Doctoral Thesis

Fault-tolerance and malleability in parallel message-passing applications

Iván Cores González

2015

UNIVERSIDADE DA CORUÑA

Fault-tolerance and malleability in parallel message-passing applications

Iván Cores González

Doctoral Thesis

September 2015

PhD Advisors:

María José Martín Santamaría
Patricia González Gómez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dra. María José Martín Santamaría
Profesora Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade de A Coruña

Dra. Patricia González Gómez
Profesora Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade de A Coruña

CERTIFICAN

Que la memoria titulada "*Fault-tolerance and malleability in parallel message-passing applications*" ha sido realizada por D. Iván Cores González bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 28 de Septiembre de 2015

Fdo.: María José Martín Santamaría
Directora de la Tesis Doctoral

Fdo.: Patricia González Gómez
Directora de la Tesis Doctoral

Fdo.: Iván Cores González
Autor de la Tesis Doctoral

Acknowledgments

I am not a man of great speeches, but that does not mean I am not hugely grateful. First of all, I would specially like to acknowledge my PhD advisors, María and Patricia, for all the time they have dedicated to me, and of course, to Gabriel, for sharing his knowledge when I needed it. Big thanks also to all the colleagues at the Computer Architecture Group for all these great moments in the Lab 0.2, those of hard work, but also fellowship and good times.

I do not want to forget my family, specially my parents, and Paula. There are no words to explain all I have to thank.

I gratefully give thanks to Emmanuel Jeannot and the people in the Runtime Group in the INRIA Bordeaux Sud-Ouest centre for hosting me during my research visit.

I also want to thank the following institutions for funding this work: Computer Architecture Group and the Department of Electronic and Systems at the University of A Coruña for the human and material support, the Spanish Government for projects TIN2010-16735 and TIN2013-42148-P (both cofunded by FEDER funds of the European Union) and FPI Grant BES-2011-045050, and the Galician Government for project 10PXIB105180PR and the Consolidation Program of Competitive Research Groups ref. GRC2013/055 also funded by FEDER funds of the EU.

Iván

Resumo

Esta tese explora solucións para tolerancia a fallos e maleabilidade baseadas en técnicas de checkpoint e reinicio para aplicacións de pase de mensaxes. No campo da tolerancia a fallos, esta tese contribúe mellorando o factor que máis incrementa a sobrecarga, o custo de E/S no envorcado dos ficheiros de estado, propoñendo diferentes técnicas para reducir o tamaño dos ficheiros de checkpoint. Ademais, tamén se propón un mecanismo de migración de procesos baseado en checkpointing. Esto permite a migración proactiva de procesos desde nodos que están a piques de fallar, evitando un reinicio completo da execución e mellorando a resistencia a fallos da aplicación. Finalmente, esta tese presenta unha proposta para transformar de forma transparente aplicacións MPI en traballos maleables. Esto é, programas paralelos que en tempo de execución son capaces de adaptarse ao número de procesadores dispoñibles no sistema, conseguindo beneficios, como maior productividade, mellor tempo de resposta ou maior resistencia a fallos nos nodos.

Todas as solucións propostas nesta tese foron implementadas a nivel de aplicación, e son independentes da arquitectura hardware, o sistema operativo, a implementación MPI usada, e de calquera *framework* de alto nivel, como os utilizados para o envío de traballos.

Resumen

Esta tesis explora soluciones de tolerancia a fallos y maleabilidad basadas en técnicas de checkpoint y reinicio para aplicaciones de pase de mensajes. En el campo de la tolerancia a fallos, contribuye mejorando el factor que más incrementa la sobrecarga, el coste de E/S en el volcado de los ficheros de estado, proponiendo diferentes técnicas para reducir el tamaño de los ficheros de checkpoint. Además, también se propone un mecanismo de migración de procesos basado en checkpointing. Esto permite la migración proactiva de procesos desde nodos que están a punto de fallar, evitando un reinicio completo de la ejecución y mejorando la resistencia a fallos de la aplicación. Finalmente, se presenta una propuesta para transformar de forma transparente aplicaciones MPI en trabajos maleables. Esto es, programas paralelos que en tiempo de ejecución son capaces de adaptarse al número de procesadores disponibles en el sistema, consiguiendo beneficios, como mayor productividad, mejor tiempo de respuesta y mayor resistencia a fallos en los nodos.

Todas las soluciones propuestas han sido implementadas a nivel de aplicación, siendo independientes de la arquitectura hardware, el sistema operativo, la implementación MPI usada y de cualquier *framework* de alto nivel, como los utilizados para el envío de trabajos.

Abstract

This Thesis focuses on exploring fault-tolerant and malleability solutions, based on checkpoint and restart techniques, for parallel message-passing applications. In the fault-tolerant field, this Thesis contributes to improving the most important overhead factor in checkpointing performance, that is, the I/O cost of the state file dumping, through the proposal of different techniques to reduce the checkpoint file size. In addition, a process migration based on checkpointing is also proposed, that allows for proactively migrating processes from nodes that are about to fail, avoiding the complete restart of the execution and, thus, improving the application resilience. Finally, this Thesis also includes a proposal to transparently transform MPI applications into malleable jobs, that is, parallel programs that are able to adapt their execution to the number of available processors at runtime, which provides important benefits for the end users and the whole system, such as higher productivity and a better response time, or a greater resilience to node failures.

All the solutions proposed in this Thesis have been implemented at the application-level, and they are independent of the hardware architecture, the operating system, or the MPI implementation used, and of any higher-level frameworks, such as job submission frameworks.

Contents

Preface **1**

1. Background **5**

1.1. Checkpointing . . . 5

1.2. The CPPC Framework . . . 9

1.2.1. The CPPC Compiler . . . 11

1.2.2. CPPC Operation . . . 12

2. I/O Optimization in the checkpointing of parallel applications **17**

2.1. Introduction . . . 17

2.2. Incremental checkpointing . . . 18

2.3. Zero-blocks exclusion . . . 19

2.4. Data compression . . . 20

2.5. Implementation . . . 22

2.5.1. Incremental checkpointing and zero-blocks exclusion . . . 22

2.5.2. Data compression . . . 24

2.6. Experimental Evaluation . . . 26

2.6.1. Checkpoint file sizes . . . 27

2.6.2. Checkpoint latency 30

2.6.3. Restart overhead 33

2.7. Related work 36

2.8. Concluding remarks 37

3. Checkpoint-based process migration 39

3.1. Introduction 39

3.2. Process migration using CPPC 40

3.2.1. Negotiation protocol 41

3.2.2. Process migration 45

3.3. Experimental evaluation 48

3.3.1. Scalability 48

Impact of the number of processes 50

Impact of the number of terminating processes 54

Impact of the application size 56

3.3.2. Overhead 56

3.4. Related work 60

3.5. Concluding remarks 61

4. Improving performance in process migration 63

4.1. Introduction 63

4.2. In-memory checkpoint-based migration 64

4.2.1. Experimental evaluation 66

Response Time 67

Evacuation time 68

Overhead . . . 69
4.3. Splitting in-memory checkpoint files . . . 70
4.3.1. Splitting the checkpoint files . . . 71
4.3.2. Implementation issues . . . 74
4.3.3. Experimental evaluation . . . 75
Memory consumption . . . 76
Migration time . . . 78
4.4. Related work . . . 82
4.5. Concluding remarks . . . 82

5. Checkpoint-based virtual malleability 85

5.1. Introduction . . . 85
5.2. Triggering the reconfiguration operation . . . 87
5.3. Scheduling algorithm . . . 87
5.3.1. Monitoring communications . . . 87
5.3.2. Mapping processes to cores . . . 88
5.4. Experimental evaluation . . . 92
5.5. Related work . . . 100
5.6. Concluding remarks . . . 101

Conclusions and Future Work 103

References 109

A. Summary in Spanish 121

A.1. Antecedentes . . . 121

A.2. Optimización de la E/S en el checkpointing de aplicaciones paralelas . 124

A.3. Migración de procesos basada en checkpointing 126

A.4. Mejorando el rendimiento en la migración de procesos 128

A.5. Maleabilidad virtual basada en checkpointing 130

List of Tables

2.1. Baseline checkpoint sizes (in MB) per process 27

2.2. Baseline checkpoint latency (s) 30

2.3. Baseline restart times (s) 33

3.1. Negotiation times and iteration times, running 16 processes 52

3.2. Iteration times (in s) for different number of total processes 52

3.3. Checkpoint file size per process (in MB) and checkpoint write and read times (in s) 52

3.4. Checkpoint file sizes per process (in MB) for different number of total processes 53

3.5. Execution times (in s) using 16 processes, with and without migration 58

3.6. Checkpoint sizes (in MB) per process (running in 16 processes) for CPPC and BLCR 59

5.1. Execution time (s) of the reconfiguration phases. 97

5.2. Scalability of the *Spawn&Rec* step vs total number of processes in the application. 98

5.3. Transfer size (checkpoint size in MB). 99

List of Figures

1.1. Example of valid and invalid recovery lines 8

1.2. Integration of a parallel application with the CPPC framework 9

1.3. Skeleton of an example of MPI code: a matrix diagonalization 13

1.4. CPPC-instrumented matrix diagonalization example code 14

1.5. Spatial coordination for checkpointing 15

1.6. Example of the CPPC configuration file 15

2.1. Example of pattern matching for data compression 21

2.2. Construction of an incremental checkpoint. 23

2.3. Restart from an incremental checkpoint. 24

2.4. Integration of the compression process in the CPPC writing layer . . 25

2.5. Checkpoint sizes per process for 16 processes normalized with respect to the ALC base case (see Table 2.1) 28

2.6. Checkpoint sizes per process for 32-36 processes normalized with respect to the ALC base case (see Table 2.1) 28

2.7. Compressed checkpoint sizes per process for 16 processes normalized with respect to the ALC base case (see Table 2.1) 29

2.8. Compressed checkpoint sizes per process for 32-36 processes normalized with respect to the ALC base case (see Table 2.1) 29

2.9. Checkpoint latency for 16 processes normalized with respect to the ALC base case (see Table 2.2) 31

2.10. Checkpoint latency for 32-36 processes normalized with respect to the ALC base case (see Table 2.2) 31

2.11. Compressed checkpoint latency for 16 processes normalized with respect to the ALC base case (see Table 2.2) 32

2.12. Compressed checkpoint latency for 32-36 processes normalized with respect to the ALC base case (see Table 2.2) 32

2.13. Restart times for 16 processes normalized with respect to the ALC base case (see Table 2.3) 34

2.14. Restart times for 32-36 processes normalized with respect to the ALC base case (see Table 2.3) 34

2.15. Compressed restart times for 16 processes normalized with respect to the ALC base case (see Table 2.3) 35

2.16. Compressed restart times for 32-36 processes normalized with respect to the ALC base case (see Table 2.3) 35

3.1. Inconsistent global state after migration in processes that are running asynchronously 41

3.2. Backward negotiation 42

3.3. Forward negotiation 43

3.4. World communicator reconfiguration 46

3.5. Actions and temporal sequence for four processes involved in a migration operation 49

3.6. Scalability impact when increasing the number of total processes. NPB class B migrating one process 51

3.7. Scalability impact when increasing the number of migrating processes. NPB class B and 16 processes 55

3.8. Impact of increasing application size via the NPB classes. Running on 16 processes 57

3.9. Overhead (in %) for NPB applications (class C - 16 processes) when a node is preemptively migrated (case of CPPC and MVAPICH) and when a node fails (case of Ckpt&Rollback) 59

4.1. Checkpoint-based migration with CPPC. 65

4.2. Modification of the CPPC Writing Layer 66

4.3. Response time (in seconds) in Pluton N1. 67

4.4. Response time for (in seconds) in Pluton N2. 67

4.5. Evacuation time (in seconds) in Pluton N1. 69

4.6. Evacuation time (in seconds) in Pluton N2. 69

4.7. Overhead migrating 1 node (8 processes) in Pluton N1. 70

4.8. Overhead migrating 1 node (8 processes) in Pluton N2. 70

4.9. Structure of the CPPC checkpoint files 72

4.10. CPPC migration with pipelined write-transfer-read steps 74

4.11. Modification of the CPPC writing layer 75

4.12. Checkpoint file size and memory consumption per process in a 32-process execution (in GB). 76

4.13. Memory consumption (in GB) per node (16 cores) in a 32-process execution when different number of the 16 processes per node are migrated. 77

4.14. Migration time in the in-memory CPPC version in a 32-process execution when 16 processes are migrated. 79

4.15. Times of the write-transfer-read steps in the in-memory and the split CPPC versions in a 32-process execution when 4, 8 and 16 processes are migrated. 80

4.16. Times of the *WriteTransferRead* step of the split CPPC version for different chunk sizes. 81

5.1. Steps in the reconfiguration operation 86

5.2. TreeMatch example 89

5.3. TreeMatch example. Step 1: identifying processes to be migrated . . 91

5.4. TreeMatch example. Step 2: identifying target nodes 92

5.5. Testbed scenarios 93

5.6. Iteration execution times in the scenarios illustrated in Figure 5.5 . . 95

5.7. Detailed iteration times in BT and FT benchmarks. Note the variation in the *Negotiation* times. 96

List of Algorithms

1. Pseudocode for the negotiation protocol 44

Preface

The current trend in computer architecture is the use of large clusters, often heterogeneous, in which the nodes are multi/many-core systems. These are highly dynamic systems, with an everincreasing number of processors, which causes relatively high hardware failure rates. For parallel programs executing on a large number of processors, this translates into frequent execution failures and a decrease in productivity.

Many fault tolerance methods for parallel applications on clusters exist in the literature, checkpoint and rollback recovery being the most popular. This method periodically saves the computation state to stable storage, so that the application execution can be resumed by restoring such a state. The overhead of saving checkpoints to disk is the main performance cost in checkpoint-recovery methods. This cost could become prohibitive for parallel applications running on large-scale facilities, where the I/O bandwidths do not increase as quickly as their computational capability and the checkpoint frequency must be increased to manage the higher failure rate.

This Thesis proposes and evaluates different techniques to reduce the size of the checkpoint files, and, thus, the computational and I/O cost of checkpointing: incremental checkpointing, zero-blocks exclusion, and data compression. The incremental checkpointing technique reduces the size of the state files by storing only data that has changed since the last checkpoint. Zero-blocks exclusion avoids storing null elements. Data compression, in turn, removes redundant information. These techniques have been implemented in CPPC (ComPiler for Portable Checkpointing), an application level checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications, obtaining important file size and

checkpoint latency reductions.

In case of failure, most of the current checkpointing and rollback solutions restart all the processes from their last checkpoint. However, a complete restart is unnecessary, since most of the nodes will still be alive. Moreover, it has important drawbacks. First, full restart implies a job requeueing, with the consequent loss of time. Secondly, since the assigned set of execution nodes is, in the general case, different from the original one, checkpoint data must be moved across the cluster in order to restart the computation, usually causing significant network contention and therefore high overheads. These limitations can be overcome if affected processes are individually restarted in case of a single node failure.

In this Thesis a solution to proactively migrate message passing interface (MPI) processes when impending failures are notified, without having to restart the entire application, is proposed. Its main features are: low overhead in failure-free executions, avoiding the checkpoint dumping associated to rolling-back strategies; low overhead at migration time by means of the design of a light and asynchronous protocol to achieve a consistent global state; transparency for the user, thanks to the use of the CPPC framework; and portability, as it is not locked into a particular architecture, operating system or MPI implementation.

Moreover, two additional techniques to reduce the I/O overhead in the checkpoint-based migration approach have been proposed. First, to take advantage of network speeds and to avoid the bottleneck of disk accesses, the storage to disk is substituted by in-memory checkpoint files and network transfers. Second, a pipeline technique is applied to overlap the different phases of a migration operation (state file writing in the terminating processes, data transfer through the network, and state file read and restart operations in the new processes) to reduce the migration time even more.

Finally, this PhD Thesis proposes a solution to automatically transform message-passing parallel applications into malleable applications, that is, parallel programs that are able to modify the number of required processors at run-time. This will allow to improve the use of resources, which will have a direct effect on the energy consumption required for the execution of applications, resulting in both cost savings and greener computing. The solution proposed is based on checkpointing and process migration and it is implemented on top of CPPC. It includes: automatic code

transformation of the parallel applications, a system to reschedule processes on available nodes, and migration capabilities based on checkpoint/restart techniques to move selected processes to target nodes.

The Thesis is organized into five chapters:

- Chapter 1 describes the checkpoint and restart technique and its main features. It also introduces CPPC, the application-level checkpointing tool used to implement the ideas proposed in this Thesis, outlining its most important characteristics.
- Chapter 2 focuses on reducing the overhead of the checkpointing operation. It analyzes different techniques to reduce the checkpoint file size: incremental checkpoint, zero-blocks exclusion, and data compression. The chapter also describes how to implement these techniques in CPPC and discusses the results obtained.
- Chapter 3 describes how to extend CPPC to provide process migration based on checkpointing techniques. This process migration is performed by storing the application data into checkpoints and creating new application processes that will be in charge of resuming the job. The chapter also includes details on how the proposed solution maintains the checkpoint consistency by means of a negotiation protocol between processes. It also provides an evaluation of the scalability and overhead of the migration proposal.
- Chapter 4 presents two improvements to the basic migration procedure. The first one avoids dumping the checkpoint files to stable storage during migrations. The second one splits the checkpoint files in order to overlap the transfer of the checkpoint files with its reading in the new processes.
- Chapter 5 describes the malleability solution proposed. It includes a scheduling algorithm that is in charge of automatically analyzing which processes have to migrate and calculate the target nodes. A description of the implementation using CPPC and the evaluation of the experimental results are also provided.

Finally, the work is concluded by summarizing the main contributions of this PhD Thesis and discussing the main research lines that can be derived from it.

Chapter 1

Background

This chapter introduces the checkpoint and restart techniques and their main properties. The CPPC (ComPiler for Portable Checkpointing) framework, used for the implementations of the proposals in this Thesis, is also briefly described in this chapter.

1.1. Checkpointing

The execution times of large-scale parallel applications on current multi/many-core systems are usually longer than the mean time between failures. Therefore, parallel applications must tolerate hardware failures to ensure that not all computation done is lost on machine failures. Checkpointing and rollback recovery is one of the most popular techniques to implement fault-tolerant applications. It periodically saves the application state to stable storage, so that, in case of failure, the stored data allows for restarting the application and continuing the execution from such state.

However, the dimension, heterogeneity, and dynamic nature of today's large computer infraestruturas open new research challenges that must still be solved, requiring proposals that are scalable, to be executed on hundreds of cores; portable, so they can deal with the heterogeneity of the platforms; and malleable to adapt to the dynamic nature thereof. To this end, we considered the following as the main

features to take into account when exploring new checkpoint solutions for parallel message-passing applications.

Granularity

There are two fundamental approaches to checkpointing: system-level checkpointing (SLC), implemented at the operating system level, and application-level checkpointing (ALC), where the application program saves and restores its own state. In SLC, the whole state of the processes (program counter, registers and memory) is saved to stable storage. The most important advantage of this approach is its transparency. However, it has two important drawbacks. First, storing the whole application state will have a higher associated cost than storing just necessary data. Second, it is inherently non-portable. A checkpointing technique is portable if it allows the use of state files to recover the state of a failed process on a different machine, potentially binary incompatible, or using different operating systems or libraries. The basic condition that has to be fulfilled in order to achieve potential portability is not to store any low-level data along with the process state. Therefore, all SLC approaches are not portable. ALC, on the other hand, is able to obtain better performance by storing only necessary data. Additionally, it enables both data portability, by storing data using portable representation formats, and communication-layer independence, by implementing the solution at a higher level of abstraction. The drawback is the need for analyses of the application code in order to identify the state that needs to be stored.

Transparency

It refers to how the users perceive the checkpointing technique. Generally, SLC corresponds with a transparent solution, because the user does not need to provide information about the application. The ALC approach is, in theory, a non transparent solution as, in this case, the user needs to have a deep knowledge of the application in order to select those variables that need to be saved in the checkpoint files. However, using compilers to automatize the variable selection process, the ALC approaches could become a transparent solution.

Portability

This property indicates if the checkpoint files can be used to recover the state of a failed process on a different machine, or using different operating systems or libraries. SLC approaches are non portable solutions because they store low level information, such as program counters, registers, etc. Furthermore, to achieve total portability in an ALC approach the checkpoint files have to be stored in an architecture independent format.

Coordination

A global state of a parallel application is a collection of individual states of all participating processes and the state of the communication channels between them. A consistent global state can be seen as one that may occur during a fault-free execution. The main difference, in terms of consistency, between parallel and sequential applications is the dependency imposed by the communications among the processes. An inconsistency could occur if the checkpoints are located in the middle of two communication sentences between processes. Note that, if the first sentence is a *send* sentence, at restart time the message will not be resent, although the other process involved in the communication expects to receive it. These kinds of messages are called *in-transit* messages. On the other hand, if the non-executed sentence is a *receive* sentence, the message will not be received, although it will be sent again. These messages are called *inconsistent* or *ghost* messages. Both types of messages are shown in Figure 1.1. In this figure every *msg* represents a communication between two processes. In the example in Figure 1.1(b) the process $P2$ has received the message $msg\ 3$ while the state of $P1$ does not reflect its shipping. This global state is inconsistent since it can not occur in a fault-free situation. In a consistent global state, if the state of a process reflects the reception of a message, then the state of the corresponding issuer reflects the sending of this message [12].

In uncoordinated checkpoint protocols processes decide independently when to store the checkpoints without negotiate with the other processes. Uncoordinated solutions have low overhead in the case of fault-free executions, but in case of failure they are susceptible to the so-called *domino effect*, which means that processes may

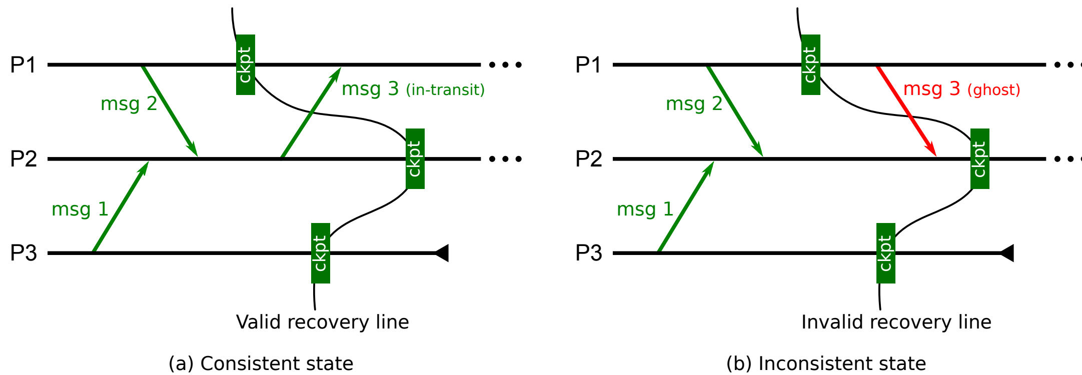


Figure 1.1: Example of valid and invalid recovery lines

be forced to rollback up to the beginning of the execution in case of restart if inconsistent messages exist. Besides, since the processes dump their states independently of each other, many of these checkpoints may be useless, as they will not be part of any valid recovery line. For these reasons, uncoordinated protocols have not been widely used in practice. As an alternative, uncoordinated checkpointing may be combined with message logging to avoid the domino effect at the expense of a high overhead in communication latencies [25].

In coordinated systems, all processes coordinate their checkpoints to produce a consistent global state, so in case of failure all processes come back to the same point. Coordinated checkpointing simplifies recovery and prevents the domino effect, since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing only requires each process to maintain one checkpoint in stable storage, reducing storage overhead.

Straightforward approaches to coordinated checkpointing are the blocking coordinated solutions. In these approaches an initiator, which may be an application process or an external entity, sends a broadcast message requesting that the processes checkpoint. Upon receiving this message, each process stops its execution, flushes all communication channels, and creates its local checkpoint. Then, each process sends an acknowledgment to the initiator and waits for a commit message before resuming its execution. The initiator will broadcast a commit message after receiving acknowledgments from all processes. This approach may lead to a significant overhead in fault-free executions.

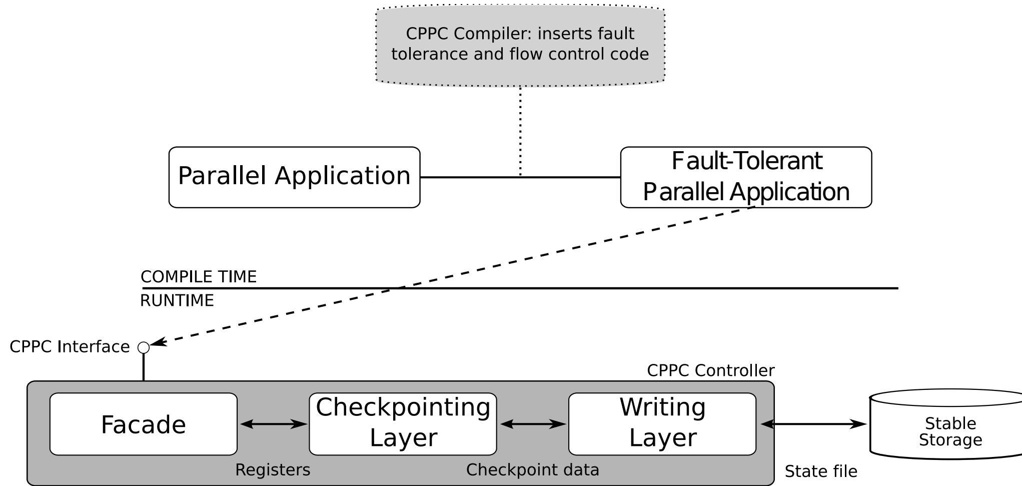


Figure 1.2: Integration of a parallel application with the CPPC framework

To reduce this overhead, non-blocking coordinated solutions have been proposed, the most popular being the distributed snapshots protocol by Chandy and Lamport [12]. This protocol starts with the initiator broadcasting a checkpoint request. Upon receiving the request, each process takes its checkpoint and rebroadcasts the request before sending any more application messages. Message logging is used to deal with in transit messages. The goal is to ensure that processes do not receive any message that could make the global state inconsistent. However, in order for this protocol to work, the communication channels need to be FIFO and reliable. For systems with non-FIFO channels some approaches resort to techniques like piggybacking [5], which causes an unacceptably high overhead in communication-intensive codes [80].

1.2. The CPPC Framework

CPPC (ComPiler for Portable Checkpointing) [74, 75] is an ALC tool focused on the insertion of fault tolerance into long-running message-passing applications. CPPC appears to the user as a compiler tool and a runtime library. The integration between the application and the CPPC framework is automatically performed by the CPPC compiler, that translates the application source files into derived files with added checkpointing capabilities. The global process is depicted in Figure 1.2.

At compile time, the CPPC compiler is used to automatically transform a parallel application into a fault-tolerant parallel application with calls to the CPPC library.

As for checkpoint consistency, CPPC minimizes the runtime overhead of classical consistency protocols by using a non-blocking spatially coordinated approach [73]. A SPMD (Simple Program Multiple Data) programming model is assumed and checkpoints are taken at the same relative code locations by all processes, but not forcibly at the same time. This ensures that all processes can select the same relative code point for restarting the parallel application, instead of restarting from different points which would render the global state inconsistent. To avoid problems caused by messages between processes, checkpoints must be inserted at points where it is guaranteed that there are no in-transit, nor ghost messages. These points will be called safe points.

Regarding memory requirements, CPPC works at the variable level (i.e. storing user variables only) and performs a live variable analysis that identifies which variable values are needed for the correct restart of the execution. Live variables are automatically detected by the CPPC compiler and marked using a CPPC function (`CPPC_Register()`) to provide such information to the CPPC controller. This process is referred to as "variable registration". Working at the variable level allows both to reduce the amount of data to be saved, which is one of the most performance impacting factors in checkpointing, and to store only portable data, hence making restart possible on different architectures. Storing only portable data on state files introduces the need for some kind of recovery mechanism, capable of regenerating the non-portable state that is not stored into state files. This mechanism is further described in the next subsection.

CPPC is designed with a special focus on portability: it uses portable code and protocols, and generates portable checkpoint files, allowing for execution restart on different architectures and/or operating systems. Currently, CPPC writes checkpoint files using the 5th version of the Hierarchical Data Format (HDF5) [85], a data format and associated library for the portable transfer of graphical and numerical data between computers.

Finally, CPPC provides multithreaded dumping, overlapping the checkpoint file writing with the computation of the application, and thus, reducing the checkpoint-

ing overhead. All the registered variables are copied to a new memory region in order to preserve their initial state while the application execution can continue. Then, the duplicated memory is stored into disk by a new thread without blocking the original MPI execution.

1.2.1. The CPPC Compiler

The CPPC compiler transforms the parallel code into fault-tolerant parallel code adding the calls to the CPPC library. The compiler is based on the Cetus platform [44].

To perform the code analysis and insert the library calls the compiler execute the next steps:

- Find *safe points*. To automatically identify safe points, the compiler performs a static analysis of inter-process communication.

- Insert the effective checkpoint function calls. A heuristic analysis, based on code complexity, identifies the most expensive loops and inserts a checkpoint function (`CPPC_Do_checkpoint()`) in the first safe point of these loops [73]. However, not all checkpoint function calls will generate checkpoint files. During runtime a checkpoint frequency may be defined in terms of number of calls to the checkpoint function.

- Analyze the data flow. The compiler performs a live variable analysis to store only the variables necessary to restore the application. Avoiding, for example, to store variables with temporary buffers or intermediate results. Depending on the considered application, applying this technique can significantly reduce checkpoint file sizes.

