Fault Tolerant Approaches for Parallel Geometric Multigrid

Georgios Diamantopoulos

Master Thesis
Fault Tolerant Approaches for Parallel Geometric Multigrid

Georgios Diamantopoulos
Master Thesis

Aufgabensteller: Prof. Dr. U. Rüde
Betreuer: Dr.-Ing. B. Gmeiner, F. Shahzad
M.Sc., M. Huber M.Sc.
Bearbeitungszeitraum: 30.4.2015 – 30.10.2015
Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30. Oktober 2015

..............................................
Abstract

Supercomputers are moving towards the Exascale era. The computing systems of that era are expected to have millions of cores and the users will be able to benefit from the enormous amounts of computational power that these machines will have. However as the number of cores is increasing, the probability for a failure to occur during a simulation is also increasing and it might even be inevitable to have a failure in a large scale simulation. Therefore HPC applications of the Exascale era are expected to be fault tolerant. This means that both the software and the algorithms should be able to show resilience against a fault, so they can recover and continue after a fault occurs. This work discusses how this can be implemented on a solver of the Poisson equation that uses a Finite Element Method on a semi-structured grid with a geometric multigrid method for the linear system. On the software side the User-Level Failure Mitigation (ULFM) library is used as a fault tolerant version of MPI. On the algorithm side checkpointing as well as Algorithm Based Fault Tolerance is used in order to recover the lost data.
## Contents

1 Introduction
   1.1 Motivation .......................................................... 6
   1.2 Problem description .................................................. 6
   1.3 Related work ........................................................ 7

2 Background
   2.1 Hierarchical Hybrid Multigrid ...................................... 8
      2.1.1 Disadvantages of unstructured grids ......................... 8
      2.1.2 The concept of HHG ............................................ 8
      2.1.3 Data structures and algorithms ............................... 9
      2.1.4 Parallel HHG .................................................. 12
   2.2 MPI background ..................................................... 13
      2.2.1 Process Groups and Communicators in MPI .................. 13
      2.2.2 Functions for Process Groups ................................ 14
      2.2.3 Functions for Communicators ................................. 15
      2.2.4 Dynamic Process Creation .................................... 16
   2.3 User Level Failure Mitigation .................................... 16
      2.3.1 Functions of ULFM ............................................. 17
      2.3.2 Error codes of ULFM .......................................... 18

3 Building a Fault Tolerant Poisson Equation Solver 19
   3.1 Introducing spare/spawned processes ............................. 19
      3.1.1 Communicator split for spare processes ..................... 19
      3.1.2 Spawned processes ............................................ 20
      3.1.3 Setup of the spare/spawned process ......................... 20
   3.2 Failure Detection .................................................. 21
   3.3 Restoring Communication .......................................... 22
      3.3.1 The Spare Processes Approach ............................... 22
      3.3.2 The Respawn Approach ....................................... 23

4 Data recovery 25
   4.1 Data recovery using Checkpoint-Restart ........................ 25
      4.1.1 Node-level checkpoint-restart using the SCR library ...... 26
   4.2 Data recovery using Algorithm Based Fault Tolerance .......... 27

5 Benchmarking and Comparisons 28
   5.1 Benchmarking system ............................................. 28
   5.2 Problem specification ............................................. 28
   5.3 Benchmarking results - No failure ............................... 28
   5.4 Benchmarking results - Without data recovery .................. 29
   5.5 Benchmarking results - Checkpoint-Restart ..................... 31
   5.6 Benchmarking results - Algorithm Based Fault Tolerance .... 33
   5.7 Remarks about the robustness of the code ...................... 33

6 Conclusions and further work 35
   6.1 Conclusions ....................................................... 35
   6.2 Suggestions for further work ..................................... 35

A Code extracts 36
1 Introduction

1.1 Motivation

The High Performance Computing (HPC) world is currently moving towards the Exascale era. Supercomputer manufacturers are constantly making the effort to create more powerful computers that can respond to the demands of modern Scientific Computing. It is only a matter of time to produce the first Supercomputers that will be able to perform computations in the scale of Exaflops. One of the significant characteristics of an Exascale platform is that the number of cores is expected to exceed one million. According to the Top500 list\(^1\), released on June 2015, there already exist two supercomputers that have more than a million cores. Moreover, all systems in the list have thousands of cores although only the top 68 have achieved a maximal performance of at least one Petaflop per second. As the number of cores in HPC platforms is constantly increasing it will soon be common for a supercomputer to have hundreds of thousands or even millions of cores. Having systems with such enormous numbers of cores increases the chance of having a failure during a simulation.

Faults that cause these failures can occur for different reasons and they can be classified in two categories. The first category is the fail-stop errors or hard faults, that lead to the permanent loss of a processes. These fault can occur for many reasons such as a permanent node crash or an error in the network of the cluster, etc. The second category is the fail-continue or soft errors. When errors of this category occur the program continues after the fault but the execution is affected because of bit-flips. In this work hard faults are concerned as the cases of failure, although many of the techniques presented here can also be used for soft faults.

All these failures can make the program abort and lose all the information of the simulation steps that have been performed until that point. According to even the most optimistic assumptions, the probability of having a failure during the next hour of simulation on an Exascale system is approaching 1 (see [2]). Hence, it becomes essential for an application to be able to recover and continue after a failure. Only applications that can tolerate faults will be able to exploit the whole computing power that an Exascale system will be able to offer. Therefore, fault tolerance is today an important topic in High Performance Computing and the development of fault tolerant software is becoming essential for large scale HPC applications.

In order to face the challenge of creating fault tolerant applications there are already techniques and algorithms that can be used in order to achieve it. However, the most widely used parallel computing standard, the MPI (Message Passing Interface), is still vulnerable to faults. A fault that might occur in an MPI application will most probably lead to an MPI-abort and an unsuccessful termination of the application. For this reason the User-Level Failure Mitigation (ULFM) library has been proposed as a fault tolerant version of the MPI standard.

This work aims to use ULFM in order to introduce fault tolerance approaches to the Terra-Neo project\(^2\). Terra-Neo is an Exascale Earth Mantle Modeling Framework. An accurate simulation of the earth mantle requires extremely large grids, fine spatial and temporal resolutions as well as many time steps. Therefore, earth mantle models are large scale applications that require enormous amounts of computational power. Terra-Neo aims to take full advantage of the computational power that Exascale systems will be able to offer. In order to achieve this, it should not be vulnerable to faults that might lead to waste of computational effort. The objective of this work is to equip Terra-Neo with fault tolerant features that can make it recover and continue after a failure.

1.2 Problem description

As this work is focused on fault tolerance, a simple problem has been chosen as the working problem. This is the Poisson equation in 3D with inhomogeneous Dirichlet boundary conditions:

\[ -\Delta u = f \text{ in } \Omega = (0,1)^3, \quad u = g \text{ on } \partial\Omega \]  \hspace{1cm} (1)

Although it is a simple problem it usually occurs after the temporal discretization of problems like the Stokes equations and many other problems. The solution of a Poisson problem is usually the

\(^1\)http://www.top500.org/
\(^2\)http://terraneo.fau.de/
most time-consuming and computationally intensive part of a simulation. Thus, the fault tolerant concepts that are presented here, can be easily extended to more complicated problems.

Terra-Neo uses the notion of the Hierarchic Hybrid Grid (HHG). The main idea behind HHG is to combine the advantages of the Finite Elements Method and the geometric multigrid solver. Multigrid solvers show excellent time-efficiency in large scale problems because they have the advantage that the solution of the problem is calculated in the coarsest grid where less computational effort is required. However, the typical finite element methods generate fully unstructured meshes, which makes the use of a geometric multigrid not possible. In HHG a coarse unstructured base mesh is created by the FEM, that is then refined on a structured manner such that coarsening and interpolation as well as stencil-based operation can be performed. This concept combines the benefits of the finite element and the multigrid method. FEM can create fine grids that are necessary for the modeling of the Earth Mantle and multigrid can solve the large scale linear systems on a time-efficient way. Moreover, HHG is easy to parallelize as spatial decomposition on the coarse level is trivial.

In order to create a fault tolerant application there are two important steps. The first is to build a fault tolerant MPI application that can detect failures, acknowledge the failure by all processes, rebuild communication, and then restart and continue the simulation. The second step is to determine a data recovery strategy with the least possible overhead. For the first step, the ULFM library is used in order to be able to detect a failure and restore communication. There are two approaches for communication restoring. The first approach is the spare processes approach, where a spare process, that is idle before the failure, undertakes the role of the failed process. The second is the respawn approach where a new MPI process is spawned by the operating system to replace the failed one. For the second step, namely the data recovery, two approaches are used. The simplest approach is to save periodic checkpoints in the parallel file-system or the memory after a finite number of steps and recover the data from there. A more sophisticated approach is the Algorithm Based Fault Tolerance (ABFT) where the data of the healthy region can be used as a boundary condition in order to re-calculate locally the solution for the faulty region.

In the following chapters more information as well as a description of the implementation of what is mentioned above can be found. Chapter 2 contains the presentation of HHG, of MPI features that are used in this work and of ULFM. In Chapter 3, it is explained how the ULFM library is used in order to create a fault tolerant MPI application. The implementation of data recovery methods can be found in chapter 4. Chapter 5, contains testing and benchmarking results of the different approaches that are used, their comparison and also a comparison with the non-fault tolerant application. Finally chapter 6 contains concluding remarks of this work as well as suggestion for further work.

1.3 Related work

Many techniques can be used in order to create a fault tolerant application. These techniques can be classified in three categories, the hardware-based fault tolerance (HBFT), the software-based fault tolerance (SBFT) and the algorithm based fault tolerance (ABFT). In this work, software-based fault tolerance and algorithm based fault tolerance are considered. The SBFT techniques used here are based on the work of the Fault Tolerance Research Hub \(^3\), the developers of ULFM. A presentation of ULFM as well as a summary of recovery techniques can be found in [1] and [2]. A presentation with helpful examples about how ULFM can be integrated in an application can be found in [3], along with a presentation of checkpointing techniques. These techniques are out of the scope of this thesis as they are used in problems that are more complex and more time-consuming than the problem considered here. On the ABFT side, a detailed explanation of fault tolerant algorithms for the Poisson problem, using geometric Multigrid, is given in [11]. Apart from fault tolerance related work, it is important to mention [10] where the design of a fast parallel multigrid solver for Stokes systems is presented.

\(^3\)http://fault-tolerance.org/
2 Background

2.1 Hierarchical Hybrid Multigrid

In this section, the concept of the Multigrid solver using Hierarchical Hybrid Grids (HHG) is presented. The main idea behind HHG is that an unstructured base mesh, arising from a Finite Element discretization, is refined using a patch-wise regular refinement method creating a hierarchy of grids where a geometric multigrid method can be applied. In this way, the flexibility of an unstructured grid is combined with stencil-based operations that can achieve superior performance in modern parallel systems.

