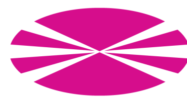

BACHELOR'S DEGREE THESIS (TFG)
DEGREE IN COMPUTER SCIENCE
MENTION IN COMPUTER ENGINEERING

Soft-fault recovery in MPI applications

Student: David Fernández Rey
Co-director: María José Martín Santamaría
Co-director: Patricia González Gómez

A Coruña, September 2020.

Thanks to all technical writers, hackers and demosceners who have inspired me to become a computer engineer, and to all the great teachers that have kept me motivated throughout the years.

Acknowledgements

- Juan Touriño for contacting me initially in order to develop this project.
- María José Martín Santamaría and Patricia González Gómez for supervising and helping me throughout the development of this project.
- CPPC developers and Nuria Losada for their ground work on this area of research.
- Computer Architecture Group (GAC) for providing me with access to the Pluton HPC cluster.

Abstract

Current high-performance computing (HPC) systems are comprised of thousands of CPU cores, and this number is expected to grow into the millions in the near future. With such an elevated number of processors, the mean time between failures (MTBF) can become so small that most scientific applications will not have time to complete their execution before a failure occurs. It is therefore critical to develop fault tolerance and resilience mechanisms in order to guarantee the completion and integrity of massively parallel applications. University of A Coruña's Computer Architecture Group (GAC) proposed a solution (Controller/comPiler for Portable Checkpointing - CPPC) in order to transparently convert generic MPI applications into fault tolerant applications, based on a checkpoint-restart scheme. CPPC was extended by Nuria Losada into CPPC-resilience in order to make resilient MPI applications, that is, those that are capable of detecting and reacting to failures without aborting the application, such that survivor processes don't have to be restarted. This was accomplished by means of a logging protocol and the usage of a proposed fault tolerance interface addition to the MPI standard (User Level Failure Mitigation). However, this system cannot handle soft errors efficiently, since it kills and respawns the failed processes entirely when it is not necessary, as these errors are transient in nature. The object of this project is to extend and adapt CPPC-resilience in order to handle soft errors in a more efficient manner, without having to respawn the failed processes. This proposal has been evaluated using 3 MPI applications with different characteristics, achieving a decrease in recovery times after a soft error ranging from 2 to 44 percent, depending on the total number of processes involved.

Resumo

Os sistemas actuais de computación de altas prestacións (HPC) están formados por miles de núcleos de procesadores, e espérase que este número aumente ata os millóns nun futuro cercano. Cun número tan elevado de procesadores, o tempo medio entre fallos (MTBF) pode chegar a reducirse tanto que a maioría de computacións científicas non terían tempo de completar a súa execución antes de que ocorrese un fallo. Polo tanto, é crítico o desenvolvemento de sistemas tolerantes e resilientes a fallos, para garantizar a finalización e integridade das aplicacións masivamente paralelas. O Grupo de Arquitectura de Computadores (GAC) da UDC propuxo unha solución (Controller/comPiler for Portable Checkpointing - CPPC) para convertir de xeito transparente aplicatcións MPI xenéricas en aplicacións tolerantes a fallos, basándose nun esquema de checkpointing e reinicio. CPPC foi posteriormente extendido por Nuria Losada en CPPC-resilience coa finalidade de crear aplicacións MPI resilientes, é dicir,

aquelas que son capaces de detectar e reaccionar a fallos sen abortar a aplicación, de xeito que os procesos superviventes non necesitan ser reiniciados. Isto logrouse mediante un protocolo de logging de mensaxes e o uso dunha interfaz de tolerancia a fallos, ULFM (User Level Failure Mitigation), proposta para adición ao estándar MPI. Sen embargo, este sistema non xestiona os errores soft de maneira eficiente, xa que mata e reinicia os procesos fallados por completo cando non é necesario, xa que este tipo de errores teñen natureza transitoria. A meta deste TFG é extender e adaptar CPPC-resilience para poder manexar os errores soft eficientemente, sen ter que reiniciar os procesos fallados. Esta proposta foi evaluada utilizando 3 aplicacións MPI con diferentes características, conseguindo unha redución nos tempos de recuperación tras un erro soft de entre un 2 e un 44 por cento, dependendo do número total de procesos involucrados.

Keywords:

- High-performance computing
- MPI
- ULFM
- Fault tolerance
- Parallelism
- CPPC
- Resilience
- Soft errors

Palabras chave:

- Computación de altas prestacións
- MPI
- ULFM
- Tolerancia a fallos
- Paralelismo
- CPPC
- Resiliencia
- Erros soft

Contents

1 Introduction **1**

1.1 Project background and goals . . . 1

1.2 Resources and planning . . . 2

1.3 Document structure . . . 3

2 Previous concepts **5**

2.1 Faults and their types . . . 5

2.2 Parallel programming: MPI . . . 6

2.3 CPPC . . . 7

3 Combining CPPC and ULFM to obtain resilience **11**

3.1 ULFM background . . . 11

3.2 CPPC-resilience overview . . . 12

3.3 CPPC-resilience workflow . . . 13

3.3.1 Failure detection and notification . . . 13

3.3.2 Communicator reconfiguration . . . 13

3.3.3 Application recovery . . . 14

3.3.4 Local rollback approach . . . 15

3.3.5 Message logging protocol . . . 16

4 Extending CPPC-resilience to cope with soft errors **19**

4.1 Introduction . . . 19

4.2 Simulating soft errors for CPPC-resilience . . . 20

4.3 Soft error detection and propagation . . . 20

4.4 Current recovery procedure flow . . . 21

4.5 Modifications for soft error handling . . . 24

5 Experimental evaluation 29
5.1 Testing environment 29
5.2 Testbed 30
5.3 Measuring spawn timings in CPPC-resilience 30
5.4 Experiments performed 33
5.4.1 Solution validation 33
5.4.2 Overhead without failures 36
5.4.3 Improvement with soft errors 37

6 Concluding remarks 41
6.1 Project conclusions 41
6.2 Lessons learned 42
6.3 Future work 42

List of Acronyms 43

Glossary 45

Bibliography 47

List of Figures

2.1 Relation between fault, error and failure . . . 5
2.2 CPPC instrumentation example . . . 8
2.3 CPPC checkpointing and recovery flow . . . 8

3.1 CPPC error checking function pseudocode . . . 14
3.2 CPPC-resilience checkpointing and recovery flow for a global rollback. . . . 15
3.3 CPPC-resilience local rollback recovery process. . . . 16

4.1 C code for a signal handler under Linux . . . 20
4.2 Sequence diagram illustrating the recovery process in CPPC-resilience. . . . 23
4.3 Sequence diagram illustrating the recovery process for soft errors in CPPC-resilience. . . . 25

5.1 MPI C pseudocode for the spawning benchmark. . . . 32
5.2 Time taken to spawn a process with different amounts of MPI processes. . . . 34
5.3 Time taken to spawn a process with different amounts of MPI processes (logarithmic scale). . . . 34
5.4 Output log for execution of CPPC_SOFT branch for the tealeaf application. . . . 35
5.5 C pseudocode for time measuring using MPI_Wtime . . . 36
5.6 Execution time comparison for all variants of each benchmark, with no failures and nprocs = 16 . . . 36
5.7 CPPC timing struct dump, for a benchmark with 16 processes. . . . 38
5.8 Execution time breakdown for a failed process for each benchmark, with nprocs = 16. . . . 38
5.9 Execution time breakdown for a failed process for each benchmark, with nprocs = 64. . . . 39
5.10 Improvement in the recovery phase when using our soft-error CPPC-resilience branch. . . . 39

List of Tables

3.1 List of ULFM fault tolerance routines. 12

4.1 Relevant ULFM, MPI and custom functions used during the respawning process. 22

5.1 Hardware platform description. 29

Chapter 1

Introduction

This first chapter offers a short description of the project at hand, indicating its main goals, alongside with the planning followed by the author in its development and ending with the structure of this document.

1.1 Project background and goals

High Performance Computing (HPC) systems have been around for many decades, however their complexity and computing power have been increasing non-stop. The constraints for the problems to be solved with these systems have also become higher, with larger datasets (e.g. Big Data from telemetry, social networks, etc.) and shorter time constraints (e.g. real-time analytics).

The growth of these systems is characterized mainly by 2 factors. On one hand these systems make use of innovative and faster technology, such as hardware extensions for vector operations (SIMD), higher clock frequencies, pipeline optimizations, among others. On the other hand HPC systems have become massively parallel, with a high number of similar hardware elements replicated and interconnected in such a way that it allows for certain problems to be partitioned and solved on many processors at the same time.

With the increase of massively parallel systems there comes however a new set of challenges and hazards that must be overcome in order to guarantee the successful, correct and efficient completion of software applications executed on these systems. One of these challenges is the reduction of times between hardware failures. It is evident from a statistical point of view that a higher amount of elements used at the same time will greatly reduce the Mean Time Between Failures (MTBF), even if the individual failure rates are very low. Dongarra et al. [1] show that if a single processor presents on average only 1 failure per century, a machine with 100000 nodes will encounter on average a failure every 9 hours, and with 1 million nodes this time shortens to 53 minutes on average.

With these numbers in mind it becomes clear that advanced fault tolerance mechanisms must be researched and developed in order to avoid application crashes, inconsistent results and a waste of electrical energy on massively parallel systems.

Faults present in these systems can be divided into hard and soft faults, the former being permanent and the latter being transient in nature. This implies that hard faults must entail some sort of complete restart while soft faults can be partially restarted. The goal of this project is to adapt an existing fault recovery tool for MPI applications, CPPC-resilience [2], to be able to deal with soft errors in an efficient manner, without restarting failed processes completely and simply going back to before the point of failure.

1.2 Resources and planning

This project was developed and tested in the Pluton cluster of the University of A Coruña [3], managed by the Computer Architecture Group (GAC). Version control of the changes made was done with Git [4], as it was the versioning tool originally used by CPPC and it allows for very efficient branching and committing.

The steps taken for the development of this project were, in chronological order:

- **Study of previous work:** as this project aims to improve an existing complex tool (CPPC), special care was taken to understand the functional principles and code structure of said tool.