  The identification of these variables can be performed at compile-time through a standard live variable analysis. A variable x is said to be *live* at a given statement s in a program if there is a control flow path from s to a use of x that contains no definition of x prior to its use. The set LV_{in} of live variables at a statement s can be calculated using the following expression:

$$LV_{in}(s) = (LV_{out}(s) - DEF(s)) \cup USE(s) \quad (1.1)$$

where $LV_{out}(s)$ is the set of live variables after executing statement s, and $USE(s)$ and $DEF(s)$ are the sets of variables used and defined by s, respectively. The live variable analysis should take into account interprocedural data flow.

Checkpoints in application-level approaches are usually triggered by an explicit call to a checkpoint function in the application code. This guarantees that checkpoints are not performed during a system call, which may have internal state unknown to the checkpointer, but rather inside user-level code. In this way, checkpoint callsites are limited and known at compile time, which allows for the live variable analysis to be bounded and not span the whole application code. For each checkpoint callsite c_i, it is only necessary to store the set of variables which are live when the control flow enters the callsite, $LV_{in}(c_i)$.

Currently, CPPC does not perform optimal bounds checks for pointer and array variables. This means that some arrays and pointers are registered in a conservative way: they are entirely stored if they are used at any point in the re-executed code.

1.2.2. CPPC Operation

For illustrative purposes, Figure 1.3 shows the C code of an MPI application (a matrix diagonalization) and Figure 1.4 details the fault-tolerant version of the same code obtained by using the CPPC source-to-source compiler.

CPPC has two operation modes: *checkpoint* and *restart*. A checkpoint mode is used during regular execution. Processes execute the code sequentially and create checkpoints according to their specified *checkpoint frequency*. Note that not all the calls to the checkpoint function generates a checkpoint file. This behaviour can be seen in Figure 1.5. This checkpoint frequency parameter and other execution related parameters can be easily modified between executions changing the CPPC configuration file. An example of this file is shown in Figure 1.6. The restart mode is used after the original execution has aborted to recover the computation state

```
1  int main(int argsc, char **argsv) {
2    // Variable definitions
3    ...
4
5    MPI_Init(&argsc, &argsv);
6
7    // Matrix data input and distribution
8    ...
9
10   for( i=0; i < niters; i++ ) {
11     // Matrix diagonalization
12     ...
13   }
14   ...
15
16   MPI_Finalize();
17 }
```

Figure 1.3: Skeleton of an example of MPI code: a matrix diagonalization

of all processes from a previously saved snapshot. Only portable state is stored into the checkpoint files. CPPC uses code re-execution to recover the application state. A section of code is defined as required execution code (REC) if it must be re-executed during a process restart to ensure correct state recovery. Each REC recovers some part of the original application state. The fundamental REC types are non-portable calls, variable registrations and checkpoint calls. A typical example of a non-portable call is a call to a function manipulating opaque library state, such as an MPI function which creates or modifies a communicator.

The CPPC compiler divides applications into pieces formed by: a block of non-relevant code, a jump target (`CPPC_EXEC` labels in the figure), a block of restart-relevant code (REC), and a conditional jump to the next jump target, which will be placed right before the following REC. Conditional jumps will only be taken when in restart mode. In this way, after a failure, CPPC is able to re-execute only relevant parts of the code, skipping the non-relevant ones.

Following the example in Figure 1.4, during *checkpoint* operation the MPI environment is initialized, then the CPPC controller through the `CPPC_Init()` function; matrix data are read and distributed; relevant variables are registered by every process (loop index, loop limit and matrix data); next the core computation of the

```
1  int main(int argsc, char **argsv) {
2    // Variable definitions
3    ...
4
5    MPI_Init(&argsc, &argsv);
6
7    CPPC_Init(&argsc, &argsv);
8    // Conditional jump to CPPC_EXEC_1
9    if( CPPC_Jump_next() ) {
10     goto CPPC_EXEC_1;
11   }
12
13   // Matrix data input and distribution
14   ...
15
16 CPPC_EXEC_1:
17   CPPC_Register(&i, ... );
18   ...
19   //Conditional jump to CPPC_EXEC_2
20
21   for(i=0; i < niters; i++){
22 CPPC_EXEC_2:
23     CPPC_Do_checkpoint( 0 );
24
25     // Matrix diagonalization
26     ...
27   }
28
29   ...
30   CPPC_Shutdown();
31 }
```

Figure 1.4: CPPC-instrumented matrix diagonalization example code

application begins with calls to the checkpoint function in every iteration and actual checkpoint dumping every n iterations depending on the specified checkpoint frequency; and, finally, the results are written and CPPC is shutdown.

In *restart* operation the execution starts normally. Upon calling the CPPC initialization function the restart is detected, and a negotiation phase is performed to identify the most recent recovery line, that is, the set of checkpoint files to be used for restart. These files are verified and read, and restart mode is entered, which activates the conditional jumps that direct the execution through the identified RECs and skip nonrelevant sections of code. In the example, the matrix data input

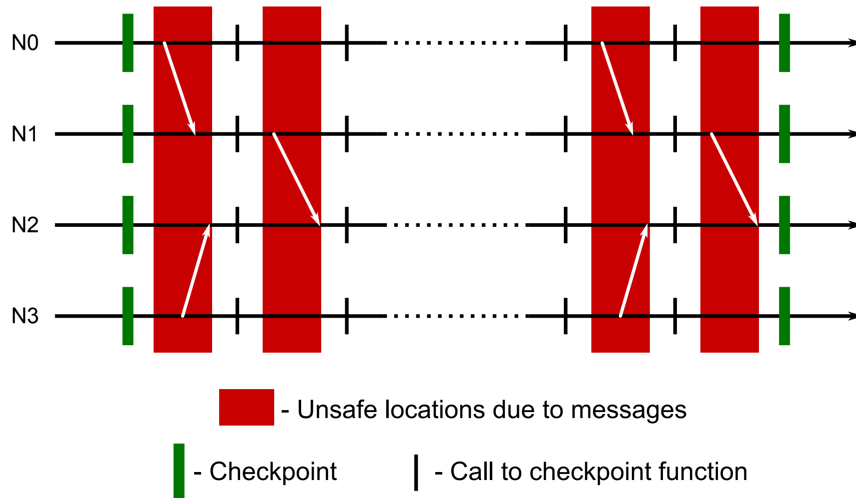


Figure 1.5: Spatial coordination for checkpointing

```
1  CPPC/Controller/RootDir=~/ckpt_cppc
2  CPPC/Controller/ApplicationName=.
3  CPPC/Controller/Restart=false
4  CPPC/Controller/Frequency=40
5  CPPC/Controller/FullFrequency=1
6  CPPC/Controller/CheckpointOnFirstTouch=false
7  CPPC/Controller/Suffix=.cppc
8  CPPC/Controller/StoredRecoverySets=10
9  CPPC/Controller/DeleteCheckpoints=false
10 ...
```

Figure 1.6: Example of the CPPC configuration file

will be skipped. The variable registration REC recovers variable values. Finally, the execution reaches the checkpoint inside the computational loop, the library is reconfigured to checkpoint mode and the application continues regular execution.

Chapter 2

I/O Optimization in the checkpointing of parallel applications

This chapter proposes and evaluates different techniques to reduce the checkpoint file sizes and, thus, the computational and I/O cost of checkpointing in ALC approaches.

2.1. Introduction

Although checkpoint/restart is the most common solution to endow scientific applications with fault tolerance, its cost in terms of computing time, network utilization or storage resources can be a limitation for large scale systems.

Checkpoint file size is the most important factor in determining checkpointing performance. As such, the reduction of the amount of stored state is one of the most frequent goals of checkpoint optimizations. However, most of the techniques described in the literature are applied to SLC approaches, since ALC solutions are less general, and they already achieve smaller checkpoint files. Nevertheless, in order to be useful for today's large scale systems, ALC approaches will also need to minimize checkpoint file sizes. The following sections describe different and

complementary techniques to optimize the checkpoint sizes in ALC solutions [21]: incremental checkpointing to store only modified data; zero-blocks exclusion to avoid storing null elements; and data compression to remove redundant information. The final sections explain the implementation details of those techniques on CPPC and evaluate the performance of the proposed methods.

2.2. Incremental checkpointing

The most popular technique for checkpoint file size reduction in SLC approaches is incremental checkpointing [2, 32, 71]. This technique involves creating two different types of checkpoints: full and incremental. Full checkpoints contain all the application data. Incremental checkpoints only contain data that has changed since the last checkpoint. Usually, a fixed number of incremental checkpoints is created in between two full ones. During a restart, the state is restored by using the most recent full checkpoint file, and applying, in an ordered manner, all the differences before resuming the execution.

There exists in the literature different solutions to implement incremental checkpointing in SLC approaches. One of them is to use the virtual memory page protection mechanisms [71]: upon starting to checkpoint, pages to be saved are marked read-only. When the page is effectively saved into the checkpoint its original status is recovered. When the application tries to write to a read-only page, the race condition is resolved by the fault handler. Another option is to use a kernel-level memory management module that employs a page table dirty bit scheme [32]. Both solutions require memory protection support from the underlying hardware along with support from the OS to be able to handle page-fault exceptions. This feature, although very common, is not universally available. An alternative to page-used checkpoint is hash-based checkpoint [2], which uses a secure hash function to obtain a unique identifier for each block of application memory to be written into state files. This value is stored and compared against the value calculated for the same block upon creating a new checkpoint. If the two hash values differ, the block contents have changed and it is stored again in the new checkpoint file.

The basic difference between SLC and ALC emerges from the fact that SLC sees

the application memory as a single continuum, while ALC distinguishes a disperse set of contiguous memory blocks. Each block contains memory allocated to one or more variables, depending on the aliasing relationships of the application data. Thus, in ALC it is not recommendable to track changes to memory blocks using the virtual memory page protection mechanism or dirty bits, as array variables do not necessarily start at page boundaries.

The solution proposed in this chapter is inspired by the hash-based approaches but is intended for ALC. The array variables are divided into chunks of memory of a previously specified size, assumed to be constant, and the changes into these chunks are detected using a secure hash function. The calculated hash value for each chunk is stored in memory and used for comparison when creating incremental checkpoints. Using an application-level approach the number of memory blocks to be checked at runtime is reduced, which minimizes the size of the hash tables to be calculated and stored, improving the overhead.

2.3. Zero-blocks exclusion

When working with real scientific applications it is well known that quite often many elements of the arrays are null, resulting in memory blocks that contain only zeros. Therefore, a possible optimization to further reduce the checkpoint file size is to avoid storage of those *zero-blocks*.

In addition, to control the changes into memory blocks, the hash function described in the previous section may also be used to detect zero-blocks. When a zero-block is detected, a small marker is saved into the checkpoint file to indicate that the block is null, instead of dumping its contents. During restart this marker is identified and the target memory is filled with zeros, which recovers the original state at a negligible cost in terms of both performance and disk usage.

The idea of not storing zero-blocks has a certain similarity to the technique used in the SLC tool Berkeley Lab's Checkpoint/Restart (BLCR) Library [49] to exclude *zero pages*, that is, those that have never been touched and logically contain only zeros.

2.4. Data compression

Another means to reduce checkpoint file sizes is data compression. This technique has been implemented, for instance, in the ickp checkpointer [69], ErrMgr [42], and the CATCH compiler [51]. In ickp, a predictive algorithm is presented that offers very low overhead, but only performs well with some highly compressible sources, as it often produces data expansion. ErrMgr uses DEFLATE (gzip) [43] and shows good results mainly for highly-compressible data. For less compressible data, the overhead offsets any compression benefit. In CATCH, the general purpose LZW [43] algorithm is used, which typically offers slightly worse performance than DEFLATE with similar overhead.

In this work a new and faster compression algorithm is proposed. It is based on particular features observed in checkpoint files and addresses the trade-off between compression efficiency and overhead. We use the well-known technique of substituting repeated chains of bytes by special codes that mark the position of the chain and its length. A string of bytes is processed sequentially. For each incoming byte, a match within the last 16 processed bytes is sought. The aim is finding the longest possible chain of matches, avoiding encoding each byte individually. Those bytes for which a match cannot be found, are known as literals. The compressed stream consists of a description of the literals, and the position and size of the matched chains. Additionally, entropy coding is applied, using shorter codes for the most common descriptors.

Figure 2.1 shows an example of the encoding process, where alphabet letters are used instead of numeric 8-bit values. A 16-byte buffer keeps the last processed bytes. A new incoming byte is compared with the content of the buffer, producing a 16-bit mask. The buffer is then updated by shifting-in the new byte. In Figure 2.1 we can see that 2 literals (*'k'* and *'l'*) are found first. Next, there are 3 candidate positions that match *'a'* and *'b'*. The length of the match keeps growing, but only 1 candidate remains. A logic AND between the current mask and the previous one is a simple way of detecting the end of the matching string. In the example, a new match starts, but it could also be a literal. Note that the length of the matches is not limited to the size of the buffer.

The number of literals (2) and their values are encoded, together with the size of

16-byte buffer	new	comparison mask	match evolution
efab cabc defg abhj	← **k**	0000 0000 0000 0000	no match , 1 literal
fabc abcd efga bhjk	← **l**	0000 0000 0000 0000	no match , 2 literals
abca bcde fgab hjkl	← **a**	**1**00**1** 0000 00**1**0 0000	new match, 2 literals
bcab cdef gabh jkla	← **b**	1001 0000 0010 0000	2 matches, 2 literals
cabc defg abhj klab	← **c**	1001 0000 00**0**0 0000	3 matches, 2 literals
abcd efga bhjk labc	← **d**	**0**001 0000 0000 0000	4 matches, 2 literals
bcde fgab hjkl abcd	← **e**	0001 0000 0000 0000	5 matches, 2 literals
cdef gabh jkla bcde	← **h**	000**0** 000**1** 0000 0000	new match, 0 literals
defg abhj klab cdeh	← **j**	0000 0001 0000 0000	2 matches, 0 literals

Figure 2.1: Example of pattern matching for data compression

the matching string (5) and its position (12 bytes from the starting point). The way in which those values are encoded was guided by the analysis of many gigabytes of data.

Essentially, an 8-bit token is built by combining the number of literals (up to 15 in a row) and the size of the match (from 1 to 16), as these values show strong correlation. Escape codes are used for longer chains of literals or matches. The tokens are then compressed using static Huffman codes [40]. The literals are not compressed, as they exhibit high entropy. And, finally, the positions are compressed using a semi-adaptive scheme.

In the example in Figure 2.1, the inputs from *'k'* to *'e'* would be encoded as: *(2,5) + k + l + 12*. The resulting bit pattern could be: *111111110100000 KKKKKKKK LLLLLLLL 111100*. Hence, 7 bytes would be encoded using 37 bits instead of 56, a 34% gain.

Compared to general purpose algorithms, this proposal allows fast parallel search instead of using iterative search guided by hash keys. Focusing on just the nearest 16 values performs well as matches separated by large distances are not as common in checkpoints as they are in text files. In general, the most common patterns are: runs of values, and repeated exponents in floating point arrays.

Also, we use static Huffman codes combined with simple adaptability. Static means that the codes are the same for all the files, assuming that they all have the same statistical distribution of positions and lengths. We have found that this is a reasonable assumption for checkpoint files, contrarily to the general case. Us-

ing fixed, static codes is significantly faster than using dynamic ones and enables building optimized decoders.

2.5. Implementation

The techniques described in previous sections have been implemented on CPPC. This section describes these implementations.

2.5.1. Incremental checkpointing and zero-blocks exclusion

For the implementation of incremental checkpointing, CPPC divides array variables into blocks of memory. The size of these memory blocks may have a great impact on the performance of the incremental checkpointing technique. CPPC allows the user to choose the size to be used for each particular application. A block size of 8K elements is selected by default when the user does not specify any size. It experimentally proved to be a good compromise value.

CPPC also calculates the hash value of each memory block. The choice of the hash function impacts upon the correctness, since many hash functions present a significant probability of *collisions*, that is, situations where two different memory blocks are assigned the same hash value. Secure hash functions should be used to implement reliable incremental checkpointing techniques [61]. The implementation of incremental checkpointing in CPPC allows the user to choose between different secure hash functions, such as MD5 or SHA. The MD5 function is selected by default.

The calculated hash functions are used to detect both zero-blocks that can be excluded in the next checkpoint, and changes in memory blocks from previous checkpoints. In order to detect zero-blocks the calculated hash values are compared to the known hash value of a zero-block. To detect changes in the memory blocks, the hash values calculated in previous checkpoints have to be stored to be compared with the new ones. In our implementation, the hash codes are stored into main memory rather than on disk to improve the performance of the technique.

Only the modified blocks with non-zero elements will be stored in the checkpoint

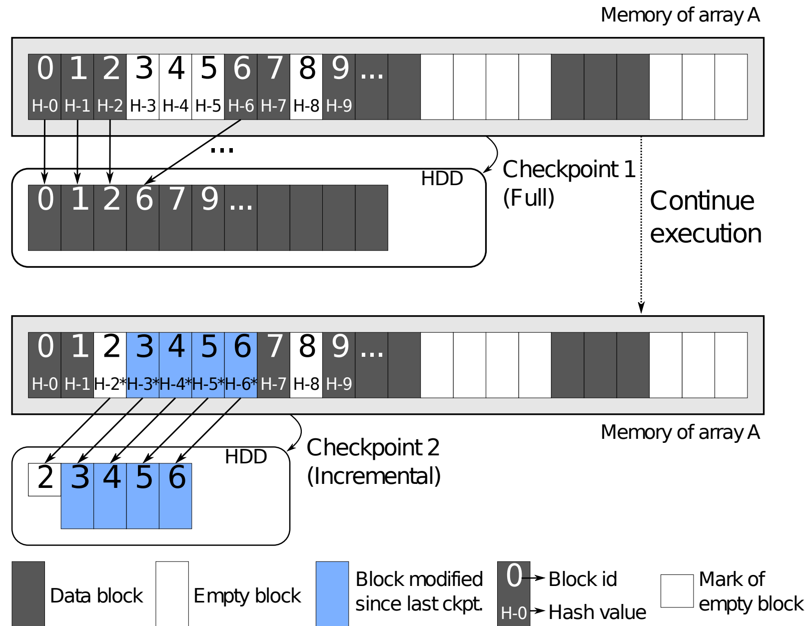


Figure 2.2: Construction of an incremental checkpoint.

file. The construction of an incremental checkpoint is depicted in Figure 2.2. In order to enable full data recovery during restart, some meta-information needs to be stored together with the checkpoint data. Specifically, an identifier is stored in the checkpoint file for each modified memory block, including modified zero-blocks. This identifier indicates the original position of the block in memory relative to the start of the array. The high-order bit of the identifier is used to mark the zero-blocks that are not included in the checkpoint file but should be restored during recovery. CPPC uses an integer array called *Block_ID* to store the meta-information. The size overhead of storing this array can be calculated as:

$$Overhead = HDF5_labels + sizeof(Block_ID) \tag{2.1}$$

Being *HDF5_labels* the number of bytes used by HDF5 to store information about the *Block_Id* array (148 if the number of elements of *Block_ID* is zero and 892 in any other case). The size of the array of identifiers can be calculated as:

$$sizeof(Block_ID) = 4\ bytes \times (\#MBlocks) \tag{2.2}$$

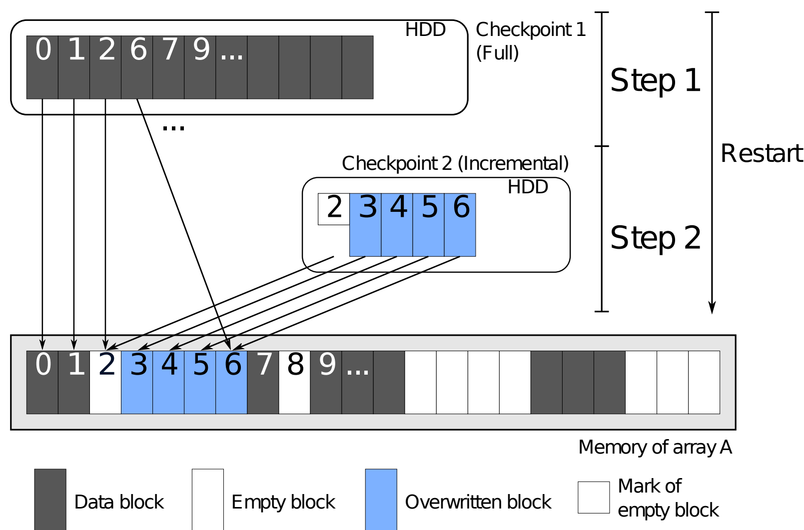


Figure 2.3: Restart from an incremental checkpoint.

Being $\#MBlocks$ the number of modified blocks. Thus, the overhead varies between 148 and $892 + (4 \times \#TBlocks)$ bytes, being $\#TBlocks$ the total number of memory blocks of the application userspace.

In addition to the checkpointing mechanism, the restart mechanism when using incremental checkpointing also varies. The process of restarting from incremental checkpoints is shown in Figure 2.3. The last available full checkpoint is restored first, and the updates contained in each incremental checkpoint are then applied in an ordered manner.

2.5.2. Data compression

To compress the checkpoint files, the CPPC writing layer seen in Section 1.2 must be extended. The HDF5 library provides users with different file drivers which map the logical HDF5 address space to different types of storage. In the current CPPC version the default file driver (`SEC2 driver`) is used to dump the HDF5 data directly to stable storage. This driver was substituted by the HDF5 `core driver` which constructs the HDF5 file in memory.

The checkpoint and recovery processes using compressed files are shown in Figure 2.4. In order to perform the compression step without storing temporary data to

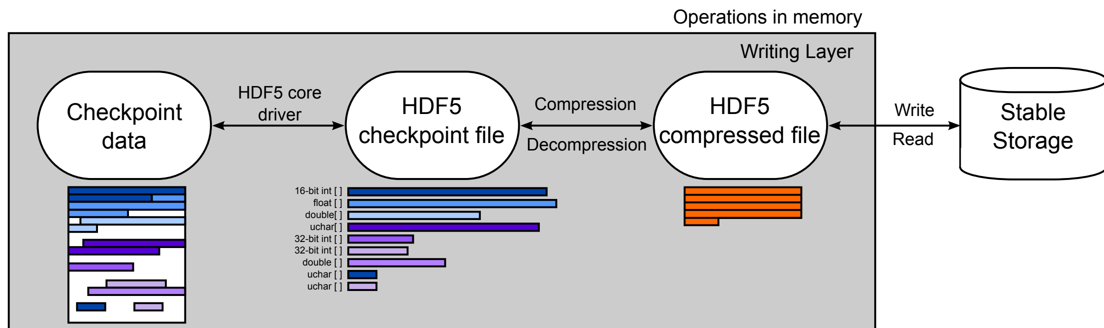


Figure 2.4: Integration of the compression process in the CPPC writing layer

the process local disk first, the `HDF5 File Image Operations` available since HDF5 v1.8.9 are used. These are a set of functions that allow working with HDF5 files directly in memory. Disk I/O is not required when files are opened, created, read from, or written to. Once the state file is committed to memory, the compression routine is invoked. Afterwards, the compressed data are stored into stable storage. The decompression process is the reverse: the compressed file is read from stable storage to local memory, data are decompressed, and finally the HDF5 is read in place. Note that if compression is disabled, the HDF5 checkpoint file in memory is directly stored into stable storage without compression.

Compression speed is crucial in order to minimize the introduced overhead. In this sense, platform-specific optimizations, such as SIMD instructions, play an important role. Then, the following optimizations are possible: fitting the buffer into one 128-bit register or two 64-bit ones; and performing 16 comparisons with 1 or 2 instructions that produce a 16-bit mask. SIMD instructions are primarily intended to accelerate multimedia processing, and they are commonplace in modern architectures. However, as they are not standardized, the optimized code is not portable. Hence, plain ANSI C code was developed together with optimized code for x86 and Itanium platforms. At compile time, directives will select which code will be used.

Note that CPPC creates a checkpoint file per process, each one containing a subset of the total data to be stored. The different checkpoint files are simultaneously compressed in the different processes. Thus, compression is implicitly performed in parallel, and thus its overhead is expected to decrease when increasing the number of processes.

2.6. Experimental Evaluation

This section assesses the impact of the described optimization techniques in the size of the checkpoint files and in the execution time overheads. A multicore cluster, Pluton (located in the Faculty of Computer Science in the University of A Coruña), was used to evaluate our proposal. Initially, it consisted of 8 nodes, each one of them powered by two Intel Xeon E5620 quad-core CPUs with 16 GB of RAM. The cluster nodes were connected through an Infiniband network and a Gigabit Ethernet network. The front-end was powered by one Intel Xeon E5502 quad-core CPU with 4 GB of RAM. The connection between the front-end and the execution nodes was an Infiniband network too. The working directory used for storing checkpoints files is connected to the cluster by a Gigabit Ethernet network and it consists of disks of 2 TB configured in RAID 6. During the realization of this thesis, Pluton received an important upgrade, so nowadays it has 16 extra computing nodes, each powered by two octa-core Intel Xeon E5-2660 CPUs with 64 GB of RAM. The cluster network was updated to an InfiniBand FDR network, providing up to 56 Gbps. The working directory is mounted via network file system (NFS) and it maintains their connection to the cluster via the Gigabit Ethernet network. To avoid misunderstandings, henceforth the oldest nodes receive the name *Pluton N1* or *N1*, and the newest *Pluton N2* or *N2*.

The application testbed was comprised of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [62] (NPB). These are well-known and widespread applications that provide a de-facto test suite. Out of the NPB suite, the biggest problem size that would fit the available memory was selected for each application. As such, the BT, LU and SP benchmarks were run using class B; the rest were run using class C. All the experiments were executed using 16 and 32-36 processes (32 processes for all the applications except for BT and SP as they require a square number of processes).

The experiments in this section were executed in Pluton N1. They can be divided into two blocks. The first block analyzes the checkpoint size reductions obtained through the use of the proposed techniques. The second block evaluates the execution overhead caused by the computation of the hash functions and data compression, and the restart overhead caused by the restart mechanism in the incremental

Table 2.1: Baseline checkpoint sizes (in MB) per process

	16 processes		32-36 processes	
NPB	*SLC*	*ALC Base*	*SLC*	*ALC Base*
BT	97.45	31.36	83.61	17.50
CG	153.05	85.85	114.50	43.30
EP	67.42	1.18	67.42	1.18
FT	514.93	256.14	290.93	128.14
IS	210.50	144.13	138.57	72.13
LU	81.03	14.78	74.86	8.54
MG	288.56	222.32	181.00	114.70
SP	99.00	32.81	86.08	19.87

technique and data decompression.

2.6.1. Checkpoint file sizes

The reduction in checkpoint file size is the main goal of the techniques described in this work. Table 2.1 allows comparing the baseline checkpoint sizes per process in the Pluton cluster. The first column (`SLC`) shows results for an SLC approach, the CKPT [89] checkpoint library was used. The second column (`ALC Base`) displays results for an ALC approach without applying any optimization technique, that is, all user variables are stored in the checkpoint file. As can be seen, ALC obtains better results than the SLC approach. Additionally, the size of the checkpoint files per process decreases more significantly for ALC approaches as the number of processes increases, which helps obtain scalable fault tolerance. As commented in Section 1.2.1, CPPC uses a method based in live variable analysis to select only those variables that are necessary to restart the application. Depending on the considered applications, this technique can significantly reduce checkpoint file sizes. Figures 2.5 and 2.6 show normalized checkpoint file sizes with respect to the ALC base case when using the live variable analysis (`LiveVar`) and the incremental checkpointing and zero-blocks exclusion techniques proposed in this work. Several incremental checkpoints (`Incr`) are created after a full checkpoint (`Full`). However, since their sizes are similar, only the first one is shown in the figures.

The live variable analysis significantly reduces checkpoint file sizes for CG (56% reduction) and FT (25%). It can be concluded that this technique may have great

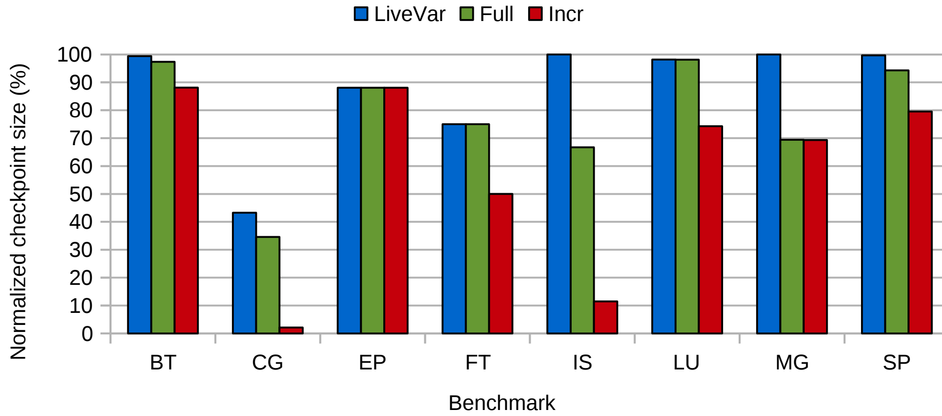


Figure 2.5: Checkpoint sizes per process for 16 processes normalized with respect to the ALC base case (see Table 2.1)

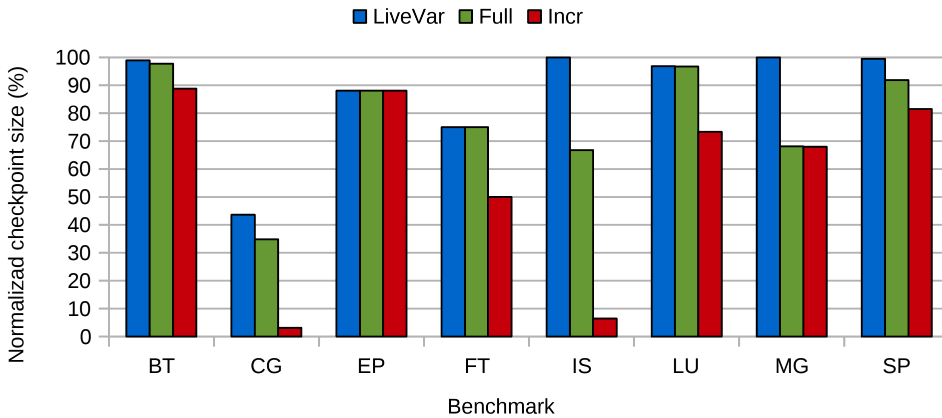


Figure 2.6: Checkpoint sizes per process for 32-36 processes normalized with respect to the ALC base case (see Table 2.1)

influence on reducing file sizes for certain applications and, as it introduces overhead only at compile time, no application can be adversely affected by its use.