2.1.1 Disadvantages of unstructured grids

In the majority of Finite Element simulations, the discretization results to an unstructured mesh. The advantage of unstructured meshes is that they are flexible and can be used for the discretization of arbitrary domains. These meshes lead to sparse matrices that are usually stored in data structures of compressed format (e.g. Compressed Row Storage) where only the non-zero elements are stored and additional matrices are allocated for indexing. As meshes are unstructured, indirect indexing is used in order to locate the non-zero entries. In this format iterative solvers can solve small or moderate sized problems in a reasonable time on modern computers. However, in large scale simulations the time for solution increases dramatically and makes unstructured grids not suitable for large scale problems.

The reason of this inefficiency is that a solver using unstructured grids with indirect indexing cannot exploit the computational power of modern CPUs in its full capacity. The use of indirect indexing makes it impossible for the compiler to determine an access pattern and pre-fetch data from the main memory to the cache. Moreover, as the grid is unstructured there is a lack of spatial locality. This means that it is unlikely that related unknowns are close to each other in the memory and they are probably not in the same cache line. These two factors reduce the probability of having cache hits during the calculations. In this case, the data are fetched from the main memory that takes longer than fetching the data from the cache. Thus, the number of flops per cycle is limited. These disadvantages can be avoided if a stencil is used, because using a stencil implies a certain data access pattern. In addition, the data for updating one unknown is often in the same cache line as the data for updating the previous or the next unknown.

2.1.2 The concept of HHG

In order to avoid the performance problems but still be able to discretize arbitrary grids the concept of Hierarchical Hybrid Grids is introduced. In HHG the initial discretization produces a purely unstructured input grid. This grid is fairly coarse and is meant to resolve the problem being solved. As the grid is coarse some type of refinement is necessary in order to reach the desired resolution for the solution of the problem. Regular refinement is applied in the coarse grid where new vertices are added along each edge of the input grid. These vertices are then connected with the appropriate vertices with new edges and faces. This refinement is done successively resulting in a hierarchy of grids on which a geometric multigrid method can be applied. An example of such a grid hierarchy in 2D can be found in figure 1. As it can be observed in the figure the initial grid is purely unstructured, however there is some regularity in the last two grids. Using suitable data structures and algorithms, stencil-based operations can be performed on the finer grids that improve the performance of the solver as the unknowns are accessed with direct addressing. Moreover, as the grid is refined using the same method in all patches of the grid, restriction and interpolation functions can be applied. Thus, a geometric multigrid method can be used. This method creates a scalable algorithm where the arithmetic cost increases linearly with the problem size, making this method suitable for large scale applications.

In order to perform refinement an algorithm for grid refinement is required. This algorithm, should produce structured meshes and use a homogeneous connection pattern between grid-points. This is necessary in order to be able to use stencils instead of matrices. For this purpose the Bey Tetrahedral Refinement algorithm is used. This algorithm works for both 2D and 3D problems. For the 2D case as it is demonstrated in figure 2a, a triangle is refined by inserting a new point in the center of each edge. Then connecting the new points with three new edges results in 4 child
Figure 1: HHG refinement. The coarse input grid on the top left is regularly refined once to obtain the hybrid grid on the top right. Then, with a second level of refinement the lower left grid is produced and with a third level of refinement the lower right grid is produced. The figure is presented in [8].

triangles. The same principle can be used for the refinement of a square element into triangular elements. As it is illustrated in figure 2c, this can be achieved either by first dissecting the square into two triangles that are refined in the next step as described above, or by dissecting the square into four squares that are triangulated in the next step. In 3D, the algorithm works under the same principle. As it is illustrated in figure 2b, the edges are bisected by new points. Then by adding edges to connect the points, four child tetrahedra are produced in the corners of the original tetrahedron and an octahedron in the center that can be split into four new tetrahedra by adding one more edge. Figure 2d shows how to split a cube into tetrahedra in a similar manner as it was explained for the square.

As it can be observed the triangles and tetrahedra that are generate by this algorithm are similar and each internal point is surrounded by the same pattern of six triangles in the 2D case or 14 tetrahedra in 3D. This results in 7-point stencils in 2D and 15-point stencils in 3D.

2.1.3 Data structures and algorithms

As it is explained above, the use of regular refinement makes it possible to perform stencil-based operations as the input grid is refined in such a way that it defines stencils for all unknowns that are stored internally in an element. However as in the unstructured input grid the elements have a different structure, it occurs that in the finer grids the stencil that is defined around one unknown might be different from the stencil around another unknown that belongs to a different element. This can be observed in figure 3 where in the two pictures the highlighted edges have a different structure. However, in the same figure it can be observed that there is a regular geometric structure along the unknowns of each edge. Thus it makes sense to decompose the grid into the geometrical primitives associated with the input grid. These primitives are the vertices, the edges, the faces and in 3D also the volumes. A separate data structure is defined for each primitive type that contains the unknowns of the refined meshes as well as the appropriate layers of ghost unknowns as it is illustrated in figure 4 for the 2D case. Now each of the different primitive object contains unknowns where the same stencil structure is defined. The data structures of each primitive allocate continuous memory in order to exploit the processor performance. The ghost layer is defined in order to resolve the dependencies that exist between objects. For example in figure 4 the unknowns of the ghost layer on the right side of the triangle are the same as the unknowns on the ghost
Figure 2: Refinement using the Bey algorithm for a triangle, a tetrahedron, a square and a cube. The figures are presented in [9].
layer on the left side of the adjacent quadrilateral, and they are exchanges through the edge object between them. Similarly the edge is exchanging its ghost point in the top or in the bottom with the respective edge through a vertex. The structure of different primitives is also illustrated in figure 5.

Figure 3: The mesh that is generated in figure 1. The unknowns of the same edge have the same neighbor pattern. This is different for unknowns on a different edge. Putting the unknowns of each primitive in different object results to objects where unknowns have the same stencil structure. The figure is presented in [8].

Figure 4: Message passing between two faces through an edge that works as an interface data structure. Here the inner nodes, where calculation is performed, are marked in grey and ghost nodes, that hold copies of values that are exchanges with neighboring nodes, are marked in black. The figure is presented in [10].

Now it is important to see how the update of the unknowns is performed in the HHG algorithm. Looping over all unknowns and performing an operation on them, will result in treating the grid as a global unstructured grid and this will cause the performance issues that are mentioned above. Moreover, there are communication dependencies because of the ghost points. Therefore, the update operation should be performed in such a way that only the newly calculated values are communicated. For this reason it makes sense to update each set primitive type separately and then communicate dependencies as it describes in algorithm 1. This technique is not only used for smoothing but also for implementing the multigrid prolongation and restriction operations.

When updating the dependencies among different objects, it is useful to have a certain communication model that cuts down the amount of communication and is not complicated. In HHG a one way communication model is used. This means that each object communicates only with the lower/simpler object type, namely volumes (in 3D) communicate only with faces, faces only with edges and edges communicate only with vertices. Objects of the same type do not communicate directly with each other. In this way each object updates its own interior points and receives the ghost points from the neighboring objects. For volumes and faces this communication model causes no problems as each object has its own unique interior points that do not depend directly on interior points of another object. However there are cases that an interior point of an edge directly depends on the interior point of a neighboring edge. For example in figure 3 the upper internal point of the
Algorithm 1 HHG iteration

1: for $i = 1,...,\text{NUM\_VERTICES}$ do
2:   smooth vertex($i$)
3: end for
4: update dependencies
5: for $i = 1,...,\text{NUM\_EDGES}$ do
6:   smooth edge($i$)
7: end for
8: update dependencies
9: for $i = 1,...,\text{NUM\_FACES}$ do
10:  smooth face($i$)
11: end for
12: update dependencies
13: for $i = 1,...,\text{NUM\_VOLUMES}$ do
14:  smooth volume($i$)
15: end for
16: update dependencies

highlighted edge of the left picture depends directly on the upper internal point of the highlighted edge of the right picture. As there is no direct communication between edges, these dependencies are updated but not communicated in the same iteration. This means that the missing dependencies will be updated in the next iteration through the adjacent vertex. In this case a Gauss-Seidel method results in a Jacobi-like iteration for those points. However, as it is proven in [8] this has only a small effect on convergence.

2.1.4 Parallel HHG

In the HHG algorithm different primitive objects perform the calculations independently and communicate only to update their dependencies. Hence, parallelization of HHG in a distributed memory MPI application is straightforward. Each process can take a set of elements (i.e. Faces in 2D, Volumes in 3D). The adjacent lower type object between them are coupled to both processes as both processes need them. These objects can be assigned to any of the two processes that are coupled to, and the information for these objects should be communicated through MPI. An example in 2D is illustrated in figure 5. The triangle on the left is assigned to process $P_0$ and the triangle on the right to process $P_1$. The unknowns on the edge between the triangles are coupled to both processes. In this case the edge (represented by an HHG edge data structure) and the respective vertices can be assigned to either of the two processes or both of them. The process on which the edge is assigned is communicating the required data between the face data structure and the edge data structure by simply copying the data within the same processor. The other processor has to communicate the data between the face and the edge through MPI messages. In case that the edge is assigned to both processes then both processes copy the data from the face to the edge and vice versa. Then, they communicate through MPI to exchange the data that should be copied to the edge object from the neighboring process.
2.2 MPI background

The Message Passing Interface (MPI) is the most widely used parallel programming model. It is based on message passing between processors with distributed memory. The first version of MPI, the MPI-1, defines the basic concepts of message-passing communication with the standard communication operations and it is based on a static-process model. MPI-2 is an extension to MPI-1 with the new concepts of dynamic process creation and management, one-sided communication and parallel I/O. A more recent version of MPI is MPI-3 that has been proposed in 2012, containing non-blocking versions of collective operations and new one-sided operations. The latest version is MPI-3.1, it was introduced on June 2015 and it is a minor update to MPI-3. There are different implementations of MPI available. The most widely used and freely available implementations are MPICH, LAM/MPI and OpenMPI. For this work the OpenMPI implementation has been chosen, as it is the only available that includes an implementation of ULFM. In order to understand the functions of ULFM and how they can be integrated in the code, two features of MPI are presented first. The first feature is the concept of Process Groups and Communicators and the second is the feature of Dynamic Process Creation.