- **Analysis:** clear definition of the scope of this project and a rough time estimation for each of the next steps.

- **Design:** layout of the new code components needed, their relationships and how the testing will be performed.

- **Development:** following the design as much as possible, implementation of the new components necessary for soft error handling into our own branch of CPPC-resilience.

- **Testing:** experimental evaluation of our new branch, and verification of the correctness of our code.

The thesis was written in parallel with all these steps, as a way to avoid forgetting small details and also to help review the work done in small batches, aiding sometimes in discovering errors or inconsistencies.

1.3 Document structure

This document is divided into 6 chapters (including this one), each containing the following information:

- **Chapter 1:** short description of the project goals, resources used, and planning and methodologies followed.
- **Chapter 2:** introduction to previous concepts necessary for understanding the whole scope of this project. Particularly, there is an explanation of the different types of faults in computer systems, of parallel programming with the MPI framework, and of the fault tolerance tool CPPC.
- **Chapter 3:** detailed description of the ULFM fault tolerance standard and its integration into CPPC in order to build CPPC-resilience, the fault tolerance tool that allows for transparent recovery from hard failures without aborting the execution.
- **Chapter 4:** soft error detection simulation and implementation of the algorithm for soft error recovery in CPPC-resilience.
- **Chapter 5:** experimental evaluation of results and details of the methodology used for testing.
- **Chapter 6:** final conclusions drawn from the development of this project, author's notes and possible future work paths.

Chapter 2

Previous concepts

This chapter illustrates the technologies and theoretical concepts necessary to understand the purpose and development of this project: types of faults, parallel programming with MPI and outline of the Fault Tolerance (FT) tool CPPC.

2.1 Faults and their types

For us to be able to develop fault tolerant systems, we must first identify and classify the different types of faults that happen in HPC environments. First, let us familiarize ourselves with the terminology employed in the industry, described by Avižienis et al. in [5] and summarized in Figure 2.1. Faults are flaws in the system (physical or logical) that are the source of errors, which are incorrect system states. These inconsistent state may propagate and lead to failures, which are what can be externally perceived. For clarity, an example follows: An electrolytic capacitor leaks (fault), this causes a bit flip which corrupts a struct in memory (error), which in turn leads to a read outside of boundaries and triggers a program crash (failure).

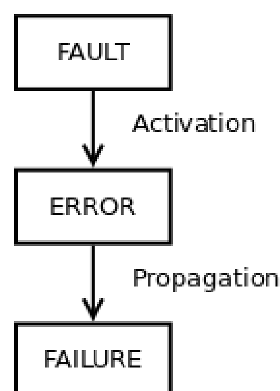

Figure 2.1: Relation between fault, error and failure

Faults can be classified in 2 major groups: hard faults and soft faults. The former group corresponds to physical faults (usually permanent), which cause errors that may be handled by hardware mechanisms and appear transparent to the software (such as RAID). Soft faults, which are the object of this project, are transient. They are most often caused by cosmic rays, and their impact has been increasing with the decrease in transistor sizes [6]. These errors can cause Silent Data Corruption (SDC) which could alter program results if not detected and corrected during runtime.

2.2 Parallel programming: MPI

The Message Passing Interface (MPI) is the de-facto standard for parallel programming in distributed memory architectures. It allows for the development of portable message-passing programs in C, C++ and Fortran. Some of the most used and efficient implementations are developed by the HPC community and are open-source, the most prominent examples being MPICH [7] and OpenMPI [8]. These implementations consist of a set of libraries that provide message passing primitives and collective operations in order to be able to share data among processors.

As the tools used and developed in this project use MPI at their core, some basic MPI concepts and terminology are required for the reader to understand and will be described here:

- **Rank:** numbering given to each process, starting incrementally from zero.
- **Communicator:** an object that connects groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its processes in an ordered topology, or in explicitly defined groups.
- **Point-to-point operations:** communications between two specific processes. The most popular examples are `MPI_Send` and `MPI_Recv`, which allow for sending and receiving data respectively (i.e. sending a double-precision floating point number to another process). Point-to-point operations can be blocking or non-blocking, depending on whether the program execution is halted after the function call.
- **Collective operations:** in contrast to point-to-point, they involve communication among all processes in a process group. A typical function is the `MPI_Bcast` call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group (in practice, communicators are used for grouping).
- **MPI datatypes:** representations of the structure of the data to be used with MPI function calls. Primitive types, such as MPI_INT, MPI_DOUBLE or MPI_CHAR are already

defined by default, but more complex user-defined datatypes can be created.

One of the issues with massively parallel systems is that the current MPI standard does not have fault tolerance support, and the default behaviour after a failure is to abort the execution of the application completely. One of the solutions that is currently under consideration to be added to the standard is the User Level Failure Mitigation (ULFM) interface designed by the Fault Tolerance Group, which integrates fault tolerance capabilities such as failure detection and notification, and communicator recreation and reconfiguration. These features are critical to the scope of this project and will be explained in depth in the next chapter.

2.3 CPPC

The Computer Architecture Group (GAC) of the University of A Coruña developed the Controller/comPiler for Portable Checkpointing (CPPC) [9] [10], an open source checkpointing tool for MPI applications written in C++ that aims to create a transparent approach to developing fault-tolerant MPI programs.

CPPC is used by the final user as a source-to-source transpiler (based on the CETUS framework [11]), which attempts to transform a normal MPI code written in C or Fortran into a fault tolerant version by adding calls to the CPPC library, although manual instrumentation is sometimes required in more complex programs. An example of said instrumentation can be found in Figure 2.2. The added calls perform the following functions which are defined in the runtime CPPC library:

- **Configuration and initialization:** at the beginning, routines are included that initialize internal data structures and variables (`CPPC_Init_configuration()`, `CPPC_init_state()`).
- **Registration of variables:** variables necessary for application recovery are identified and marked here.
- **Checkpointing:** after an user-defined number of calls to the `CPPC_Do_Checkpoint()` function, which is introduced in code hotspots, the state will be dumped to a portable file format (HDF5 [12]) in stable storage for future failure recovery.
- **Shutdown:** added at the end of the program, it ensures consistent and integral system shutdown.

The behavior after a failure, as seen in Figure 2.3, is to relaunch the application. Then, there is a negotiation phase among the processes in order to identify the most recent valid recovery

```
1  int main(int argc, char* argv[]) {
2      CPPC_Init_configuration();
3      MPI_Init( &argc, &argv );
4      CPPC_Init_state();
5
6      if (CPPC_Jump_next()) goto REGISTER_BLOCK_1;
7      [ ... ]
8
9      REGISTER_BLOCK_1:
10     <CPPC Register(...) block>
11     [ ... ]
12     if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1
13     [ ... ]
14
15     for(i = 0; i < nIters; i++) {
16     CKPT_BLOCK_1:
17     CPPC_Do_checkpoint();
18     [ ... ]
19
20     }
21     <CPPC_Unregister(...) block>
22     CPPC_Shutdown();
23     MPI_Finalize();
24 }
```

Figure 2.2: CPPC instrumentation example

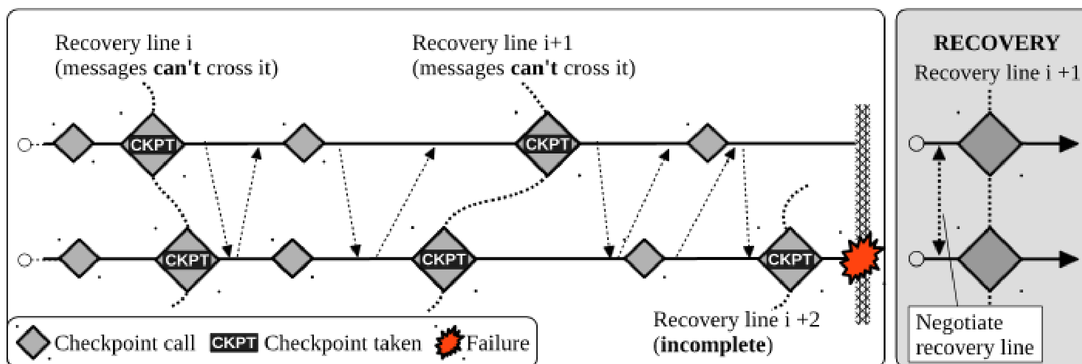

Figure 2.3: CPPC checkpointing and recovery flow

point, which corresponds to the most recent checkpoint file available to all processes. Finally, the checkpoint data is loaded into memory and the state is recovered through re-execution of critical parts of the program such as initialization, variable registration and creation of communicators.

The issue with CPPC is that the execution is aborted completely upon a single process failure. This is unnecessary in most cases as the majority of processes would still be alive and with correct partial results. Furthermore, a complete restart introduces major overheads since the application must be re-queued and the checkpoint files must be moved to the newly assigned computing nodes.

Therefore, in the next chapter we will introduce CPPC-resilience, an extension of CPPC that allows us to obtain resilient MPI applications, that is, applications capable of detecting failures in one or more processes and recovering from them without completely aborting program execution.

Chapter 3

Combining CPPC and ULFM to obtain resilience

RESILIENT applications are a subset of fault-tolerant applications. The latter is a more general term to describe programs that can continue functioning after a failure, whereas the former also imply that the execution continues without causing a program abortion. This generally entails a quicker and more efficient recovery, without re-initializing any unnecessary parts of the program.

In this chapter we will describe in detail the mechanisms used and the steps taken by CPPC-resilience in order to guarantee resilient failure recovery without respawning non-failed processes.

3.1 ULFM background

As described in Section 2.2, the current MPI standard as of June 2020 does not have any built-in fault tolerance capabilities. This has led to the birth of the ULFM (User Level Failure Mitigation) interface, currently being proposed for addition into the standard.

With vanilla MPI, the default behavior is to abort the execution of the entire application upon failure of a single process, so the only option would be to restart it from the beginning. CPPC improves this default behavior by creating periodic checkpoints in code hotspots, therefore even though all the involved processes have to be restarted upon a failure, they do so from a checkpoint, avoiding all of the computations that had already been done up to that point.