The incremental checkpointing and zero-blocks exclusion techniques achieve important file size reductions for almost all the applications. Note that these techniques were applied together and in addition to the live variable analysis. Thus, reductions achieved in the full checkpoint relative to the live variable technique are only due to the elimination of zero-blocks. These reductions vary with the size of the memory block. Figures 2.5 and 2.6 show results for the default value of 8K elements per block. Reductions with respect to the ALC base case range from 3% (BT) to 65%

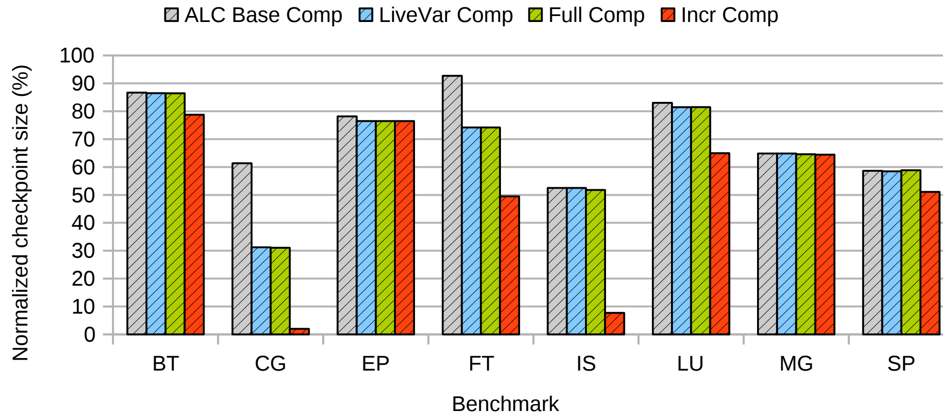


Figure 2.7: Compressed checkpoint sizes per process for 16 processes normalized with respect to the ALC base case (see Table 2.1)

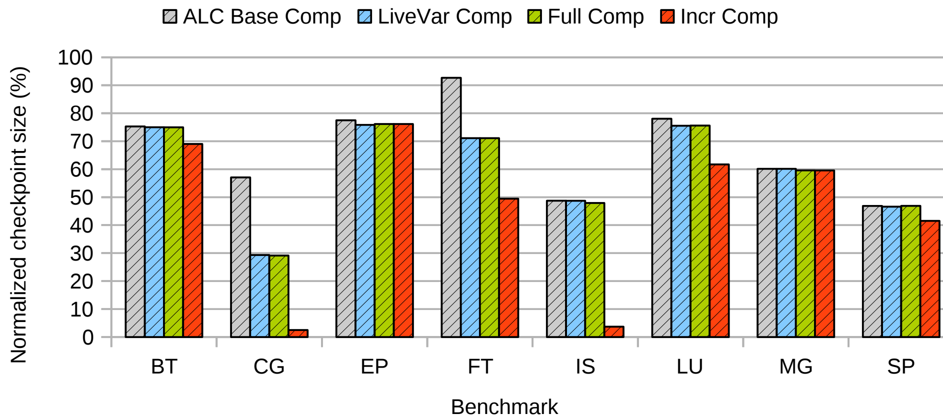


Figure 2.8: Compressed checkpoint sizes per process for 32-36 processes normalized with respect to the ALC base case (see Table 2.1)

(CG) for the full checkpoint and from 12% (BT) to 98% (CG) for the incremental checkpoints.

The results of compressing the checkpoints are given in Figures 2.7 and 2.8 (`ALC Base Comp`, `Live Var Comp`, `Full Comp`, `Inc Comp`). On average, size reductions of 20% and 25% are achieved for 16 and 32-36 processes, respectively. For some benchmarks, like IS (Integer Sort), it is easy to discern why compression performs better for 32 processes: sorting removes entropy, helping the compressor to find repeated patterns. For other benchmarks the underlying reason may not be so obvious. There are also important differences among benchmarks, as some of

Table 2.2: Baseline checkpoint latency (s)

	16 proc.	32-36 proc.
NPB	*ALC Base*	*ALC Base*
BT	5.29	6.65
CG	14.38	15.33
EP	0.25	0.30
FT	37.67	38.56
IS	21.90	21.93
LU	2.56	2.69
MG	34.92	35.64
SP	5.71	7.08

them, like FT, are hardly compressible. Also, incremental checkpoints are generally less compressible than the others, as much of the redundant data have already been removed. Comparatively, DEFLATE and LZMA would offer an additional 7-10% gain, but increasing overhead, as will be discussed in Section 2.6.2.

2.6.2. Checkpoint latency

The checkpoint latency is defined as the ellapsed time between the call to the checkpointing function and the return of control to the application. Table 2.2 shows the baseline checkpoint latency obtained for the different NPB applications for 16 and 32-36 processes. Note that the increase in the number of processes does not have a great influence in the latency times, since a shared filesystem is used and the total amount of data to be dumped remains almost constant. All tables and graphs in this section and the next one display the average data of at least 10 executions.

As regards the incremental checkpointing, some extra time is spent in the computation of the hash functions and the inspections needed. The hash function selected for these experiments was MD5. From the results shown in Figures 2.9 and 2.10, it can be observed that the overhead introduced by the incremental checkpointing technique is hidden by the gain obtained from the reduction in checkpoint size. Results for the creation of the full checkpoint in the incremental technique also allow to assess the obtained gain when solely applying the zero-blocks exclusion.

Data compression (Figures 2.11 and 2.12) also allows reducing checkpointing

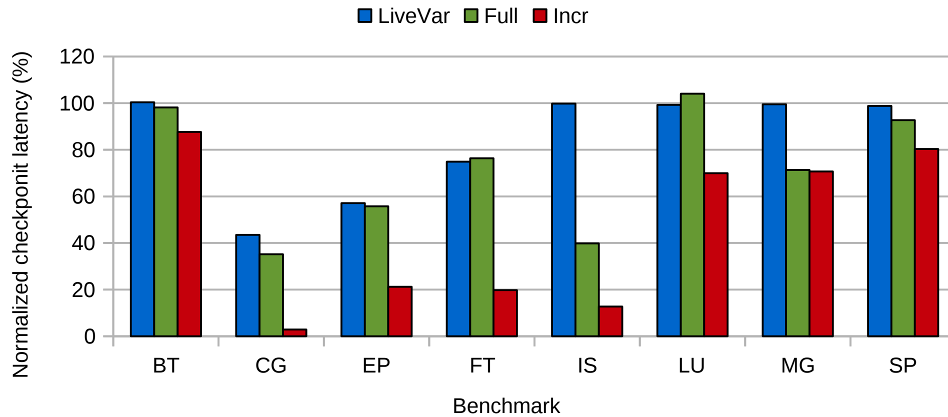


Figure 2.9: Checkpoint latency for 16 processes normalized with respect to the ALC base case (see Table 2.2)

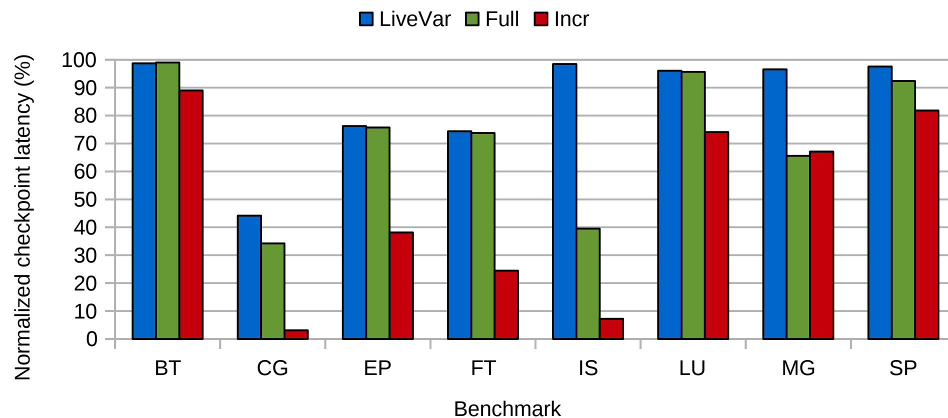


Figure 2.10: Checkpoint latency for 32-36 processes normalized with respect to the ALC base case (see Table 2.2)

latency in virtually all the tests. The main exception is FT, which contains poorly-compressible data. This gain is possible by the combination of two factors. Firstly, compression allows a significant size reduction, as seen in Section 2.6.1. Consequently, storage overhead is proportionally reduced. Secondly, compression speed is 85-90 MB/s on average. That is close to the maximum bandwidth of the Gigabit network on which the testbed storage system is based upon. Therefore, our compression system adds very little overhead to checkpointing, 7% on average (the compression overhead is labeled as `Compression` in the Figures). In comparison, DEFLATE and LZMA are, respectively, 3 and 12 times slower, which makes them

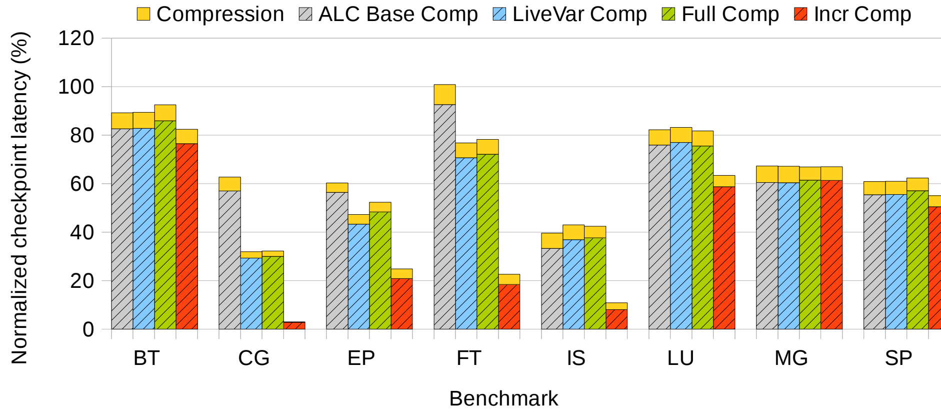


Figure 2.11: Compressed checkpoint latency for 16 processes normalized with respect to the ALC base case (see Table 2.2)

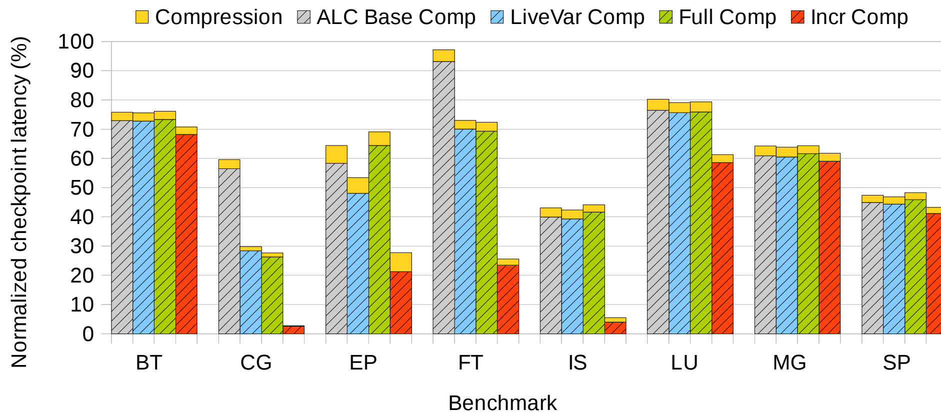


Figure 2.12: Compressed checkpoint latency for 32-36 processes normalized with respect to the ALC base case (see Table 2.2)

impractical alternatives (the gain does not compensate the overhead introduced by the compression).

Although the compression algorithm proposed in this chapter has been applied to an ALC approach, it could equally be applied to SLC. We have experimentally tested that it is also viable for compressing SLC checkpoint files as, unlike the DFLATE and LZMA algorithms, it is fast enough to provide a performance benefit. Nevertheless, compressing SLC checkpoint files will always be computationally more expensive than compressing ALC files, due to the significantly larger sizes. Additionally, the resulting compressed files will be also larger than their ALC counterparts. Thus,

Table 2.3: Baseline restart times (s)

	16 proc.	32-36 proc.
NPB	*ALC Base*	*ALC Base*
BT	4.52	5.63
CG	12.23	12.88
EP	0.19	0.36
FT	36.54	36.39
IS	20.53	20.49
LU	2.15	2.46
MG	31.69	32.56
SP	4.73	6.38

starting from ALC checkpointing files will be always a better solution.

As can be seen by comparing Figures 2.11 and 2.12, compression overhead drops as the number of processes is increased. This is due to the fact that the total compression workload is shared by more processors. Hence, checkpoint compression in large scale supercomputers will allow reducing the volume of stored data with almost negligible overhead.

In general, all the proposed techniques perform better than the ALC base approach. In some cases the reduction in latency can be as high as $92 - 97\%$ (IS or CG).

As commented in Section 1.2, CPPC can be configured so that the checkpoint file is created in parallel with the execution of the application by creating new threads [74]. Thus, the application execution does not need to be stalled until the checkpoints are created, and the latencies may be hidden.

2.6.3. Restart overhead

Restart times include the read of the checkpoint files and the restart of the application up to the point where the checkpoint was dumped. In all the experiments, write buffers were flushed before each execution to avoid the effect of page cache and to guarantee that checkpoint files are read from disk.

Figures 2.13 and 2.14 show the normalized restart time with respect to the baseline restart times shown in Table 2.3. Columns labeled `Full` show the restart

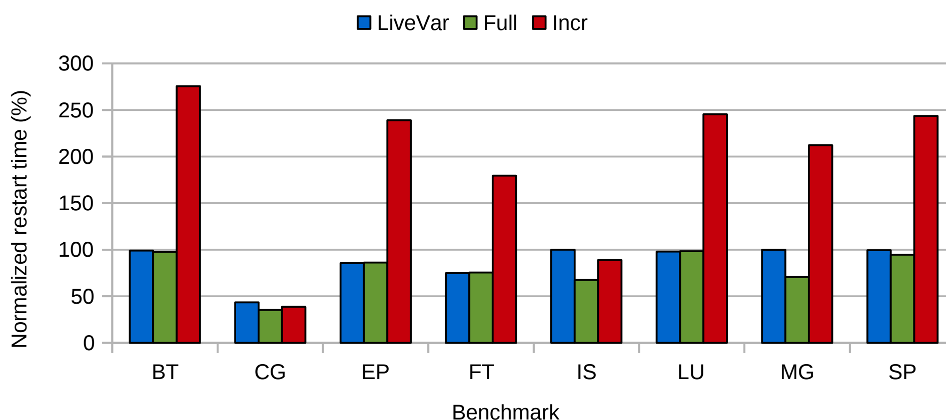


Figure 2.13: Restart times for 16 processes normalized with respect to the ALC base case (see Table 2.3)

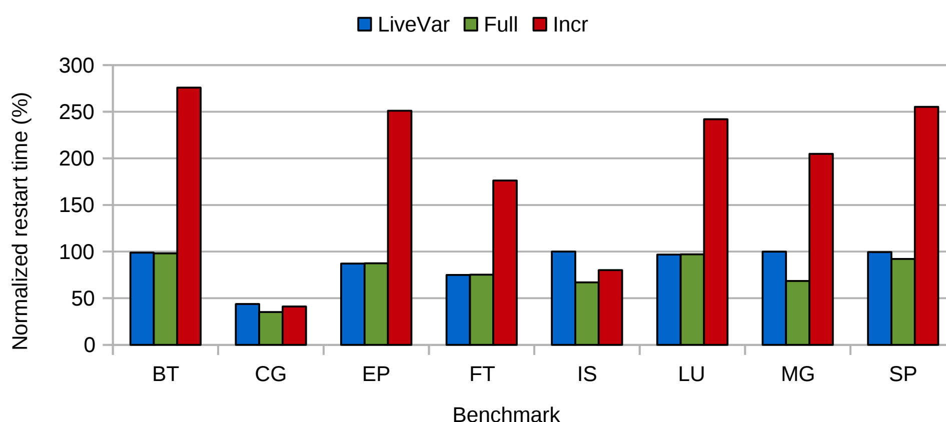


Figure 2.14: Restart times for 32-36 processes normalized with respect to the ALC base case (see Table 2.3)

overhead when there are no incremental checkpoint files, just the full one. These correspond to the overhead when applying only the zero-blocks exclusion, which is always less than the overhead of the base approach.

The incremental checkpointing technique presents a higher restart overhead compared to the others. This is due to a larger volume of data being moved and read, which can be calculated as the sum of the incremental and full checkpoint file sizes. In these experiments two incremental checkpoints were created after a full one.

Compression (Figures 2.15 and 2.16) has also a positive impact in restart over-

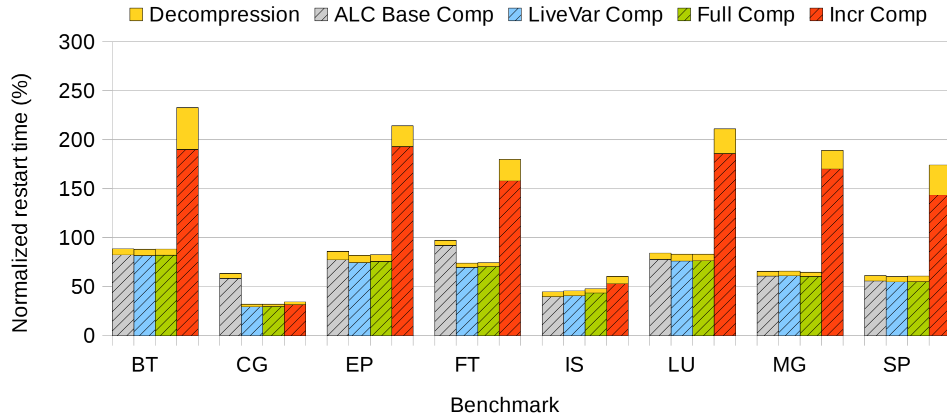


Figure 2.15: Compressed restart times for 16 processes normalized with respect to the ALC base case (see Table 2.3)

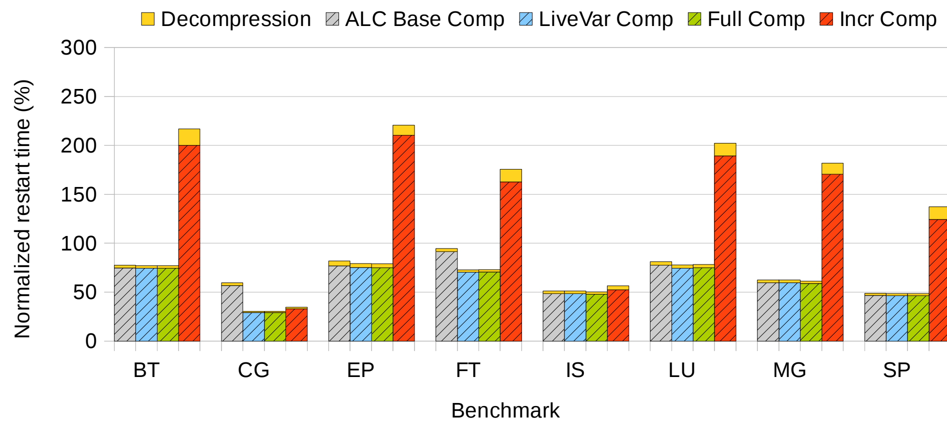


Figure 2.16: Compressed restart times for 32-36 processes normalized with respect to the ALC base case (see Table 2.3)

head. As data decompression is very fast, the restart process benefits from data reduction with a minimal decompression overhead (see `Decompression` in the Figures). On average, a 20-25% time saving is achieved. When restarting from incremental checkpoints, the overhead is large, and the benefits of compression are more noticeable.

Due to the high influence that the read of the checkpoint files can have on the performance of the restart operation, recent studies are focused on reducing this impact. A post-checkpointing tracking mechanism is presented in [53] to reduce

restart latency by overlapping application recovery with the retrieval of checkpoint files. In the case of incremental approaches, the number of incremental checkpoints has great influence in the restart overhead. There exist studies [60] that provide a model to determine the optimal number of incremental checkpoints between two consecutive full checkpoints. One possible approach to reduce the restart overhead would be to merge the full checkpoint file and the incremental ones into a single file at the checkpoint server before a restart is required [2]. Nevertheless, it must be considered that in a traditional fault tolerance context the main objective is accelerating checkpoints storage, which is performed several times per execution. Restart, however, is a secondary target, as it may never be necessary.

2.7. Related work

All the techniques mentioned in this chapter focus on reducing checkpoint file sizes. Another way to optimize the computational and I/O cost of checkpointing is to avoid the storage of checkpoint files in a parallel file system. Plank et al. proposed to replace stable storage with memory and processor redundancy [70]. Recent works [13, 14, 34, 93] have adapted the technique, known as *diskless checkpointing*, to contemporary architectures. The main drawback of diskless checkpointing are its large memory requirements. As such, this scheme is only adequate for applications with a relatively small memory footprint at checkpoint. Other recent solutions focus on the use of non-volatile memory technology, like solid-state disks (SSDs) to keep checkpoint data [52]. SSDs offer excellent read/write throughput when compared to secondary storage and thus they can help to reduce disk I/O load. Moody et al. propose a multi-level checkpoint system that writes checkpoints to RAM, Flash, or disk on the compute nodes in addition to the parallel file system [57].

Other works focus on minimizing the network and file system contention caused by the parallel checkpointing by reducing the number of simultaneous checkpoints. Norman et al. identify at compile-time recovery lines formed by staggered checkpoint calls so that the concurrent writing of checkpoint files is minimized at runtime [63]. In [47] the data layout of the checkpoint files are rearranged to reduce the number of files serviced by each I/O server. Additionally, the write operations of concurrent checkpoints are serialized on each computer node to further improve

the checkpointing performance.

Accelerators have been also considered for reducing checkpointing overhead [27, 31]. Mainly, these works focus on computing hash functions using GPUs. Whereas significant speed-ups are obtained, hash calculation is not a bottleneck in the checkpointing process. Data compression could be an interesting target for hardware accelerators as up to the present moment, we have not found any implementation in the literature.

2.8. Concluding remarks

This chapter has analyzed different alternatives to reduce the size of the checkpoint files generated by ALC approaches: live variable analysis, incremental checkpointing, zero-blocks elimination, and data compression. These techniques have been implemented in an ALC tool, CPPC, obtaining important file size and checkpoint latency reductions.

The reduction of the checkpoint sizes will be particularly useful for parallel applications with a large number of parallel processes, where the transference of a large amount of checkpoint data to stable storage can saturate the network and cause a drop in application performance.

The implementation in the application-level is a key aspect of the proposal. On one hand, it allows a more efficient implementation of the proposed techniques. On the other hand, it does not make any assumptions about the underlying system hardware/software characteristics, thus enabling portable operation.

Though all these techniques were incorporated to CPPC, in the following chapters only live variable analysis and zero-block exclusion will be used. Both incremental checkpoint and checkpoint compression present drawbacks with the techniques explored in next chapters: the former concerning high restart overheads, and the latter regarding portability issues.

Chapter 3

Checkpoint-based process migration

This chapter extends CPPC to proactively migrate MPI processes when impending failures are notified, without having to restart the entire application.

3.1. Introduction

Most of the approaches present in the literature use the checkpoint files to respond in the event of a failure. In a practical scenario, checkpoints should be frequently stored in order to cope with unpredicted failures in a traditional reactive way. The determination of optimal checkpointing intervals for reactive approaches has been studied extensively in the past [28, 29, 88]. In these solutions, all the processes are restarted from their last checkpoint in case of failure. However, a complete restart is unnecessary, since most of the nodes will still be alive. Moreover, it has important drawbacks. First, full restart implies a job requeueing, with the consequent loss of time. Second, since the assigned set of execution nodes is, in the general case, different from the original one, checkpoint data must be moved across the cluster in order to restart the computation, usually causing significant network contention and therefore high overheads. These limitations can be overcome if affected processes can be individually restarted in case of a single node failure [90].

With the recent advances in monitoring systems, and thus in the prediction of hardware failures [78], solutions that use checkpointing to implement *proactive* policies have emerged [11, 91]. In these approaches tasks are preemptively migrated from processors that are about to fail. Thus, only terminating processes need to dump their state, reducing the usually high I/O overhead associated to checkpointing solutions. Studies show failure avoidance to be more efficient than traditional fault tolerance [9]. Moreover, both techniques can complement each other, reducing checkpoint frequency when the success rate of failure prediction is high [91].

To be effective, proactive solutions require that failures can be anticipated accurately. However, this issue should not be seen as a limitation nowadays. Health monitoring has become a common feature in servers and HPC (High Performance Computing) components. Such monitors range from processor temperature sensors to baseboard cards with a variety of sensing capabilities, including fan speeds, voltage levels and chassis temperatures. Recent studies show that, assisted by such capabilities, node failures may be predicted in large-scale systems with a high degree of accuracy [54, 77, 37, 83, 35].

The following sections describe how to extend the CPPC framework to proactively migrate processes when impending failures are notified.

3.2. Process migration using CPPC

The basic idea behind dynamic migration in parallel applications is to spawn new processes that will be in charge of continuing the work of the terminating processes on other computation nodes. Migration is preferably performed to spare nodes, although the use of already allocated ones is also possible. In a checkpoint-based solution, when a signal with a migration request is received, the terminating processes need to write their state to checkpoint files, while newly spawned processes need to read these files and recover the state of the terminating processes. In addition, before resuming the execution, communication groups must be rebuilt to exclude terminating processes and include the newly spawned ones [15].

The reconstruction of the communication groups is a critical step, since replacing communicators may lead to an inconsistent global state: messages sent/received us-

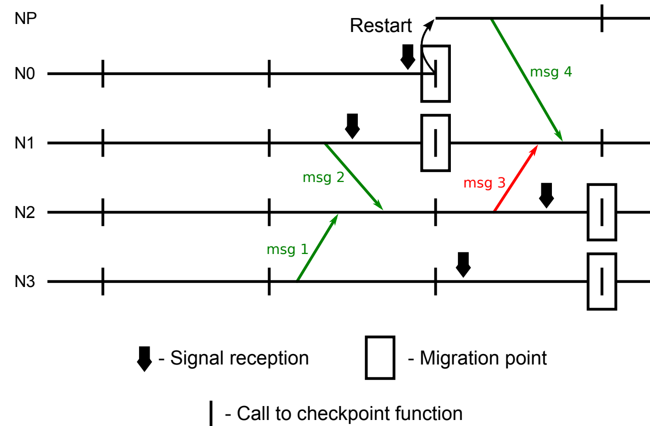


Figure 3.1: Inconsistent global state after migration in processes that are running asynchronously

ing the old communicators cannot be received/sent using the new ones. One possible solution to this problem is to make the reconstruction of the communicators, and thus the migration, in locations where there are no pending communications, i.e. safe points. The CPPC compiler automatically detects safe points, thus facilitating the implementation of this approach. Moreover, based in a heuristic evaluation of computational cost, it places calls to the checkpoint function in selected safe locations. These calls could be used as migration points. However, conducting the migration from different checkpoint calls in different processes may lead to inconsistencies, since messages may be sent in the code executed in between the two calls. The communication labeled *msg. 3* in Figure 3.1 is an example of such a situation. In order to implement proactive process migration, processes need to dynamically engage in a negotiation to decide which checkpoint call to select as migration point.

Summarizing, there are two main phases on process migration using CPPC: a negotiation to reach consensus on the migration point; and the process migration itself, which includes the communicator reconstruction.

3.2.1. Negotiation protocol

The negotiation protocol must ensure that, when a migration is initiated, all processes are able to converge to a single selected checkpoint location to achieve global coordination. There are different approaches that can be used towards this

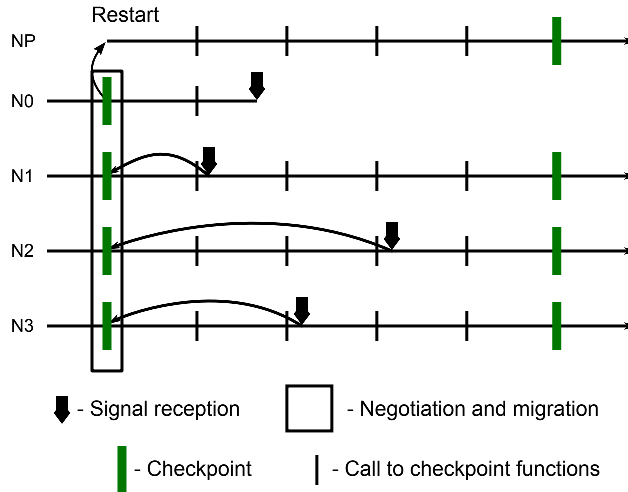


Figure 3.2: Backward negotiation

end. A first possibility is *backward negotiation*, shown in Figure 3.2. Using this strategy all processes agree to restart their execution from the most recent recovery line. Another solution is *forward negotiation*, detailed in Figure 3.3, in which processes agree to coordinate at the next checkpoint call to be reached by the process that has advanced the farthest in the execution. Backward negotiation uses previously created checkpoint files. As such, its greatest advantage is avoiding the overhead of creating new snapshots during migration. This, in turn, incurs higher overheads during a failure-free execution, given that processes need to checkpoint often. Backward negotiation can be thought of as being roughly equivalent to a stop and restart approach but avoiding the job requeueing. All processes need to recover a previous state, causing a loss of computation and higher total execution overhead. Due to these shortcomings of backward negotiation, forward negotiation will be the approach followed in this work.