2.2.1 Process Groups and Communicators in MPI

As MPI is used for message passing between processes the existence of an object that performs communication is necessary. This object is called communicator and it performs communication among the processes in the underlying process group (or group in short). A process group is an ordered set of processes. Each process within a group has a unique rank that is an non-negative integer in the range between zero and the number of processes in the group minus one. A process can belong to more than one groups and can have different ranks in different groups. In MPI, each point-to-point or collective communication is executed in a communication domain. A communicator represents locally the corresponding communication domain for each process in a group. There is a communicator for each group and each communicator defines a group. There are two types of communicators. The intracommunicators that execute point-to-point and collective communication are executed in a communication domain. A communicator represents locally the corresponding communication domain for each process in a group. There is a communicator for each group and each communicator defines a group. There are two types of communicators. The intracommunicators that execute point-to-point and collective communication between processes in the same group and the intercommunicators that execute only point-to-point communication between two process groups. For the user a communicator is an object of the type MPI_Comm and the group an object of the type MPI_Group.

In many MPI applications the only communicator and group that is used is the predefined global communicator MPI_COMM_WORLD and the corresponding group that contains all existing processes. These setting are sufficient as long as only SIMD algorithms are concerned. However, in the case of task parallel algorithms, creating subsets (groups) of the global communicator can make the implementation easier and prevent errors. In this case each group can perform different tasks in
parallel with the task of the other groups. One such case in fault tolerance is when spare processes are introduced that will undertake the role of failed process. The spare processes should not participate in the computation before a failure occurs. Therefore, a subset of the global communicator containing only the active process should be defined for this purpose. MPI defines functions that deal with operations on groups and communicator. The most important ones are presented below.

### 2.2.2 Functions for Process Groups

Before presenting the functions that can perform operations over groups, the functions that can provide information about a group are presented. Note that the predefined empty group `MPI_GROUP_EMPTY` can be used in all functions:

```c
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group)
```

The underlying group of a communicator can be obtained by calling this function. The parameter `comm` is the given communicator and `group` is a pointer to a previously defined `MPI_Group` object.

```c
int MPI_Group_size (MPI_Group group, int *size)
```

Returns the size of `group`.

```c
int MPI_Group_rank (MPI_Group group, int *rank)
```

Returns the rank of the calling process in `group`.

```c
int MPI_Group_free (MPI_Group group)
```

This function can be used to free a group that is no longer needed. The group handle is set to `MPI_GROUP_NULL`.

```c
int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *res)
```

Checks if two groups are identical (i.e. have the same processes in the same order). The parameter `res` with the value `res = MPI_IDENT` is retuned in case that this is true, the value `res = MPI_SIMILAR` if both groups have the same processors but in a different order and the value `res = MPI_UNEQUAL` in any other case.

```c
int MPI_Group_translate_ranks (MPI_Group group1, int n, const int *ranks1, MPI_Group group2, int *ranks2)
```

Translates the rank of processes in `group1` to those in `group2`. The integer arrays `ranks1` and `ranks2` have `n` entries each. The entries of `ranks1` correspond to the rank of `n` process in `group1`. The ranks of the corresponding processes in `group2` are returned in `ranks2`. For a correct execution valid ranks and properly allocated arrays should be provided.

The following functions are used to perform operation over groups and define new groups based on existing ones:

```c
int MPI_Group_incl (MPI_Group group, int p, int *ranks, MPI_Group *new_group)
```

Creates a subgroup of the given `group`. `ranks` is an integer array with `p` entries where each entry corresponds to the rank of a process in `group`. The `new_group` that is created has the `p` processes that their ranks can be found in the array. The order of the ranks in the new group is in the same order as in `ranks` (i.e. the rank of process `ranks[i]` in `group` is `i` in the `new_group`). For a correct execution valid and unique ranks should be provided.

```c
int MPI_Group_excl (MPI_Group group, int p, int *ranks, MPI_Group *new_group)
```

This function provides a `new_group` removing the processes with ranks `ranks[0],...,ranks[p-1]` in `group`. Again valid and unique ranks should be provided for a correct execution.

```c
int MPI_Group_union (MPI_Group group1, MPI_Group group2, MPI_Group *new_group)
```

Creates a `new_group` with the union of the processes in `group1` and `group2`. In the new group the processes of `group1` preserve their ranks and the process of `group2` which are not in the first group get subsequent ranks with respect to the order of their ranks in `group2`. 

---

14
int MPI_Group_intersection (MPI_Group group1, MPI_Group group2, MPI_Group *new_group)
Creates a new group with the intersection of the processes in the two groups. The ranks in the new group are defined according to the order of the ranks in group1.

int MPI_Group_difference (MPI_Group group1, MPI_Group group2, MPI_Group *new_group)
Creates a new group with the set difference of the two groups. The processes in the new group are the processes which belong to group1 and do not belong to group2. Again the order of the ranks in group1 specifies the order in the new group.

2.2.3 Functions for Communicators

Now the most important functions for Communicators can be presented. In all functions apart from the last one (MPI_Intercomm_merge) only intracommunicators are considered.

MPI_Comm_Create (MPI_Comm comm, MPI_Group group, MPI_Comm *new_comm)
Generates a new intracommunicator for a given process group. The parameter group is an existing process group that is a subset of the group associated with comm. For a correct execution all processes who belong to group should call the function and each process should specify the same group argument. A pointer to the new communicator will be returned in new_comm for the processes belonging to group. Processes who are not in group will receive MPI_COMM_NULL as return value in new_comm.

MPI_Comm_size (MPI_Comm comm, int *size)
Returns the size of the process group associated with the communicator comm.

MPI_Comm_rank (MPI_Comm comm, int *rank)
Returns the rank of the calling process in the group associated with the communicator comm.

MPI_Comm_free (MPI_Comm *comm)
deallocates the communicator comm. The deallocation is performed when all pending operations performed with this communicator are completed.

MPI_Comm_dup (MPI_Comm comm, MPI_Comm *new_comm)
Creates a new communicator new_comm which is an exact copy of comm.

MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int *res)
Compares two communicators. If both comm1 and comm2 denote the same data structure then the parameter ret returns with res = MPI_IDENT. In case that the associated groups of both communicators contain the same processes in the same order then the functions returns with res = MPI_CONGRUENT. If the two associated groups of the communicators contain the same processes in a different rank order then the functions returns with res = MPI_SIMILAR. In any other case the result of the function is res = MPI_UNEQUAL.

MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *new_comm)
This function is used in order to split the group associated with comm into disjoint subgroups and create a new communicator new_comm for each group. The parameter color specifies in which group each process will be placed. Processes with the same color are placed in the same group. The number of subgroups is the same as the number of different colors. Processes with color = MPI_UNDEFINED are not placed in any of the subgroups. The parameter key specifies the rank order of the process within the new subgroup. In case that two processes have the same color and the same key then they are ordered with respect to their rank in the group associated with comm. The subgroups that are created are not explicitly returned but each processes gets a pointer new_comm to the communicator of the subgroup where it belongs. Processes with color = MPI_UNDEFINED will receive a null pointer new_comm = MPI_COMM_NULL.

MPI_Intercomm_merge (MPI_Comm intercomm, int high, MPI_Comm *newintragrup)

15
This function creates a new intracommunicator that contains the processes of both groups that the intercommunicator intercomm is associated with. The parameter intercomm is an intercommunicator and newintracomm is a pointer to the new intracommunicator. The parameter high is a boolean variable that is used to order the groups of the intercommunicator within the new intracommunicator. The processes of the group with high equal to false receive the lower ranks (starting from zero) and the processes of the group with high equal to true receive the subsequent ranks. In case that both groups provide the same value then the groups are ordered arbitrarily.

2.2.4 Dynamic Process Creation

In most of the MPI applications the number of processes is specified in mpirun and remains constant until the end of the execution. However there are cases where creating processes dynamically is useful. For example in a fault tolerant application, a failure can result to an uneven distribution of the workload that affects the performance. In this case it is useful, after the failure, to create a new process that undertakes the role of the failed process. Dynamic Process Creation is a feature that was introduced in MPI-2.

New MPI processes (one or more) can be started by calling the function:

```c
int MPI_Comm_spawn (char *command, char *argv[], int maxprocs, MPI_Info info, int root,
                    MPI_Comm comm, MPI_Comm *intercomm, int errcodes[])
```

The parameter command specifies the name of the program that will be executed by each of the new processes (in a C / C++ code this can be argv[0] from the command line arguments). The parameter argv[] contains the arguments for this program, excluding the name of the executable (in a C / C++ code this can be a pointer to the second string of argv, i.e. argv+1). In case that no arguments are required then this parameter can be equal to MPI_ARGV_NULL. The parameter maxprocs specifies the number of processes to be started. The parameter info is an MPI_Info object. It is a data structure with (key, value) pairs that provide instructions to the MPI runtime system about how to start the new processes. For example, the host node of the new process can be specified if the pair ("host", <host name>) is contained at info. Specifying this kind of information can be useful in case that the user wants to have full control about where and how the new processes is started. These information may be provided in such a way that the portability of the application is not affected. If none of this information is provided, then the parameter info should be equal to MPI_INFO_NULL. The parameter root specifies the root process from which the new processes are spawned. Only the root process specifies values to the parameters mentioned before, however all processes of the communicator comm should call the function as this is a collective operation. The parameter intercomm returns an intercommunicator in case that the new processes are spawned successfully. The spawned processes belong to a new group that communicates with the group associated with comm through intercomm. The parameter errcodes is an array, with maxprocs entries, which contains the the error status of the spawned processes. In case that a process is spawned successfully, the error code MPI_SUCCESS is returned to the corresponding entry in errcodes.

The spawned processes can communicate with the group of the old processes through the intercommunicator created by MPI_Comm_Spawn which can be accessed by calling the function:

```c
MPI_Comm_get_parent (MPI_Comm *parent)
```

In order to place all processes in the same communicator, the function MPI_Intercomm_merge can be used. The old processes specify the intercommunicator using the pointer intercomm returned at MPI_Comm_spawn and the spawned processes using the pointer parent returned by MPI_Comm_get_parent. An implementation of this in a can be found in section 3.3.2, where new processes are spawned to replace the failed processes.

2.3 User Level Failure Mitigation

The User-Level Failure Mitigation (ULFM) library has been proposed as a fault tolerant version of the MPI standard. ULFM is an MPI recovery API that introduces fault-tolerant constructs in the MPI standard. ULFM provides the necessary APIs that can handle a failure. ULFM can detect failures, it has the functionality to inform all processes about the failure and can abort all pending communications (without aborting MPI) in order to avoid deadlocks. Moreover, ULFM
provides the necessary functions that can restore communication omitting the failed processes, in
such a ways that the MPI application can complete without problems despite of failures. It is
important to mention that ULFM is not an application recovery API, this means that once the
communication is restored an application recovery strategy should be used in order to restart
the application. This last feature makes ULFM simple and flexible in term of usage and lightweight in
terms of performance.

Note that ULFM is not included in the latest version of MPI (MPI 3.1), but it is implemented
in OpenMPI. Therefore, in all code extracts that follow in the next chapters, ULFM functions start
with OMPI instead of MPI that is how MPI standard functions start.