The downside to this approach is that in real life scenarios it is usually just one or a reduced amount of processes that fail at the same time, so restarting all of them carries a significant amount of overhead. A possible solution to this is to use the capabilities provided by the ULFM interface in order to create a more efficient approach to recovering from failures.

It is important to note that ULFM follows the same low-level interface style as MPI, there-

fore it provides mechanisms to detect failures, revoke and recreate communicators, but not to recover or relaunch the application, as this is generally a very specific process for each different program (i.e. recovery is different in a master-slave paradigm and a grid division, since in one case the program can continue with the missing processes whereas in the other they must be recreated in the exact same manner as before). ULFM is simply a set of functions and data structures and adding fault tolerance capabilities to existing MPI programs would imply manually refactoring the code in order to introduce these new FT operations, which is a very difficult and expensive task.

Some routines provided by ULFM are shown in Table 3.1, the full specification of which is provided by a draft of the standard [13] written by the Fault Tolerance Group.

We will now describe how these routines are integrated into CPPC in order to allow for transparent application recovery.

Function	Description
MPI_Comm_failure_ack(comm)	Acknowledge process failures on communicator comm. Resumes MPI_ANY_SOURCE operations.
MPI_Comm_get_acked(comm, *failedgroup)	Returns the group of processes acknowledged to have failed.
MPI_Comm_agree(comm, *mask)	Collective, agrees on the AND value on binary mask, ignoring failed processes (reliable AllReduce).
MPI_Comm_revoke(comm)	Non-collective with effect on entire comm, all communications on comm are interrupted with MPI_ERR_REVOKED.
MPI_Comm_shrink(comm, *newcomm)	Collective, creates a new communicator without the failed processes (all ranks are preserved).

Table 3.1: List of ULFM fault tolerance routines.

3.2 CPPC-resilience overview

CPPC-resilience is a tool developed by Nuria Losada [2] that combines CPPC and ULFM in order to optimize performance when recovering from failures. The issue with CPPC is that a single process failure leaves MPI in an undefined state, therefore it cannot continue its execution properly and all processes must be restarted from the checkpoint, even if they have partial correct results. This is a waste of resources, and can be improved by integrating fault tolerance mechanisms from the ULFM extensions to MPI into CPPC. The integration allows us to create a non-shrinking, local backward recovery scheme based on checkpointing:

- **Non-shrinking**: the number of running processes stays the same, to allow for the same data distribution.

- **Backward recovery**: after a failure the application is restored from a checkpoint. This is in contrast to forward recovery, where there is an attempt to continue and find a new state (this is, however, very application-dependent and can not be proposed as a general approach).

- **Local recovery**: only failed processes are restored. Due to interprocess communication dependencies, this solution requires a message logging protocol (that we will describe in detail later) in order to replay communications from the last checkpoint to the current state.

3.3 CPPC-resilience workflow

In the domain of this project (scientific MPI applications), plain CPPC is a fault-tolerance mechanism, as it allows a program to survive hard process failures, but it restarts all running processes during recovery. CPPC-resilience, on the other hand, takes care to avoid restarting healthy processes and uses instead a logging protocol that replays communications from the point of failure.

3.3.1 Failure detection and notification

With the function `MPI_Comm_set_errhandler(communicator, handler)` the user can specify whether error codes should be returned to the application or a user-specified error handling function should be invoked.

In CPPC, after every MPI function call the routine `CPPC_Check_errors()` is called, whose pseudocode is shown in Figure 3.1. Its purpose is to check whether the returned value from the MPI call corresponds to a failure or not and if it does, to start the recovery process among the surviving processes. In order to do this, in the face of failure detection it calls the ULFM function `MPIX_Comm_revoke(comm)` on all communicators, which assures global failure detection as it interrupts all messages on comm with error `MPIX_ERR_REVOKED`.

3.3.2 Communicator reconfiguration

In most MPI applications, the ranks (identifiers of the processes) must be preserved after a failure in order to guarantee correct program execution. Since we are trying to obtain resilience and not just fault tolerance, only the failed processes will be restarted. This implies that all custom communicators will be recreated by CPPC by re-executing the MPI calls used to create them in the first place.

```
bool CPPC_Check_errors(int error_code)
{
  if (error_code == process failure) {
    /* Revoke communicators */
    for comm in application_communicators {
      MPI_Comm_revoke(..., comm, ...);
    }

    /* Shrink global communicator */
    MPI_Comm_shrink(...);
    /* Re-spawn failed processes */
    MPI_Comm_spawn_multiple(...);
    /* Reconstruct communicator */
    MPI_Intercomm_merge(...);
    MPI_Comm_group(...);
    MPI_Group_incl(...);
    MPI_Comm_create(...);

    /* Start the recovery */
    return true;
  }

  return false; /* No error detected */
}
```

Figure 3.1: CPPC error checking function pseudocode

First, all failed processes are excluded from the communicator by means of the ULFM function `MPI_Comm_shrink()`, which also preserves the original ranks. Then, after respawning the processes, the communicators are merged and reordered in order to include the new ranks. In order to ensure that the correct global communicator is used (i.e. one that does not contain failed processes), the one returned by `CPPC_Get_comm()` is stored as a global variable and used instead of the default `MPI_COMM_WORLD`.

3.3.3 Application recovery

CPPC-resilence can recover the application from a failure in two ways: using a global or a local rollback approach. In this section, the global approach, which does not require a message logging protocol, will be explained. In order to recover the application properly, all processes must be in a consistent global state. This implies that while the respawned process is already in such a state after reinitializing the communicators, all running processes must re-execute certain blocks of code (RECs). This is done by reversed conditional jumps after calls to `CPPC_Check_errors()` and `CPPC_Go_init()`, by means of goto statements and return instructions. Non-local jumps (`setjmp` and `longjmp`) are not used in order to provide compatibility with Fortran.

Once all the processes have reached the beginning of the application code, a regular CPPC

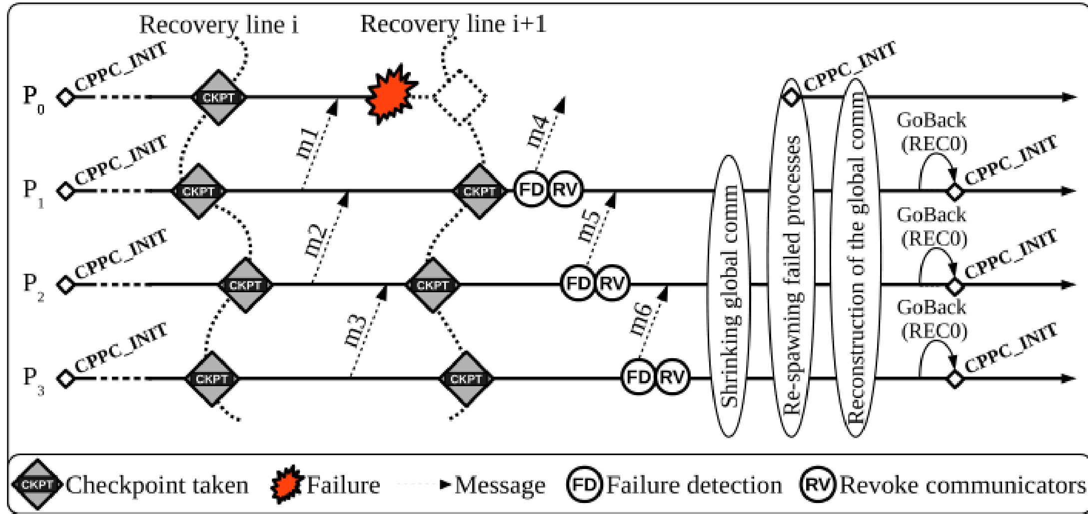

Figure 3.2: CPPC-resilience checkpointing and recovery flow for a global rollback.

restart is performed. First, processes negotiate the recovery line as shown in Figure 3.2, and then the checkpoint files are restored and non-portable state is recovered through the ordered re-execution of RECs.

3.3.4 Local rollback approach

The approach described in Section 3.3.3 relies on a global rollback scheme. This is inefficient in many cases as failures are often localized and it is unnecessary to restart all process from the latest common checkpoint. However, as communications from the latest checkpoint must be replayed at least for the failed process, local rollback protocols necessarily imply some sort of message logging capabilities. Figure 3.3 shows an outline of the steps taken by the local recovery branch of CPPC-resilience in order to recover from failures using a local rollback protocol.

The first important thing to note in the figure is that the execution of the survivor process (represented by the top line) is never interrupted. Immediately after a failure is detected, the communicators are revoked and shrunk in order to exclude the failed processes, by means of the ULFM functions `MPI_Comm_revoke()` and `MPI_Comm_shrink()` respectively. Afterwards, recovery can be divided into 2 major parts: processes recovery and consistency recovery. In the first part, processes which have been marked as failed are re-spawned with regular MPI functions (e.g. `MPI_Comm_spawn()`) and the global and custom communicators are rebuilt using CPPC's internal structures to match the ones present before the failure.

The second major part is denoted as consistency recovery, and its function is to recreate the application state as it was prior to any failures. This is accomplished by first negotiat-

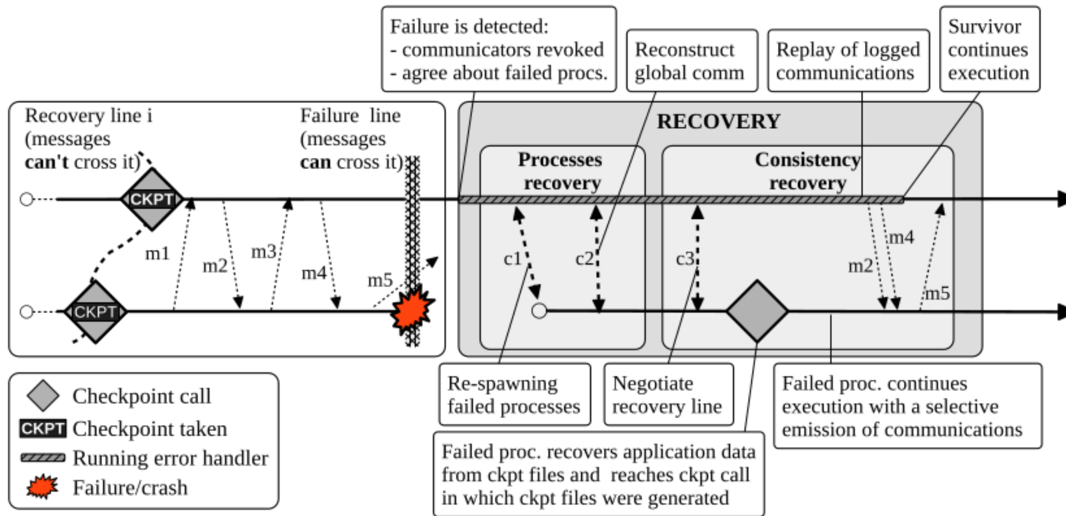

Figure 3.3: CPPC-resilience local rollback recovery process.

ing the recovery line, which entails finding the most recent checkpoint that is present in all processes. Once this is done, the checkpoint is read into memory and all data stored on it is assigned to the correspondent variables. Lastly comes the replay of logged communications. The failed processes request the survivor to re-send all logged communications from the last checkpoint until the point of failure. This is possible thanks to the message logging protocol that we will now describe more thoroughly. When the communications have been replayed, all processes are in a consistent and up to date state and can resume normal execution.