We are assuming that the `mpirun` process receives a migration request from a user or batch scheduler and propagates it by sending a signal to each of the spawned MPI processes. This external signal triggers a handler which activates a migration flag in the CPPC controller to change to *migration* mode, a new operation mode added to CPPC besides the already explained checkpoint and restart ones (see Section 1.2.2). In migration mode each MPI process has to coordinate with the others to find out the farthest checkpoint location that has been reached by any of them. As CPPC

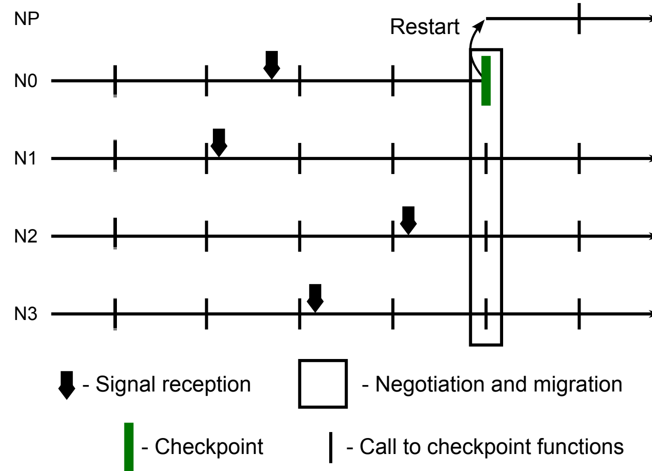


Figure 3.3: Forward negotiation

stores in an internal parameter (`touchedCheckpoint` variable) the number of times the CPPC_Do_checkpoint() function is called in each process, a direct and simple solution is to use an MPI reduction operation inside the signal handler to calculate the maximum `touchedCheckpoint` value. Unfortunately, according to the MPI standard, implementations may prohibit the use of MPI calls from signal handlers. Thus, for the sake of robustness and portability an alternative negotiation protocol was built outside of the signal handler.

One-sided MPI communications are used so that processes may continue running asynchronously during the negotiation. Prior to invoke a one-sided MPI communication operation, each process has to specify the memory region (window) that it exposes to others. The window for the proposed negotiation algorithm comprises two values for each process: `flag` and `touched`. The `flag` value indicates whether a process is actively engaged in the negotiation. It is activated when a checkpoint function is reached after migration mode has been enabled. It will not be deactivated until the migration is finished. The `touched` value contains the value of the `touchedCheckpoints` parameter and it is kept up to date throughout the execution when in checkpoint mode. When in migration mode this value is not updated, and contains the value it had when the migration mode was enabled by the external signal.

Algorithm 1 shows the pseudocode of the negotiation algorithm. This code is

Algorithm 1: Pseudocode for the negotiation protocol

```
value[2];
for allRemoteProcesses do
    flag = 0;
    while !flag do
        LockWindow();
        GetRemoteWindow(&value);
        UnlockWindow();
        if value.touched > touchedCheckpoints then
            return; %continue to next checkpoint
        end
        flag = value.flag;
    end
end
```

included inside the checkpoint function and executed only in migration mode. Each process p reads the exposed `flag` and `touched` values of every other process q (`GetRemoteWindow()` in the figure). In this way, all processes have a global picture of the execution status. As explained before, each process must advance up to the farthest reached checkpoint location. If process q is more advanced than process p (i.e. $\texttt{touched}_q > \texttt{touched}_p$), then p must continue its execution until the next checkpoint location, regardless of the value of $\texttt{flag}_q$. Otherwise, a deadlock would occur if process q were waiting for a message from process p sent in the application code in between the checkpoint call number $touched_p$ and the one number $touched_q$ (hence unable to reach the next checkpoint location and activate its flag). If process q has not yet advanced beyond checkpoint $touched_p$, then p waits for q to enable its `flag` value. This indicates that q is aware that a migration is to take place. Once all processes are verified to be aware of the negotiation process and not more advanced than process p a consensus migration point has been discovered, and process p has arrived at it. Note that other processes may be behind in their execution, and will arrive later at the same migration location in an asynchronous way.

Local updates to the window values use exclusive locks (`MPI_LOCK_EXCLUSIVE`) to guarantee consistency, whereas remote reads (Get operations on Figure 1) use shared locks (`MPI_LOCK_SHARED`), which allow for concurrent read accesses.

3.2.2. Process migration

At this point, a migration point has been agreed upon and processes begin arriving at that location independently. Still, several issues remain to be solved: saving the terminating processes state; spawning new processes to continue the computation done by the terminating ones; updating and managing the communication groups; and restoring the terminating processes state in the newly spawned processes. Whenever possible, these actions will be taken by each process without coordination. All the required steps are fully explained below.

The state of the terminating processes is saved using native CPPC capabilities. Checkpoint file creation begins once the terminating process reaches the migration point. Note that, due to the spatial coordination protocol employed by CPPC, there is no need to coordinate processes at the migration point before the state dump can start. Checkpoint creation is managed by a new ad-hoc thread, which allows for the reconfiguration to occur concurrently.

The newly spawned processes are created using the `MPI_Comm_spawn_multiple()` MPI-2 function. This call is collective over the communicator, that is, it must be performed by all the processes in the communication group involved in the migration (that is, the world communicator). Depending on the implementation, `MPI_Comm_spawn_multiple()` may not return until `MPI_Init()` has been called in the spawned processes. Similarly, `MPI_Init()` in the spawned processes may not return until all processes in the original communicator have called `MPI_Comm_spawn_multiple()`. As such, `MPI_Comm_spawn_multiple()` in the original processes and `MPI_Init()` in the spawned ones form a collective operation over the union of parent and child processes that may imply a synchronization during the migration operation.

Spawning new processes creates an inter-communicator between the original and the newly created processes. Old communicators should be reconstructed, replacing terminating processes with the newly created ones. The approach used is to reconfigure the world communicator (`MPI_COMM_WORLD`) using the dynamic communicator management facilities provided in MPI-2. Other communicators, which derive from `MPI_COMM_WORLD`, will be reconstructed by re-executing the MPI calls used for creating them in the original execution. Figure 3.4 details the reconfiguration phase for the world communicator for an example where four processes take part

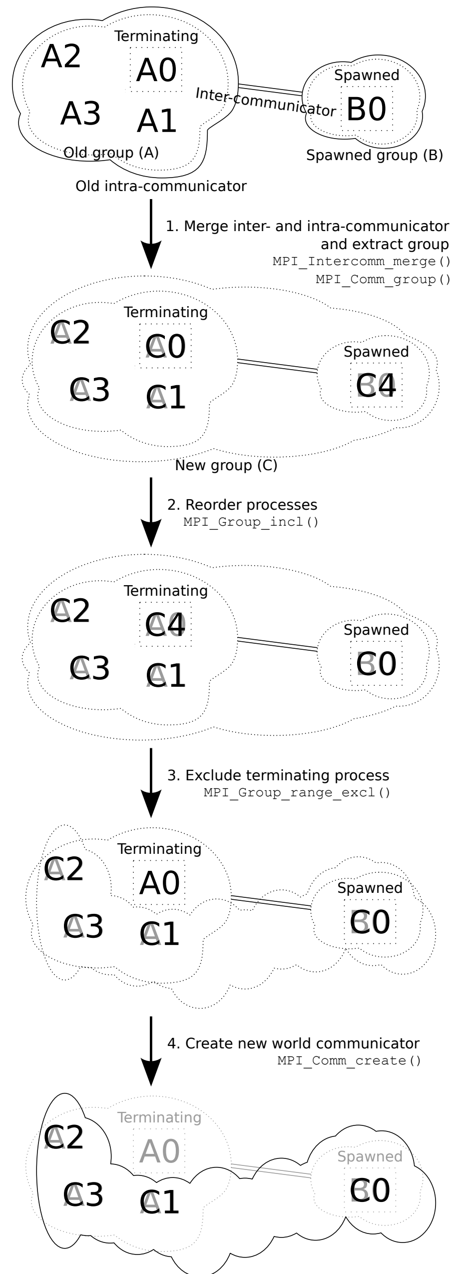


Figure 3.4: World communicator reconfiguration. Process A0, in the world communicator A, migrates to a new execution node. B0 is the newly created process to support the migration. The old world communicator A is reconfigured into a new world communicator C

of the migration operation and only one process is migrated to a new execution node. First, the two intra-communicators that contain the original and the new pro-

cesses need to be merged into a single one. The `MPI_Intercomm_merge()` function is used for this purpose. Afterwards, the group of processes that form the new intra-communicator is extracted via `MPI_Comm_group()`. Ranks in this group are reordered using `MPI_Group_incl()`, in such a way that the spawned processes will take over the ranks of the terminating processes. Afterwards, terminating processes are excluded from the group using `MPI_Group_range_excl()`. Finally, `MPI_Comm_create()` is used to build the new world communicator from the reconfigured group.

As described, this process only reconfigures the world communicator. However, in order for the migration to succeed, communicators which include any migrating process have to be rebuilt as well. Using the same approach for reconfiguring these communicators would require the participation of the terminating processes, which would in turn require the soon-to-fail nodes to be up for a longer time, reducing the chances of successful migration. In order to avoid this, the CPPC restart capabilities are used. MPI calls that result in the creation of new communicators (such as split operations) are identified and logged by CPPC both into memory and created checkpoint files. The set of communicators in an MPI application can be seen as a tree in which each node is created from another one by using a certain MPI operation (i.e. `MPI_Comm_split()`, `MPI_Comm_dup()`, etc.). The root of this tree is the world communicator. Taking advantage of the operation log provided by CPPC, this communicator tree is reconstructed from its root by orderly re-executing the logged operations. Regular processes (those that do not migrate) read the log from memory and re-execute its contents right after the reconfiguration of the world communicator. Spawned processes do so after reading the checkpoint file contents during their restart phase. Note that if these MPI operations are blocking in the MPI implementation used, a synchronization between the processes involved will be imposed.

Terminating processes, in turn, participate in the reconfiguration of the world communicator and wait until the creation of their checkpoint file is completed. When this happens, they notify the spawned processes (using the inter-communicator created by `MPI_Comm_spawn_multiple()`) that checkpoints may now be read (assuming a shared file system). Finally, they safely finish their execution.

Spawned processes still have to recover the terminating processes state from their snapshots contents. This involves reading the appropriate checkpoint file and

executing the necessary RECs to regenerate non-portable state. This is achieved by delegating to CPPC and employing its native capabilities.

3.3. Experimental evaluation

Experiments were performed to evaluate both the scalability of the proposed solution and the total overhead associated to the migration. Pluton N1 cluster (presented in Section 2.6) was used to carry out these experiments. All the checkpoint files were stored into the working directory, mounted via NFS and connected to the cluster by a Gigabit Ethernet network.

The application testbed was comprised of six out of the eight applications in the NPBs compiled with the OpenMPI library version 1.5.4. The IS and MG benchmarks were discarded due to their low execution times.

The experimental results obtained are classified in two subsections. The first one evaluates the scalability of the solution, analyzing the duration of the different phases of a migration operation in relation to different impacting factors. The second subsection evaluates the migration overhead of the proposed approach, and compares it with other different solutions.

3.3.1. Scalability

The scalability of the solution can be analyzed from three points of view: the impact of the total number of processes in the execution, the effect of the number of migrating processes, and the influence of the application memory footprint. In all these experiments the migration time is broken down into 5 parts (see Figure 3.5):

- *Negotiation*: execution time between the `mpirun` migration request and the call to the spawn function. This time is measured in the worst possible case, that is, when the signal is received by at least one of the processes just after a checkpoint function call.
- *Spawn&Rec*: execution time of the spawn function and the reconfiguration of the world communicator.

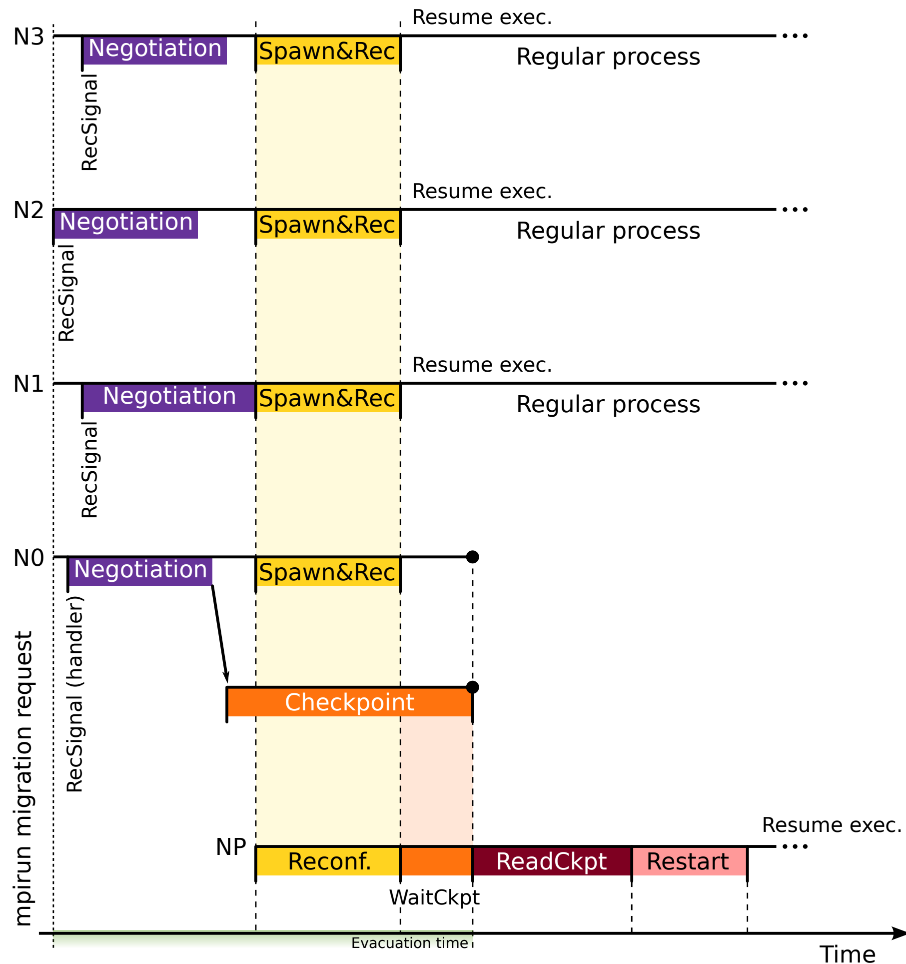


Figure 3.5: Actions and temporal sequence for four processes involved in a migration operation. Process $N0$ migrates to a new execution node. NP is a newly created process to support the migration. $N1 - N3$ are regular processes that passively participate in the migration

- *WaitCkpt*: average execution time between the end of the reconfiguration phase in the newly spawned processes, and the start of the checkpoint file read.

- *ReadCkpt*: average time it takes to read the checkpoint file from disk in the newly spawned processes.

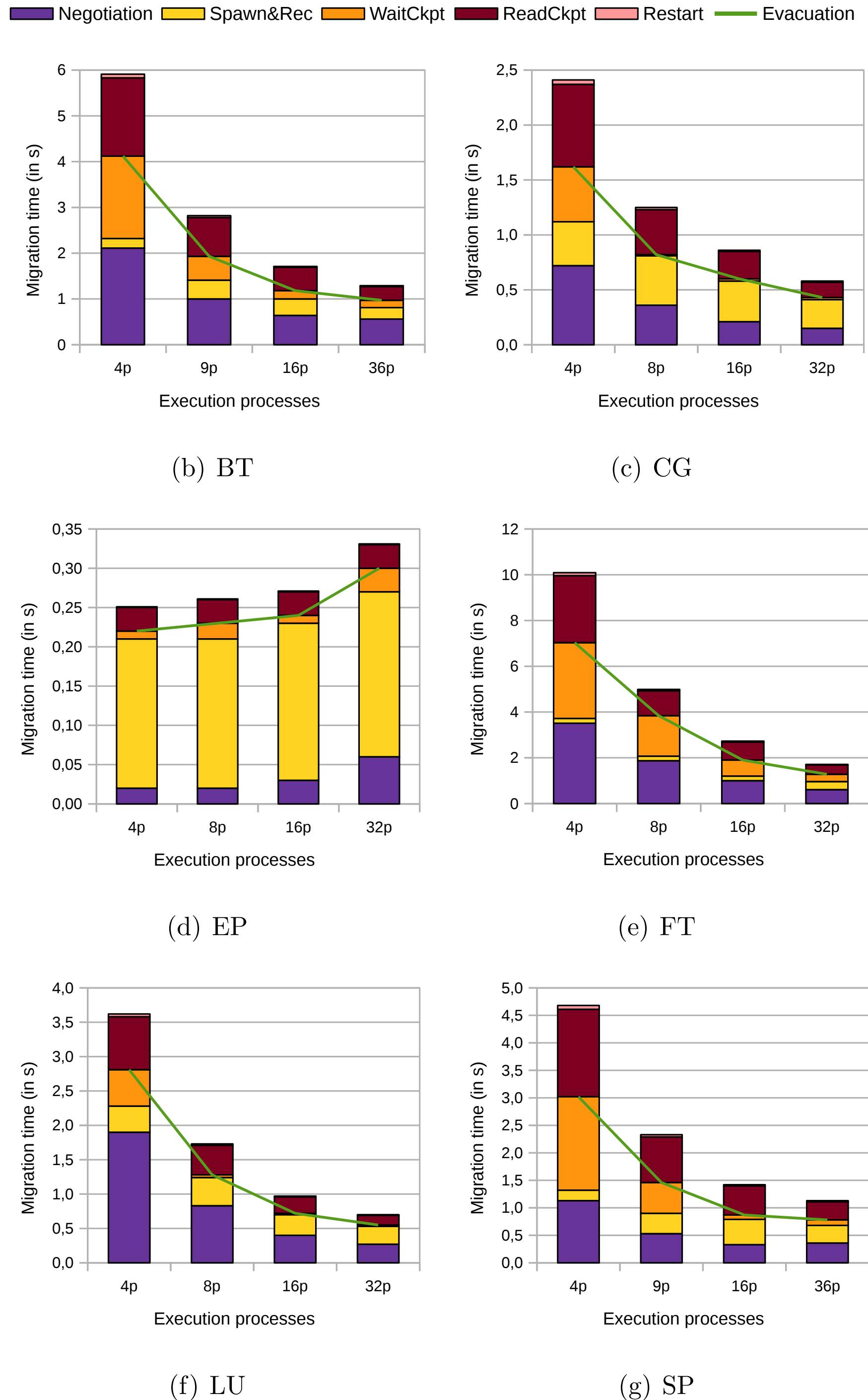
- *Restart*: average time for restarting the application once the checkpoint file has been read. It includes the execution of the RECs.

Impact of the number of processes

These tests measure the scalability of the migration solution when increasing the number of total processes. Experiments were carried out using 4, 8, 16 and 32 processes, except for BT and SP that need a square number of processes, thus using 4, 9, 16 and 36. Although each node was running at least 4 processes, in this experiment only one process was migrated each time (equivalent to what would occur if an imminent failure was predicted in one node with one process running on it). In all the cases the terminating process was migrated to a spare node. The results obtained using the NPB applications using class B are shown in Figure 3.6.

The *Negotiation* time depends on how often the CPPC checkpoint function is called, the inherent synchronization between the processes during the execution of the application, and the overhead introduced by the negotiation protocol. Given that the migration signal is received just after a checkpoint call, the *Negotiation* time will be at least the time between two consecutive checkpoint calls. In all NPB, the CPPC checkpoint function is called once in each iteration of the main computational loop. The *Negotiation* time, as well as the execution time per iteration, for 16 processes and class B, are shown in Table 3.1. Except for EP, the processes of all the NPB applications are inherently synchronized in every internal iteration of the application. This means that, during the negotiation phase, one process will never advance more than one iteration before reaching the migration point. Results in Table 3.1 prove that in these cases the overhead associated to the negotiation protocol is almost negligible, the *Negotiation* time being mainly determined by the iteration time. As the number of processes increases, the iteration time decreases, thus achieving a reduction in the *Negotiation* time. Table 3.2 shows the execution time per iteration for different number of processors. However, when the processes are not synchronized, such as in EP, it may take several checkpoint calls to reach the migration point.

The *Spawn&Rec* time increases slightly with the total number of processes, since this phase involves different collective communications. However, as can be observed in Figure 3.6, this time is at most 0.5 seconds in these experiments.



(b) BT (c) CG

(d) EP (e) FT

(f) LU (g) SP

Figure 3.6: Scalability impact when increasing the number of total processes. NPB class B migrating one process

Table 3.1: Negotiation times and iteration times, running 16 processes

NPB. Class B	*Negotiation (s)*	*Iteration time (s)*
BT	0.64	0.56
CG	0.21	0.20
EP	0.03	0.01
FT	1.00	1.00
LU	0.40	0.39
SP	0.33	0.28

Table 3.2: Iteration times (in s) for different number of total processes

NPB. Class B	*4p*	*8/9p*	*16p*	*32/36p*
BT	2.05	0.94	0.56	0.28
CG	0.73	0.38	0.20	0.15
EP	0.01	0.01	0.01	0.01
FT	3.59	1.85	1.00	0.61
LU	1.46	0.73	0.40	0.24
SP	1.14	0.50	0.27	0.14

Table 3.3: Checkpoint file size per process (in MB) and checkpoint write and read times (in s)

NPB. Class B	*Size*	*Write T.*	*Read T.*
BT	30.69	0.51	0.58
CG	14.23	0.33	0.25
EP	1.10	0.04	0.03
FT	48.10	0.87	0.79
LU	14.49	0.25	0.24
SP	30.93	0.52	0.53

As can be seen in the figure, in most of the cases, the biggest contribution to the migration overhead is due to the write and read of checkpoint files. In these experiments checkpoint files are stored to shared disk via NFS, using a Gigabit Ethernet network. In this situation, checkpoint file sizes are critical to minimize the I/O time. As explained in Chapter 2, CPPC applies live variable analysis and identification of zero-blocks to decrease checkpoint file sizes [17]. These sizes, as well as the checkpoint write and read times, for NPB class B and 16 processes, are shown in Table 3.3. When the number of processes grows, the checkpoint files use

Table 3.4: Checkpoint file sizes per process (in MB) for different number of total processes

NPB. Class B	*4p*	*8/9p*	*16p*	*32/36p*
BT	106.61	52.10	30.69	17.27
CG	47.48	24.79	14.23	7.64
EP	1.10	1.10	1.10	1.10
FT	192.12	96.11	48.10	24.10
LU	48.79	26.62	14.49	8.24
SP	96.35	50.19	30.93	18.24

to become smaller and, thus, the time to write or read the file from disc decreases. Table 3.4 shows the checkpoint file sizes for the different number of processes. Again, the EP application is a special case, since the checkpoint file sizes does not decrease when the number of processes grows. Thus, the checkpoint write and read times for EP remains constant in Figure 3.6. Note that the write of the checkpoint file is overlapped with the spawn and reconfiguration phase (see Figure 3.5), and the time shown in these figures is the one consumed in the non overlapped part (*WaitCkpt*).

Finally, the *Restart* time is very small for all the tested applications. It depends on the amount of state recovered using code re-execution (RECs) on the newly spawned processes. As occurs for the checkpoint file sizes, by increasing the total number of processes, the amount of state to be recovered usually decreases, and likewise the *Restart* time.

Note that the sum of *Negotiation*, *Spawn&Rec* and *WaitCkpt* corresponds to the *evacuation* time, that is, the time needed after the migration request to free the nodes that are about to fail. This time is represented with a line in Figure 3.6. Evacuation time decreases when scaling the number of processes, except for EP. Although the evacuation time in EP increases slightly with the number of processes, its variance in absolute value is lower than 0.1s. This behavior can be explained by the negative impact of the scaling in the *Negotiation* time for the EP application.

In order to make the proposed solution practical, the evacuation time should be smaller than the lead-time (time ahead of the potential occurrence of a failure) of the prediction mechanism. In all the experiments the evacuation time was only of a few seconds. In [83] lead-times between tens of seconds and several minutes

are reported. Thus, the evacuation time observed in these experiments proves the viability of the solution.

Impact of the number of terminating processes

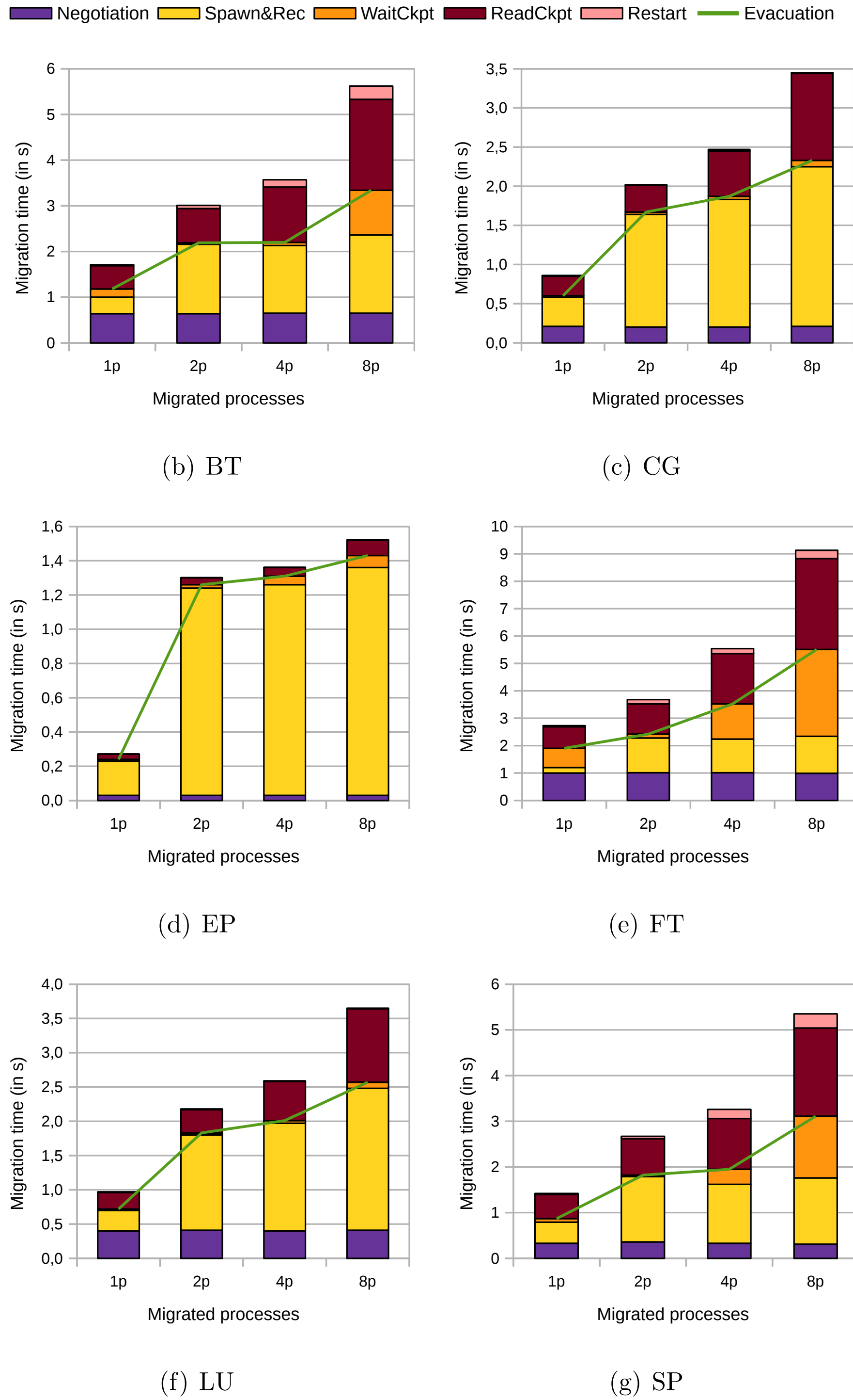
When a node is about to fail, all the processes running on it have to be migrated to a new location. The previous subsection shows experimental results obtained assuming only one terminating process. In this subsection experiments were carried out varying the number of terminating processes (from 1 to 8) and maintaining the number of total processes at 16. In these experiments each node runs 2 processes and the terminating processes are migrated to spare cores of nodes. The results are shown in Figure 3.7.

As expected, the *Negotiation* time remains constant, as the execution time between two consecutive calls to the CPPC checkpoint function does not change.

The *Spawn&Rec* time increases with the number of migrating processes. It is particularly low for migrating a single process and augments significatively when going from 1 to 2 migrating processes. This is probably due to internal OpenMPI optimizations of the `MPI_Comm_spawn_multiple()` function when spawning only one process.

The checkpoint write and read times also grow because the number of checkpoint files dumped to the NFS shared directory increases. Note that, in most cases, the *WaitCkpt* time is negligible because the time needed to dump the checkpoint files is overlapped by the increase in the time spent in the *Spawn&Rec* phase.

Finally, the *Restart* time increases with the number of migrating processes for those applications that recover non-world communicators during their restart phases (BT, FT, and SP). The reason for this is that, as mentioned in Section 3.2.2, the collective operations executed during this recovery are blocking. As such, these operations impose a synchronization that becomes more costly as the number of migrated processes increases.



(b) BT (c) CG

(d) EP (e) FT

(f) LU (g) SP

Figure 3.7: Scalability impact when increasing the number of migrating processes. NPB class B and 16 processes

Impact of the application size

In this subsection NPBs of classes A, B and C are used to evaluate the impact in the migration time of different application memory footprints. Figure 3.8 depicts the experimental results obtained when scaling the problem size, using 16 processes and migrating only one.

As expected, for most cases an increase in the migration duration is observed, since the problem scaling results in larger data per process. The *Negotiation* time grows due to the increase of the iteration time. The *Spawn&Rec* time remains almost constant, since it does not depend on the memory footprint of the application. Both the checkpoint write and read times and the *Restart* time augment due to the increase in the checkpoint file sizes and in the amount of state to be recovered, respectively.