2.3.1 Functions of ULFM

MPI_Comm_revoke(MPI_Comm comm)
This is a non-collective function that is called in case that a failure has been detected. It declares
the underlying communicator as improper and interrupts all communication operations. When a
failure occurs it is likely that not all processes are aware about it. In point-to-point communica-
tion only the processes that are directly communicating with the failed process (e.g. neighboring
processes) are aware about the failure and in collective communication some processes detect the
failure whereas others might consider the operation successful. As a result some processes might
continue on their normal failure-free execution path when others are already at the recovery path.
This inconsistency can lead to a deadlock, therefore propagation of the failure is necessary. When
this function is called the communicator is revoked, all pending and future communications are
interrupted and the error code MPI_ERR_REVOKED is returned by all functions that use (or will use)
this communicator. After a call to this function, all processes are aware about the failure and
they can proceed with rebuilding the communicator. It is important to mention that although this
function is non-collective, it affects all the processes in the underlying communicator.

MPI_Comm_shrink(MPI_Comm comm, MPI_Comm* newcomm)
This is a collective function that creates a new communicator eliminating the failed processes from
a revoked communicator. This function shrinks an existing communicator by creating a duplicate
of it leaving out all the processes that have failed. Note that this function cannot return an error
because if a failure is detected during the operation then the newly failed process is not included
in the new communicator.

MPI_Comm_agree(MPI_Comm comm, int* flag)
This is a collective function that calculates the conjunction of a boolean variable (flag) over all
processes on the communicator. Dead processes "participate" in this function with a default 'false'
variable. It is an extremely costly function and it is used in order to ensure a consistent state of the
application. For example this function can be called in order to solve any inconsistency issues after
a failure or to make sure that a part of the algorithm has been completed uniformly. It is important
to note that this function will complete even if there are failed processes in the communicator, if a
failure occurs during the process or if the communicator is revoked.

MPI_Comm_failure_ack(MPI_Comm comm) &
MPI_Comm_failure_get_acked(MPI_Comm comm, MPI_Group* group)
These two functions are non-collective and they are used to determine which processes within a
communicator have failed. In case that a failure occurs an application can continue in case that
only point-to-point communication is used and the task of the failed processes can be undertaken
by another one. In this case it is important to have an overview of the failed process in order to
avoid communicating with them. The function MPI_Comm_failure_ack acknowledges a new failed
process and the function MPI_Comm_failure_get_acked returns the group of processes that are
known as failed at the last call of the acknowledge function. This operations can also complete
on revoked communicators. When using these functions it is important to notice that the group
of failed processes is local to each process which means that not all processes are aware about all
failures.
2.3.2 Error codes of ULFM

MPI_ERR_PROC_FAILED
This error code is returned in case that a failed process has been discovered. It is important to mention that during collective communication not all processes will return with this status but some processes might return with a successful status even though a failure has occurred.

MPI_ERR_REVOKED
This error code is returned in case that the function MPI_Comm_revoke has been called by one of the processes on the underlying communicator. This means that the communicator has been marked as improper for further communications because it contains failed processes.

MPI_ERR_PROC_FAILED_PENDING
This error code is returned when a non-blocking MPI_ANY_SOURCE potential sender has failed.
3 Building a Fault Tolerant Poisson Equation Solver

It can now be presented how ULFM is used in order to build a fault tolerant code for the solution of the Poisson equation. The objective is to have an MPI application that will not abort after a failure but it will be able to restore communication and restart the simulation. The code that is given is a non-fault tolerant Poisson solver using the HHG classes of TerraNeo. In order to make the code fault tolerant, some changes are introduced. The first change is to introduce processes that can undertake the role of the failed process. There are two options that can be used to implement this. The first option is to occupy spare processes when the application starts and keep them idle until a failure occurs upon which they will be triggered to undertake the role of the failed process. In this case, special care should be taken that the spare processes are not involved in the computation before any failure and that they will return in case that the simulation completes without failures. The second option is to use dynamic process creation as described in section 2.2.4 and start new processes that will undertake the role of the failed process. In both options there should be some changes in the code in order to bypass communication with other processes during the setup phase. The second concept that is introduced is failure detection. A failure can be detected using ULFM, but the error should be propagated to the functions who called the MPI routine where the failure was detected in order to handle the error and start the recovery of MPI. The last change introduced is restoring communication using ULFM and replacing the failed process with a spare or a spawned process.

Before presenting how this is implemented it is important to mention that the error handler of all communicators should be set to MPI_ERRORS_RETURN otherwise MPI aborts in case of a failure. It is better to do this in the beginning of the application. By setting the error handler to MPI_ERRORS_RETURN, MPI continues even if errors occur and the errors can be handled in the code. The command bellow shows how this is implemented:

```c
MPI_Comm_set_errhandler( comm, MPI_ERRORS_RETURN )
```

3.1 Introducing spare/spawned processes

3.1.1 Communicator split for spare processes

In case that the spare processes approach is used, using the global communicator MPI_COMM_WORLD should be avoided as the spare processes are not participating in the computations before a failure occurs. The processes in this case are placed in two categories. The active processes that participate in the computation from the beginning, and the spare processes which stay idle until one of the active processes fail, when they undertake its role. Therefore it is useful to create a process group that is a subset of the global group (the group associated with MPI_COMM_WORLD) and contains only the active processes. The function MPI_Comm_split is used in this case. The active processes receive a color variable equal to 1 in the function call and the spare processes the undefined color status MPI_UNDEFINED. After calling the split function the active processes belong to the new working subgroup from which the spare processes are excluded. An illustration of this can be found in Figure 6a. Listing 1 shows how this is implemented. Here globrk is the rank of the process in MPI_COMM_WORLD, np the total number of processes, SPARES the number of spare processes, spare the color argument of MPI_Comm_split and working is an MPI_Comm object that for the active processes is the new communicator and for the spare processes is equal to MPI_COMM_NULL. After MPI_Comm_split completes, working can be set as the communicator of the whole program by giving a reference to it at the object of the hhgMPIController class that performs all communication in the program.

Listing 1: Communicator split

```c
spare = (globrk>np−SPARES−1)? MPI_UNDEFINED : 1;
rc = MPI_Comm_split( MPI_COMM_WORLD, spare, globrk, &working );

hhgMPIController &mpiCtrl = hhgMPIController::instance ();
mpiCtrl.comm = working ;
```

With this setting the spare processes can be separated from the active ones. The active processes proceed to the setup phase and the V-cycles iteration using the new working communicator, but the
spare processes wait until a failure occurs. This is implemented in the following way. After calling the split function the spare processes enter a loop where they have to wait at MPI_Comm_agree until they are triggered by the active processes in case that the program completes successfully or in case that a failure occurs. If the program completes successfully (i.e. completed = 1 after the agree function) then the spare processes can return, otherwise (i.e. completed = 0) they call the function MPIX_Comm_replace that is restoring communication (presented in section 3.3.1). As there might be more spare processes than the number of failures, this procedure is wrapped within a do-while loop because some processes might still have a spare status after restoring communication. In case that a process joins the group of active processes it should proceed to the setup phase and join the other active processes in the data recovery and the V-cycle iteration. Listing 2 shows how the spare processes are handled. The statement mpiCtrl.fail=true is useful for the setup of the spare process and it is explained in section 3.1.3.

Listing 2: Spare processes code

```c
if (mpiCtrl.comm == MPI_COMM_NULL) {  // Spare processes
    do {
        // unused spares are always ready to complete
        completed = 1;
       OMPI_Comm_agree( MPI_COMM_WORLD, &completed );
        if (completed) return;
        MPIX_Comm_replace( MPI_COMM_WORLD, mpiCtrl.comm, &newcomm );
    } while (newcomm == MPI_COMM_NULL);

    if (!completed){
        MPI_Comm_free( &mpiCtrl.comm );
        mpiCtrl.comm = newcomm;
        mpiCtrl.fail=true;
    }
}
```

// else active process

3.1.2 Spawned processes

In case that the spawned processes approach is used there is no need for a split of the communicator as only active processes exist. However it is necessary to make a duplicate of MPI_COMM_WORLD and work with this as the main communicator. This is necessary, because in case of a failure the communicator is revoked and is not used anymore after restoring communication. The new processes are spawned in the function MPIX_Comm_replace (presented in section 3.3.2) and this should be the first function that they call after initializing MPI. The spawned processes can be identified by calling the function MPI_Comm_get_parent( &comm ). As explained in section 2.2.4 the parameter comm returns with a pointer to the group of processes that spawned the process, or it is equal to MPI_COMM_NULL if the processes are not spawned but started at mpirun.

3.1.3 Setup of the spare/spawned process

After the new process joins the active processes group, it proceeds with setup and then data recovery. As data recovery methods are presented in chapter 4, in this chapter the case of no recovery is discussed where the unknowns vector of the new process is initialized to default (zero) values.

Setting up the new processes is trivial in case that no communication is present during the setup processes. However, in the case of the HHG Poisson equation solver there are communication operations during setup. The setup procedure is the same that the old active processes follow in the beginning of the program. As the spare/spawned process is doing going through the setup phase alone most of the communication should be skipped. For this reason, an after-failure state is introduced, during which the new process is not performing any communication with the old processes. In order to realize this state in the code a new boolean variable fail is introduced as a member of the hhgMPIController class. This variable is set to true when the new process starts.
the setup procedure and is set again to false when setup is completed. The functions that contain communication operations are now aware about the failure and they can skip communication.

In HHG most of the communication takes place when the processes update their dependencies. This also happens during the setup phase. The class hhgMesh is the one who deals with the operations in the HHG mesh. The function updateDMPDependencies of the class hhgMesh is calling the required MPI routines to perform the dependencies update. In order to skip communication in the setup phase, the function returns in the very beginning if the variable fail of the hhgMPIController object is true. There is also another point in the code where communication takes place and it should be skipped. This is at the constructor of the hhgUGLiFilereader class. This is an interface class responsible for reading the input file. In order to read the data faster only the process with rank zero reads from the file and broadcasts the data to the other processes. In order to skip this communication, the new process reads the data directly from the input file if the variable fail in the MPI controller is true.

Although most of the communication is skipped, there is a point at the end of the setup phase where communication should be performed in order to have correct results. This is when the right-hand side of the equation is calculated with respect to the boundary conditions. Moreover, communication might be necessary for the data recovery strategy. To perform this the fail variable of the MPI controller is set to false and the old processes call the same functions that new process calls.

3.2 Failure Detection

As mentioned in chapter 2.3, ULFM provides functionality that can detect failures. In case of failure MPI functions return with the error code MPI_ERR_PROC_FAILED. However, not all processes are aware about the failure and they might return with a successful status. Therefore, the communicator is revoked because all the active processes must become aware of the failure and all communication towards the faulty process should be interrupted. Then all processes call the function that performs the communication recovery.