3.3.5 Message logging protocol

Message logging protocols must record two fundamental kinds of data: events and the content of messages. An event corresponds with a computational or communication step of a process that, given a preceding state, leads the process to a new state. The content is the payload that an event is supposed to deliver: it is the actual message data to be exchanged by two processes.

By executing all these messages in the correct order a process can be brought from the latest checkpoint to its original state before the failure. Two kinds of approaches could be made for message storage: (1) Receiver-based and (2) sender-based. In (1) processes make the local copy of the messages in the receiver side. This method has the advantage of having the log locally available upon recovery, but the disadvantage of having to save the messages to stable storage in order to not lose them upon process failure. Approach (2) stores the payload in the sender side. This provides better performance [14] as the local copy can be made in parallel with the network transfer, and the log can be kept in memory. If the process fails, the log will

be lost but it will be recreated during recovery. This strategy has the drawback of having to request the replay of messages by survivor processes during recovery. The message logging protocol in CPPC-resilience uses a sender-based approach, due to increased performance.

There are also certain implementation details that must be taken into account when designing a message logging protocol. An important one is that point-to-point operations (send, recv and their asynchronous variants) can be logged directly using the MPI protocol, whereas collectives would have to be saved as their corresponding individual point-to-point variants. This has 2 problems: (1) the log size would become unnecessarily large and (2) it does not allow for hardware-specific optimizations. Therefore the actual function call will be logged at an application level instead, together with the communicator IDs (CIDs) of the communicators involved, in order to allow for translation into CPPC's custom communicators.

CPPC-resilience was designed for hard failures where a certain amount of processes die completely, therefore, after a failure, they are replaced with new processes by means of respawning. Process respawning is the most costly step of the recovery process, as shown by Nuria Losada et. al. in [15]. However, when dealing with soft errors, the inconsistent process state is temporary and respawning would be unnecessary. In the next chapter we will illustrate how we expanded CPPC-resilience in order to deal with soft errors in this manner, thereby increasing the efficiency of the recovery process.

Chapter 4

Extending CPPC-resilience to cope with soft errors

In this chapter we will detail the main object of this project, which is the design and implementation into CPPC-resilience of a mechanism to recover from soft errors without having to respawn all failed processes, thereby avoiding initialization overheads.

4.1 Introduction

In Section 2.1 we have shown the differences between hard and soft errors. The former consist of physical errors that cause repeated corruption, such as a bit stuck in RAM at ”0” or ”1” or more severely a bus error, whereas the latter are transient in nature and can be caused by phenomena such as cosmic rays, where a high-energy particle carries enough energy to alter the state of a single bit for a short amount of time.

Studies by IBM [16] have shown that these events are not uncommon and that computers typically experience about one cosmic-ray-induced error per 256 megabytes of RAM per month. Mechanisms such as ECC memory are used in order to detect and correct such soft errors whenever possible by means of Hamming codes or TMR (Triple Modular Redundancy).

However in cases where these mechanisms are not available (e.g. in some GPUs) or where the corruption spans too many bits for error correction algorithms to correct or even detect, soft errors can cause Detectable Uncorrectable Errors (DUE) and/or Silent Data Corruption (SDC), which can in turn lead to undefined behavior in an MPI program, such as total program abortion or possibly worse, incorrect results without knowing (e.g. wrong orbital calculations for a space probe causing a fatal crash).

In this project we will extend CPPC-resilience to deal with DUEs and SDCs that are detected by software mechanisms such as the ones described in [17] and [18].

4.2 Simulating soft errors for CPPC-resilience

Since soft error detection techniques and means of notification to the Operating System are still under research and outside of the scope of this project, we will resort to simulating them. As CPPC is intended for UNIX-like systems, we can use signals, which are a form of inter-process communications (IPC).

Under Linux, we can define a signal handler in the manner shown in Figure 4.1. The aforementioned code snippet registers a hypothetical signal SIG_SOFT_ERROR so that the function sig_handler() will be called whenever that signal is raised by the OS or another process by means of the function raise().

```
#include <signal.h>

void sig_handler(int signum)
{
    if (signum == SIG_SOFT_ERROR) {
        printf("Soft error detected");

        // perform appropriate action
    }
}

int main()
{
    signal(SIG_SOFT_ERROR, sig_handler);

    // wait (i.e. busy loop)
}
```

Figure 4.1: C code for a signal handler under Linux

In practice we will use signal SIGUSR1 (predefined Linux signal for user-defined actions) as no signal currently exists that is raised with soft errors, and one would have to compile a custom kernel in order to add new ones [19].

4.3 Soft error detection and propagation

As mentioned before, soft errors will be assumed to raise a UNIX signal with code SIGUSR1, so we will capture this signal. In CPPC, as shown in Figure 2.2, the function CPPC_Init_configuration() is executed before MPI initialization, therefore this is an adequate place to register the signal handler.

One of the first challenges is that CPPC runs as a multi-process program, as each of the MPI processes run by the target application has its own CPPC thread. Therefore, the signal will only be triggered in the process where it was raised and we would normally

not have any way to notify all the other processes of it. We will use the ULFM function `MPIX_Comm_revoke(*global_comm)`, which is described in Table 3.1, inside of the signal handler. This function allows us to interrupt all communications and trigger the custom MPI error handling routine set by CPPC (`mpi_error_handler`) in all processes, thereby giving us a starting global control point from where to handle the recovery.

4.4 Current recovery procedure flow

We will use the local recovery branch of CPPC-resilience as a starting point for the development of our project. As stated previously CPPC-resilience is currently only designed to deal with hard failures, so it assumes that when it has to recover, failed processes are already dead. The flow that CPPC-resilience follows in order to recover from hard failures is illustrated in the sequence diagram in Figure 4.2, that we will now proceed to describe in detail.

Whenever a failure occurs in one of the processes, the error handler `mpi_error_handler` is triggered. Inside this function it is checked whether the error code is equal to the ULFM codes MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED, being raised when the process dies and when the communicators are revoked respectively.

The first relevant function called from inside the error handler by survivor processes (as respawning ones are not available yet) is `relaunchProcesses`, whose job is to detect the broken communicators and shrink them, as shown in Figure 3.2, as an initial step after failure. This is done with the ULFM function `MPI_Comm_shrink()`. Afterwards the respawning takes place inside of the custom function `MPIX_Comm_replace()`, which takes as parameters the old communicator, the new communicator to act as a replacement, and a structure containing meta-information about CPPC.

The first important step inside this function is to detect the number of dead processes. This is done by checking the difference between the sizes of the old communicator (which has not been shrunk and therefore still contains the dead processes) and the new shrunk one. After this is done, the parameters for the spawn function `MPI_Comm_spawn_multiple()` (shown in Table 4.1) are prepared and said function is called. Once the new processes have spawned, their original ranks are preserved using the MPI function `MPI_Comm_split()`. From here onwards the new processes continue the program flow alongside survivors.

Before returning, `relaunchProcesses()` also sets the global flags `migrationParameter` and `goBackMigration` to true. Later on, this will signal to other functions that there is a migration in process.

Function	Parameters
MPI_Comm_replace	oldcomm: old communicator to be replaced newcomm: new communicator state: CPPC controller state information
MPI_Comm_spawn_multiple	count: number of commands array_of_commands: programs to be executed (as string) array_of_argv: arguments to previous programs array_of_maxprocs: max processes to be spawned for each command root: rank of process in which previous arguments are examined comm: intracommunicator containing group of spawning processes intercomm: intercommunicator between original group and newly spawned group array_of_errcodes: one error code per process
MPI_Comm_split	comm: communicator to be split color: processes with the same color are in the same new communicator key: control of rank assigment newcomm: new communicator

Table 4.1: Relevant ULFM, MPI and custom functions used during the respawning process.