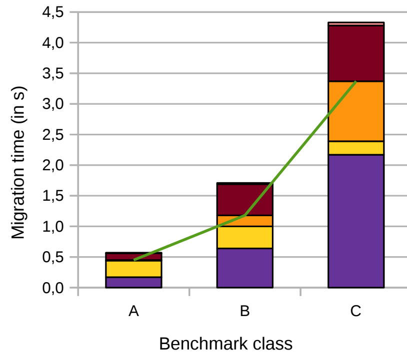
Though the problem scaling leads to an increase in the migration duration, considering the total execution time of the application, a decrease in percentage terms is observed when the problem size increases.

3.3.2. Overhead

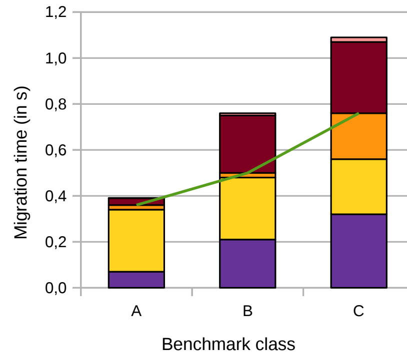
The overhead was studied for each of the NPB codes by running class C with 16 processes divided in 8 nodes. In this configuration, when a node is about to fail, 2 processes are migrated. Class C was chosen to get a more realistic execution time. The results are shown in Table 3.5 where the column labeled *Orig.* shows the execution time of the original application in a fault-free execution and the *Instr.* one the time of the application instrumented with CPPC, again in a fault-free execution. In most cases the instrumentation overhead is minimal, generally less than 1%. The application execution times when a failure is imminent and the migration of a node is performed are shown in the *Migr.* column. As seen in the previous subsections, the migration time is mainly dominated by the times to read and write the checkpoint files. Thus, FT is the application with the highest relative overhead (10% with respect to the original code) due to their larger checkpoint files (see Table 3.3).

Table 3.5 also includes the execution time of the checkpoint and rollback solution using CPPC. Checkpoint files are periodically dumped and, in case of failure,

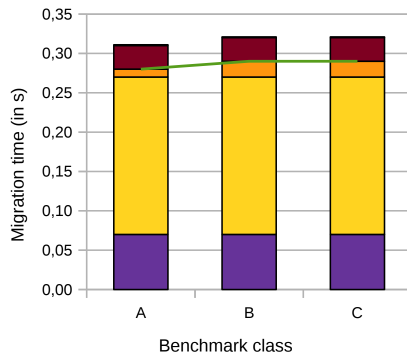
Negotiation Spawn&Rec WaitCkpt ReadCkpt Restart Evacuation



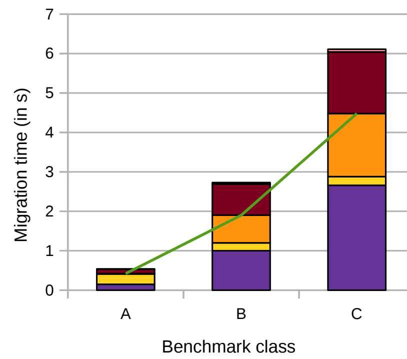
(b) BT



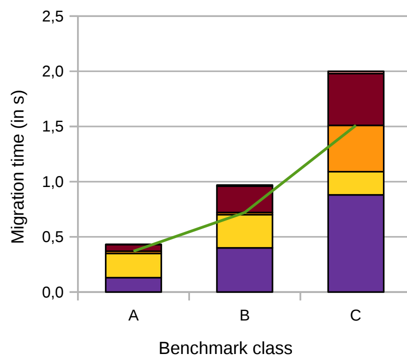
(c) CG



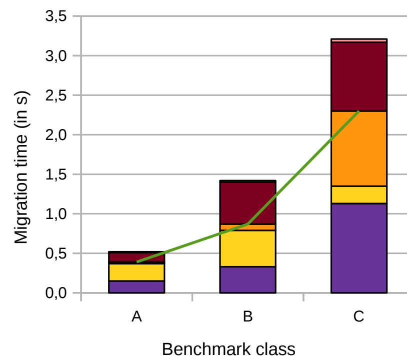
(d) EP



(e) FT



(f) LU



(g) SP

Figure 3.8: Impact of increasing application size via the NPB classes. Running on 16 processes

Table 3.5: Execution times (in s) using 16 processes, with and without migration

	CPPC			Ckpt&	MVAPICH	
NPB. Class C	*Orig.*	*Instr.*	*Migr.*	Rollback	*Orig.*	*Migr.*
BT	449.94	450.55	459.63	482.27	471.48	486.39
CG	43.91	43.87	45.93	53.75	47.32	59.04
EP	33.27	33.29	34.59	33.70	34.35	36.01
FT	99.49	99.42	109.56	150.73	152.38	184.95
LU	418.41	419.17	425.47	433.39	435.47	444.47
SP	518.74	520.17	532.18	547.37	548.52	568.72

the complete execution is restarted in new nodes. The Checkpoint&Rollback times shown in this table are the optimal ones for this approach, that is, only one checkpoint file is dumped before the failure occurs, thus avoiding additional overhead due to useless dumps; and the rollback is performed just after the checkpoint, thus avoiding loss of work on restart. The overhead associated to proactive migration is lower than the overhead associated to the checkpoint and rollback solution, the only exception being EP. The relatively high overhead of the EP benchmark (1.21%) is due to the high number of MPI window updatings as a consequence of the high number of internal iterations (more than 500 iterations in approximately 5.5 seconds). Fortunately it is rather improbable to find this behavior in a real application. It can be concluded that the proactive migration approach can significantly decrease the cost to survive a node failure.

For comparative purposes, Table 3.5 also shows the execution time using MVAPICH version 1.8. MVAPICH provides process migration based on BLCR (Berkeley Lab's Checkpoint/Restart Library) [49] and FTB (Fault Tolerant Backplane) [36] libraries for Infiniband, iWAPP and RoCE architectures [1]. The table shows the MVAPICH original execution time, that is, a fault-free execution, and the execution time when one node needs to be migrated. When these experiments were performed, MVAPICH process migration support was only available for Mellanox Infiniband adapters. Unfortunately, cluster Pluton has a QLogic card, and the Mellanox interface over the QLogic cards did not achieve its top performance, resulting in a MVAPICH fault-free execution slower than the OpenMPI execution. Regardless, the solution proposed in this chapter results advantageous for all the NPB applications also in percentage terms as shown in Figure 3.9. This figure shows the

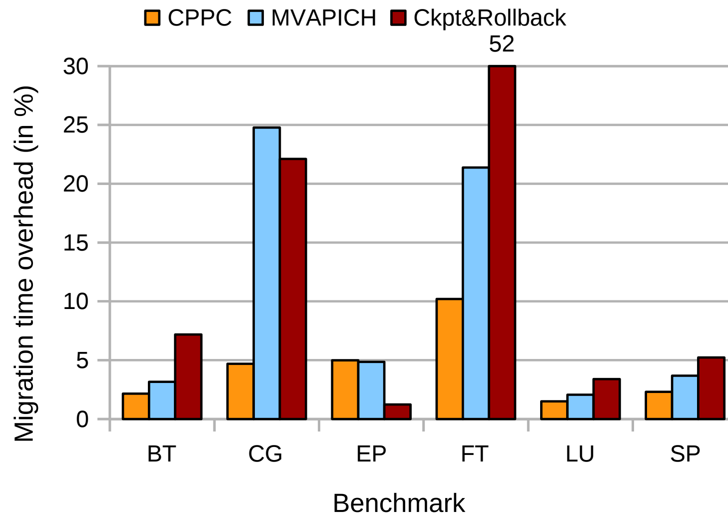


Figure 3.9: Overhead (in %) for NPB applications (class C - 16 processes) when a node is preemptively migrated (case of CPPC and MVAPICH) and when a node fails (case of Ckpt&Rollback)

Table 3.6: Checkpoint sizes (in MB) per process (running in 16 processes) for CPPC and BLCR

NPB. Class C	*CPPC*	*BLCR*
BT	110,23	111.04
CG	27.74	72.19
EP	1.04	2.36
FT	192.12	450.36
LU	50,89	51.60
SP	103,14	100.61

overhead with respect to a fault-free execution. In order to provide a fair comparison, the MVAPICH overhead is calculated with respect to the MVAPICH fault-free execution. The benefit obtained using CPPC versus MVAPICH can be in part explained by the smaller size of the checkpoint files. Table 3.6 shows the sizes of the checkpoint files per process generated for each application using CPPC and BLCR. Those applications where CPPC achieves significant reductions in checkpoint sizes (CG and FT) also present lower overhead under migration.

In Section 3.3.1 it can be seen that EP migration time with 32 processes is

negatively affected by an increase in the Negotiation time. Results in Figure 3.9 evidence that the coordination between processes to perform the migration operation is a bottleneck for EP, since for this application the best performance is obtained by the checkpoint and rollback approach, that avoids the coordination during execution time.

When comparing MVAPICH with the CPPC-based proposal it must be noted that one of the most important features of the latter, besides its efficiency, is its portability, as it does not need any specific architecture, operating system (OS), MPI implementation or system file to work.

Finally, results in Figure 3.9 also provide an idea of the impact that false-positives associated with the failure prediction may have on system performance. The overhead of migration is, for most of these benchmarks, less than 3% when migrating only one process. In [35] a percentage of false positives smaller than 10% is reported. Thus, we can conclude that the overhead due to this issue will not be very significant.

3.4. Related work

Process migration may be implemented either through dynamic migration or based on the simple stop-and-restart approach [55, 8]. In this section we will focus on proposals that, like the one proposed in this chapter, address dynamic process migration.

Some existing approaches rely on operating system virtualization techniques. Hacker et al. [38] investigate the use of OpenVZ to perform dynamic migration of parallel applications. Chackravorty et al. [11] use Charm++ [48] and Adaptive MPI (AMPI) [39] to implement a transparent proactive fault tolerance approach. In [59] the live migration mechanism in Xen [3] is exploited to implement a live migration solution. However, the same authors reported in a later work [26] that, in HPC, solutions at the process level are more widely accepted than those based on virtualization, mainly due to the lower penalty in performance.

In [81] a process level solution through checkpointing using the previously men-

tioned system level checkpointing tool BLCR is presented. The proposal extends both BLCR and LAM/MPI to allow process migration. Although the authors do not present experimental results, the paper explicitly states that the checkpoint file writing has a high overhead. Wang et al. [91] reduce this overhead through a live migration solution (execution proceeds while a process image is asynchronously transferred to a spare node) at the expenses of an increase in evacuation time. The migration mechanism implemented in MVAPICH2 [1] also relies on BLCR. It takes advantage of the Remote Direct Memory Access (RDMA) in Infiniband to reduce the I/O overhead [65]. Another proposal that uses a different migration mechanism is MPI_Mitten [22], an MPI library implemented on the HPCM (High Performance Computing Mobility) middleware [23] which achieves some independence from the underlying MPI implementation. All these solutions are based on a coordinated checkpointing approach to reach a consistent global state.

3.5. Concluding remarks

The approach presented in this chapter extends CPPC to proactively migrate processes from processors when impending failures are notified, without having to restart the entire application. It has been proved to be more efficient than the classical checkpoint and rollback solution. Besides, the proposed approach makes improvements on the two most important overhead factors in other migration solutions: process coordination and I/O overhead.

A light and asynchronous protocol has been designed to achieve a global consistent state during the migration operation, avoiding, when possible, operations that lead to stalls in the processes execution.

The approach improves efficiency through the reduction of the checkpoint read-/write overhead, since the use of CPPC allows for reduction in the checkpoint file sizes, and the dumping of the terminating processes state is overlapped to a certain extent with other stages of the migration operation. The experimental validation performed has shown the efficiency and scalability of the proposal.

Another remarkable feature is that the solution is implemented at the application level, and it is independent of the hardware architecture, the OS or the MPI

implementation used, and of any higher-level frameworks, such as job submission frameworks.

Despite the reduced checkpoint file size of the CPPC migration approach, write/read of the processes state continues to be the main cause of overhead. For this reason, the next chapter will focus on the reduction of the I/O cost of the migration operation.

Chapter 4

Improving performance in process migration

This chapter describes two optimizations to improve the performance in process migration. The first one is the in-memory migration that avoids write checkpoint files in disk. The second one optimizes the in-memory migration by splitting the checkpoint files, overlapping write, transfer and read phases in a migration operation.

4.1. Introduction

Process migration provides many benefits for parallel environments including dynamic load balance, by migrating processes from loaded nodes to less loaded ones; data access locality, by moving processes closer to the data that they are processing; and/or fault tolerance, by preemptively migrating processes from nodes that are about to fail.

The previous chapter described an application-level checkpoint-based approach to achieve process migration in MPI codes. The proposal was built on top of CPPC and it was based on the dumping of the state of the migrating processes to disk.

The bandwidth of a hard disk is small as compared to network bandwidth, even when using RAID configurations. To take advantage of network speeds and avoid the bottleneck of disk accesses, a new migration approach is proposed in Section 4.2.

The new solution substitutes storage to disk by in-memory checkpoint files and network transfers.

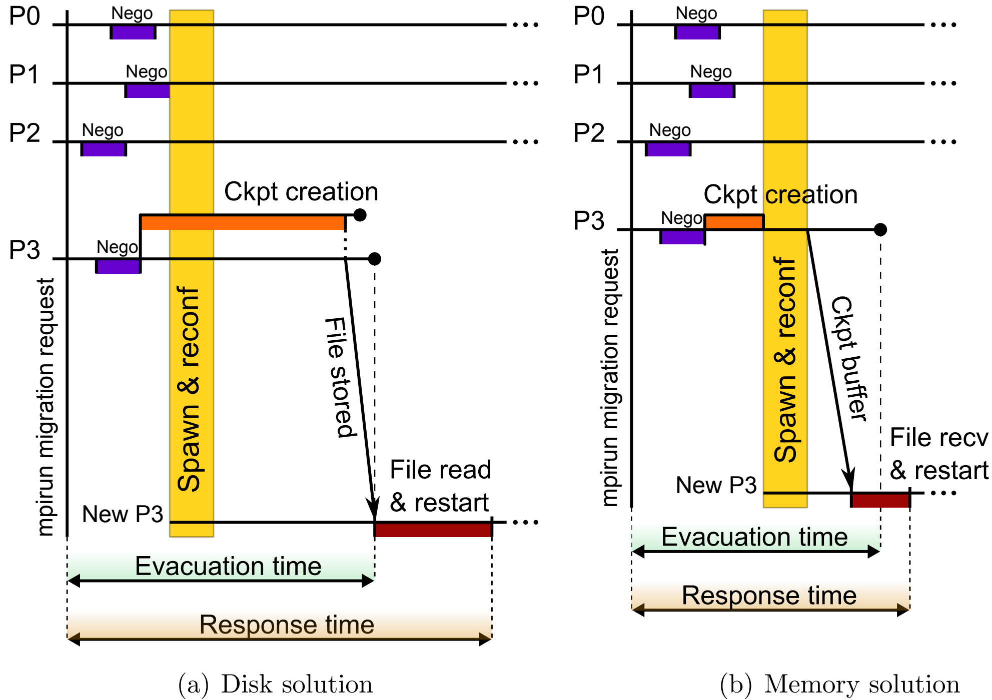
In-memory migration significantly reduces the I/O cost of the disk migration process. However, the significance of this reduction depends heavily on the network used. Additionally, it has large memory requirements, as a copy of the whole checkpoint has to be stored into memory.

Another way to optimize the I/O cost of migration based on checkpointing is to overlap the I/O operations. Section 4.3 proposes the split of the generated checkpoint files to overlap the different phases of a migration operation (state file writing in the terminating processes, data transfer through the network, and state file read and restart operations in the new processes), thus, reducing the migration time. Moreover, the split of the state files also allows for reduction in memory consumption, since it avoids storing the complete checkpoint file. Both improvements will be especially important for those applications with large checkpoint files.

4.2. In-memory checkpoint-based migration

To reduce the I/O overhead of a process migration, CPPC can store its checkpoint file in memory, instead of in disk, and sent it to the target node using the network, via MPI functions. As HDF5 allows to store files in memory, CPPC can use this in-memory HDF5 files to improve the migration operation.

The steps followed by the in-memory checkpoint-based solution for four processes running on different nodes, one of which is having its process migrated to a new node, are shown in Figure 4.1b). Figure 4.1a) shows the original behaviour. The first step is the negotiation to reach a consensus on the migration point. This protocol was presented in Section 3.2.1. Then, the migrating processes save their process state, storing it in memory (Ckpt creation in the figure). Afterwards, new processes are spawned in the target nodes to replace the migrating ones and the global communicator is reconstructed using the same MPI-2 functions as those mentioned in Section 3.2.2. Afterwards, the checkpoint files of the terminating processes are sent using MPI communications. At this point the terminating processes can safely finalize. The new processes receive the checkpoint files via an MPI communication



(a) Disk solution

(b) Memory solution

Figure 4.1: Checkpoint-based migration with CPPC.

and CPPC is used to recover the stored state.

For the implementation of this new approach, the CPPC writing layer has been modified. As mentioned in Section 2.5.2, the HDF5 library provides different file drivers which map the logical HDF5 address space to different types of storage. In the original CPPC version the default file driver (`SEC2 driver`) was used to dump the HDF5 data directly to stable storage. This driver is substituted by the HDF5 `core driver` which allows constructing the HDF5 files in memory. Additionally, the `HDF5 File Image Operations` [84], available since HDF5 v1.8.9, are used to work with these files in the same way that users currently work with HDF5 files on disk. The new writing layer is shown in Figure 4.2. When a signal with a migration request is received, the process state is stored in memory using the HDF5 core driver. Then, the HDF5 `H5Fget_file_image` function is used to obtain a buffered copy of these data in memory to be used as a file. This function returns a pointer to the buffer and its size. This information is used to send the data through the network to the newly spawned process using the MPI library. The data are received in a memory buffer in the newly spawned process via an MPI function. Finally, the `H5LTopen_file_image`

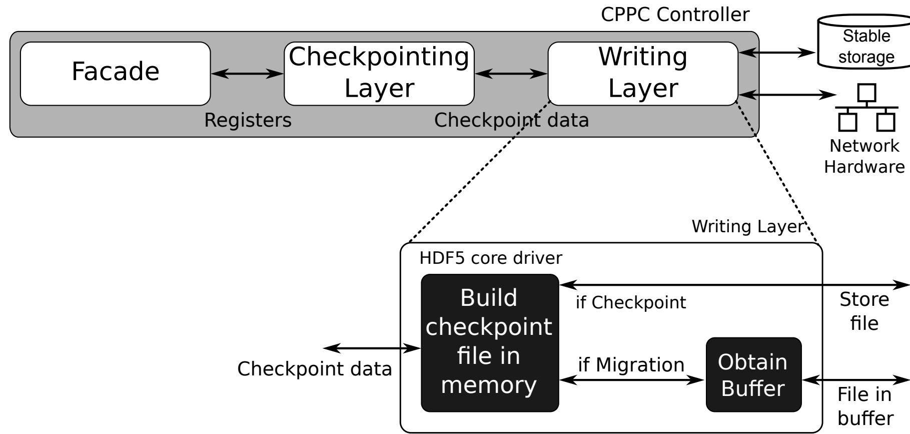


Figure 4.2: Modification of the CPPC Writing Layer

function is used by the new process to convert the buffered memory to an HDF5 file. Then, CPPC is used to read the file and restore the application state using its standard mechanisms. The `HDF5 core driver` also allows storing the files directly to disk. This feature is used to store the files to disk when creating a checkpoint file in a non-migrating situation, thus enabling fault tolerance.

4.2.1. Experimental evaluation

In this section the efficiency of the in-memory solution is evaluated. Comparisons with the disk-based approach are also shown. Both clusters, Pluton N1 and N2 (see Section 2.6 for more details), were used to carry out these experiments. To remark upon the time differences in this experimental evaluation, the files used in the in-memory scheme were transferred via the Gigabit Ethernet network in N1 and the InifiniBand FDR network in N2.

The application testbed was composed of the six out of the eight applications in the NPBs. The MPI implementation used was OpenMPI v1.5.4 for Pluton N1 and OpenMPI v1.6.4 for Pluton N2.

All the experiments were carried out using 16 and 32 processes and 8 processes per node (except BT and SP that need a square number of processes and were ran with 36) in order to also evaluate the scalability of the solution.

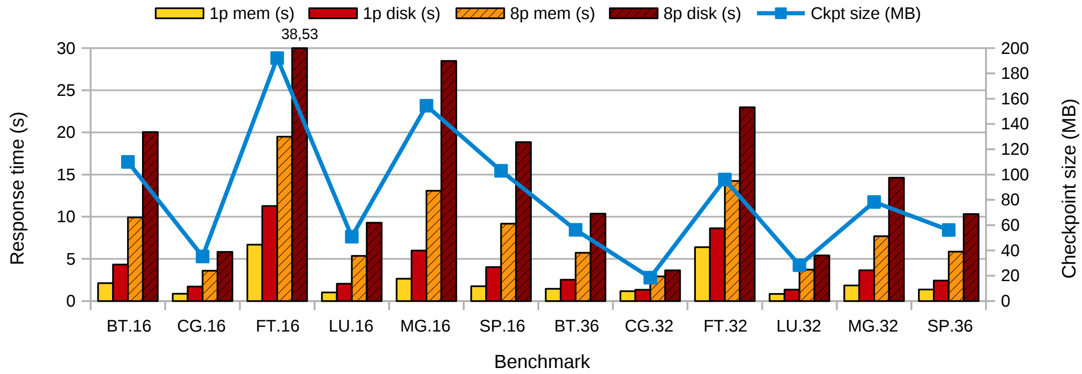


Figure 4.3: Response time (in seconds) in Pluton N1.

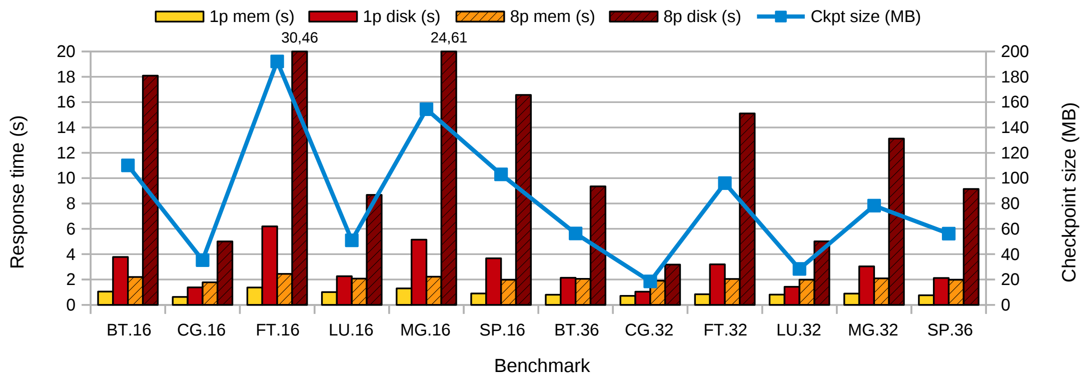


Figure 4.4: Response time for (in seconds) in Pluton N2.

Response Time

The response time is the execution time between the migration request and the actual restart in the new process (see Figure 4.1). Experiments in both clusters were carried out with two different configurations: migrating only one process and migrating the eight processes of a node. Figures 4.3 and 4.4 show response times in Pluton N1 and Pluton N2 respectively, migrating 1 and 8 processes via memory (*1p mem* and *8p mem* bars) and via hard disk (*1p disk* and *8p disk* bars). The checkpoint file sizes are also represented in the figures (line labeled *Ckpt size* referenced to the right axis). They vary between 1.04 MB and 192.12 MB per process.

The biggest contribution to the response time is the write and read of checkpoint

files. Thus, the response time increases with the checkpoint file size and the number of migrating processes. When the total number of processes grows, the checkpoint files tend to become smaller and, thus, the response time decreases. For the smallest checkpoint files and migrating only 1 process the response time is negligible (see for instance EP).

Note that the response time via hard disk is always higher than the time needed to migrate via memory. The maximum benefits are obtained for applications with large checkpoint files and, as expected, the differences are more significant for Pluton N2 due to its faster network. For instance, the response time for FT with 16 processes, out of which 8 are migrated, decreases by 50% when using the in-memory solution over the Gigabit Ethernet network (Pluton N1) and by 90% over the InfiniBand one (Pluton N2).

Evacuation time

The evacuation time is the time needed after the migration request to safely finish the migrating processes (see Figure 4.1). Reducing this time is specially important when migration is used to implement proactive fault tolerance approaches (tasks are migrated in a preventive way when node failures are anticipated).

In the migration via hard disk the evacuation time is the time between the migration request and the dumping of the checkpoint files to stable storage. As for the in-memory migration, the evacuation time is the time between the migration request and the return of the MPI function used to send the in-memory checkpoint file.

Figures 4.5 and 4.6 show the evacuation times in Pluton N1 and Pluton N2 respectively. As can be seen, the in-memory solution is always better than the disk-based one except for the smallest checkpoint sizes, the difference in these last cases not being significant (less than 0.5 seconds). The best results are obtained for applications with large checkpoint files using Pluton N2. For instance, the evacuation time for FT in Pluton N2 with 16 processes, out of which 8 are migrated, decreases from 16.34 s for the disk-based solution to 2.22 s for the in-memory one (an 86%).

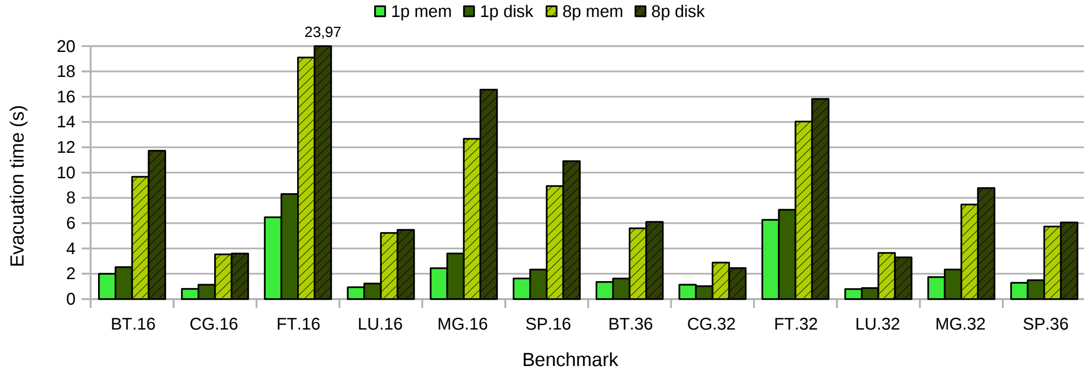


Figure 4.5: Evacuation time (in seconds) in Pluton N1.

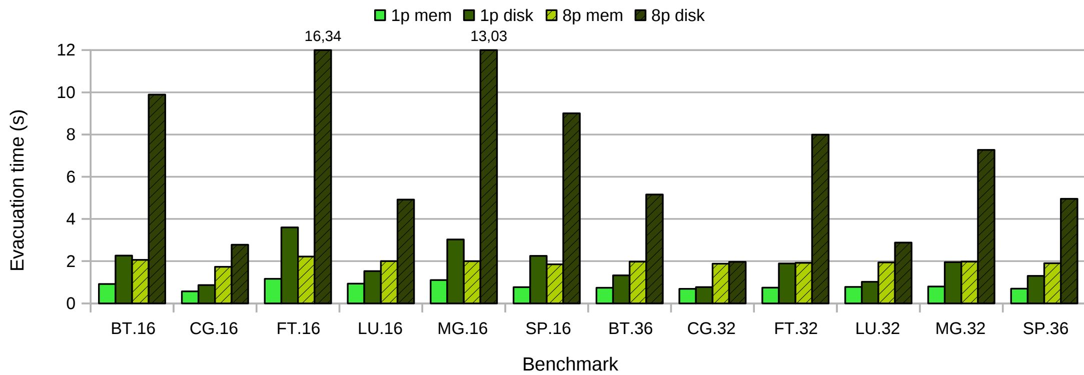


Figure 4.6: Evacuation time (in seconds) in Pluton N2.

Overhead

The total execution times of the NPBs were measured to analyze the overhead introduced by the migration solutions. Figures 4.7 and 4.8 show the original execution times (line labeled *Execution time* referenced to the right axis) and the overheads in percentages of the memory-based (*Memory* bars) and disk-based (*Disk* bars) proposals for 16 and 32/36 processes in Pluton N1 and Pluton N2 respectively and migrating 8 processes. Note that the highest overheads are due to small execution times and large checkpoint files. As expected, the in-memory migration approach is always faster than the migration via hard disk. The benefit is more pronounced for fast networks and applications with large checkpoint files. For instance, for the FT

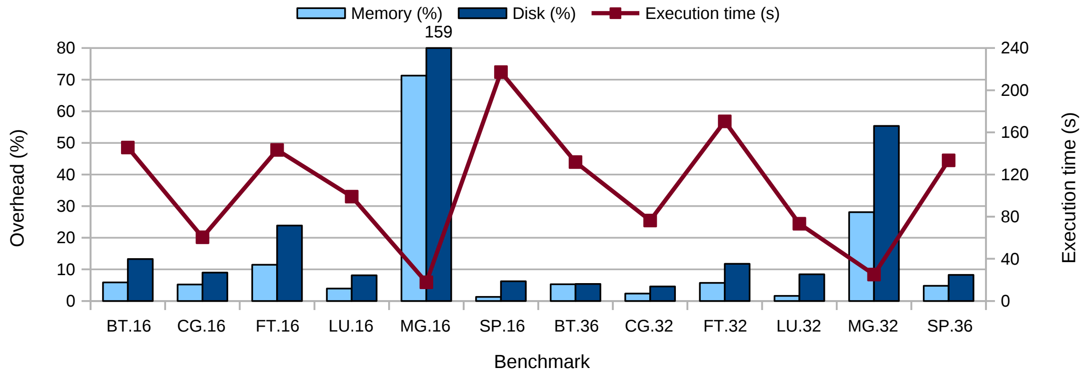


Figure 4.7: Overhead migrating 1 node (8 processes) in Pluton N1.