Run-time errors are handled in C++ with exception objects. In the case of a fault tolerant application, two exceptions are introduced. One exception that is thrown in case that a failure is detected and one exception that is thrown in case that the communicator has been revoked. The exception classes (defined as derived classes of the standard exception class) failed.proc and revoked.comm represent these two cases. The first is thrown in case that an MPI function returns with the error code MPI_ERR_PROC_FAILED and the second in case that the error code is MPI_ERR_REVOKED.

In order to know whether any of the exceptions must be thrown, the error code returned by MPI functions is checked. A simple way to do it in the object-oriented concept is to call the MPI functions through an inline wrapper function that checks which error code is returned and throws an exception if necessary. Not all MPI functions are called through a wrapper function because in point-to-point communication the send routine, and in case of asynchronous communication even the receive routine, are not aware of the failure. For this reason, and as asynchronous point-to-point communication is used in this application, the functions that are called through a wrapper function are MPI_Waitall as well as the collective communication functions (in the HHG Poisson solver the only collective communication function involved in the V-cycles iteration is MPI_Allreduce). For MPI_Waitall the wrapper function is slightly different because in case of an error the return value is MPI_ERR_IN_STATUS and the actual error code can be found in the array of MPI_Status objects that the function has as parameter.

After a failure is thrown it bounces back to the previous function in the call order until it is caught by a try-catch statement or until it reaches the main function from where it is returned to the system causing the program to abort. For this reason, a try-catch block is introduced in the Poisson solver. All commands in the V-cycle for-loop are called within a try block. In case that a failure occurs, the exception thrown moves the control flow to the catch block where the exception is handled. In the catch block the communicator is revoked, this revoke will cause all processes that have not detected the error to enter the catch block as the revoked.comm exception is thrown. It is important to mention that all processes will enter this block even if they are in different parts of the V-cycle or even in the previous or the next V-cycle because all communication is interrupted and an exception is thrown by all processes. In this way all active processes are synchronized. In
case that the spare processes approach is used then the spare processes should be triggered using
the ULFM function MPI_Comm_agree with an unsuccessful status (i.e. completed = 0) in order to
call the function for restoring communication. The agree step is skipped in the spawned processes
approach. Listing 3 shows how this is implemented in the V-cycle iteration with the try-catch block.

In case that data recovery is used, then the respective functions are called inside the catch block
after communication is restored. Also all functions that are used to synchronize with the setup of
the new process are called in the data recovery part.

Listing 3: The V-cycle iteration and the try-catch statement for failure detection and after-failure
recovery.

```c
for(int cycle = 0; cycle < cycles; cycle++) {
    try {
        // V-Cycle functions
    } catch (std::exception& e) {
       OMPI_Comm_revoke(mpiCtrl.comm);
        int complete = 0;
       OMPI_Comm_agree(MPI_COMM_WORLD, &completed);
        MPIX_Comm_replace(MPI_COMM_WORLD, mpiCtrl.comm, &newcomm);
        MPI_Comm_free(&mpiCtrl.comm);
        mpiCtrl.comm = newcomm;

        // Data recovery
    }
}
```

3.3 Restoring Communication

Application recovery takes place in the MPIX_Comm_replace function. The role of this function is
to replace the revoked communicator with a new one that contains the active processes which have
survived and merge with the spare/spawned processes which take the role of the failed ones. The
procedure for achieving this is different between the spare and the spawned processes approach.
For this reason they are presented separately. The function for both approaches was presented at
[3] and it is used here with some modifications.

3.3.1 The Spare Processes Approach

First step of this function is to shrink, using the ULFM function MPI_Comm_shrink, the global
communicator that in this case is MPI_COMM_WORLD. Next, the MPI_ERRORS_RETURN is set as the
error handler of the shrinked communicator. Then follows a procedure to define the ranks of
the processes in the new communicator. What is needed in this case is that the survived active
processes preserve their ranks and the spare processes take the rank of the failed ones. Therefore,
after checking if there are enough spare processes, each processes recalls the rank that had in the
revoked communicator. The node with rank=0 finds the rank of the failed processes and sends it
to the spares. If there are more spare processes than the number of failures then the extra spares
will receive an undefined status again. After all processes have the rank that is needed, the new
working communicator is defined using the function MPI_Comm_split. The new communicator has
now as many processes as the revoked one, hence failure mitigation is completed. Before ending
the function it is necessary to check if any failure occurred during the procedure and repeat the
whole communication replacement if necessary. If no failure occurred then the function can return
successfully. An illustration of this procedure can be found in figure 6. The function is shown in
listing 4 in Appendix A.

After communication is restored we can proceed with restarting the application. The first step
is to set the new communicator as the working communicator and free the revoked one (see the
code extracts in sections 3.1.1 and 3.2).
3.3.2 The Respawn Approach

In case that the spawned processes approach is used, the first step is the same as in the spare processes approach and it is to shrink the communicator that contains the failed processes (in this case the duplicate of the global communicator). The second step is to spawn the new process(es) using MPI_Comm_spawn. The newly spawned process(es) belong to a separate group communicating through an intercommunicator with the group of old processes. Therefore, a merge of the two groups is necessary. Before this, the old processes should find the rank that they had before the failure. The process with rank=0 is doing this for the spawned process(es) that take the rank of the failed process(es). In order to do this the difference of failure group and the shrinked group is found and then the rank can be recalled using the function MPI_Group_translate_ranks. Then the process with rank=0 sends the ranks of the failed processes to the spawned processes via the inter-communicator. Afterwards, the two groups are merged using the function MPI_Intercomm_merge. Now all processes belong to the same group, but they probably do not have the correct ranks. Using the function MPI_Comm_split the new communicator is created and all processes have now the correct rank. In this case the split function is called with the role to give the correct rank to each process and not to split the communicator. After every critical point, the processes are using the function MPI_Comm_agree in order to agree whether a failure has occurred during the recovery procedure. The whole procedure is wrapped within a while loop and it is repeated in case that a failure is detected. An illustration of the procedure can be found in figure 7. The code of the function can be found in listing 5 in Appendix A.

After the communication is restored we can proceed with restarting the application in the same way as described in 3.3.1.
Figure 7: Step-by-step restoring communication after a failure using the respawn approach. In this case 4 active processes are used.
4 Data recovery

As it is discussed above it is possible to create fault tolerant software for the HHG Poisson solver, recover communication and start the V-cycles again. One issue that have not been discussed yet is how to recover the data that have been calculated by the failed process and have been lost after the failure. If no special recovery strategy is chosen, then this case can be called the no-recovery case or the do-nothing job. Intuitively it can be assumed that in the no-recovery case, the performance of the solver is seriously affected because the residual of the faulty domain pollutes the global residual. Hence, more V-cycles are required until convergence is reached. This assumption is justified in [11]. In the fault scenario that is considered, in a simulation with 16 million unknowns, the failure of one processes causes the loss of 0.3 million unknowns. The unit cube geometry $\Omega = (0, 1)^3$ is considered and the faulty subdomain is located in the interior of the cube. Figure 8b shows the distribution of the residual after the failure and setup and figure 8c after a global multigrid cycle is performed. As it can be observed, the error of the faulty subdomain is polluting the healthy subdomain, hence the global residual is increasing. Figure 8a shows how the pollution effect is affecting convergence. In this case a failure occurs after 5 V-cycles. Comparing with the failure-free case, 4 additional V-cycles are required in order to reach a residual of the order $10^{-15}$. According to [11], for a do-nothing job the number of additional multigrid steps is almost always equal to the number of V-cycles before the failure minus one.

![Figure 8: Pollution effect for the no-recovery case. Residual decay after a failure is presented in (a). Figure (b) shows the residual distribution after the failure and setup and figure (c) the residual after a global V-cycle. $\alpha = \log_{10}(|\text{Residual}|)$. The figure is presented in [11].](image)

After these observation it is important to define data recovery strategies that add a smaller overhead to the runtime. In this work two recovery strategies are implemented. The first recovery strategy is checkpointing where the unknowns’ vector is saved in a file after a certain number of iterations and in case of a failure all processes read the last checkpoint and restart from the iteration where the last checkpoint was saved. The second recovery strategy is algorithm based, in which the lost data of the failed process can be recalculated locally by applying boundary conditions on the faulty domain using the values of the surrounding healthy domain.

4.1 Data recovery using Checkpoint-Restart

The concept of data recovery using checkpoint-restart is simple and straightforward. After a certain number of multigrid iterations each processes writes in a file the current value of the unknowns’ subset that is calculating. In case of a failure all processes read the last checkpoint and restart from the iteration that follows the iteration where the last checkpoint was saved (i.e. last checkpoint iteration + 1). Checkpoints are saved using an object of the class `hhgVariableFileInterface`. This class is used to save variables in a file (specified at the constructor) or load variables from a previously saved file and copy them to a variable object. Checkpoints are saved using the function `saveVariable` and loaded using the function `loadVariable`. In order to minimize the number of data written, only the unknowns of the finest mesh are saved in the checkpoints. The end of a V-cycle is the most suitable place to save a checkpoint. In order to keep track of when the last checkpoint was
saved the number of the iteration is saved in a variable. Checkpoint frequency can be specified in the parameter file.

In case of a failure all variables read from the last checkpoint. This is important in order to have a consistent state of the application. Before calling the function that reads the checkpoint, the process with rank=0 is broadcasting the variable that keeps track of when the last checkpoint was saved, in order to provide this information to the new process. In case that this variable is equal to zero, the processes are not reading the checkpoint but they proceed without recovery. Having this variable equal to zero can occur in two cases. The first case is if the failure occurs in an early stage of the simulation when no checkpoint has been saved yet. The second case is if the process with rank=0 fails. In this case the variable that is broadcasted has the default value zero. This happens because of the assumption that the process with rank=0 is not failing and therefore it has the task of broadcasting the variable. However as in reality all processes have the same probability to fail the application should not abort in case that the process with rank=0 fails, therefore the processes continue without recovery. An alternative to this is to save, in each checkpoint, an information file with the number of the iteration where the checkpoint is saved.