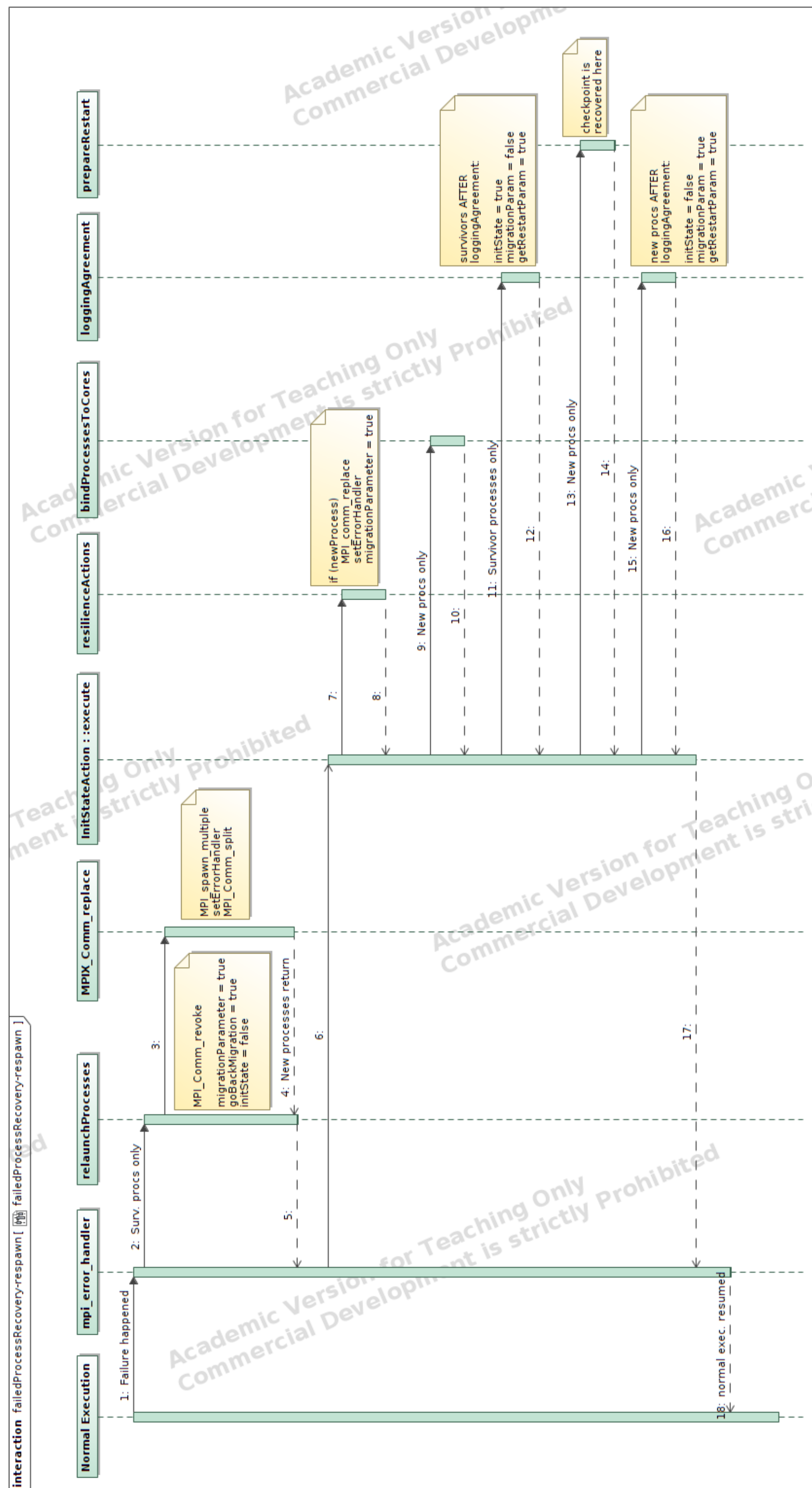

Figure 4.2: Sequence diagram illustrating the recovery process in CPPC-resilience.

Next, `InitStateAction` is instanced, which is one of many ”xxxAction” classes defined in CPPC with a common interface that exposes the class method `execute()`. This method coordinates the recovery, and starts by calling `resilienceActions()`. Here, new processes get their communicators replaced with the new CPPC ones and their error handler set to the custom one. The global flag `migrationParameter` is also set to true for new processes at this point.

The next routine called by `InitStateAction::execute()` is `bindProcessesToCores()`, which, as the name implies, sets the bindings between processes and physical processor cores for the new processes. Afterwards, `loggingAgreement()` is called. This function is responsible for the local recovery using the message logging protocol and it is to be called twice, first for the survivor processes and then for the newly created ones when they have already recovered the checkpoint. On its first execution, survivors agree here about the last complete checkpoint line from which to recover, and they are instructed to wait for requests in order to reply to logging requests (i.e. to replay communications since the last checkpoint).

Following the sequence diagram, we can see that new processes now call `prepareRestart()`, whose function is to recover from the checkpoint. First it determines which actual file on the filesystem contains the latest valid checkpoint, and then it reads it into memory and restores all required program state.

Now comes the second execution of `loggingAgreement()` (this time by the new processes) in order to request the replay of communications by the survivors. As commented in Section 3.3.4, CPPC-resilience uses a sender-based approach to storing messages because it allows for a more efficient implementation. Communications are replayed from the checkpoint until the point of failure in the original execution. Once this is done, the recovery is complete and the program resumes normal execution.

4.5 Modifications for soft error handling

Figure 4.3 shows the sequence diagram for the recovery when handling a soft error, although it must be kept in mind that hard failures can still be handled, it just will not be represented. We will now highlight the differences between this and the diagram for hard failures (Figure 4.2).

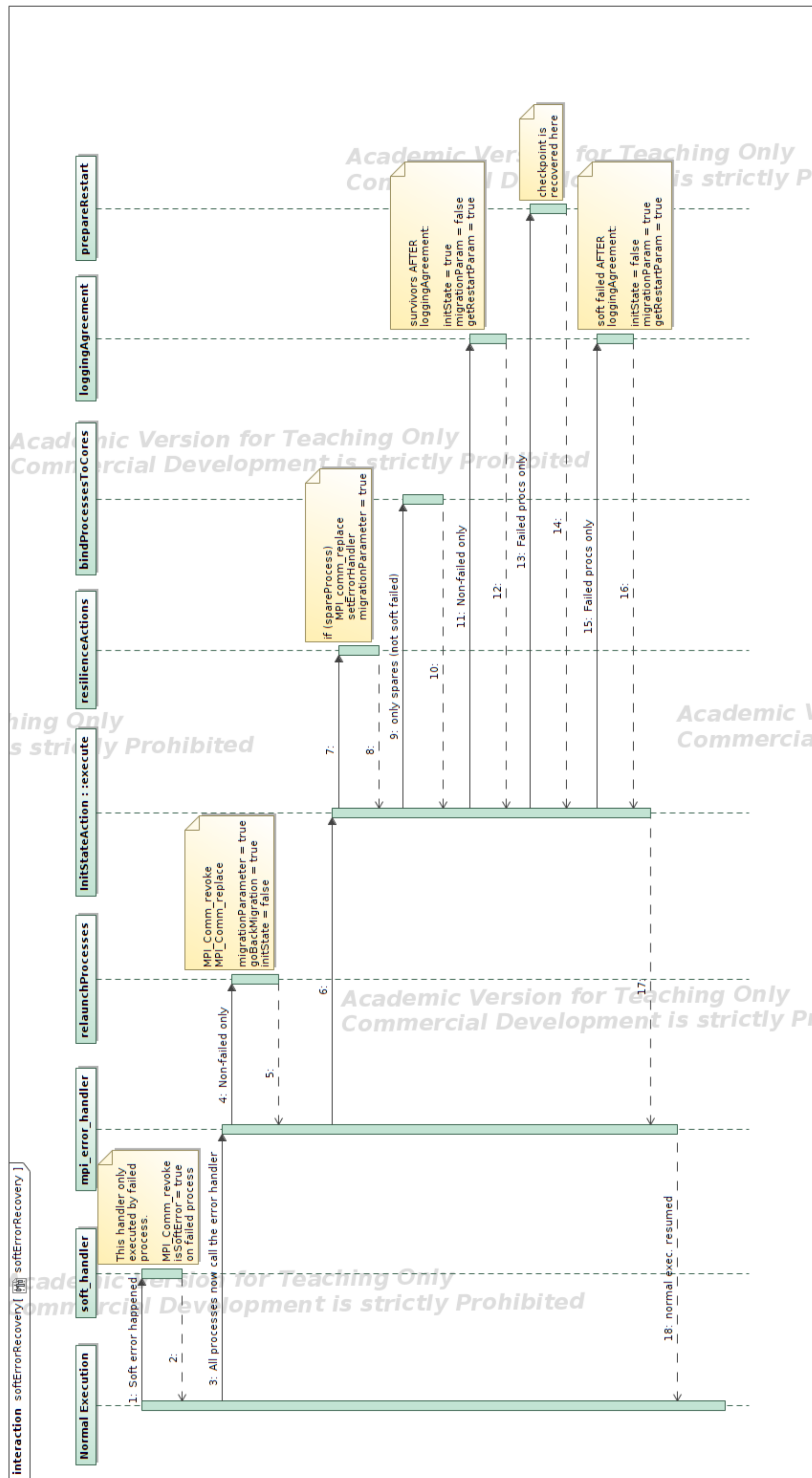

Figure 4.3: Sequence diagram illustrating the recovery process for soft errors in CPPC-resilience.

In order to simulate hard failures, CPPC-resilience used an iteration counter that killed a process with `raise(SIGKILL)` inside of the checkpointer class. The frequency and number of processes to be killed is configured with environment variables prior to execution. We can take advantage of this structure and use it to raise the soft error signal, therefore we will create our own routine `softSimFailure()` and substitute the call `raise(SIGKILL)` with `raise(SIGUSR1)`, this particular signal code being used for the reasons stated in Section 4.2.

The next step is to catch the signal and propagate it as described in Section 4.3, because, as contrary to hard failures, this is only detected by the single process which receives the signal. We propagate this error by interrupting the communicator with the ULFM function `MPI_Comm_revoke`, so the MPI error handler is called on all processes. However, before exiting the signal handler we set a global flag `isSoftError` to true in the failed process.

The first action inside of the MPI error handler is to check whether the flag `isSoftError` is set, and if so, `setFailedFlag()` is called in order to set the flag that is used by the other functions to identify this as a failed process and perform checkpoint recovery on it.

Afterwards, `relaunchProcesses()` is called in the non-failed processes, which does the same thing as it did with hard failures, revoking and reconfiguring communicators. This function is not called on soft failed processes since the communicators have already been revoked and they will be reconfigured later, when the checkpoint is to be recovered. The global flags `migrationParameter` and `goBackMigration` are also set to true. Later on, this will signal to other functions that there is a migration in process. Since the soft failed process is marked as failed, it is simply ignored inside this function.

In `InitStateAction::execute`, the same procedure as before is used to recover from hard failures. The difference now is that processes marked as soft failed are prevented from calling `bindProcessesToCores()`, as unlike processes to be respawned they are alive and already allocated in the proper core. Soft failed processes also do not go into `loggingAgreement()` in its first execution. Recall that `loggingAgreement()` must be run twice, first by the non-failed processes in order to agree about the checkpoint line and to wait for replay requests from the logging protocol, and afterwards by the failed ones in order to actually send the replay commands.