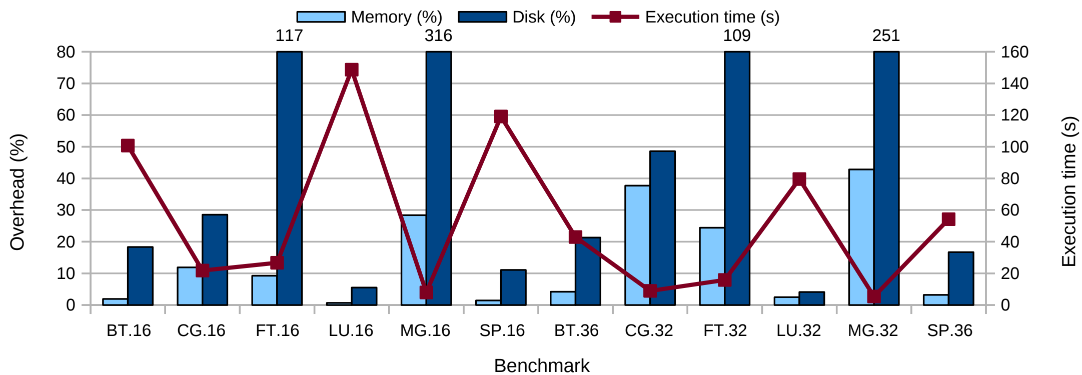


Figure 4.8: Overhead migrating 1 node (8 processes) in Pluton N2.

application in the Pluton N2 with 16 processes the overhead decreases from 31.26 s to only 2.47 s.

4.3. Splitting in-memory checkpoint files

Despite the reduction in the checkpoint file size obtained with the live variable analysis and the zero-block exclusion techniques analyzed in Chapter 2, and the in-memory optimization described in previous section, write/read of processes state continues to be the main cause of overhead in the migration operation.

In the proposal described in the previous section the writing of the state file, the

transfer of this file to the new process and the read and recovery of the saved state are executed in a sequential way. When the size of the checkpoint files is large, the time due to the serialized execution of these three phases may become unreasonable. To reduce this time, these three steps could be executed in a pipeline fashion [76]. To that end, the checkpoint files are split into multiple smaller files so that the new process can start with the read step while the original process continues to write other checkpoint fragments. In this way, the time that the new process has to wait to begin the restart operation is shortened, thus reducing the whole time of the migration operation. Moreover, less memory storage is required, since only those fragments that are being written and transferred in each step need to be temporarily stored.

The following subsections describe the structure of the checkpoint files generated by CPPC and how these files are split to make the pipeline operation possible.

4.3.1. Splitting the checkpoint files

In CPPC, checkpoint files are stored using HDF5 [85], a data format and associated library for the portable transfer of graphical and numerical data between computers. HDF5 files consist of two primary types of objects: groups and datasets. An HDF5 group is a container structure which can hold datasets and other groups. An HDF5 dataset is a multidimensional array of data elements. Both types support metadata. Any HDF5 group or dataset may have an associated attribute list, which is a user-defined HDF5 structure that provides extra information about an HDF5 object.

CPPC checkpoint files contain not only the relevant data needed to continue with the execution, but also all the context information needed for the correct restart of the application. They are hierarchically structured using HDF5 as depicted in Figure 4.9. Each state file is divided in two different parts: a metadata section and an application data section. The metadata section consists of three main parts, a dataset called Header and two groups: FileMap and Context. The dataset Header contains information about the checkpoint file, such as the compression type. The FileMap group records all the files opened during the execution. The Context group keeps track of execution context changes. Each context object represents a call

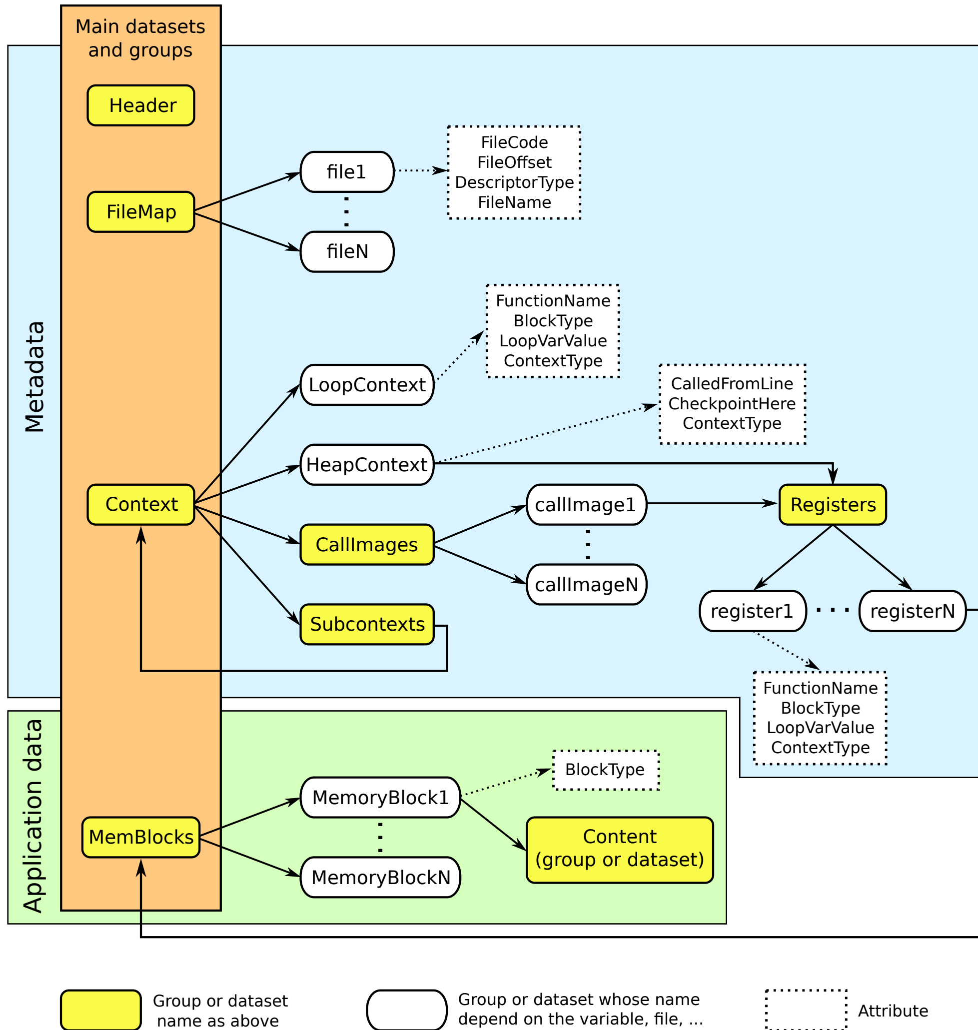


Figure 4.9: Structure of the CPPC checkpoint files

to a procedure, and contains the information required for recovering data in that procedure scope. Contexts can contain subcontexts, created by nested calls to the same or other procedures. This hierarchical representation allows for the sequence of procedure calls made by the original execution to be recreated upon restart. In this way, the application stack is rebuilt, and the relevant state is recovered inside its appropriate scope. Finally, the application data section contains the MemBlocks group, which includes the value of all the registered variables. This group contains a subgroup, henceforth referred to as MemoryBlock, for each variable. Each subgroup includes one or more datasets. The MemBlocks group is the largest portion of the

checkpoint file.

Considering the CPPC restart operation, the checkpoint files are split into two main parts. The first part includes three objects: Header, FileMap and Context. Once the new process receives this part, it can begin to rebuild the application stack and open the necessary files. Thus, this first part is transferred in a single fragment. Although the size of this fragment depends on the application, and, more specifically, on the context changes of the application, it is always negligible compared to the total checkpoint file size (inferior to 1% for all the applications used in this chapter).

The second part contains the object MemBlocks, which represents the bulk of the checkpoint file. This part, in turn, can be split in multiple chunks. The maximum size of each chunk can be specified into the CPPC configuration file. Its default value is set to 64 MB, that demonstrated a good cost/efficiency relation in the experimental tests. Attending to the maximum size of each chunk, the MemBlocks group will be split at a different structure level:

- *MemoryBlock level*, when the MemBlocks group is split into two or more files, each one containing one or more MemoryBlocks.
- *Dataset level*, if a MemoryBlock contains several datasets, it can be split by grouping one or more datasets in different files.
- *Element level*, when each dataset is divided into two or more chunks. In this level, when a dataset represents a block of zeros, it will never be split, since, thanks to the zero-blocks exclusion technique applied by CPPC, it will only contain one element.

When the splitting is done at dataset or element level, some extra information has to be added to allow the new process to restore the MemoryBlock accurately. This information is inserted in the checkpoint file using three new defined HDF5 attributes:

- `isNotFirstChunk`, associated with de group Content, indicates that this MemoryBlock has already been created. Therefore, during restart, the new process should search for it rather than create a new one.

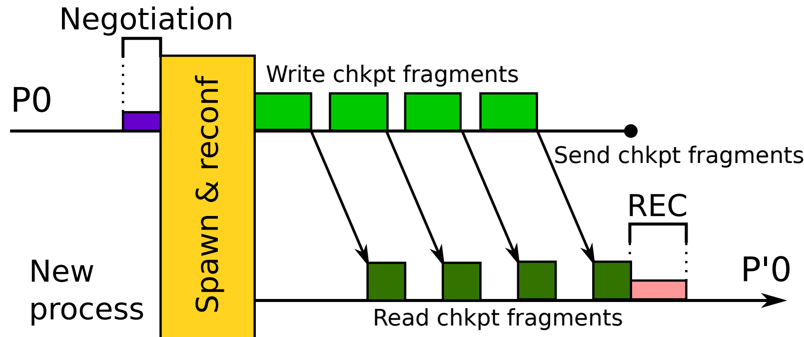


Figure 4.10: CPPC migration with pipelined write-transfer-read steps

- `FragmentTotalElements`, included only in the first fragment of a fragmented dataset, it indicates total number of items in the dataset and it is used by the new process to allocate memory for the complete dataset.
- `PositionOfFragment`, indicates the starting position of the elements in the dataset. This attribute is included in all dataset chunks except for the first.

4.3.2. Implementation issues

To implement the overlap in writing, transfer and read phases of the migration process, the CPPC migration operation seen in Section 4.2 (see Figure 4.1) is modified as shown in Figure 4.10. Negotiation phase remains unchanged. However, the spawn of the new process and the reconfiguration of communicators is brought forward at the beginning of the operation. In this way, the original process is able to send the different chunks, step by step, instead of sending the complete state file at the end.

Additionally, the CPPC writing layer (see Figure 4.2) is modified to split the MemoryBlocks complying with the chunk size specified in the CPPC configuration file and to add the HDF5 attributes commented in previous subsection. Now, every time a MemoryBlock is dumped, the writing function checks whether it should be split at dataset level, or even at element level. The new writing layer is depicted in Figure 4.11. The process state is stored chunk by chunk to memory using the HDF5 core driver. Then, a buffered copy of each chunk is sent through the network to the newly spawned processes using the MPI library. In each step, HDF5 starts to dump

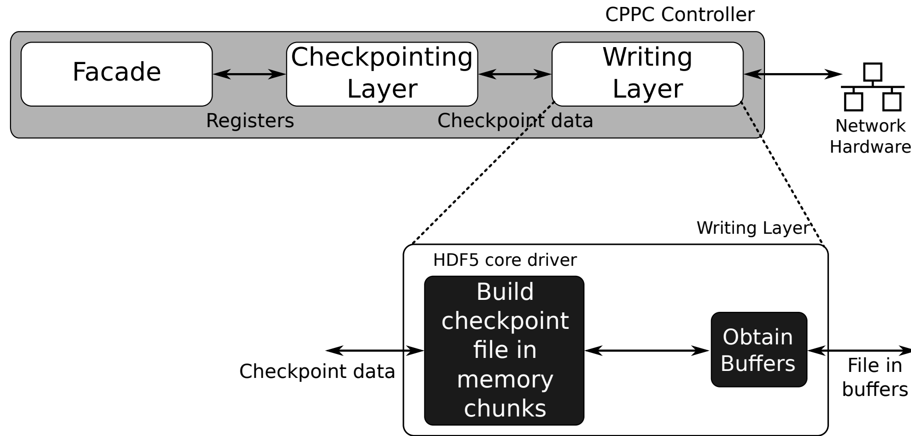


Figure 4.11: Modification of the CPPC writing layer

the next chunk at the same time that the chunk copied into the buffer is sent to the new process. Thus, the memory overhead is twice the chunk size.

The CPPC reading function is also modified. Now, whenever a chunk is read, it is necessary to check if the MemoryBlock has been split, and at which level. If it has been split, CPPC checks whether it is the first chunk or not. In the first case, space for the whole MemoryBlock will be reserved. In the second case, the position of that chunk in the MemoryBlock is stated.

After the checkpoint file is read, the memory addresses for the registered variables will be back-calculated to point to the addresses containing its data. Therefore, the memory used to store the transferred chunks will be the final memory used by the restored variables, and there will be no memory overhead in the new spawned process.

Finally, the blocking MPI communications used for the transfer of the checkpoint files in the previous version [19, 20] are replaced by non-blocking MPI calls so that they can be overlapped with writing and reading steps.

4.3.3. Experimental evaluation

This section explores the efficiency of the proposed solution in Pluton N2. The MPI implementation used was OpenMPI v1.6.4.

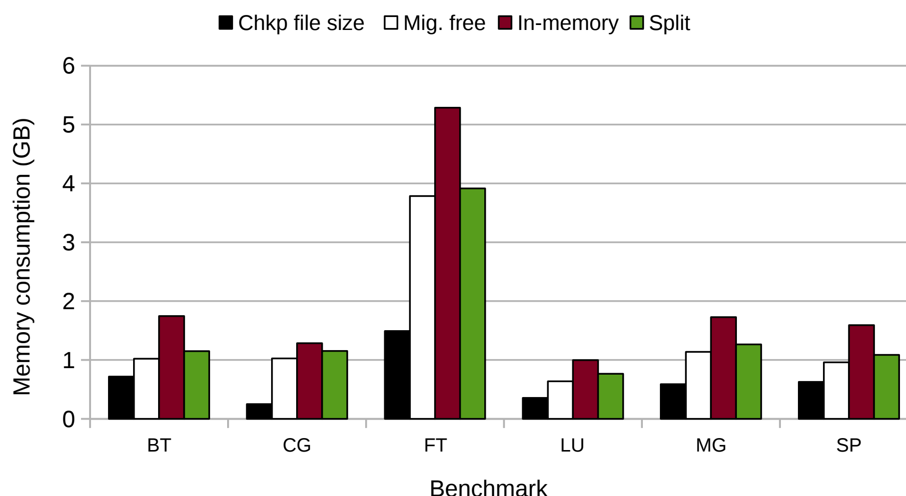


Figure 4.12: Checkpoint file size and memory consumption per process in a 32-process execution (in GB).

The application testbed is composed of the same six out of the eight applications in the NPBs shown in the previous section in this chapter. For all the executions, the benchmark size used in this section is class D.

All the experiments were carried out using 32 processes in 2 nodes (except BT and SP that need a square number of processes and, thus, they were executed with 36 processes using a third node). The experiments were focused on the performance evaluation both in terms of reduction in memory consumption and reduction in migration time with respect to the previous CPPC version. The experiments were designed so that, for each application, the migration were triggered always at the same point for all the executions.

Memory consumption

One of the most valuable goals associated with the proposed solution is that it requires considerably less memory space than the serialized classical version, which might be decisive to assure the feasibility of the migration.

Figure 4.12 shows the results of memory consumption per process for a 32-process execution of: an execution without migration (Mig. free); an execution performing

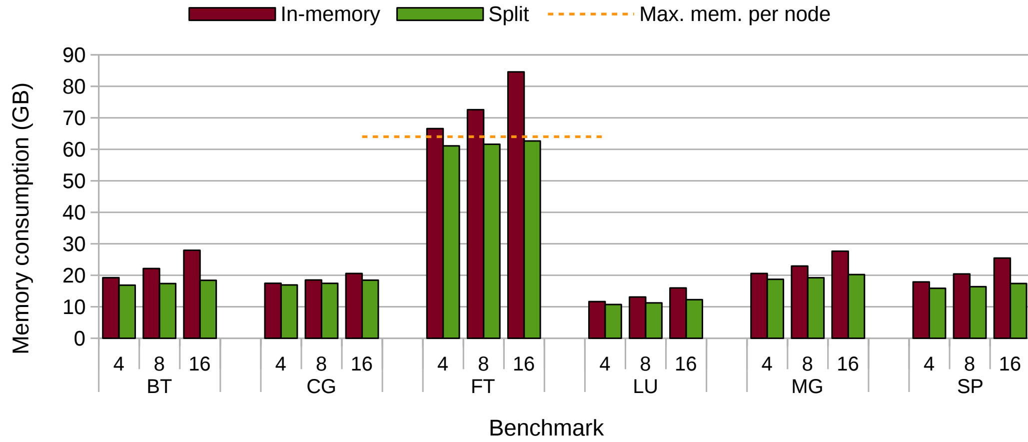


Figure 4.13: Memory consumption (in GB) per node (16 cores) in a 32-process execution when different number of the 16 processes per node are migrated.

one migration based on dumping the complete checkpoint file into memory before transferring it to the new spawned process (In-memory); and an execution performing one migration where the checkpoint file is split and the write, transfer and read phases are overlapped (Split). The increase in memory consumption during the migration is due to the memory requirements to save the state file. In the previous non-split version, the increase in the memory consumption is according to the size of the checkpoint file per process (shown also in this figure). When the splitting technique proposed in this paper is applied, only the checkpoint data that corresponds with the segment that is being written or read at that moment need to be stored. In the experiments shown in this figure a chunk size of 64 MB (the default value) has been used. In this case, the actual memory requirements are double that of the segment size, since, as commented in Section 4.3.2, while a copy of the written segment is being transferred, another segment begins to be dumped in the HDF5 buffer.

Note that the increase in memory consumption only arises in those processes that are being migrated. Thus, the memory consumption per computational node depends on the number of migrating processes in each node. Figure 4.13 shows the memory consumption per node when different number of processes are migrated. The system used has 64 GB of RAM per node which means that FT (class D) application cannot be migrated using in-memory storage unless the splitting technique

is employed. Therefore, the savings in memory consumption is particularly relevant because, as illustrated in next subsection, if the size of the complete checkpoint file is too large to fit into memory, the pipeline solution allows for avoiding the use of disk storage that would represent a significant bottleneck in the migration operation.

Migration time

The migration time is the execution time between the migration request and the completion of the restart operation in the new process. For the previous version, the migration time is broken down into 5 parts (see Figure 4.1):

- *Negotiation*: execution time between the mpirun migration request and the arrival to the migration point.
- *WriteCkpt*: time required for the checkpoint writing in memory.
- *Spawn*: execution time of the spawn function and the reconfiguration of the communicators.
- *TransferRead*: time required for the transfer of the checkpoint files and the read of these files in the newly spawned processes.
- *Restart*: time required for the restart of the application once the checkpoint files have been read. It includes the execution of the RECs, as explained in Section 1.2.

Figure 4.14 shows the migration time in the previous CPPC approach in a 32-process execution when all the processes of a node (16 processes) are migrated. The breakdown of the migration time shown in the figure corresponds to the slowest migrating process.

The contributions having the most impact on the migration overhead are: *Write-Ckpt*, and *TransferRead*, despite the fact that checkpoint files are stored in memory. Note that the migration using the original approach in FT cannot be performed *in-memory* (see Figure 4.13), and disk storage is used instead, thus, significantly increasing the *WriteCkpt* and *TransferRead* times.

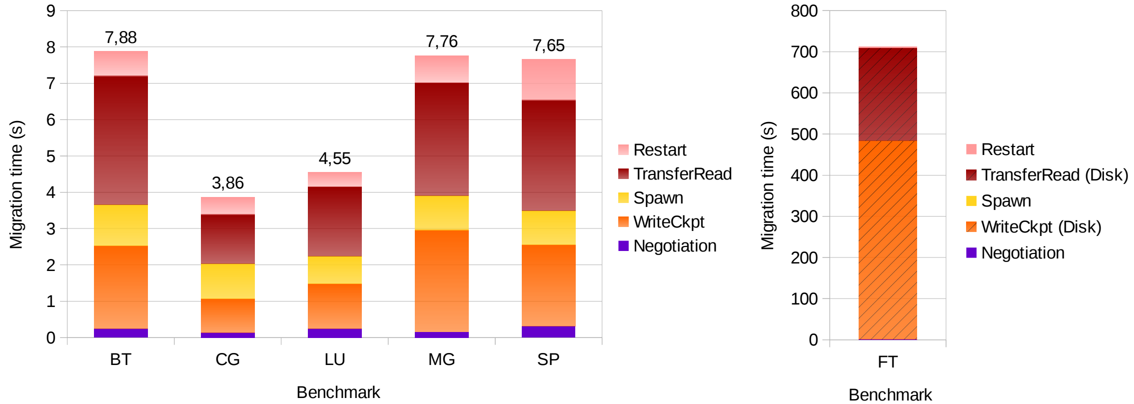


Figure 4.14: Migration time in the in-memory CPPC version in a 32-process execution when 16 processes are migrated.

In the new proposal *WriteCkpt* is overlapped with *TransferRead* in order to reduce the migration times and it will be labeled as *WriteTransferRead* in the following figures. It corresponds to the time between the end of the reconfiguration phase in the newly spawned processes and the actual restart (see Figure 4.10). Note that the maximum improvement that can be achieved with the pipeline proposal is limited by the minimum between the *WriteCkpt* time and the *TransferRead* time in the in-memory one.

Subsequent subsections show the improvements obtained in the write-transfer-read times with the split approach in function of: (a) the number of processes that migrate from one node, and (b) the chunk size considered.

(a) Impact of the number of migrating processes

Note that the most realistic situation would be that in which several processes in one node have to be migrated (if the node is overloaded), or even the whole node needs to be migrated (if the node is about to fail). Figure 4.15 shows the results for each benchmark using 32 processes and migrating 4, 8 and 16 processes from the same node. The times shown in the figure correspond to the slowest migrating process.

The amount of data to be dumped into the node memory and to be transferred through the network increases with the number of migrating processes. Therefore,

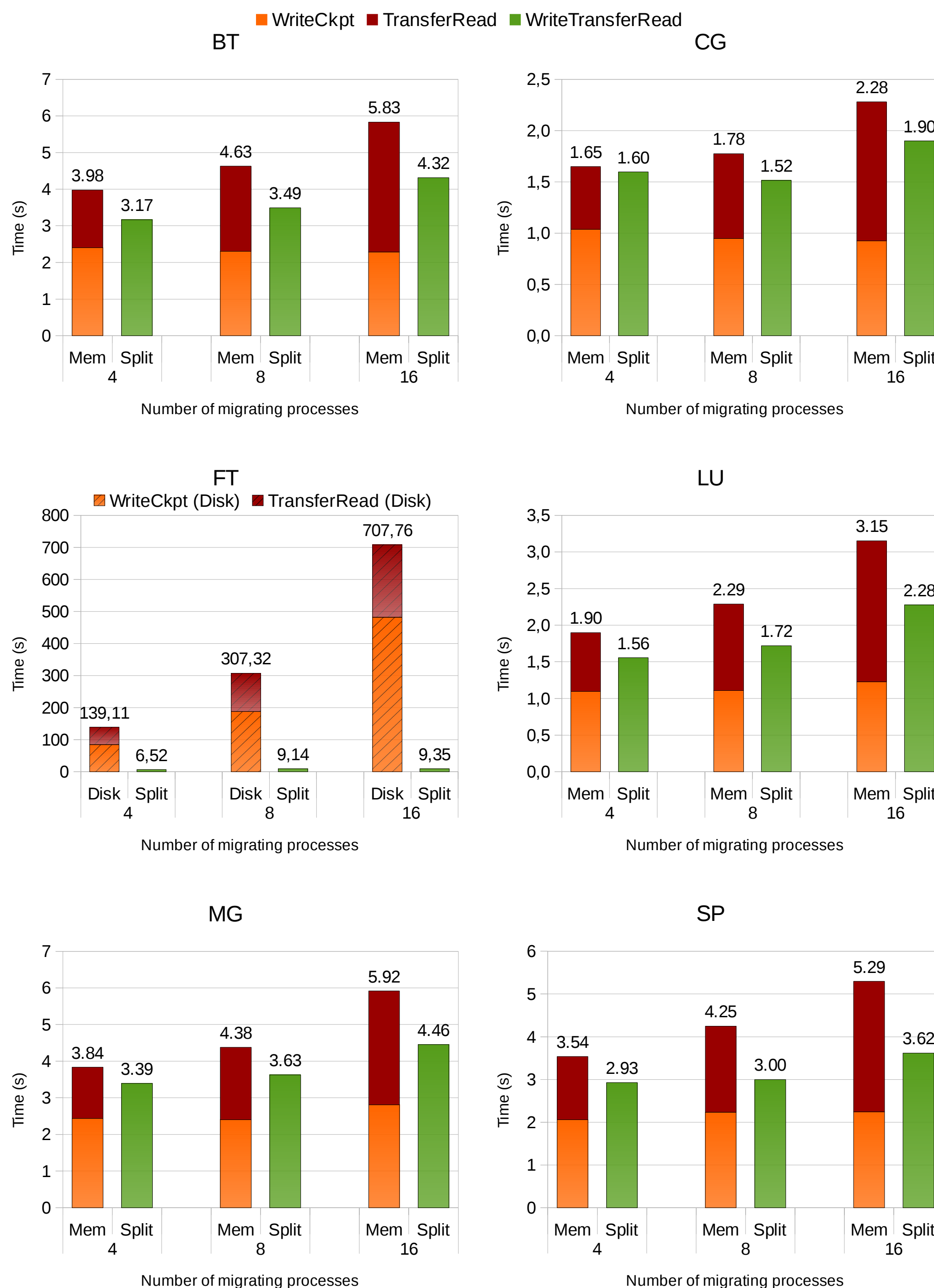


Figure 4.15: Times of the write-transfer-read steps in the in-memory and the split CPPC versions in a 32-process execution when 4, 8 and 16 processes are migrated.

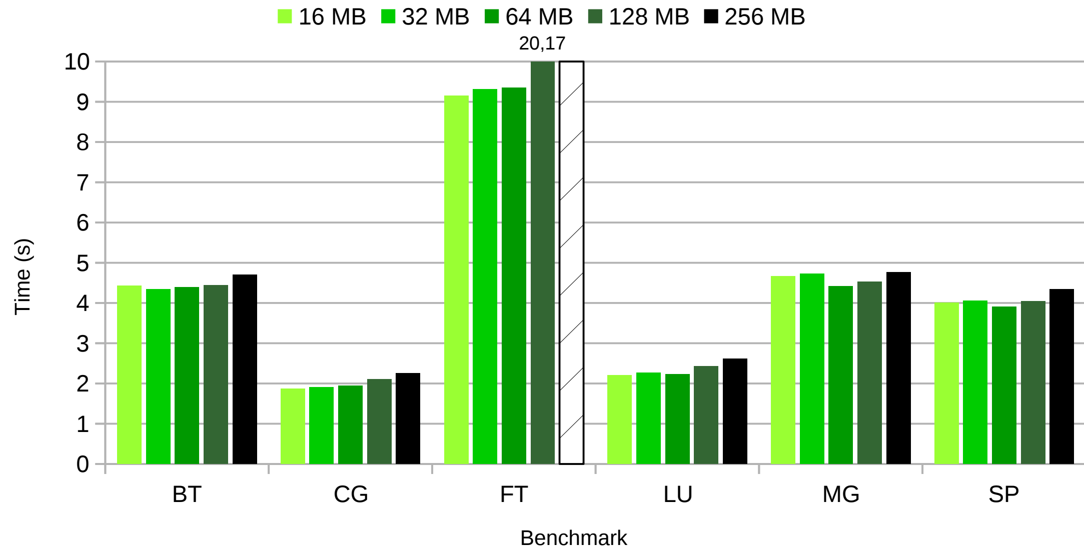


Figure 4.16: Times of the *WriteTransferRead* step of the split CPPC version for different chunk sizes.

the overhead of the *WriteCkpt* and *TransferRead* phases in the previous approach, and the *WriteTransferRead* step in the split solution increases when the number of migrating processes per nodes grows.

The new approach outperforms the previous one in all the cases. The overall improvement achieved with the split approach becomes bigger when the number of migrating processes per node grows, since the contribution of the *WriteCkpt* and *TransferRead* phases in the in-memory approach also increases. Particularly significant is the case of FT, because in the previous version the checkpoint files have to be written to disk due to insufficient memory.

(b) Impact of the chunk size

The impact of the chunk size in the *WriteTransferRead* time is analyzed in this section. Figure 4.16 shows the *WriteTransferRead* times for each benchmark running 32 processes and migrating 16 processes from the same node, for chunk sizes from 16 to 256 MB. The times for FT with a chunk size of 256 MB are not shown as there is not enough memory to store that chunk size.

The biggest improvement have been obtained mainly using intermediate chunk sizes, being 64 MB (the default value) the optimal value in most of the experiments. For smaller chunks the migration time is higher because more messages are needed, and, thus, the transfer overhead is bigger. On the other hand, when the chunks are larger, the overlap between writing and reading is smaller, since the writing of the first chunk and the reading of the last one cannot be overlapped. Regardless, the sum of the *WriteCkpt* and the *TransferRead* in the previous CPPC approach is always larger than the *WriteTransferRead* times of the pipeline version for all the chunk sizes considered. Thus, the new proposal reduces the write-transfer-read times in all the cases.