4.1.1 Node-level checkpoint-restart using the SCR library

The simplest way to save a checkpoint is to save it at the file-system. Many clusters are equipped with parallel file-systems (e.g. LXFS) that have large bandwidth sufficient for some applications. However, in large-scale jobs, as several processes are used, the total size of the checkpoint data can be very large that the bandwidth of a parallel file-system becomes insufficient. Moreover, parallel file-systems are shared among different jobs running on the same cluster simultaneously. Therefore, the file-system might be busy serving other jobs when our job needs to checkpoint. In both cases the result is that the performance of the jobs is affected as they remain idle while they wait for the file-system to save the checkpoints. Therefore in large-scale jobs it might be better to avoid using the file-system but save the checkpoint in the memory (HDD, SSD, RAM or cache) of the neighboring nodes. The Scalable Checkpoint-Restart (SCR) \(^4\) library provides this functionality. It is based in two important observations, firstly that an application needs only the information of the latest checkpoint, secondly that in most of the cases a fault disables only a small part of the system but leaves the rest of the system intact. SCR can save checkpoints to the HDD, SSD, RAM or cache (depending on the system) instead of the file-system. In case of a failure the checkpoint is read from the memory, otherwise it is discarded when the next checkpoint is saved. Moreover, after a finite number of checkpoints the data of the checkpoints are saved to the file-system in order to recover from failures that affect a larger portion of the system.

As checkpoints in SCR are written on compute nodes that may fail, it is necessary to save data redundantly in order to recover the lost data after a failure. The SCR library defines three redundancy schemes. The first redundancy scheme is called Local, where each checkpoint is saved on the local node only. The second scheme is called Partner, where each checkpoint is saved on the local node and a full copy is saved in the memory of a partner node. In this case every node has a partner node with a certain hop distance defined by the environmental variable SCR_HOP_DISTANCE. The third redundancy scheme is called XOR, where each checkpoint is saved on the local node and a second copy of the checkpoint is saved among a set of nodes defined by an XOR parity. In this work the Partner redundancy scheme is used. More information about the SCR library can be found at [7].

In the implementation of SCR described at [7] the library considers only the global communicator MPI_COMM_WORLD. However as in this work more communicators are considered due to our fault tolerance strategy, there are some small changes made in the code of SCR, so that it can work with any intracommmunicator.

To use SCR some small changes to the file-system checkpoint code are required. After the working communicator is defined, SCR is initialized using SCR_Init, passing the communicator as a parameter. SCR is finalized using SCR_Finalize at the end of the code. In order to find the path to the checkpoint file in the memory, the function SCR_Route_file is used. The path returned by this function can be used to save or read a checkpoint on the same way as it is done for the file-system checkpoint. In order to save a checkpoint the function SCR_Start_checkpoint is called before opening the output file and the function SCR_Complete_checkpoint after writing

\(^4\)http://computation.llnl.gov/project/scr/
and closing the file. In the last function a parameter valid is specified that should be true (i.e. non-zero) if no problem occurred when writing the checkpoint and false (i.e. equal to zero) if the checkpoint was not saved successfully. In case of a failure SCR is finalized after the communicator is revoked and re-initialized when the new healthy communicator is built. When SCR is re-initialized it rebuilds the redundancy scheme based on the last checkpoint.

4.2 Data recovery using Algorithm Based Fault Tolerance

Recovering using a checkpoint is possible as explained above, however it has the disadvantage that the time to save checkpoints adds some overhead to the runtime even for the failure-free case. For this reason an algorithm based recovery method is considered. This method does not add any overhead in case of a failure-free run, because it runs only in case of a failure. The main idea behind this method is that the data of the faulty region are recalculated locally using the data of the healthy domain as boundary conditions, as explained in [11].

First of all, some basic concepts are considered in order to explain the algorithm. The unit cube \( \Omega = (0, 1)^3 \) is considered here as the domain. Following the concept of HHG presented in section 2.1 the domain is triangulated resulting in an unstructured base mesh \( \mathcal{T}_{-2} \) with tetrahedral elements. This mesh is successively uniformly refined to produce a hierarchy of meshes \( \mathcal{T} \) that are used for the multigrid solver. Setting \( \mathcal{T}_0 \) as the coarsest grid of the solver means that the input grid is refined twice in order to produce the coarsest grid of the solver. This is necessary in order to have at least one inner degree of freedom on the coarsest level. As it is explained in 2.1.4 the domain distribution among the processes is based on the elements of \( \mathcal{T}_{-2} \). In each level of the multigrid a linear system is defined.

\[
A_l u_l = f_l; \quad l = 0, ..., L
\]  

In case of a failure the following domains are defined. The faulty subdomain \( \Omega_F \subset \Omega \) that is the domain that is assigned to the failed processes and the healthy or intact subdomain that is defined as \( \Omega_I := \Omega \setminus \Omega_F \). Between these two subdomains the interface \( \Gamma := \partial \Omega_I \cap \partial \Omega_F \) is defined. Moreover the subsets of unknowns \( u_F, u_I \) and \( u_T \) are defined which are the unknowns belonging to \( \Omega_F, \Omega_I \) and \( \Gamma \) respectively. Assuming that one element of the input grid is assigned to every process, then in 3D \( u_F \) are the unknowns in the volume assigned to the faulty process including its ghost layer. Moreover, \( u_I \) are the unknowns in the ghost layer of \( u_F \), that are also saved in the faces, edges and vertices surrounding the domain \( \Omega_F \). These data structures are saved in more than one processes as more than one processes have dependencies on them. Hence, the unknowns \( u_I \) can be easily recovered from the faces, edges and vertices that are saved in the processes neighboring to the failed process. Now the ghost layer of the volume that belongs to \( \Omega_F \) is known. Recovering the data for the inner nodes is not possible as they are not saved in the neighboring processes. However, the values for \( u_I \) that are located on the boundary \( \Gamma \) of \( \Omega_F \) are known. This gives a sub-problem in \( \Omega_F \) with Dirichlet boundary conditions. By applying the multigrid solver locally in the faulty subdomain an approximation of the lost data can be calculated.

While the local V-cycles are performed on the faulty subdomain, the processes that are responsible for the healthy subdomain remain idle. In order to affect the performance as little as possible the time required to calculate the local V-cycles should be as small as possible. Therefore, it is necessary to accelerate the calculations in the faulty subdomain. This can be achieved using the local superman approach, namely over-balancing the compute power in the faulty subdomain. This approach can be implemented in different ways. For example, the new process can occupy a full compute node and calculate the solution on the subdomain in parallel using OpenMP shared-memory parallelization in the node. This will accelerate the computation because the work that was executed by one core before the failure is now executed by several cores. An other option is to re-partition \( \Omega_F \) and assign the computation to more than one MPI processes.

Another improvement that can accelerate the data recovery procedure is to run V-cycles on the healthy domain simultaneously with the computation on the faulty domain. A simple approach to this is to fix \( u_I \) to the pre-failure values and solve a sub-problem with Dirichlet boundary conditions on the healthy subdomain simultaneously with the sub-problem on the faulty subdomain. Two algorithms using the approach of simultaneous V-cycles in the faulty and healthy subdomains are presented in [11]. However, the implementation of these algorithms as well as the superman approach are out of the scope of this thesis.
Table 1: Testcases. In order to have weak scaling the number of unknowns and the number of processes of the initial problem (1a6) is multiplied with 8, 27 and 125. The number of processes per node is selected in such a way that the processes are evenly distributed among the nodes.

## 5 Benchmarking and Comparisons

### 5.1 Benchmarking system

Benchmarking and testing is performed at the Emmy cluster of the Regional Computing Center Erlangen (RRZE). The cluster consists of 560 compute nodes, each with two Xeon 2660v2 "Ivy Bridge" chips (10 cores per chip + SMT) running at 2.2 GHz with 25 MB Shared Cache per chip and 64 GB of RAM per node. It is using a parallel file-system (LXFS) with capacity of 400 TB and an aggregated parallel I/O bandwidth of at least 7000 MB/s. It has an Infiniband interconnect fabric with 40 GBit/s bandwidth per link and direction. It can achieve overall peak performance of approx. 234 TFlop/s (191 TFlop/s LINPACK).

### 5.2 Problem specification

The problem that is used for all tests and benchmarking is the Poisson equation in the unit cube as defined in equation (1) and described in detail in (3):

\[-\Delta u = f \text{ in } \Omega = (0,1)^3, \quad u = g \text{ on } \partial \Omega\]

with \( f = 6\pi^2 \sin \left( \left( x + \sqrt{2}y \right) \pi \right) \sin \left( \sqrt{3}\pi \right) \)

and \( g = \sin \left( \left( x + \sqrt{2}y \right) \pi \right) \sin \left( \sqrt{3}\pi \right) \) \hspace{1cm} (3)

For the Multigrid HHG solver, linear interpolation and its adjoint operator are used. A Gauss-Seidel updating scheme with three pre- and post-smoothing steps is used for smoothing and a conjugate gradient (CG) method as the coarse grid solver. The base tetrahedral mesh \( T_{-2} \) is refined twice to produce the coarsest grid \( T_0 \), that has at least one inner degree of freedom. This grid is refined 5 more times to produce the grid hierarchy \( T_1, \ldots, T_5 \) that is used for the Multigrid solver. The Bey Tetrahedral Refinement Algorithm, explained in section 2.1.2, is used for refinement. The number of processes is equal to the number of tetrahedral element in \( T_{-2} \). All tetrahedra are equally refined in order to have equal workload for each process. The cases mentioned in table 1 are used in all tests that follow. The number of processes per node is defined in such a way that the processes are distributed evenly among the nodes (i.e. the number of processes per node divides exactly the total number of processes). Even distribution of processes among the nodes is necessary for using SCR.

In the experiments presented in section 5.4 and 5.5 all processes of one node fail simultaneously. In reality this is not always the case. However, in case that a failure occurs in one process it is likely that more failures will occur on the same node. Therefore, it makes sense to kill all processes on the node and restart in a new healthy node. In case that a checkpoint-restart method is used the effort for data recovery is the same regardless of the number of failures because all processes have to read the checkpoint. In case of algorithm based fault tolerance, special care must be taken in this case in order to limit the amount of lost data.

### 5.3 Benchmarking results - No failure

Before benchmarking the cases with a failure, it is important to measure the time for one V-cycle in order to understand the weak scaling results that follow. In weak scaling it is expected that the...
Table 2: Time (in seconds) for 1 V-cycle

<table>
<thead>
<tr>
<th>Testcases:</th>
<th>1a6</th>
<th>8a6</th>
<th>27a6</th>
<th>125a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for 1 V-cycle [sec.]</td>
<td>0.245</td>
<td>0.35</td>
<td>0.36</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Table 3: Number of V-cycle required to achieve a residual smaller than $10^{-15}$. All processes of a node fail after 7 V-cycles.

<table>
<thead>
<tr>
<th>Testcases:</th>
<th>1a6</th>
<th>8a6</th>
<th>27a6</th>
<th>125a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>No failure</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1 Failure - No recovery</td>
<td>19</td>
<td>18</td>
<td>18</td>
<td>17</td>
</tr>
</tbody>
</table>

time for solution is not varying significantly as the size of the problem and the number of processes are increased proportionally. In the results presented in table 2, this is true in most of the cases. The test-case 1a6 is an exception here as the problem that is calculated is small and there are only 6 processes in total. In this case, most of the processes are adjacent to the boundary. Thus, the average number of neighbors per process is smaller comparing to the other test-cases. Hence less communication is required, that allows a faster run.