Between the `loggingAgreement()` calls, soft failed processes now also call `prepareRestart()`, which is the function that actually loads the checkpoint file from disk to memory. The procedure from here onwards is the same as if it was a hard failure (aside from the minor detail of removing the soft failure flag), because at this point the checkpoint is already recovered and the communications replayed.

In the next chapter we will study the efficiency of these changes by means of benchmark-

ing with 3 MPI applications with different characteristics, verifying that our proposal works and indeed allows us to significantly reduce the overhead time associated to program recovery when dealing with soft errors.

Chapter 5

Experimental evaluation

This chapter is dedicated to the experimental evaluation of the proposed extension to CPPC-resilience. We will present in detail the working environment, the different benchmarking methods used and the results obtained.

5.1 Testing environment

As CPPC-resilience is intended to run on HPC systems, if we want to be realistic with our experiments we must do the testing in such a system. Therefore, the Pluton HPC cluster [3] was used. This cluster has the specifications shown in Table 5.1, and uses the Slurm Workload Manager [20] as a job scheduling system for running HPC applications in its computing nodes. A special frontend node is provided for SSH access, and then individual jobs must be scheduled with Slurm, specifying the resources required (RAM, number of cores, GPUs, etc.) and the commands to be executed.

As for software, we have used the CPPC-resilience branch of CPPC available at `https://bitbucket.org/nuriallv/cppc`, working with HDF5 version 1.8.11 and GCC

Pluton HPC cluster	Hardware details
Nodes	23 computing nodes + 1 frontend node
Processor	2x Intel(R) Xeon(R) Silver 4110 (on each node)
Cores/threads	16/32 per node
Memory	64 GB per node
Storage	Total 16 TiB in RAID10 configuration
Network	Infiniband FDR@56Gbps and Gigabit Ethernet
OS	Rocks 7 (CentOS 7 based)
MPI version	ULFM development branch
GCC	v4.8.5

Table 5.1: Hardware platform description.

version 4.8.5. The MPI version used corresponds with the ULFM development branch of the OpenMPI distribution, located at `https://bitbucket.org/nuriallv/ulfm2`.

5.2 Testbed

In order to test the efficiency of CPPC and our soft error handling modifications, we will use a series of benchmarks. The testbed will consist in the execution of 3 different scientific MPI applications on the hardware platform described in Section 5.1:

- MOCFE-bone [21]: written in Fortran, simulates the main procedures in a 3D method of characteristics (MOC) code for the numerical solution of the steady state neutron transport equation. 3D-MOC was chosen in exascale computing because of its heterogeneous geometry capability, high degree of accuracy, and potential for scalability.
- Himeno [22]: written in Fortran, designed to evaluate the performance of incompressible fluid analysis code. This benchmark program takes measurements to proceed major loops in solving the Poisson's equation solution using the Jacobi iteration method.
- TeaLeaf [23]: written in C, solves the linear heat conduction equation on a spatially decomposed regularly grid using a 5 point stencil with implicit solvers.

The scripts to run the benchmarks have a defined set of parameters shown here:
`run.sh [bmname] [class] [nprocs] [times] [killnum] [place]`

- `bmname`: benchmark name {tealeaf, himeno, mocfe}.
- `class`: dataset to use as input to the program.
- `nprocs`: total number of processes.
- `times`: number of times to send soft error signals.
- `killnum`: number of signals to send each time.
- `place`: where to store the checkpoint files in the filesystem.

5.3 Measuring spawn timings in CPPC-resilience

The current version of CPPC-resilience has 2 operating modes, CPPC_LOGGING and CPPC_LOGGING_SPARES. The former would spawn new processes whenever a failure occurs, whereas the latter keeps a pool of "hot spares" (already initialized processes), such that the failed ones are migrated to the spares at runtime. The spare processes version avoids the

spawning overhead during recovery but it adds its own overheads: extra processes need to be allocated at the beginning of the execution regardless of whether there are failures or not and, in case of failure, data must be recovered and transferred to the replacement processes before the execution can continue.

Our proposed branch, CPPC_SOFT, is a modification of the CPPC_LOGGING branch in order to avoid the re-spawning of alive processes in case of soft errors. Thus, to measure the efficiency of our proposal, we will need to compare the execution time of the CPPC_LOGGING version and the CPPC_SOFT one. Unfortunately, the CPPC_LOGGING branch is currently in an unusable state, due to bugs with the code that surfaced with the new ULFM specifications and implementations.

To overcome this issue, we will estimate the execution time needed by CPPC_LOGGING, using the execution time needed by CPPC_LOGGING_SPARES and adding the time needed to spawn new processes. A small benchmark program will be used in order to determine time spent in the creation of new processes. The pseudocode of said program is shown in Figure 5.1. It begins by selecting a random rank from within the total set to be the one who will be the root when spawning the new process. This rank is sent to all other processes by means of a broadcast collective, `MPI_Bcast`. The parameters of this function are, in order: the buffer with the variables to send, the number of variables, the type of said variables, the root process for the collective (the one that initially contains the variable) and the communicator to be used. Afterwards, all ranks make the collective call to spawn the new process, which, in our case, consists of a bogus worker that simply prints out its assigned rank. Finally there is a call to `MPI_Barrier`, in order to ensure that the collective finishes in all ranks before ending the time measurement.

The time elapsed from before the collective call until after the barrier is reached by all processes is measured, and it serves as an estimation of the spawn times in CPPC-resilience. It must be kept in mind that CPPC-resilience performs many more tasks when recovering from a failure, such as rebuilding communicators, internal state structures, and so on. However, those are also performed by the CPPC_LOGGING_SPARES branch and need not be added to the recovery time.

In order to mitigate statistical outliers, this program was ran 5 times for 2^x slaves, where $x = \{1 - 8\}$. The average time of these 5 executions was taken and the resulting graph is shown in Figure 5.2.

If we look at the graph on a logarithmic scale (Figure 5.3), we can see that the time scales linearly with the number of processes until it reaches 16, where it starts to take considerably longer. This is due to two facts: (1) if there are more processes involved in a collective operation, the total number of communications increases; and (2) each computing node has a total of 16 processor cores, and after this number, inter-node communications start occurring,

```
// spawn.c
int main()
{
    MPI_Init...
    MPI_Comm_size(&nprocs, ...
    MPI_Comm_rank(&myrank, ...

    // which rank should spawn the new process
    int ranktospawn;

    if (myrank == 0)
        ranktospawn = random(0, nprocs - 1);

    t1 = get_time();

    // send which rank should be the root of the collective
    operation
    MPI_Bcast(&ranktospawn, 1, MPI_INT, 0, MPI_COMM_WORLD);
    // spawn collective operation
    MPI_Comm_spawn("worker", 1, ...);
    // wait for it to finish everywhere
    MPI_Barrier(MPI_COMM_WORLD);

    t2 = get_time();
    print("time to spawn = %d", t2 - t1);
    MPI_Finalize();
}

// worker.c
int main()
{
    MPI_Init...
    MPI_Comm_rank(&myrank, ...

    print("worker %d spawned", myrank);
    MPI_Finalize();
}
```

Figure 5.1: MPI C pseudocode for the spawning benchmark.

which are significantly more expensive.

5.4 Experiments performed

In this section we will describe all the different experiments that have been done in order to evaluate different aspects of our custom CPPC-resilience branch. As done in Section 5.3, the methodology will be to run each benchmark variation 5 times and to take the average execution time, for all 3 applications.

5.4.1 Solution validation

Our first step was to check that our soft error branch functions correctly. In order to do this, the presence of various soft errors was simulated during the execution of each of the benchmark applications. Then, it was verified that in all cases said applications continued with the execution flawlessly after the failures and finished with the correct results.

In order to validate our program a series of debug log statements were inserted throughout the program to inform us about the state of the recovery. To check for correct results, the test values provided by the benchmark datasets were used to compare against the ones returned by our custom branch. An excerpt from the execution log of `tealeaf` with 16 processes can be found in Figure 5.4. In this log we see:

- Lines [1-3]: all processes are initialized in CPPC-resilience.
- Lines [5-7]: normal steps of the benchmark algorithm.
- Lines [9-11]: after a configured number of iterations, a checkpoint is performed in all processes.
- Lines [15-18]: a soft error signal has been detected by the error handler, failed and survivor processes are enumerated.
- Lines [20-22]: the checkpoint line (last valid checkpoint in all processes) is selected.
- Lines [25-27]: the failed process' state is restored using the checkpoint file.
- Lines [28-34]: the message log is attached in the survivors that communicated with the failed process, and those communications are replayed.
- Lines [36-41]: benchmark execution completes with correct results.

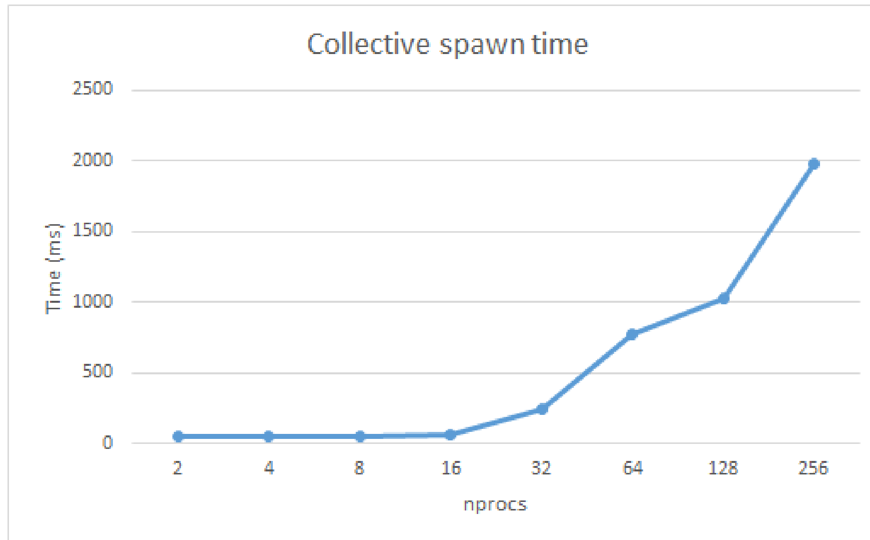

Figure 5.2: Time taken to spawn a process with different amounts of MPI processes.