4.4. Related work

A significant part of the overhead in a checkpoint-based migration approach is the long time required to write and read the checkpoint files. Techniques to reduce the checkpoint file size present in the literature, such as data compression of the checkpoint files [51, 69], or memory exclusion [68], could be used to improve migration solutions.

Another way to optimize the I/O cost of migration based on checkpointing is to overlap the I/O operations. In [66] a pipelined process migration with RDMA is presented. The proposed protocol pipelines checkpoint writing, and checkpoint transfer and read using data streaming through RDMA transport.

Other recent solutions focus on the use of non-volatile memory technology, such as solid-state disks (SSDs) [52], to store checkpoint data. SSDs offer excellent read-/write throughput when compared to secondary storage and thus they can help reduce disk I/O load.

4.5. Concluding remarks

The main drawback of checkpoint-based migration is its high overhead, in terms of I/O cost. To overcome this issue this chapter proposes two optimizations: transfer

HDF5 checkpoint files directly to the remote memory without storing them to stable storage; and split the checkpoint files into multiple smaller files, in order to overlap the writing of the state in the terminating processes with the read and restarting operation in the newly spawned processes.

The in-memory approach moves the bottleneck of the system to the network bandwidth, reducing, in all cases, the overhead and the evacuation time. The obtained benefits are more significant for fast networks. The main issue concerning the in-memory migration is the high overheads in terms of memory consumption.

Experimental results also prove the efficiency of the splitting solution, both in terms of reduction in memory consumption and I/O migration times. The solution proposed conserves the hierarchy of checkpoint files used in the original version, preserving the most important feature of the CPPC framework, the portability.

The reduction in migration time is always worthwhile, but it becomes of particular importance in solutions where the migration of processes are used to prevent application failures proactively, i.e., the processes are migrated away from those nodes that are about to fail. Also, the reduction in memory consumption, when the application presents very large checkpoint files, allows for avoiding the use of disk storage that would inflict a significant penalty in the migration time. Proposals like this one, that aims to reduce the overhead of the migration operation, can make a difference when determining whether the migration operation is viable or not.

Chapter 5

Checkpoint-based virtual malleability

This chapter presents a proposal, based on checkpointing, to automatically transform MPI applications into malleable applications, that is, parallel programs that are able to adapt their execution to the number of available processors at runtime. The proposal includes a mapping algorithm to reschedule processes on available nodes.

5.1. Introduction

The resources availability of large-scale distributed systems may vary during a job execution, making malleable applications specially appealing. Malleable jobs provide important advantages for the end users and the whole system, such as higher productivity and a better response time [7, 41], or a greater resilience to node failures [30]. These characteristics will allow improving the use of resources, which will have a direct effect on the energy comsuption required for the execution of applications, resulting in both cost savings and greener computing.

Most MPI applications follow the SPMD programming model and they are executed in HPC systems by specifying a fixed number of processes running over a fixed number of processors. The resource allocation is statically specified during job sub-

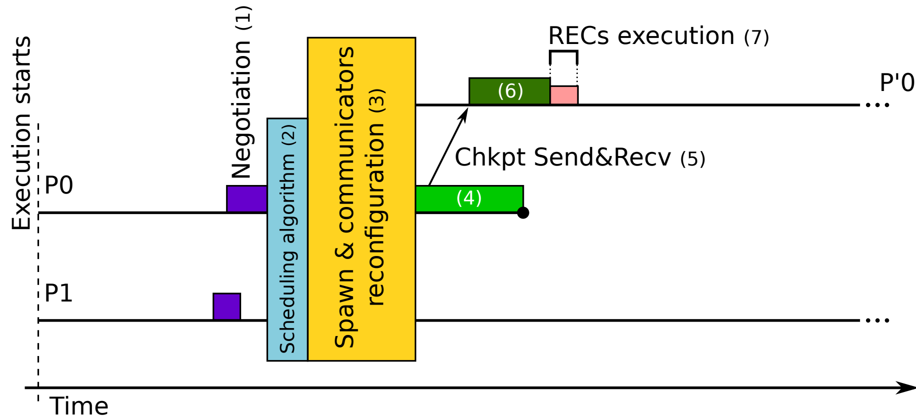


Figure 5.1: Steps in the reconfiguration operation: (1) Negotiation to select a single safe location to trigger the reconfiguration; (2) Scheduling algorithm to decide the processes to be migrated and the new allocations; (3) Spawning of new processes and reconfiguration of the communicators; (4) Checkpointing of the migrating processes; (5) Sending of the checkpoint files; (6) Recovering the state from the checkpoint files; and (7) Execution of the RECs. Steps (4) to (6) are partially overlapped.

mission, and maintained constant during the entire execution. Thus, applications are unable to dynamically adapt to changes in resource availability.

The objective of this chapter is to transparently transform MPI applications into malleable jobs that are capable of adapting their executions, without user or system interaction, to changes in the environment. The proposed solution is based on process migration [16]. If a node becomes unavailable, the processes on that node will be migrated to other available ones, overloading nodes when necessary. To achieve this aim the application should: (a) be aware of the changes and trigger the reconfiguration operation; (b) autonomously decide which processes should be migrated and their new appropriate node allocations; and (c) migrate processes.

The main steps of the reconfiguration process are depicted in Figure 5.1. The following sections describe the main new components of the proposed solution: the triggering of the reconfiguration operation; and the scheduling algorithm implemented to allow the application to decide which processes should be moved and to which target nodes. The migration operation will be performed using the pipeline in-memory approach described in Chapter 4.

5.2. Triggering the reconfiguration operation

The reconfiguration operation will be triggered when a change in the availability of the resources occurs. This proposal relies on a monitoring system that provides dynamical information about the available resources. The reasons why a node becomes available/unavailable are varied. For instance, malleable jobs could be used in a fault tolerance context to preemptively migrate processes from processors that are about to fail, or in a non-dedicated environment to release nodes to be used for a higher priority user. The monitoring information could be provided to the running application through different methods. There are in the literature many proposals for different environments and objectives [4, 56, 67, 72, 79]. For this work we assume that an *availability file* is set up for each malleable MPI job. This file contains the names of all the nodes that are likely to execute the MPI job together with their number of available cores. The format of this file may be as simple as `hostname:numCores`. If a node becomes unavailable, its number of cores will be set to zero.

The MPI application will be aware of changes in the availability of the resources through a periodical polling to the availability file. A change in this file activates a flag in the CPPC controller to change to migration mode and to start the reconfiguration of the execution. The same negotiation protocol explained in Section 3.2.1 is used to select a single safe location to trigger the reconfiguration protocol.

5.3. Scheduling algorithm

Once the MPI processes reach the negotiated reconfiguration location, and previous to the start of the migration operation, the processes to be migrated and their mapping to the available resources need to be selected.

5.3.1. Monitoring communications

To be able to migrate the process efficiently, it is important to have a precise knowledge of the application behavior. In particular, as will be shown in the

next section, the *affinity* between the processes needs to be known so as to map those with a high communication rate as close as possible. To this end, a dynamic monitoring component included in OpenMPI is used[1]. It is integrated in an MCA (Modular Component Architecture) framework called pml (point-to-point management layer). This component, if activated at launch time (through the `mpiexec` option `--mca pml_monitoring_enable`), monitors all the communications at the lower level of OpenMPI (i.e. once collective communications have been decomposed into send/recv communications). Therefore, contrary to the MPI standard profiling interface (`PMPI`) approach where the MPI calls are intercepted, the actual point-to-point communications that are issued by OpenMPI are monitored, which is much more precise.

Internally, this component uses low-level process id and creates an associative array to convert sender and receiver ids into ranks in `MPI_COMM_WORLD`. Therefore, it is oblivious to communicator splitting and merging. Each time a message is sent, the sending process increment two arrays: the number of messages and the size (in bytes) sent to the receiver. Moreover, there are two temporary arrays (one for number of messages and one for communication size) used to monitor communication between two specified points in the application. Therefore, the memory overhead of this component is 4 arrays of N x 64 bits items, where N is the number of MPI processes. Several callback functions are made available at the application level to gather the internal state of the monitoring. Therefore, at any moment, it is possible to know the amount of data exchanged between two pairs of processes since either the beginning or a given point of the application. It is also possible to gather all the local view onto a given process to get the full communication matrix.

5.3.2. Mapping processes to cores

To optimize the mapping of processes to cores, TreeMatch [46] and Hwloc [6] are used. TreeMatch is an algorithm that obtains the optimized process placement based on the communication pattern among the processes and the hardware topology of the underlying computing system. It tries to minimize the communications at all

[1]This monitoring component was developed by the Runtime team in the INRIA Bordeaux research centre. It is under review by the OpenMPI consortium and is planed to be released this year. A prototype is available on the OpenMPI github platform.

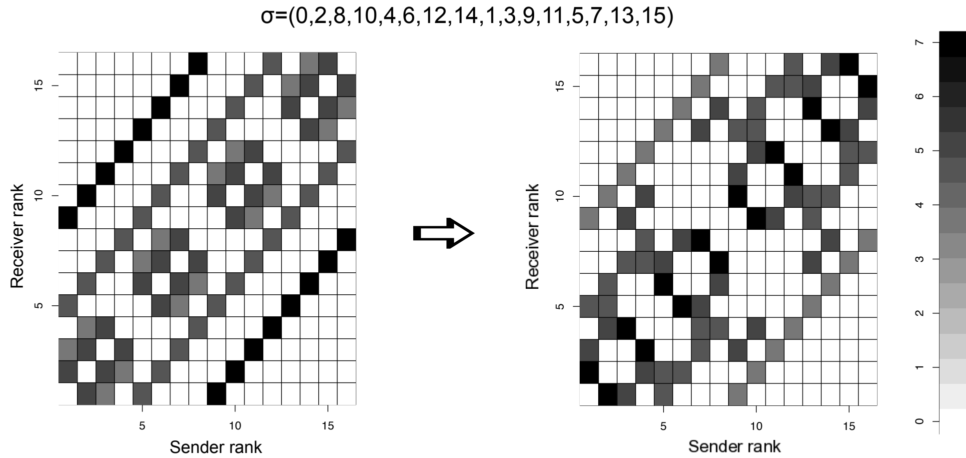


Figure 5.2: TreeMatch example. Figure provided by Emmanuel Jeannot from the Runtime team in the INRIA Bordeaux research centre.

levels, including network, memory and cache hierarchy. It takes as input both a matrix modeling the communications among the processes, and a representation of the topology of the system. CPPC obtains the matrix dynamically just before the scheduling algorithm is triggered using the monitoring component explained in the previous section. The topological information is provided by Hwloc (represented as a tree) and it is also obtained dynamically during application execution. TreeMatch returns as output an array with the core ID that should be assigned to each process.

An example of the output of TreeMatch is given in Figure 5.2. On the left, a communication matrix representing the affinity between processes is given as input. The darker the dot the higher the communication volume and hence the affinity. TreeMatch computes the permutation of the processes such that the cores with high affinity are mapped close together on the tree representing the target topology. In this example, the permutation *sigma* says that process 0 is mapped on core 0, process 1 on core 2, 3 on 8, . . . , 9 on 3, etc. Thanks to this new mapping, processes 1 and 9, with a high communication rate, are mapped very close to each other. This can be seen on the right when the permutation *sigma* is applied to the communication matrix.

An interesting feature of TreeMatch is that the topology given as an input can be a real machine topology or a virtual topology designed to separate groups of processes into clusters such that communication within clusters are maximized while

communication outside the clusters are minimized.

TreeMatch focuses on minimizing the communication cost in a parallel execution. Thus, if TreeMatch is directly applied to find the processes mapping during a reconfiguration phase, it could lead to a complete replacement of all the application processes. This would involve unnecessary process migrations and, thus, unnecessary overheads. To avoid this behavior, a two-step mapping algorithm was designed. The first step decides the number and the specific processes to be migrated. The second step finds the best target nodes and cores to place these processes.

- **Step1: identify processes to migrate**. A process should be migrated either because it is running in nodes that are going to become unavailable, or because it is running in oversubscribed nodes and new resources have become available. To know the number of processes that need to be migrated all processes exchange, via MPI communications, the node and core in which they are currently running. Then, using this information, each process calculates the current computational load of each node listed in the availability file associated to the application. A *load* array is computed, where $load(i)$ is the number of processes that are being executed in node n_i. Besides, each process also calculates the maximum number of processes that could be allocated to each node n_i in the new configuration:

$$maxProcs(i) = \left\lceil nCores(i) \times \frac{N}{nTotalCores} \right\rceil$$

  where $nCores(i)$ is the number of available cores of node n_i, N is the number of processes of the MPI application, and $nTotalCores$ is the number of total available cores. If $load(i) > maxProcs(i)$ then $load(i) - maxProcs(i)$ processes have to be migrated. If the node is no longer available, $maxProcs(i)$ will be equal to zero and all the processes running in that node will be identified as migrating processes. Otherwise, TreeMatch is used to identify the migrating processes. The aim is to maintain in each node the most related processes according to the application communication pattern. Figure 5.3 illustrates an example with two 16-core nodes executing a 56-process application in an oversubscribed scenario. When two new nodes become available, 12 processes per node should be migrated to the new resources. To find the migrating pro-

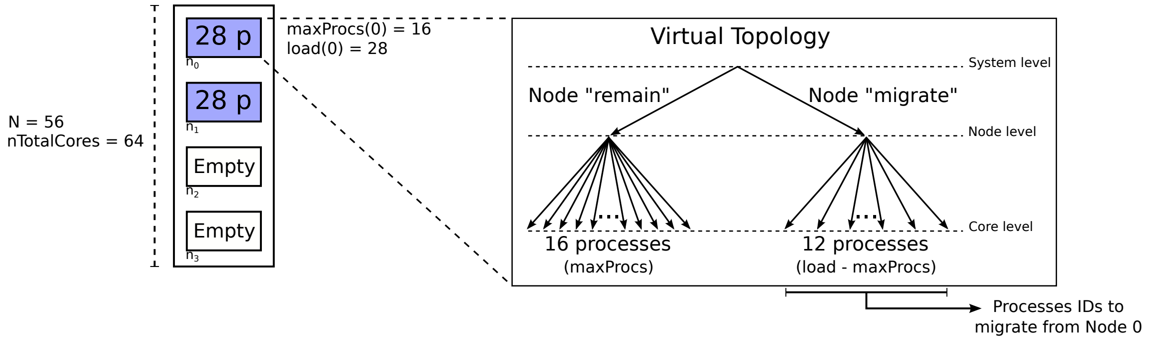


Figure 5.3: Step 1: identifying processes to be migrated. Virtual topology built to migrate 12 processes from a 16-core node where 28 processes are running (16 processes remain and 12 processes migrate).

cesses in each oversubscribed node, a virtual topology that models two nodes is built: one simulates the node for non-migrating processes, with $maxProcs(i)$ cores, and the other one simulates a target node for migrating processes, with $load(i) - maxProcs(i)$ cores. TreeMatch uses this virtual topology, and a sub-matrix with the communication pattern between the processes involved, to identify the processes to be migrated, that is, those mapped to the second virtual node.

- **Step 2: identify target nodes**. Once the processes to be migrated are identified, CPPC has to find the target nodes (and the target cores inside the target nodes) to place these processes. To find the best placement for each migrating process, CPPC relies again on TreeMatch. It uses a sub-matrix with the communication pattern of the migrating processes, and a virtual topology built from the real topology of the system but restricted to use only the potential target nodes in the cluster. The potential targets are those nodes that satisfy $load(i) < maxProcs(i)$. They can be empty nodes, nodes already in use but with free cores, or nodes that need to be oversubscribed. Since TreeMatch only allows the mapping of one process per core, if there are no sufficient real target cores to allocate the migrating processes, CPPC will build the virtual topology simulating $maxProcs(i) - load(i)$ extra cores in the nodes that need to be oversubscribed. Figure 5.4 illustrates the second step of the algorithm for the same example of Figure 5.3. In this example, the virtual topology used consists of the new available nodes in the system, two 16-core

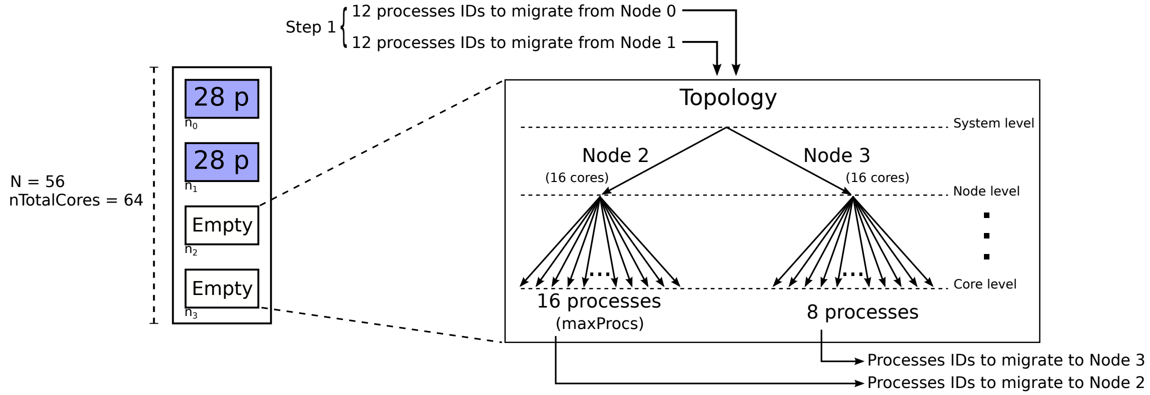


Figure 5.4: Step 2: identifying target nodes. Topology built to map the migrating processes selected in step 1 to the empty cores in the system.

nodes to map the 24 processes. After executing TreeMatch, CPPC knows the target cores and therefore, the target nodes for the migrating processes obtained in the step 1.

Once the mapping of migrated processes to available resources is decided, the migration operation can start. The processes to be migrated write their state to checkpoint files and terminate their execution, while newly spawned processes read these files and recover the state of the terminating ones. To minimize the overhead associated to the I/O operations needed for the migration, the pipeline in-memory technique seen in Chapter 4 is used. Note that initially, the newly spawned processes are not bound to any specific core. The TreeMatch assignment is sent to the new processes together with the checkpoint file and CPPC performs the binding via the Hwloc library.

5.4. Experimental evaluation

This section aims to show the feasibility of the proposal and to evaluate the cost of the reconfiguration whenever a change in the resource availability occurs. Pluton N2 cluster described in Section 2.6 was used to carry out these experiments.

The application testbed is composed of six out of the eight applications in the NPB. For all the executions the benchmark size used was class C. The Himeno

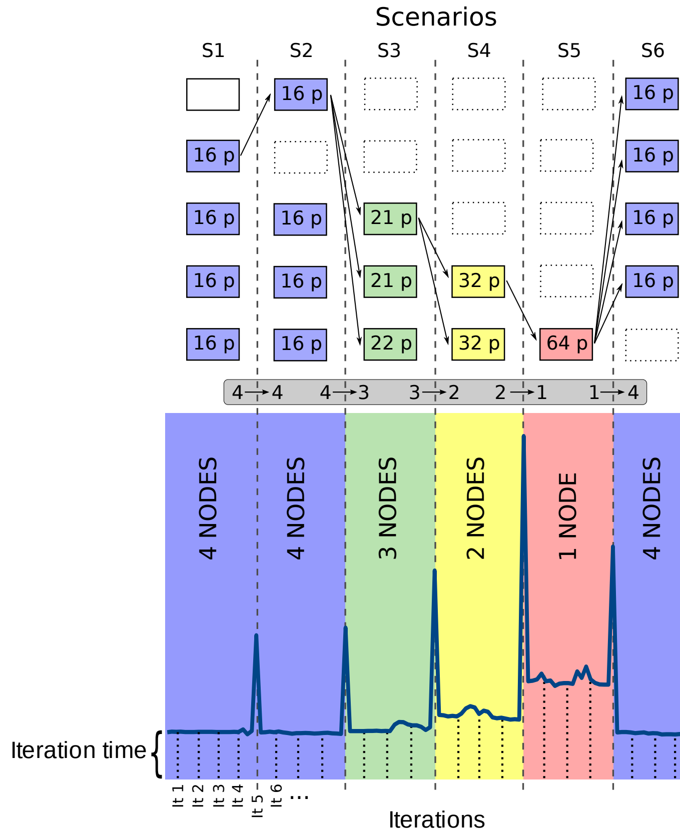


Figure 5.5: Testbed scenarios

benchmark [45] was also tested. Himeno uses the Jacobi iteration method to solve the Poissons's equation, evaluating the performance of incompressible fluid analysis code, being a benchmark closer to real applications.

The MPI implementation used was OpenMPI v1.8.1 [64]. The `mpirun` environment has been tuned using MCA parameters to allow the reconfiguration of the MPI jobs. Specifically, the parameter `orte_allowed_exit_without_sync` has been set to allow some processes to continue their execution when other processes have finished their execution safely. The parameter `mpi_yield_when_idle` was also set to force degraded execution mode and, thus, to allow progress in an oversubscribed scenario.

To evaluate the feasibility of the proposed solution and its performance, different scenarios have been forced during the execution of the applications. Figure 5.5 illustrates these scenarios and displays the representation of the iteration time over the timeline. The applications were initially launched in a 64-process configuration

running on 4 available nodes of the cluster (16 cores per node). Then, after a time, one of the nodes becomes unavailable. In this scenario, the 16 processes running on the first node should be moved to the empty node, and the application execution continues in a 4 node configuration. After a while, the 4 nodes where the application is running start to become unavailable sequentially, first one, then another, without spare available nodes to replace them, until only one node is available and the 64 processes are running on it. Finally, in a single step, the last node fails but 4 nodes become available again, and the processes are migrated to return again to using 4 nodes. In all benchmarks the CPPC compiler automatically detects a safe point at the beginning of the iteration of the main loop of the application. Thus, this will be the point where reconfiguration will take place. To demonstrate the feasibility of the solution, the iteration time was measured across the execution in those scenarios. Measuring iteration time allows us to have a global vision on the instantaneous performance impact.

Figure 5.6 shows the benchmarks results in the scenarios illustrated in Figure 5.5. These results demonstrate that, using the proposed solution, the applications are capable of adjusting the execution to changes in the environment. The high peaks in these figures correspond to reconfiguration points. As shown in Figure 5.1, the iteration time when a reconfiguration is performed can be broken down into the following stages:

- *Negotiation*: execution time of the negotiation protocol used to reach consensus on the reconfiguration point.
- *Scheduling*: execution time of the scheduling algorithm to identify processes to be moved and target nodes.
- *Spawn&Rec*: execution time of the spawn function and the reconfiguration of the communicators.
- *ChkptTransf&Recv*: average time to write the checkpoint files in the terminating processes, transfer them to target nodes, and read them in newly spawned processes.
- *Restart*: average time to complete the restart of the application once the checkpoint files have been read. It includes the execution of RECs.

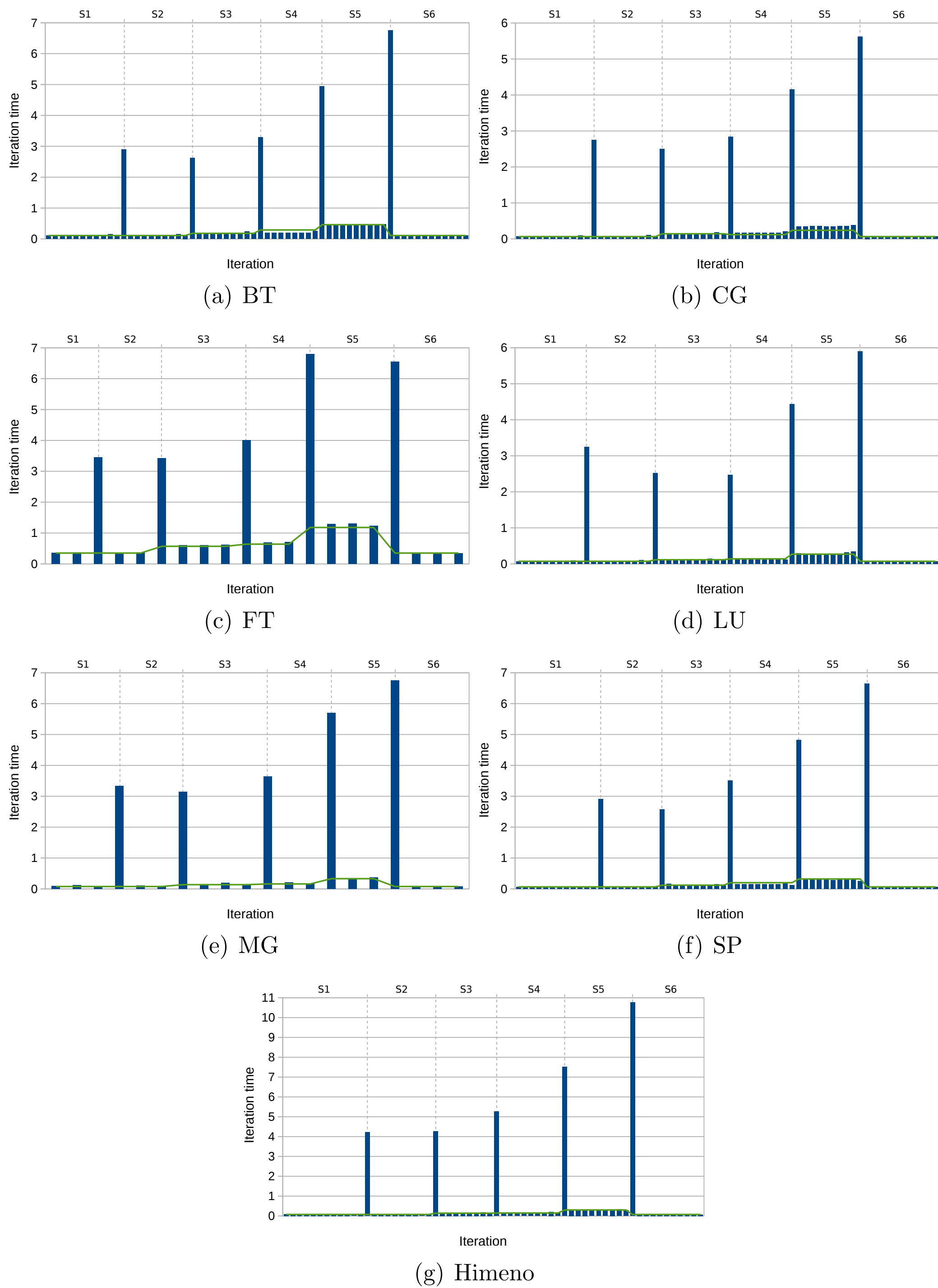
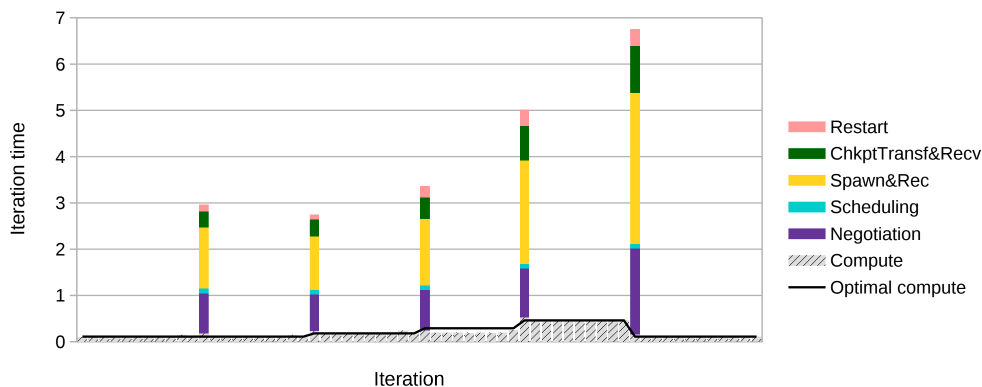
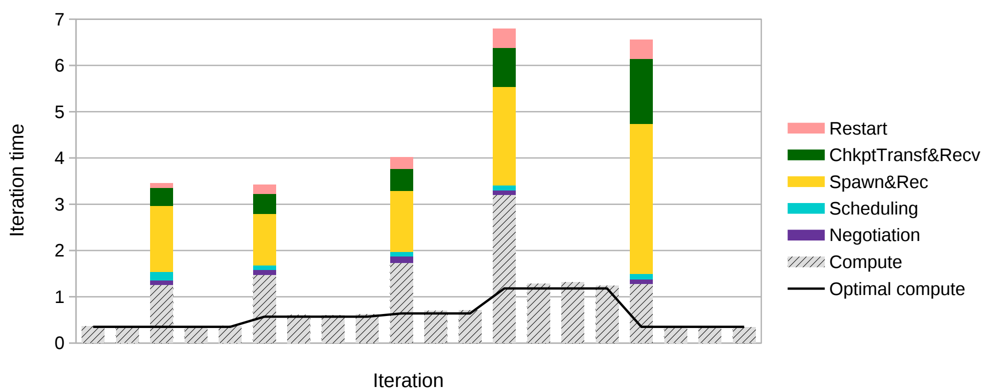


Figure 5.6: Iteration execution times in the scenarios illustrated in Figure 5.5



(a) BT



(b) FT

Figure 5.7: Detailed iteration times in BT and FT benchmarks. Note the variation in the *Negotiation* times.

- *Compute*: the computational time of the iteration where the reconfiguration takes place.

Figure 5.7 illustrates detailed iteration times for BT and FT benchmarks, breaking down the overhead of the iterations involved in a reconfiguration operation into the aforementioned 6 parts. For comparison purposes, the compute times that would be attained if the application could adjust its granularity to the available resources[2], instead of oversubscribing them without modifying the original number of processes

[2]This time is measured executing the application with different number of processes depending on the hardware available (16 processes version when only 1 node is available, 32 processes version when 2 nodes are available, etc.)

Table 5.1: Execution time (s) of the reconfiguration phases.