Another important remark about the failure free case is found in figure 10. The bottom lines (blue and green) of the figure consist of the timings for the no-failure cases, with and without fault tolerance enabled on `mpirun`. These two lines are almost identical, hence ULFM fault tolerance does not add any overhead to the runtime in case that no failure occurs.

5.4 Benchmarking results - Without data recovery

Now benchmarking results for a do-nothing job (i.e. without data recovery) are presented. Table 3 shows the total number of V-cycles required to achieve a residual smaller than $10^{-15}$. A failure occurs after 7 V-cycles and all processes of one node fail. Hence, a significant portion of the data is lost. It was observed during testing that a residual of the same order as before the failure is achieved after 6 post-failure V-cycles for the cases 1a6 and 125a6 and 7 post-failure cycles for the cases 8a6 and 27a6. However, in all test-cases a residual close to the pre-failure residual is achieved after 6 post-failure V-cycles. From the results of table 3 it is important to mention that in case of a failure at least 7 additional V-cycles are performed in order to achieve the required residual. Hence, in case that no data recovery is used, all the computational effort that has been made before the failure is lost.

Figure 9 shows the weak scaling results for the comparison of the two communication restoring approaches, and table 4 the time required to restore communication. For the respawn approach the new processes are spawned in a node that is idle before the failure in order to preserve load-balancing. As it is clear from the figure and the table, the spare processes approach requires much less time to restore communication in comparison to the respawn approach. However, using the spare processes approach an application can recover from a fixed number of failures, whereas for the respawn approach the number of recoveries is only limited by the number of available nodes.

Table 4: Time (in seconds) for communication recovery for the spare processes and the respawn approach.

<table>
<thead>
<tr>
<th>Testcases:</th>
<th>8a6</th>
<th>27a6</th>
<th>125a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spare processes [sec.]</td>
<td>0.12</td>
<td>0.33</td>
<td>0.43</td>
</tr>
<tr>
<td>Respawn [sec.]</td>
<td>0.72</td>
<td>1.42</td>
<td>4.75</td>
</tr>
</tbody>
</table>
Figure 9: Weak scaling. Measuring runtime (in seconds) with 1 failure and no recovery strategy, for the spare processes and the respawn approach. Runtime is measured for the testcases 8a6, 27a6 and 125a6. In case of a failure all processes of a node fail after 7 V-cycles. 11 V-cycles are executed for the no-failure case and 18 (7 pre-failure and 11 post-failure) V-cycles for the 1-failure cases.
5.5 Benchmarking results - Checkpoint-Restart

In this section benchmarking results for the Checkpoint-Restart strategy are presented. In figure 10 weak scaling results for the following cases are presented: the failure free case without fault tolerance enabled on `mpirun`, the failure free case with fault tolerance enabled, the case of 1 failure and no data recovery, the cases of 1 failure and file-system checkpointing and the case of 1 failure and node-level checkpointing. The respawn approach is used for communication recovery because it is easier to combine it with SCR. A failure occurs after 7 iterations, hence 2 iterations are repeated for the checkpoint-restart cases. As checkpoints are saved after 5 iterations, a total number of 2 checkpoints is saved at the checkpoint-restart test-cases.

First of all, as it is discussed in section 5.3 the baseline shows a small linear increment of the runtime as the size of the problem and the number of processes increase. This increment is higher for all cases where a failure occurs because of the overhead added by communication recovery as discussed in section 5.4. For the case of file-system checkpointing, the linear increment becomes dramatically high because of the time required to write the checkpoints as it is presented on the tables 5 and 6. The size of the binary file, where the checkpoint is saved, is 28 MB for each process in all test-cases. As the size of the problem and the number of processes increases, the total amount of data also increases. For the test-case 125a6 the total amount of data becomes extremely large (approx. 20.5 GB) that the file-system can not process them in a reasonable time, leading to a bottleneck. On the other hand, using SCR this bottleneck is avoided as the checkpoints are saved on the RAM of the nodes. As a result, the time to save 2 checkpoints in the file-system takes more than 7 seconds for the test-case 125a6 whereas for node-level checkpoints only 0.32 seconds are required. Hence, saving checkpoints in the file-system regularly can seriously affect the performance even in case that no failures occur. On the other hand, the overhead that node-level checkpointing is adding is much smaller. Thus, using an optimal checkpoint strategy adds a small overhead to the runtime that is not affecting the performance significantly. However, considering that SCR is not able to recover in case that more than one neighboring nodes fail simultaneously, it might be important for some applications to use a combination of file-system and node-level checkpoints. This functionality is provided by SCR. In this case an optimal checkpointing strategy is required where file-system checkpoints are saved seldom and node-level checkpoints more frequently.

From the information provided on the tables 5 and 6, there are some things to notice. Reading a checkpoint takes a very small amount of time. However, failure detection is more time consuming. As there is not an accurate way to measure failure detection, the measurements presented here is the time between the moment where a process is killed and the moment where the control flow enters the catch block. This time interval includes more operations than just failure detection as the processes proceed with the computations until the failure is detected during communication. As it can be observed in the tables, the time for restoring communication is also significant but it can be reduced in case that the spare processes approach is used as discussed in 5.4. The time for setup is also important here because it can not be reduced.

From the information extracted from figure 10 and the tables 5 and 6, it can be concluded that a failure adds an overhead to the run-time of the application. For the Poisson equation that is a simple problem with small runtime, this overhead is significant. However, this overhead is not severely increased in case that more complicated problems are considered. Therefore, adding some seconds to a simulation that runs for one hour or longer is not affecting its performance significantly.
Figure 10: Weak scaling. Measuring runtime (in seconds) without FT, with FT but no failure, with 1 failure and no recovery strategy, with 1 failure and file-system checkpoint-restart and with 1 failure and node-level checkpoint-restart. Runtime is measured for the testcases 8a6, 27a6 and 125a6. In case of a failure all processes of a node fail after 7 V-cycles. 11 V-cycles are executed for the no-failure case and 18 (7 pre-failure and 11 post-failure) V-cycles for the no recovery case. For the checkpoint-restart cases, checkpoints are saved every 5 iterations, 11 V-cycles are executed plus 2 who are repeated after the failure.

Testcases: 8a6 27a6 125a6

<table>
<thead>
<tr>
<th>Processes per node</th>
<th>12</th>
<th>18</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint size [GB]</td>
<td>1.3</td>
<td>4.4</td>
<td>20.5</td>
</tr>
<tr>
<td>Write CP [sec.]</td>
<td>1.04</td>
<td>1.97</td>
<td>7.27</td>
</tr>
<tr>
<td>Read CP [sec.]</td>
<td>0.041</td>
<td>0.028</td>
<td>0.03</td>
</tr>
<tr>
<td>Failure Detection [sec.]</td>
<td>1.06</td>
<td>1.17</td>
<td>1.1</td>
</tr>
<tr>
<td>Restore Comm. [sec.]</td>
<td>0.73</td>
<td>1.43</td>
<td>4.81</td>
</tr>
<tr>
<td>Setup [sec.]</td>
<td>1.5</td>
<td>1.68</td>
<td>1.71</td>
</tr>
<tr>
<td>Total time [sec.]</td>
<td>10.24</td>
<td>12.19</td>
<td>21.61</td>
</tr>
</tbody>
</table>

Table 5: Timing (in seconds) per operation for file-system checkpoint-restart, using the respawn approach. Checkpoints are saved every 5 V-cycles. All processes of a node fail after 7 V-cycles. 11 V-cycles are executed plus 2 who are repeated after the failure.

Note: The time for failure detection is the time between the point where the failure is simulated and the point where the processes enter the catch block.
### Table 6: Timing (in seconds) per operation for node-level checkpoint-restart, using the respawn approach. Checkpoints are saved every 5 V-cycles. All processes of a node fail after 7 V-cycles. 11 V-cycles are executed plus 2 who are repeated after reading the checkpoint.

<table>
<thead>
<tr>
<th>Testcases:</th>
<th>8a6</th>
<th>27a6</th>
<th>125a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes per node</td>
<td>12</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>Checkpoint size [GB]</td>
<td>1.3</td>
<td>4.4</td>
<td>20.5</td>
</tr>
<tr>
<td>Write CP [sec.]</td>
<td>0.24</td>
<td>0.34</td>
<td>0.32</td>
</tr>
<tr>
<td>Read CP [sec.]</td>
<td>0.024</td>
<td>0.0235</td>
<td>0.023</td>
</tr>
<tr>
<td>Failure Detection [sec.]</td>
<td>1.064</td>
<td>1.206</td>
<td>1.076</td>
</tr>
<tr>
<td>Restore Comm. [sec.]</td>
<td>0.713</td>
<td>1.41</td>
<td>4.66</td>
</tr>
<tr>
<td>Setup [sec.]</td>
<td>1.42</td>
<td>1.39</td>
<td>1.32</td>
</tr>
<tr>
<td>Total time [sec.]</td>
<td>9.77</td>
<td>11.33</td>
<td>15.11</td>
</tr>
</tbody>
</table>

Note: The time for failure detection is the time between the point where the failure is simulated and the point where the processes enter the catch block.

### Table 7: Weak scaling timing. Checkpoints containing lower dimension objects (faces, edges, vertices) are saved in the file-system after every iteration. After the failure, local V-cycles over the volumes are performed. One processes fails after 7 V-cycles. 11 global V-cycles are executed plus 7 local V-cycles on the faulty domain after the failure. The spare processes approach is used for communication recovery. Here the processes are not distributed evenly among the nodes but the maximum of 20 processes per node is used.

<table>
<thead>
<tr>
<th>Testcases:</th>
<th>1a6</th>
<th>8a6</th>
<th>27a6</th>
<th>125a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint size [GB]</td>
<td>0.07</td>
<td>0.56</td>
<td>1.9</td>
<td>8.79</td>
</tr>
<tr>
<td>Write CP (total) [sec.]</td>
<td>4.87</td>
<td>5.79</td>
<td>5.81</td>
<td>6.01</td>
</tr>
<tr>
<td>Data Recovery [sec.]</td>
<td>1.53</td>
<td>1.52</td>
<td>1.53</td>
<td>1.38</td>
</tr>
<tr>
<td>Total time [sec.]</td>
<td>13.46</td>
<td>15.56</td>
<td>16.63</td>
<td>26.19</td>
</tr>
</tbody>
</table>

### 5.6 Benchmarking results - Algorithm Based Fault Tolerance

For the Algorithm Based data recovery the following strategy is used. After every V-cycle a checkpoint containing the lower dimension objects (i.e. faces, edges and vertices) is saved on the file-system. In case of a failure, the new process who undertakes the faulty subdomain reads the last checkpoint and then performs local V-cycles over the volumes only. The values of the lower dimension objects and the boundary domain $\Gamma$ are kept constant during the iteration. By performing a number of local V-cycles equal to the number of the pre-failure V-cycles, the residual is reaching the order that was achieved before the failure. Table 7 shows the time required to save all checkpoints, the time for data recovery (i.e. the local V-cycles) and the total time for the solution of the Poisson problem with 1 failure. Comparing the total time with the weak scaling timings of figure 10 it can be observed that the total time is in this case higher than any of the other approaches. The main reason for this is that the time to save the checkpoints is highly time-costly because a checkpoint is saved in every iteration. Here the size of a checkpoint per process is 12 MB. This is smaller than the checkpoints mentioned at section 5.5 as the amount of information saved is smaller. Hence the bottleneck observed in table 5 does not occur here. On the other hand, the time for data recovery is relatively small and it can be improved if the superman approach is used as it is explained in section 4.2. Moreover, the small checkpoints that are saved here can be avoided if the local V-cycles are performed over all objects. However, this improvement is out of the scope of this thesis.