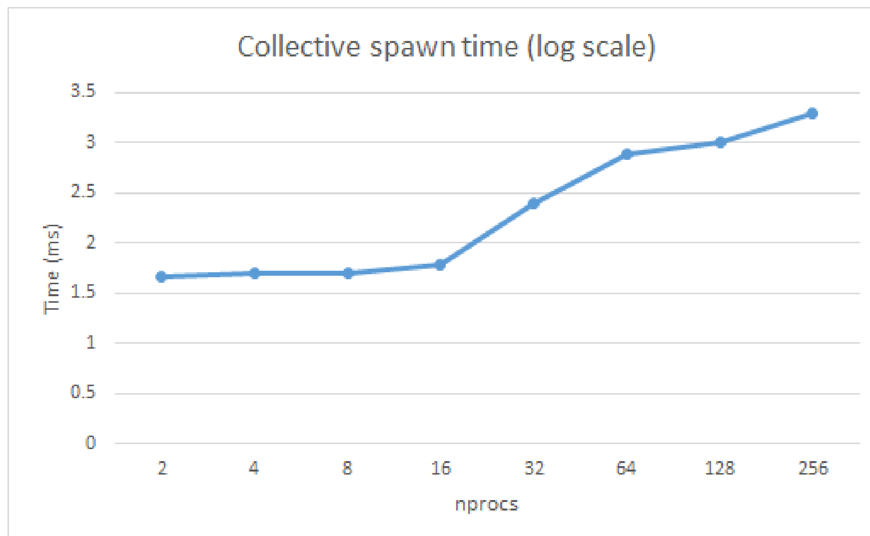

Figure 5.3: Time taken to spawn a process with different amounts of MPI processes (logarithmic scale).

```
// process initialization
Rank 0 entered CPPC_Init_state
Rank 1 entered CPPC_Init_state
...
// steps of the benchmark algorithm
Timestep 1
Timestep 2
...
// checkpoint performed in all processes
CKPT_AUXILIARYTHREAD_PROC0|0.0232508|
CKPT_AUXILIARYTHREAD_PROC1|0.0235608|
...
Timestep 13
// soft error detected
**CPPCSOFTRESET[P15,it15]:afterRunning|10.2873|
Detected soft fault signal at rank 15
// survivor processes enumerated
MPI_ERR_HANDLER: PROCESSES 1,2, ... SURVIVED (isSoftError = 0)
...
// recovery line decided in all survivors
P2 survivor chooses ckpt 1
P3 survivor chooses ckpt 1
...
// checkpoint file found
isFullCheckpoint: p15 checkpoint path: /scratch/CPPC/15_0/0.cppc
// and loaded in soft failed process
P15_T0: Restarting from full checkpoint
// log attached in required procs, ready to replay commands
43344|Survivor 1 attach log in|0.000000|size|35577120|
43354|Survivor 14 attach log in|0.000000|size|35577120|
...
// processes finished replay
**PROC1 FINISH CPPC LOG
**PROC14 FINISH CPPC LOG
...
Timestep 20
// run finished
Checking results...
Expected 1.012100932683400e+02
Actual   1.012100932683323e+02
This run PASSED (Difference is within 0.00000000%)
```

Figure 5.4: Output log for execution of CPPC_SOFT branch for the tealeaf application.

5.4.2 Overhead without failures

Next, we measured the overhead of our proposal without failures. For that, we ran the 3 benchmarks without any sort of FT capabilities (orig), with CPPC-resilience respawning failed processes (respawn), and with special soft error handling (soft).

The procedure taken to measure time is to use the MPI function `MPI_Wtime()`, which returns a double-precision floating point number representing the time since the last call to itself. The pseudocode of this procedure is shown in Figure 5.5.

```
int main()
{
    double t1, t2;

    MPI_Init...
    t1 = MPI_Wtime();
    // do work...
    t2 = MPI_Wtime();
    printf("Time measured by MPI_Wtime: %1.2f\n", t2-t1);
    MPI_Finalize();
}
```

Figure 5.5: C pseudocode for time measuring using MPI_Wtime

Figure 5.6 shows the results obtained for this experiment with 16 processes. We can observe that the introduction of FT capabilities has a significant amount of overhead on any version, although it is important to note that this scenario is not particularly realistic as it employs a set of very short-lived benchmarks. These kinds of fault-tolerant solutions are designed for programs running for hours or even days, where a total program crash would imply large electricity and hardware costs.

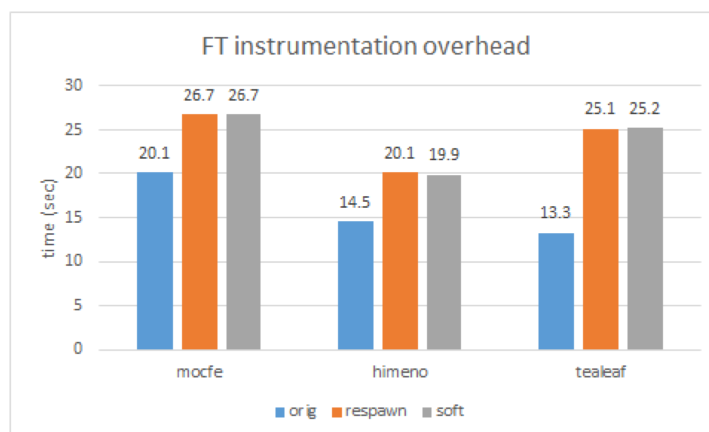

Figure 5.6: Execution time comparison for all variants of each benchmark, with no failures and nprocs = 16

Aside from that, there is no major difference between CPPC-resilience branches, since as there is no recovery involved the program does not even branch into the code that we have modified, so the differences are mere statistical deviations.

Another thing to point out are the disparities between overheads in the different benchmarks (e.g. in `tealeaf` is 11.8 seconds, or 47%). This can be attributed to various factors:

- **Complexity of the actual work performed:** if the calculations performed take up a lot more CPU cycles for one of the benchmarks, this will mask the time spent on FT tasks.

- **Complexity and size of data:** certain benchmarks have more intricate data structures, which in turn take up more space and take longer to store. This is especially relevant when saving/restoring checkpoints to/from an optical hard drive, which is a high latency device.

- **Cache invalidation:** if the benchmark is optimized to fit in the cache, the added instructions dedicated to FT tasks can severely impact the performance of the actual calculations.

5.4.3 Improvement with soft errors

In this section we will measure the improvement obtained with our proposal when handling soft errors. Both in the CPPC_LOGGING and CPPC_SOFT branches, the execution flow of a program consists of the following steps:

- **Spawn:** time it takes the MPI library to launch the actual process on the operating system, for the first time.

- **Initialization:** setup of all CPPC-related structures and variables, and MPI initialization.

- **Work:** actual calculations, different for each benchmark.

- **Checkpoint:** time taken to write all of the checkpoints to stable storage.

- **Recovery:** time taken to recover from a failure and resume the execution.

- **Finalization:** freeing of CPPC and MPI-allocated resources.

Aside from the spawning times already measured in Section 5.3, these steps will be measured using the timing structures already inside CPPC. These consist of a series of C `structs`

storing times at different points of the execution, such as after initialization, after each checkpoint, after recovery, etc. An example of some of the content of this struct (output reduced for brevity, many more measurements in the real program) is provided in Figure 5.7.

```
TIMES|RANK|TChkpt|TFinDetect|Trevoke|TShrink|
TIMES|0|0.00451507|10.4762|1.387e-05|0.52582|
TIMES|1|0.00456191|10.4762|1.8256e-05|0.525821|
TIMES|2|0.00398097|10.4762|1.5674e-05|0.525797|
TIMES|3|0.00423796|10.4762|1.4766e-05|0.525762|
TIMES|4|0.00413456|10.4763|2.0942e-05|0.525714|
TIMES|5|0.00391636|10.4762|1.4854e-05|0.525826|
TIMES|6|0.00520496|10.4762|1.2049e-05|0.52578|
TIMES|7|0.00435946|10.4763|5.436e-06|0.52572|
TIMES|8|0.00431812|10.4762|1.3589e-05|0.525819|
TIMES|9|0.00432258|10.4762|8.602e-06|0.525816|
TIMES|10|0.00447206|10.4762|2.0819e-05|0.525819|
TIMES|11|0.00504054|10.4762|1.367e-05|0.525776|
TIMES|12|0.00446098|10.4762|1.2414e-05|0.525849|
TIMES|13|0.00485647|10.4762|1.2045e-05|0.525805|
TIMES|14|0.00488027|10.4762|1.1871e-05|0.525832|
TIMES|15(FAILED)|0.0053019|-1|-1|0.000338895|
```

Figure 5.7: CPPC timing struct dump, for a benchmark with 16 processes.

All 3 benchmarks were run on the CPPC_LOGGING branch with a total of 16 and 64 processes, and the results obtained can be seen in Figures 5.8 and 5.9. Even with such a small number of processes, we can see that the recovery phase constitutes a big chunk of the total execution time, and this chunk becomes more significant the more processes are involved.

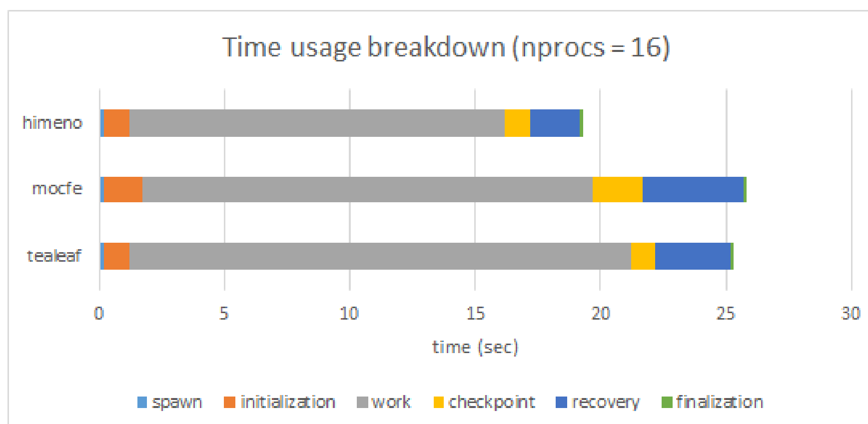

Figure 5.8: Execution time breakdown for a failed process for each benchmark, with nprocs = 16.

