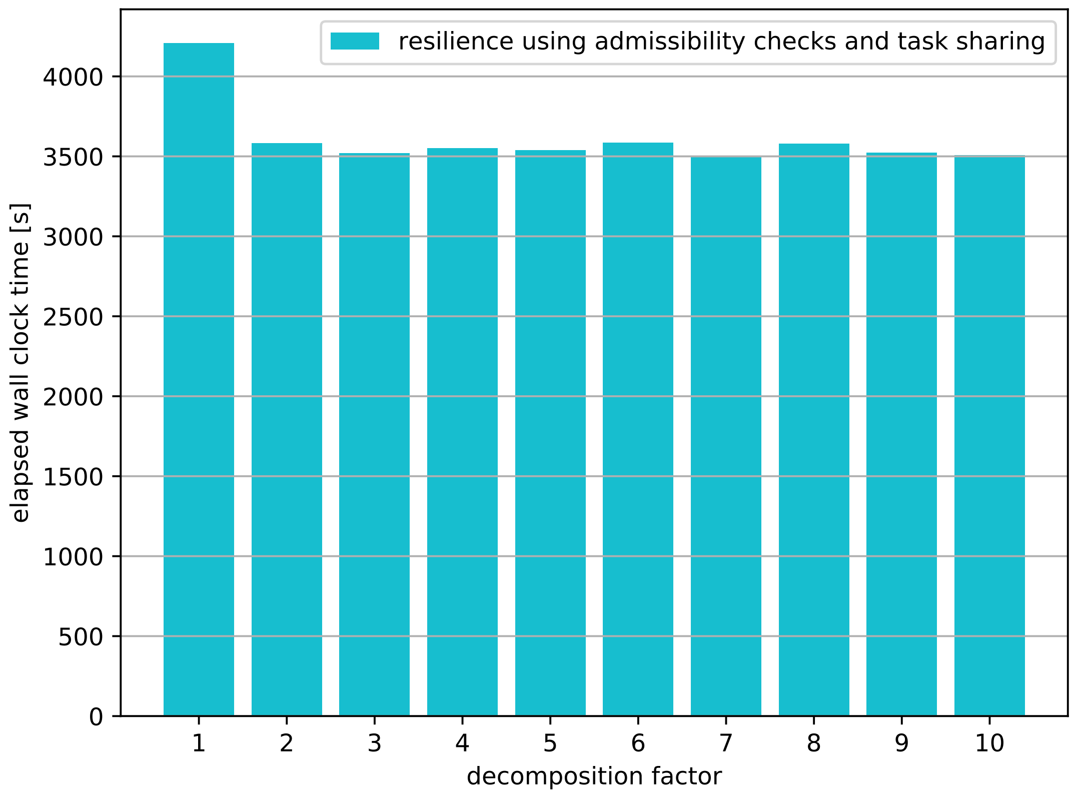

Figure 5.9: Decomposition factor scaling using 6000 by 6000 cells.

6 Further Work

6.1 Improvement

Our method *Hashes* that we introduce in section 4.1 can only provide an early SDC detection. The method can be improved to support more than two teams with a voting mechanism similar to the mechanism defined in RedMPI (2.1) to identify healthy teams within the TeaMPI library and recover the detected corrupted teams. This would additionally provide recovery to the method.

A second improvement could be done by adding additional admissibility criteria to our admissibility checks, especially for the update arrays. The previous values in the update arrays that are computed in previous simulation times could maybe be saved and used for a prediction or guess of their current state, similar to the DMP that we use for the data arrays. In our SWE applications there are more update arrays than the data arrays in total, so adding an admissibility criterion for the update arrays could greatly improve the DUE and Correctable outcome rates of our methods *Sharing* and *Redundant* in random bit-flip injection evaluations.