### 5.7 Remarks about the robustness of the code

The code that was developed for this work, with all implementation details mentioned in chapters 3 and 4, was tested for its robustness. Recovering from a failure is possible in case that the failure occurs during the V-cycles iteration, namely when the control flow of all processes is within the try block. In case that a failure occurs outside this block, then the failure is fatal. This cases are: a failure that occurs in the setup phase, a failure that occurs during the setup of the new process(es),
a failure that occurs during data recovery, and a failure that occurs when a checkpoint is saved. In case that a failure occurs during communication restoring, it is managed by the communication restoring function as explained in 3.3.

The code was tested for different failure scenarios that occur during the V-cycles iteration and it was possible to recover from all of them.

At this point it is important to mention that as ULFM is on an experimental level, there are problem that occur because of bugs in the ULFM implementation. For example the following problem occurs, using a release build of ULFM 1.0: A failure is detected during asynchronous point-to-point communication by MPI_Waitall, the first processes that detect the failure revokes the communicator, then all processes are expected to return with a revoked error status. However, some processes do not return from MPI_Waitall at all, causing a deadlock. When this failure scenario with exactly the same settings was tested in a debug build of an older ULFM version this problem did not occur. These problems can be reported to the developers through the ULFM mailing list at http://groups.google.com/forum/#!forum/ulfm. All these problem affect the resilience of the application, however they are not expected to be present when ULFM becomes part of the MPI standard.
6 Conclusions and further work

6.1 Conclusions
As it is discussed in the previous chapters, it is possible to introduce fault tolerance to the Hierarchical Hybrid Multigrid solver, within the object oriented framework of Terra-Neo. Using ULFM it is possible to detect a failure, make all processes aware about it, restore communication and restart the iteration. Restoring communication is implemented in two different ways. The first is the spare processes approach, where some additional processes are started in the beginning and they are ready to undertake the role of the failed process(es) in case of a failure. The second is the respawn approach, where in case of a failure new processes are started in order to replace the failed process(es). As it is discussed above, restoring communication requires less time in case that the the spare processes approach is used. On the other hand, using the respawn approach the number of possible recoveries is only limited by the number of available resources (i.e. processes and nodes). It is important to mention that ULFM is still on experimental level and it has some defects (i.e. bugs) that affect the resilience of the application, however all these problems are expected to be solved in the future. The data that are saved in the memory of the failed process and are lost when the failure occurs, are recovered using a checkpoint-restart method or an algorithm based fault tolerance technique. Using checkpoints adds an overhead to the runtime even in case that no failures occur, nevertheless by using node-level checkpoint-restart and an optimal checkpointing strategy, this overhead can be reduced. On the hand, using ABFT techniques this overhead can be avoided, however these techniques are not easy to implement and there is not a general techniques that can be applied to every problem as it happens for the checkpoint-restart.

6.2 Suggestions for further work
An important topic for further work is the improvement of the algorithm based fault tolerance. The first step is the implementation of the local V-cycles without saving the small checkpoints, and the acceleration of the this procedure using the superman approach as explained in 4.2. Then the next step is the implementation of the Dirichlet-Dirichlet and the Dirichlet-Neumann algorithm for data recovery, presented in [11]. The second suggestion is to extend the fault tolerant concepts to a problem that requires a longer time for solution. Considering a problem that requires some hours to compute the solution, different checkpoint strategies can be investigated as well as an efficient combination of node-level and fail-system checkpointing.

Acknowledgments This work would not be possible without the help and the support of the chair for System Simulation (LSS) of Prof. Dr. Ulrich Rüde and the RRZE High Performance Computing group of Prof. Dr. Gerhard Wellein. I would like to express my cordial thanks to my main supervisor Dr.-Ing. Björn Gmeiner for assigning me the topic of this work and for his constant support during the last months, to Faisal Shahzad for his support on the high performance computing side of this work and his help on dealing with the problems of ULFM and SCR, and to Markus Huber.
A Code extracts

Listing 4: Communication restore using spare processes (Section 3.3.1)

```c
int MPIX_Comm_replace(MPI_Comm worldwspares, MPI_Comm comm, MPI_Comm *newcomm)
{
    MPI_Comm shrinked;
    MPI_Group cgrp, sgrp, dgrp;
    int rc, flag, i, nc, ns, nd, crank, srank, drank;

    while(1) { /* Repeat in case that a failure occurs during the process */
       OMPI_Comm_shrink(worldwspares, &shrinked);
        MPI_Comm_set_errhandler(shrinked, MPI_ERRORS_RETURN);
        MPI_Comm_size(shrinked, &ns);
        MPI_Comm_rank(shrinked, &srank);
        if(MPI_COMM_NULL != comm) { /* I was not a spare before. */
            /* not enough processes to continue, aborting. */
            MPI_Comm_size(comm, &nc);
            if(nc > ns) MPI_Abort(comm, MPI_ERR_INTERN);
            /* Use the former rank for the active processes */
            MPI_Comm_rank(comm, &crank);
            if(0 == srank) {
                MPI_Comm_group(comm, &cgrp);
                MPI_Comm_group(shrinked, &sgrp);
                MPI_Group_difference(cgrp, sgrp, &dgrp);
                MPI_Group_size(dgrp, &nd);
                /* Assign the rank of the failed process to the spare */
                for(i=0; i<ns-(nc-nd); i++) {
                    if(i < nd)
                        MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
                    else drank=-1; /* still a spare */
                    /* sending their new assignment to all spares */
                    MPI_Send(&drank, 1, MPI_INT, i+nc-nd, 1, shrinked);
                }
                MPI_Group_free(&cgrp);
                MPI_Group_free(&sgrp);
                MPI_Group_free(&dgrp);
            }
            else { /* I was a spare, waiting for my new assignment */
                MPI_Recev(&crank, 1, MPI_INT, 0, 1, shrinked, MPI_STATUS_IGNORE);
            }
        }
        rc = MPI_Comm_split(shrinked, crank<0?MPI_UNDEFINED:1, crank, newcomm);

        /* Check for failures during the process */
        flag = (MPI_SUCCESS==rc);
       OMPI_Comm_agree(shrinked, &flag);
        MPI_Comm_free(&shrinked);
        if(!flag) MPI_Comm_free(newcomm);
        else break;
    }
    return MPI_SUCCESS;
}
```

Listing 5: Communication restore using spawned processes (Section 3.3.2)

```c
int MPIX_Comm_replace(MPI_Comm comm, MPI_Comm *newcomm, char ** argv) {
    MPI_Comm shrinked, icomm, mcomm;
    MPI_Group cgrp, sgrp, dgrp;
    int rc, flag, rflag, i, nc, ns, nd, crank, srank, drank;
    static int failures = 0;

    while(1){ /* Repeat in case that a failure occurs during the process */
        MPI_Comm_get_parent(&icomm);
        if( comm != icomm ) { /* not a spawnee */
            MPI_Comm_size(comm, &nc);
           OMPI_Comm_shrink(comm, &shrinked);
            MPI_Comm_set_errhandler( shrinked, MPI_ERRORS_RETURN );
            MPI_Comm_size(shrinked, &ns);
            MPI_Comm_rank(shrinked, &srank);
            rc = MPI_Comm_spawn(argv[0], argv+1, nc-ns, MPI_INFO_NULL, 0, shrinked, 0, icomm, MPI_ERRCODES_IGNORE);
            flag = (MPI_SUCCESS == rc);
           OMPI_Comm_agree(comm, &flag);
            if( (MPI_SUCCESS == rc) != flag ) {
                if( MPI_SUCCESS == rc ) {
                   OMPI_Comm_revoke(icomm);
                    MPI_Comm_free(&icomm);
                }
                MPI_Comm_free(&shrinked);
                continue;
            }
            MPI_Comm_rank(comm, &crank);
            if(0 == srank) {
                MPI_Comm_group(comm, &cgrp);
                MPI_Comm_group(shrinked, &sgrp);
                MPI_Group_difference(cgrp, sgrp, &dgrp);
                MPI_Group_size(dgrp, &nd);
                /* Computing the rank assignment for the newly spawned processes */
                for(i=0; i < nd; i++) {
                    MPI_Group_translate_ranks(dgrp, 1, &i, cgrp, &drank);
                    MPI_Send(&drank, 1, MPI_INT, i, 1, icomm);
                }
                MPI_Group_free(&cgrp);
                MPI_Group_free(&sgrp);
                MPI_Group_free(&dgrp);
            } else { /* I am a new process, waiting for my new assignment */
                MPI_Recv(&crank, 1, MPI_INT, 0, 1, icomm, MPI_STATUS_IGNORE);
                shrinked = MPI_COMM_WORLD;
            }
            MPI_Comm_set_errhandler( icomm, MPI_ERRORS_RETURN );
            rc = MPI_Intercomm_merge(icomm, 1, &mcomm);
            flag = (MPI_SUCCESS==rc);
            rflag = flag;
           OMPI_Comm_agree(icomm, &rflag);
           OMPI_Comm_agree(shrinked, &flag);
            MPI_Comm_free(&icomm);
            if!(flag & rflag) {
```
if ( MPI_SUCCESS == rc ) { MPI_Comm_free(&mcomm); }
MPI_Comm_free(&shrinked);
continue;
}
rc = MPI_Comm_split(mcomm, crank<0?MPI_UNDEFINED:1, crank, newcomm);

MPI_Comm_set_errhandler(*newcomm, MPI_ERRORS_RETURN);
flag = (MPI_SUCCESS==rc);
OMPI_Comm_agree(mcomm, &flag);
MPI_Comm_free(&mcomm);
if ( !flag ) { MPI_Comm_free( newcomm ); }
else break;
return MPI_SUCCESS;
}
References