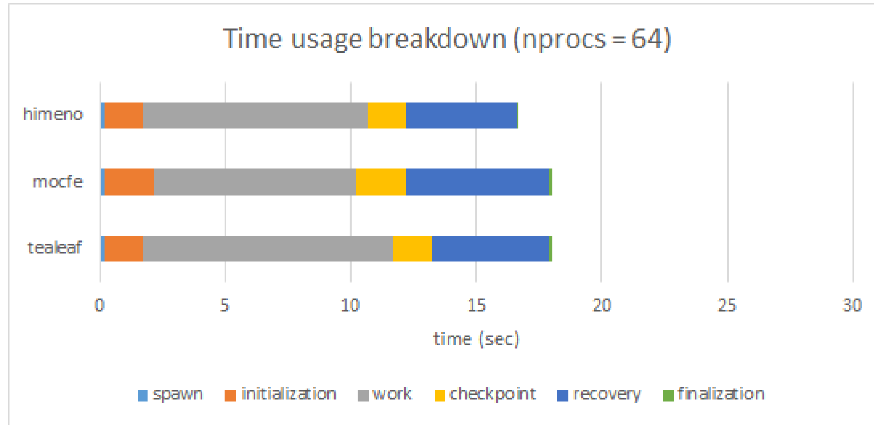

Figure 5.9: Execution time breakdown for a failed process for each benchmark, with nprocs = 64.

In CPPC_SOFT, all broken down times match, except for recovery time, which will be smaller with our proposal as process respawning is avoided. Thus, to address the main object of this project which is the efficiency of handling soft errors as a special case, we have conducted a last experiment that compares the differences of recovery times after a failure between the respawning and the soft error branches of CPPC-resilience.

Figure 5.10 shows the reduction for the recovery time for nprocs = {16, 64, 128, 256}. The data for this graph was taken from the same structs used earlier that annotate times at different points of the program, and the percentage was calculated as $p = \frac{t_{respawn} - t_{soft}}{t_{respawn}} * 100$, where $t_{respawn}$ is the time taken to complete the recovery step by the CPPC_LOGGING version and t_{soft} by the CPPC_SOFT version.

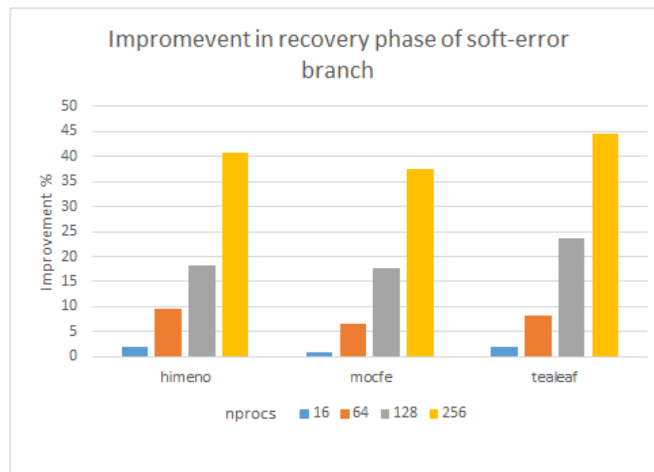

Figure 5.10: Improvement in the recovery phase when using our soft-error CPPC-resilience branch.

We can draw various conclusions from this graph: (1) There is a clear improvement in the handling of soft errors as a special case. (2) This improvement is highly dependant on the total number of processes involved, as evidently reconfiguring communicators with 256 ranks is slower than with 16, and the same is true about coming to an agreement about which checkpoint line to use. (3) The overhead percentages have disparities between the different benchmarks (e.g. for himeno it is only a 10% increase with 128 processes whereas for tealeaf it is nearly 20%), this indicates that the complexity of the data and communicator topologies to be recovered have an effect on recovery performance. This is application-specific and in principle unavoidable without ad-hoc solutions.

Chapter 6

Concluding remarks

In this last chapter, the final implications of this project along with extra information about the author's experience and thoughts during the development of this project will be shown.

6.1 Project conclusions

This project presents the extension of the fault tolerance tool CPPC-resilience in order to be able to deal efficiently with soft errors in HPC systems. Due to the increasing number of processors involved per system and the decrease in transistor sizes, soft errors are expected to become a common occurrence in the near future. The problem is that when soft errors are treated as hard failures, that is, fully restarting the failed processes by means of a stop-and-restart approach, it leads to significant execution overheads and energy waste.

Our proposal treats soft errors as a special case, making sure that those processes where such an error has happened are recovered without re-spawning, because the nature of soft errors is transitory and the processes do not need to be restarted. The results obtained in Chapter 5 show the following facts:

- There is a clear **improvement** in the **recovery phase** when handling soft errors as a special case in fault tolerance recovery frameworks. As an example, for the `tealeaf` benchmark with 256 processes, we have reduced recovery time by nearly 45 percent.

- This improvement is highly **dependant** on the total **number of processes** involved, larger amounts of processes having significant performance gains. This is specially important as this project is oriented towards preparing for failures in future exascale systems where failures are statistically unavoidable and the number of processes could be in the order of tens of thousands.

6.2 Lessons learned

In the academic domain, this project has given me the opportunity to learn new concepts in various disciplines. I have learned about MPI's cutting-edge fault tolerance interfaces, working with HPC clusters and job scheduling systems, and about writing a technical document with LaTeX in order to keep a record of all the work done.

In the technical domain, CPPC is a complex and large project and the source code is written in a rather convoluted way, with global variables that affect code in other files in a cascading manner. Due to this, a lot of time was wasted trying to understand all the small details and intricacies in the code. If I were to start this project over from scratch, I would have spent a good amount of time at the start re-writing certain parts of CPPC-resilience, in order to give myself an easier time trying to integrate my new changes into the codebase.

6.3 Future work

While we saw in Chapter 5 that there is already an improvement in the special handling of soft errors, there are still some unaddressed overheads with communicator reconfiguration and recreation that could be avoided, as the transitory nature of soft errors implies that the communicators could still be valid after a failure. This may require more in-depth research in order to make sure that the data structures that hold the communicator metadata do not get corrupted by the soft error and are rendered useless. It was not addressed in this project as it would require a major redesign and rewrite of CPPC-resilience and it would be far too much work for a Bachelor's thesis.

List of Acronyms

HPC *High Performance Computing.*

MPI *Message Passing Interface*

ULFM *User Level Failure Mitigation*

MTFB *Mean Time Between Failures*

SIMD *Single Instruction Multiple Data*

ECC *Error Correcting Code*

FT *Fault Tolerant/Tolerance*

DUE *Detectable Uncorrectable Error*

SDC *Silent Data Corruption*

CPPC *Controller/comPiler for Portable Checkpointing*

Glossary

Refactoring The process of restructuring existing source code without changing its behavior.

Application resilience The resistance to failures and ability to continue its execution normally after one occurs.

Handler A function to be called after a certain event is triggered.

struct A data structure containing different data members, common in C-like languages.

Vanilla Said of something unmodified, default.

Bibliography

[1] R. Y. Dongarra J., Herault T., "Fault tolerance techniques for high-performance computing," *Springer International Publishing*, 2015.

[2] N. Losada, "Application-level fault tolerance and resilience in HPC applications," *PhD Thesis, University of A Coruña*, 2018. [Online]. Available: http://hdl.handle.net/2183/21451

[3] "Pluton HPC cluster." [Online]. Available: http://pluton.des.udc.es/

[4] L. Torvalds, "Git, distributed version control system." [Online]. Available: https://git-scm.com/

[5] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental concepts of dependability." [Online]. Available: https://pld.ttu.ee/IAF0530/16/avi1.pdf

[6] A. Geist, "Supercomputing's monster in the closet." *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[7] "MPICH." [Online]. Available: https://www.mpich.org/

[8] "Open MPI." [Online]. Available: https://www.open-mpi.org/

[9] G. Rodríguez, M. J. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[10] "The CPPC project." [Online]. Available: http://cppc.des.udc.es/

[11] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[12] "HDF5 website." [Online]. Available: https://support.hdfgroup.org/HDF5/whatishdf5.html

[13] "ULFM specification draft," 2017. [Online]. Available: https://fault-tolerance.org/wp-content/uploads/2012/10/20170221-ft.pdf

[14] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," *Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, 2000.

[15] N. Losada, "Fault-tolerance of MPI applications in exascale systems: the ULFM solution," *Elsevier*, 2019.

[16] J. F. Ziegler, "Terrestrial cosmic ray intensities," *IBM Journal of Research and Development*, vol. 42, no. 1, pp. 117–140, 1998.

[17] L. Bautista-Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 595–602, 2015.

[18] G. Pawelczak, S. McIntosh-Smith, J. Price, and M. Martineau, "Application-based fault tolerance techniques for fully protecting sparse matrix solvers." *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 733–740, 2017.

[19] L. Torvalds, "Kernel source code for signal number definitions." [Online]. Available: https://github.com/torvalds/linux/blob/6f0d349d922ba44e4348a17a78ea51b7135965b1/include/uapi/asm-generic/signal.h

[20] SchedMD, "Slurm workload management system." [Online]. Available: https://slurm.schedmd.com/archive/slurm-19.05.2/overview.html

[21] E. Wolters and M. Smith, "Mocfe-bone: the 3d moc mini-application for exascale research," *University of North Texas Libraries, UNT Digital Library*, 2013.

[22] R. Himeno, "Himeno fortran benchmark." [Online]. Available: http://accc.riken.jp/en/supercom/documents/himenobmt/

[23] UK-MAC, "Tealeaf C benchmark." [Online]. Available: https://uk-mac.github.io/TeaLeaf/