Our reporting and recovery system for the primary blocks (4.8) as well as for the secondary blocks (4.9) cannot handle some soft error cases explained in sections 4.2.3. We have e.g. not implemented any recovery mechanisms for the multiple soft errors that can corrupt the application at different replicas at the same time. Also, our secondary block recovery mechanism can be improved to recover from the cases where a soft error corrupts a primary block of a rank after its validation and causes the admissibility criteria to fail in the secondary block validation right after the task sharing in our method *Sharing*. This situation can be handled e.g. by storing the received blocks from the task sharing without overwriting the current states of the secondary blocks and trying to recover using them in case a possible SDC is detected. It is also possible to integrate additional mechanisms to our recovery like bit-wise comparison of the blocks to make sure that the replicas hold the exact same block information after the recovery.

One study did use CR together with replication as explained in section 2.2. Application level CR mechanisms or different communication mechanisms that can communicate the "in-memory" checkpoint via MPI can similarly be combined with our methods to recover a whole team.

Lastly, one important thing we have noticed is that in some cases the injected error can be spread to other blocks that are held in different ranks of the same team, and the recovery of a corrupted rank should therefore include the recovery of all of its team. This is especially important for our method *Redundant* because it has the potential to recover from such cases. However in our recovery mechanisms we did not include the recovery of all the ranks in the corrupted teams, but we only recover the corrupted ranks that could notice the error using our admissibility criteria. So, their recovery does not depend on each other. Yet, we have noticed this too late and did not have the time to extend our recovery mechanism. Fortunately this has no effect on our soft error outcome rate evaluations in section 5.1 because our method *Sharing* cannot recover if it does not immediately detect the injected error anyways because of the task sharing, and our method *Redundant* runs the teams independent from each other and this just affects the corrupted teams' current states and does not affect the healthy teams' outputs. However, extending this is important for our method *Redundant* since it could guarantee us a complete recovery of the whole team in case an error could eventually be detected by our admissibility checks at any iteration. Additionally, a recovered team could e.g. later help us recover other corrupted teams. Therefore this extension needs to be added to our recovery mechanism by e.g. modifying our primary/secondary block reporting and recovery system to recover all the ranks in a team in case any of them is potentially corrupted.

It is hard to cover all the error cases especially when it comes to soft error resilience since soft errors can typically cause SDCs which do not directly raise an error in the application. Therefore our methods should definitely be tested with more random bit-flip injections – also at different places in the code – for better identification of their weaknesses which could then be improved accordingly.

6.2 Hard Error Resilience

Providing hard error resilience was out of scope for this thesis, however our method *Hashes* can easily be integrated to also provide additional hard error resilience using the TeaMPI library by adding an additional heartbeat pair that is bound to wall-clock time. It should then be able to detect any slowing or failing rank. Also we have already prepared our methods *Sharing* and *Redundant* for hard error integration using the TeaMPI-heartbeats but they need to be tested and evaluated using different set-ups and techniques in order to make sure that our soft error resilience does not intervene with hard error resilience mechanisms.

6.3 Testing on Different and Larger Applications

It would also be very interesting to test our methods on larger and different applications. ExaHyPE [16] can e.g. use different solvers that has different schemes and can maybe help improving our multi-node performances by studying the decomposition factor. Our methods can also be tested with different simulation types e.g. in areas like seismology and astrophysics to study the effect of different simulation types on the soft error outcome rates.

7 Conclusion

We have introduced three different methods for soft error resilience, which are implemented for the simulation software SWE using the TeaMPI library. Each method makes use of replication by employing two teams. The resilience in our first method (*Hashes*) depends on the direct hash value comparisons of the results and can detect any SDC that can occur in the largest data structures of our application, however it cannot provide error correction and always needs to be restarted after an error detection. Our second (*Sharing*) and third (*Redundant*) methods can provide both soft error detection and recovery. They both use an *a-posteriori* approach to detect a potential error by applying predefined admissibility checks which are not always accurate with their predictions. In order to make sure that no potential error is detected at other replicas, they exchange byte-size reports and can recover each other by sending their block information via MPI communication in case a potential error is detected. Our method *Sharing* additionally employs a task sharing mechanism between its replicas, which can reduce the compute costs by up to 43%. Our methods *Hashes* and *Redundant* however make use of redundant computation and therefore have worse runtimes than method *Sharing*. And since method *Redundant* employs admissibility checks which are more expensive than hashing, it has the worst runtime overall.

We have also evaluated our methods *Sharing* and *Redundant* by injecting randomly selected bit-flips into the four largest data arrays (*b,h,hu,hv*) and eight net-update arrays of a wave propagation (*radialBathymetryDamBreak*) scenario at a fixed place in the source code and achieved 4.36%/0.43% DUE and 15.22%/19.06% Correctable outcome rates. Our methods could detect and correct all the errors injected into the array *b* since it should stay constant, and if we only focus on relatively larger errors that are injected only into the left 10 bits of randomly selected floating-point numbers we can get a 50.2% Correctable outcome rates for the data arrays for both of the methods. Our method *Sharing* could however never correct the errors that were injected into the update arrays since the task sharing makes them propagate to other teams, which disables the recovery for the errors that are detected in later simulation times. In this case the redundant computation in method *Redundant* can help us achieve a Correctable outcome rate of about 15.34% of the errors that are injected into the update arrays even though we do not have any sensible admissibility criterion for the update arrays. Any additional admissibility criterion for the update arrays could improve our methods' correction rates. Also our recovery mechanisms need to be improved to provide better soft error detection and recovery since they cannot recover from some soft error cases that we are aware of. We have also found out that the SDC outcome rates of our methods can really depend on a lot of parameters such as the relaxation factor in our relaxed DMP criterion, total simulation duration, and the simulated scenario type. In our analysis of different parameters we have reported higher DUE and Correctable outcome rates for lower relaxation factors and for total simulation durations that are greater than the simulation time in which the error was injected. We have also seen that the type of the simulation scenario can have a significant effect on the correction rates of our methods.

List of Figures

List of Tables

Bibliography

[1] J. Ahrens, B. Geveci, and C. Law. "ParaView: An End-User Tool for Large-Data Visualization." In: *The Visualization Handbook*. 2005.

[2] R. Baumann. "Radiation-induced soft errors in advanced semiconductor technologies." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316.

[3] W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, and J. J. Dongarra. "Post-failure recovery of MPI communication capability: Design and rationale." In: *IJHPCA* 27.3 (2013), pp. 244–254.

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

[5] D. Eastlake and P. Jones. *RFC3174: US Secure Hash Algorithm 1 (SHA1)*. Tech. rep. USA, 2001.

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

[7] 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: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12.

[8] 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: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–12.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation." In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

[10] T. Herault and Y. Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st. Springer Publishing Company, Incorporated, 2015.

[11] A. Hölzl. "Integrating TeaMPI with ULFM for Hard Failure Tolerance in Simulation Software." Bachelor's Thesis. Technical University of Munich, 2020.

[12] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design." In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, pp. 111–122.

[13] D. Montezanti, A. De Giusti, M. Naiouf, J. Villamayor, D. Rexachs, and E. Luque. "A Methodology for Soft Errors Detection and Automatic Recovery." In: *International Conference on High Performance Computing Simulation (HPCS)*. 2017, pp. 434–441.

[14] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 4.0*. available at: `http://www.mpi-forum.org` (access: June 2021). June 2021.

[15] S. Mukherjee, J. Emer, and S. Reinhardt. "The soft error problem: an architectural perspective." In: *11th International Symposium on High-Performance Computer Architecture*. 2005, pp. 243–247.

[16] 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).

[17] A. Reinarz, J.-M. Gallard, and M. Bader. "Influence of A-Posteriori Subcell Limiting on Fault Frequency in Higher-Order DG Schemes." In: *IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2018, pp. 79–86.

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

[19] 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.

[20] B. Schroeder and G. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *Dependable and Secure Computing, IEEE Transactions on* 7 (Jan. 2011), pp. 337–351.

[21] S. Schuck. "Integrating Task Sharing with Team Recovery for Hard Failure Tolerance in teaMPI and SWE." Bachelor's Thesis. Technical University of Munich, 2021.

[22] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. "Addressing Failures in Exascale Computing." In: *The International Journal of High Performance Computing Applications* 28.2 (2014), pp. 129–173.