		NPB applications						
	scenarios	*BT*	*CG*	*FT*	*LU*	*MG*	*SP*	*Himeno*
Negotiation	4→4	0.88	1.18	0.01	1.08	0.93	0.94	1.12
	4→3	0.80	0.85	0.01	0.86	0.94	0.81	0.90
	3→2	0.88	0.89	0.14	0.53	0.94	0.86	0.95
	2→1	1.06	1.11	0.02	1.15	1.07	1.04	1.09
	1→4	1.86	1.94	0.02	1.90	1.88	1.87	1.95
Spawn&Rec	4→4	1.32	1.18	1.44	1.61	1.28	1.29	1.13
	4→3	1.15	1.18	1.11	1.08	1.09	1.14	0.99
	3→2	1.43	1.43	1.32	1.31	1.35	1.59	1.48
	2→1	2.23	2.14	2.14	2.17	2.38	2.07	2.12
	1→4	3.27	3.19	3.25	3.28	3.20	3.22	3.23
ChkptTransf &Recv	4→4	0.35	0.10	0.39	0.16	0.44	0.39	1.44
	4→3	0.37	0.11	0.43	0.18	0.42	0.40	1.59
	3→2	0.48	0.14	0.47	0.21	0.54	0.54	1.96
	2→1	0.75	0.23	0.84	0.36	0.84	0.93	2.92
	1→4	1.01	0.28	1.41	0.47	1.41	1.06	4.96
Restart	4→4	0.15	0.01	0.04	0.01	0.04	0.12	0.25
	4→3	0.05	0.01	0.20	0.02	0.03	0.08	0.46
	3→2	0.24	0.01	0.26	0.02	0.03	0.30	0.47
	2→1	0.36	0.02	0.41	0.02	0.06	0.30	0.57
	1→4	0.36	0.01	0.41	0.01	0.03	0.29	0.45

are also shown (black line). The overhead that would introduce the data distribution needed to adjust the application granularity is not shown in the figure. The computational time of the iteration involved in the reconfiguration is higher than the computational time of the rest of the iterations. This is in part due to the cache misses caused by the data movement, and in part due to an overhead observed in the first MPI communication after the communicators are reconfigured. The largest contribution to the reconfiguration overhead is due to the *Spawn&Rec* step. Table 5.1 details the main impacting steps in the reconfiguration overhead for all the NPB applications.

The *Negotiation* phase depends on the application as in this phase MPI one-sided communications are used and the progress of these remote operations is affected by the MPI calls inside the application. These times could be lower using other MPI

Table 5.2: Scalability of the *Spawn&Rec* step vs total number of processes in the application.

	Number of total processes			
NPB	*16*	*32/36*	*64*	*128/121*
BT	0.97	0.99	1.32	1.57
CG	0.98	1.01	1.18	1.79
FT	0.96	1.07	1.44	1.75
LU	1.00	1.01	1.61	1.89
MG	1.02	1.01	1.28	1.63
SP	0.99	1.00	1.29	1.72
Himeno	0.99	1.01	1.13	1.96

implementations and/or computer architecture [18].

The time spent in the spawn function depends on the number of spawned processes and the degree of oversubscription. The more processes to be migrated, the larger the overhead of this phase. This can be observed comparing the overhead associated to the reconfiguration from scenario 1 to scenario 2 ($4 \rightarrow 4$), where 16 processes are moved to an empty target node, and the overhead associated with the reconfiguration from scenario 5 to scenario 6 ($1 \rightarrow 4$), where 64 processes are moved to 4 empty target nodes. When target nodes are oversubscribed, the computation time of each process is penalized and so is the *Spawn&Rec* phase, specially affected due to their collective communications. This can be observed in the increase that the overhead of the *Spawn&Rec* phase suffers in the reconfiguration from scenario 2 to scenario 3 ($4 \rightarrow 3$), from scenario 3 to scenario 4 ($3 \rightarrow 2$), from scenario 4 to scenario 5 ($2 \rightarrow 1$), and from scenario 5 to scenario 6 ($1 \rightarrow 4$), where 16, 21, 32 and 64 processes are migrated each time, oversubscribing the surviving nodes. Finally, since this phase involves different collective communications, its time depends on the total number of processes in the application. This can be observed in Table 5.2 that shows the overhead of the *Spawn&Rec* step when migrating 16 processes to an empty target node with different number of processes in the application.

The *ChkptTransf&Recv* step also impacts significantly in the reconfiguration overhead. The I/O operations are recognized to be one of the main impacting factors in the performance of migration operations, specially in those associated to checkpoint solutions. Checkpoint file sizes are critical to minimize the I/O time. As

Table 5.3: Transfer size (checkpoint size in MB).

NPB	Checkpoint size per process	Total data size migrated *16 proc.*	*32 proc.*	*64 proc.*
BT	33.15	530.45	1060.90	2121.80
CG	7.93	126.97	253.95	507.90
FT	48.09	769.50	1539.01	3078.02
LU	15.48	247.74	495.48	918.96
MG	39.26	628.19	1256.39	2512.78
SP	32.12	513.99	1027.99	2055.98
Himeno	166.71	2667.36	5334.72	10669.44

commented before, CPPC applies live variable analysis and identification of zero-blocks to decrease checkpoint file sizes. Table 5.3 shows the checkpoint sizes per process and the total data size transferred between nodes when migrating 16, 32 and 64 processes. The total amount of data varies between 127 MB for CG migrating a single node (16 processes) and 10.42 GB for Himeno when migrating 4 nodes (64 processes). By means of the pipelined approach (see Section 4.3) that overlaps the state file writing in the terminating processes, the data transfer through the network, and the state file read in the new processes, the proposed solution is able to significantly reduce this impact.

The *Restart* step is a small contributor to the reconfiguration overhead. An important part of this time is due to the RECs associated to the negotiation protocol. During the negotiation phase each process specifies a memory region (window) that it exposes to others. Since the MPI communicators of the application have been reconfigured, at restart time the old MPI windows have to be closed and new ones have to be created. Although not as impacting as the spawning function, the overhead of this operation is not negligible.

Finally, the *Scheduling* phase, which includes the monitoring of communications and the mapping of processes to cores, is negligible for all the NPB applications, always being smaller than 0.1 s. For this reason, these times are not included in the table.

5.5. Related work

There exists in the literature two main approaches to provide malleability to MPI applications. Virtual Malleability, where the number of processes is preserved and the application adapts to changes in the number of resources by oversubscribing processors; and Real Malleability, where the number of processes changes to adapt to the number of available processors.

Regarding virtual malleability, AMPI [39] is an adaptive implementation of MPI built on top of the CHARM++ runtime environment which supports dynamic load balancing through processor virtualization. The computation is divided into a large number of virtual processors, independent of the number of physical processors, the number of virtual processors being typically much larger than of physical processors. The runtime system takes over the responsibility for mapping virtual processors to physical cores. Chakravorty et al. [11] propose a proactive fault tolerant solution using AMPI. Utrera et al. [86] describe another strategy to obtain malleable MPI applications based on the combination of moldability (a job can be executed with different number of processes in different executions, but the number remains fixed during the whole execution), folding (execution of processes overloading physical cores) and co-scheduling techniques. Unfortunately it only works in shared memory multiprocessors.

With respect to real malleability, all the proposals present in the literature are very restrictive regarding the kind of applications they support. Only iterative [24, 33, 58, 82, 87, 92] or master-slave [10, 50] applications are considered, as the modification of the number of processes is much easier than in a general application. Furthermore, in these approaches reconfiguration can only take place in very specific points within the applications. This may not be a serious concern in load balance contexts, however, for fault tolerance purposes reconfiguration points should be selected in a much more flexible way to allow a fast reconfiguration when needed.

The solution proposed in this chapter provides virtual malleability. It is implemented at the application level, and thus it is independent of the hardware architecture, the OS or the MPI implementation used. As the number of processes is preserved, there are no restrictions on the type of applications that can benefit from

this proposal.

5.6. Concluding remarks

This chapter presents an effective and comprehensive approach to achieve automatic virtual malleability in MPI applications, including necessary code transformations, rescheduling, and migration capabilities. Using this proposal, applications are capable of dynamically adapting themselves to the changing conditions of the underlying HPC environment, increasing application performance and system throughput.

The experimental evaluation of the proposal shows successful and efficient operation, with an overhead of a few seconds during reconfiguration, which will be negligible in large applications with a realistic reconfiguration frequency. The higher impact in performance is observed to be due to the MPI operations involved in the migration, specially the MPI spawn function. Future MPI-3 capabilities are expected to improve the performance of migration operations. The proposed scheduling algorithm, based on TreeMatch and Hwloc, obtains well balanced nodes and preserves performance as much as possible. The application-level nature of the checkpoint-and-restart system provides not only good migration performance, but also portability in heterogeneous environments.

Malleable jobs have multiple practical applications. For instance, they can be used in a non-dedicated environment to release resources that need to be assigned to other uses; in a fault tolerance context to implement a proactive fault-tolerant approach through preemptive migration; or in an infrastructure with a bid-based pricing model, as in the Amazon EC2 Spot Instances, to dynamically select the most cost-effective option. They will be also of particular interest in future large scale computing systems, since applications that are able to dynamically reconfigure themselves to adapt to different resource scenarios will be key to achieve a tradeoff between energy consumption and performance.

Conclusions and Future Work

High-performance computing systems tend to increase their number of processors from year to year. Failure rates depend mostly on system size and are roughly proportional to the number of processors in a system. Thus, fault-tolerance techniques need to be applied to parallel applications running in HPC environments to guarantee computation progress.

Many fault tolerance methods for parallel applications exist in the literature, checkpoint-recovery being the most popular. However, the dimension, heterogeneity and dynamic nature of today's large computer infrastructures open new research challenges that must still be solved, requiring proposals that are scalable, to be executed on hundreds of cores; portable, so they can deal with the heterogeneity of the platforms; and malleable to adapt to the available resources. To this end, this Thesis makes the following contributions:

- Different optimization techniques to reduce the I/O cost of checkpointing in application-level approaches: incremental checkpointing, zero-blocks exclusion and data compression. The incremental checkpointing technique stores only that data that have changed from the last checkpoint. The zero-blocks exclusion avoids storing variable blocks with only zeros. Finally, the data compression algorithm compresses the data before dumping it to disk. All the techniques proposed reduce the total amount of data stored, which will be particularly useful for parallel applications with a large number of parallel processes, where the transference of a large amount of checkpoint data to stable storage can saturate the network and cause a drop in application performance.
- A proposal to allow the transparent migration of MPI processes when im-

pending failures are notified, without having to restart the entire application. The proposed solution attains: a low overhead in failure-free executions, by avoiding the checkpoint dumping associated to rolling back strategies; a low overhead at migration time, by means of the design of a light and asynchronous protocol to achieve a consistent global state.

- An in-memory checkpoint-based migration approach to reduce the I/O cost of the migration processes. The in-memory migration avoids storing checkpoint files in stable storage transferring them from memory to memory, and, thus, it moves the bottleneck of the system to the communication network, which in HPC systems is usually a high speed network with low latency.

- A pipeline technique to overlap the different phases of the migration operation, hiding the transferring time. The proposal consists on splitting the checkpoint files into multiple smaller files to overlap the checkpoint file writing in the terminating process, with data transferring through the network, and state file read and restart operations in the new spawned processes.

- A comprehensive proposal to automatically transform MPI applications into malleable jobs. The proposed solution is based on checkpointing and migration and it includes a scheduling algorithm based on the TreeMatch tool to move selected processes to target nodes.

All the proposals were implemented at the application level using CPPC and preserving its main characteristics: portability, not being locked into a particular architecture, operating system or MPI implementation; and transparency for the user.

The results of this research work have been published in the following journals and conferences:

- Journal Papers (4):

  - Iván Cores, Mónica Rodríguez, Patricia González and María J. Martín. Reducing the overhead of an MPI application-level migration approach. *Parallel Computing.* Submitted, currently under minor revision.

- Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. In-memory application-level checkpoint-based migration for MPI programs. *The Journal of Supercomputing.* 70(2): 660–670, 2014.
- Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. Failure avoidance in MPI applications using an application-level approach. *The Computer Journal.* 57(1): 100–114, 2014.
- Iván Cores, Gabriel Rodríguez, María J. Martín, Patricia González and Roberto R. Osorio. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Computing.* 31(3): 163–185, 2013.

- International Conferences (6):

  - Iván Cores, Patricia González, Emmanuel Jeannot, María J. Martín and Gabriel Rodríguez. Checkpoint-based virtual malleability of MPI applications. Submitted to an International Conference. 2015.
  - Mónica Rodríguez, Iván Cores, Patricia González and María J. Martín. Improving an MPI application-level migration approach through checkpoint file splitting. In *Proc. of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014).* Pages 33–40. Paris, France. 2014.
  - Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. High-performance process-level migration of MPI applications. In *Proc. of the 2013 International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'13).* Pages 456–466. Cabo de Gata, Almería, España. 2013.
  - Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. Achieving checkpointing global consistency through a hybrid compile time and runtime protocol. In *Proc. of the 13th International Conference on Computational Science (ICCS'13).* Pages 169–178. Barcelona, España. 2013.
  - Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. In *Proc. of the 10th IEEE International Symposium on*

Parallel and Distributed Processing with Applications (ISPA'12). Pages 371–378. Leganes, Madrid, España. 2012.

- Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. An application level approach for proactive process migration in MPI applications. In *Proc. of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11).* Pages 400–405. Gwangju, Corea del Sur. 2011.

- National Conferences (1):

  - Iván Cores, Gabriel Rodríguez, María J. Martín and Patricia González. Checkpoint size reduction in application-level fault-tolerant solutions. In *Actas de las XII Jornadas de Paralelismo.* Pages 713–718. La Laguna, Tenerife, España. 2011.

Future work

The proactive fault-tolerance approach presented in Chapter 3 has proved to be useful to reduce the overhead of traditional checkpointing and rollback-recovery solutions. Unfortunately, this proposal is not able to cope with already happened failures. Recently, the Fault Tolerance Working Group within the MPI forum proposed the ULFM (User Level Failure Mitigation) interface to integrate resilience capabilities in the MPI 4.0. It includes new semantics for process failure detection, and communicator revocation and reconfiguration. Thus, it enables the implementation of resilient MPI applications, that is, applications that are able to recover themselves from failures. Nevertheless, incorporating the ULFM capabilities in already existing codes is a complex and time-consuming task.

As future work, the migration technique presented in Chapter 3 could be enhanced and extended to use the new functionalities provided by ULFM to transparently obtain resilient MPI applications from generic MPI SPMD programs. The solution would use ULFM to detect failures in one or more processes, whereas the migration protocol proposed in Chapter 3 could be reused to generate new processes

and continue the execution. This would allow to the MPI applications to recover from failures, without stopping nor re-queuing the MPI job.

Bibliography

[1] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu`. Last accessed September 2015.

[2] S. Agarwal, R. Garg, and M. S. Gupta. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*, pages 277–286, Saint Malo, France, 26 June–01 July 2004.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP 03*, pages 164–177, Bolton, NY, USA, 19–22 October 2003.

[4] J. Brandt, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong. Resource monitoring and management with OVIS to enable HPC in cloud computing environments. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–8, Rome, Italy, 2009.

[5] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the 2003 ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 84–94, 2003.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 180–186, Feb 2010.

[7] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *Proceedings of the 2007 International Conference on Cluster Computing, CLUSTER*, pages 372–381, Austin,TX,USA, Sept 2007.

[8] J. Cao, Y. Li, and M. Guo. Process migration for MPI applications based on coordinated checkpoint. In *Proceedings of ICPADS 05*, pages 306–312, Fukuoka, Japan, 20–22 July 2005.

[9] F. Cappello, H. Casanova, and Y. Robert. Checkpointing vs. migration for post-petascale supercomputers. In *Proceedings of ICPP 10*, pages 168–177, San Diego, CA, USA, 13–16 September 2010.

[10] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. Navaux. Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*, pages 242–257, Kolkata, India, 2010.

[11] S. Chakravorty, C. L. Mendes, and L. V. Kale. Proactive fault tolerance in MPI applications via task migration. In *Proceedings of HiPC 06*, pages 485–496, Bangalore, India, 18–21 December 2006.

[12] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[13] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005.

[14] G.-M. Chiu and J.-F. Chiu. A new diskless checkpointing approach for multiple processor failures. *IEEE Transactions on Dependable and Secure Computing*, 8(4):481–493, 2011.

[15] I. Cores, G. Rodríguez, P. González, and M. J. Martín. An application level approach for proactive process migration in MPI applications. In *Proceedings of PDCAT 11*, pages 400–405, Gwangju, Korea, 20–22 October 2011.

[16] I. Cores, G. Rodríguez, P. González, and M. J. Martín. Failure Avoidance in MPI Applications Using an Application-Level Approach. *The Computer Journal*, 57(1):100–114, 2014.

[17] I. Cores, G. Rodríguez, M. J. Martín, and P. González. Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2012)*, pages 371–378, Madrid, Spain, 10–13 July 2012.

[18] I. Cores, G. Rodríguez, M. J. Martín, and P. González. Achieving checkpointing global consistency through a hybrid compile time and runtime protocol. In *2013 International Conference on Computational Science (ICCS)*, volume 18, pages 169–178, 2013.

[19] I. Cores, G. Rodríguez, M. J. Martín, and P. González. High-performance process-level migration of MPI applications. In *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2013*, pages 456–466, Almería, Spain, 24–27 June 2013.

[20] I. Cores, G. Rodríguez, M. J. Martín, and P. González. In-memory application-level checkpoint-based migration for MPI programs. *The Journal of Supercomputing*, 70(2):660–670, 2014.

[21] I. Cores, G. Rodríguez, M. J. Martín, P. González, and R. R. Osorio. Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes. *New Generation Computing*, 31(3):163–185, 2013.

[22] C. Du and X.-H. Sun. MPI-Mitten: Enabling migration technology in MPI. In *Proceedings of CCGRID 06*, pages 11–18, Singapore, 16–19 May 2006.

[23] C. Du, X.-H. Sun, and K. Chanchio. HPCM: A pre-compiler aided middleware for the mobility of legacy code. In *Proceedings of CLUSTER 03*, pages 180–187, Hong Kong, China, 1–4 December 2003.

[24] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. Malleable iterative MPI applications. *Concurrency and Computation: Practice and Experience*, 21(3):393–413, Mar. 2009.

[25] E. Elnozahy and J. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[26] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of PDP 09*, pages 252–257, Weimar, Germany, 18–20 February 2009.

[27] K. B. Ferreira, R. Riesen, R. Brightwell, P. G. Bridges, and D. Arnold. libhashckpt: Hash-Based Incremental Checkpointing Using GPU's. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, pages 272–281, 2011.

[28] E. Gelenbe, D. Finkel, and S. K. Tripathi. Availability of a distributed computer system with failures. *Acta Inform.*, 23(6):643–655, 1986.

[29] E. Gelenbe and M. Hernández. Optimum checkpoints with age dependent failures. *Acta Inform.*, 27(6):519–531, 1989.

[30] C. George and S. S. Vadhiyar. ADFT: An adaptive framework for fault tolerance on large scale systems using application malleability. *Procedia Computer Science*, 9(0):166–175, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

[31] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010*, pages 167–178, 2010.

[32] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA*, page 9, 2005.

[33] E. Godard, S. Setia, and E. White. DyRecT: Software support for adaptive parallelism on NOWs. In *Proceedings of the 14th International Parallel &*

Distributed Processing Symposium, IPDPS, pages 1168–1175, Cancun, Mexico, 2000.

[34] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 63–72, Washington, DC, USA, 2010.

[35] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, and B.-H. Park. Dynamic meta-learning for failure prediction in large-scale systems: A case study. In *Proceedings of ICPP 08*, pages 157–164, Portland, OR, USA, 8–12 September 2008.

[36] R. Gupta, P. Beckman, B. H. Park, E. Lusk, P. Hargrove, A. Geist, A. Lumsdaine, and J. Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *Proceedings of ICPP 09*, pages 237–245, Vienna, Austria, 22–25 September 2009.

[37] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput.*, 69(7):652–665, 2009.

[38] T. J. Hacker, F. Romero, and J. J. Nielsen. Secure live migration of parallel applications using container-based virtual machines. *Int. J. Space-Based and Situated Comput.*, 2(1):45–57, 2012.

[39] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC*, pages 306–322, College Station, TX, USA, October 2003.

[40] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, pages 1098–1101, September 1952.

[41] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD*, pages 206–213, Foz do Iguaçu, PR, Brazil, Oct 2004.

[42] J. Hursey and A. Lumsdaine. A composable runtime recovery policy framework supporting resilient HPC applications. Technical Report TR686, Indiana University, Bloomington, Indiana, USA, August 2010.

[43] IEEE Global History Network. History of lossless data compression algorithms. `http://www.ieeeghn.org/wiki/index.php/History_of_Lossless_Data_Compression_Algorithms`. Last accessed September 2015.

[44] S. ik Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Proocedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03), College Station, TX, USA, 2003*, pages 539–553, 2003.

[45] Information Technology Center, RIKEN. HIMENO Benchmark. `http://accc.riken.jp/2444.htm`. Last accessed September 2015.

[46] E. Jeannot and G. Mercier. Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. In *Euro-Par 2010 - Parallel Processing*, pages 199–210, Ischia, Italy, Aug. 2010.

[47] H. Jin, T. Ke, Y. Chen, and X.-H. Sun. Checkpointing orchestration: Toward a scalable hpc fault-tolerant environment. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, pages 276–283, 2012.

[48] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, Cambridge, MA, USA, 1996.

[49] Laurence Berkeley National Laboratory. Berkeley Lab Checkpoint/Restart. https://ftg.lbl.gov/CheckpointRestart/. Last accessed September 2015.

[50] C. Leopold, M. Süß, and J. Breitbart. Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. In *Proceedings of the 20th European Conference on Modelling and Simulation, ECMS*, pages 665–670, Bonn, Germany, 2006.

[51] C. J. Li and W. K. Fuchs. CATCH: Compiler-Assisted Techniques for Checkpointing. In *20th International Symposium on Fault Tolerant Computing (FTCS-20)*, pages 74–81, 1990.

[52] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. M. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–12, 2010.

[53] Y. Li and Z. Lan. FREM: A fast restart mechanism for general checkpoint/restart. *IEEE Transactions on Computers*, 60(5):639–652, 2011.

[54] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in IBM BlueGene/L event logs. In *Proceedings of ICDM 07*, pages 583–588, Omaha, NE, USA, 28–31 October 2007.

[55] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. University of Wisconsin-Madison, Madison, WI, USA, 1997.

[56] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[57] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11, 2010.

[58] J. Moreira and V. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303–330, May 1997.

[59] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of ICS 07*, pages 23–32, Seattle, WA, USA, 17–21 June 2007.

[60] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 783–788, 2008.

[61] H.-C. Nam, J. Kim, S. J. Hong, and S. Lee. Secure checkpointing. *Journal of Systems Architecture*, 48(8-10):237–254, 2003.

[62] National Aeronautics and Space Administration. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB`. Last accessed September 2015.

[63] A. Norman and C. Lin. A scalable algorithm for compiler-placed staggered checkpointing. In *Proceedings of the 23rd International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, 2012.

[64] Open MPI. Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`. Last accessed September 2015.

[65] X. Ouyang, S. Marcarelli, R. Rajachandrasekar, and D. K. Panda. RDMA-Based job migration framework for MPI over InfiniBand. In *Proceedings of CLUSTER 10*, pages 116–125, Heraklion, Greece, 20–24 September 2010.

[66] X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. K. Panda. High performance pipelined process migration with RDMA. In *Proceedings of CCGRID 11*, pages 314–323, Newport Beach, CA, USA, 23–26 May 2011.

[67] K. Park and V. S. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):65–74, 2006.

[68] J. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[69] J. S. Plank and K. Li. Ickp: A consistent checkpointer for multicomputers. *IEEE Parallel Distrib. Technol.*, 2(2):62–67, June 1994.

[70] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[71] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: an algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, Department of Computer Science, Aug. 1995.

[72] T. Repantis, Y. Drougas, and V. Kalogeraki. Adaptive resource management in peer-to-peer middleware. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS*, page 132b, Denver, CO, USA, April 2005.

[73] G. Rodríguez, M. Martín, P. González, and J. Touriño. A heuristic approach for the automatic insertion of checkpoints in message-passing codes. *Journal of Universal Computer Science*, 15(14):2894–2911, 2009.

[74] G. Rodríguez, M. J. Martín, P. González, and J. Touriño. Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study. *The Computer Journal*, 54(11):1821–1837, 2011.

[75] 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*, 22(6):749–766, 2010.

[76] M. Rodríguez, I. Cores, P. González, and M. J. Martín. Improving an MPI application-level migration approach through checkpoint file splitting. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014)*, pages 33–40, Paris, France, 2014.

[77] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of KDD 03*, pages 426–435, Washington, DC, USA, 24–27 August 2003.

[78] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, 2010.

[79] J. M. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D'Arcy, and A. Chervenak. Monitoring the Grid with the Globus Toolkit MDS4. In *Journal of Physics: Conference Series*, volume 46, page 521, 2006.

[80] M. Schulz, G. Bronevetsky, and B. Supinski. On the performance of transparent MPI piggyback messages. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 194–201. 2008.

[81] R. Singh and P. Graham. Performance driven partial checkpoint/migrate for LAM-MPI. In *Proceedings of HPCS 08*, pages 110–116, Québec City, Canada, 9–11 June 2008.

[82] R. Sudarsan and C. Ribbens. Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In *Proceedings of the International Conference on Parallel Processing, ICPP*, page 44, Xi-An, China, Sept 2007.

[83] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *Proceedings of PODC 10*, pages 173–182, Zurich, Switzerland, 25–28 July 2010.

[84] The HDF Group. HDF5 File Image Operations. `http://www.hdfgroup.org/HDF5/doc/Advanced/FileImageOperations/HDF5FileImageOperations.pdf`. Last accessed September 2015.

[85] The HDF5 Group. HDF-5: Hierarchical Data Format. `http://www.hdfgroup.org/HDF5/`. Last accessed September 2015.

[86] G. Utrera, J. Corbalan, and J. Labarta. Implementing malleability on mpi jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 215–224, Washington, DC, USA, 2004.

[87] S. S. Vadhiyar and J. J. Dongarra. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. In *In: Parallel Processing Letters. Volume 13*, pages 291–312, 2002.

[88] N. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, 46(8):942–947, 1997.

[89] Victor C. Zandy. CKPT process checkpoint library. `http://pages.cs.wisc.edu/~zandy/ckpt/`. Last accessed September 2015.

[90] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proceedings of IPDPS 07*, pages 1–10, Long Beach, CA, USA, 26–30 March 2007.

[91] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration and back migration in HPC environments. *J. Parallel Distrib. Comput.*, 72(2):254–267, 2012.

[92] D. Weatherly, D. Lowenthal, M. Nakazawa, and F. Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC*, pages 5–5, Phoenix, AZ, USA, Nov 2003.

[93] G. Zheng, X. Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–6, 2012.

Appendix A

Summary in Spanish

La tendencia en arquitectura de computadores hoy en día es la utilización de grandes clusters de computación, en muchos casos heterogéneos, en los cuales los nodos son sistemas *multi/many-núcleo*, siendo la mayoría de las aplicaciones de computación intensiva que se ejecutan en estos entornos programas paralelos, y más concretamente de paso de mensajes.

En la bibliografía existen muchos métodos de tolerancia a fallos para sistemas distribuidos, siendo el checkpoint y la replicación los métodos más populares. Sin embargo, la dimensión, heterogeneidad y naturaleza dinámica de las grandes infraestructuras de computación actuales abren nuevos retos de investigación que deben ser todavía resueltos, necesitándose propuestas que sean: escalables, para poder ser ejecutadas sobre cientos de núcleos; portables, para que puedan tratar con la heterogeneidad de las plataformas; y maleables, para que se adapten a la naturaleza dinámica de las mismas. Esta tesis se centra en desarrollar soluciones de tolerancia a fallos y maleabilidad para aplicaciones paralelas de paso de mensajes basadas en checkpoint.

A.1. Antecedentes

Con el aumento constante del tamaño de los grandes sistemas de computación la probabilidad de fallos en los componentes hardware también se incrementa. Para

evitar que estos fallos causen la pérdida de la computación realizada hasta ese momento, surgen las técnicas de tolerancia a fallos, siendo el checkpointing una de las más utilizadas. El checkpointing es una técnica de tolerancia a fallos que consiste en almacenar periódicamente en el disco duro (u otros medios de almacenamiento permanente) todo el estado de la computación para, en caso de fallo, poder reiniciar la aplicación desde el último punto guardado.

Las principales características a tener en cuenta a la hora de diseñar una solución de checkpointing son:

- Granularidad. Hace referencia al nivel en el que se guardan los datos de la aplicación. En el checkpointing a nivel de sistema (o SLC) se almacena todo el estado de la aplicación, incluyendo contadores de programa, registros, etc. Esto se corresponde con una granularidad gruesa. La otra opción es el checkpointing a nivel de variable (o ALC) que corresponde con una granularidad fina, donde se guardan únicamente los datos que se corresponden con las variables que utiliza la aplicación.

- Transparencia. Se refiere a cómo los usuarios ven la técnica. Generalmente las técnicas SLC son soluciones transparentes porque el usuario no necesita conocer la aplicación para generar los checkpoints, simplemente se vuelca toda la memoria a disco. Por contra, las técnicas ALC son soluciones generalmente no transparentes, pues se necesita conocer qué variables de la aplicación es necesario guardar en los checkpoints y cuáles no.

- Portabilidad. Indica si los ficheros de checkpoint permiten reiniciar las aplicaciones en máquinas con hardware y/o software distintos a los de la máquina en la que fueron generados.

- Coordinación. En sistemas no coordinados cada proceso genera los checkpoints independientemente de los demás. En los sistemas coordinados todos los procesos se coordinan para generar los checkpoints al mismo tiempo o en el mismo punto del código.

Las soluciones propuestas en esta tesis son todas a nivel de aplicación, portables y transparentes al usuario. Además, se evita, siempre que sea posible, la coordinación

en tiempo de ejecución entre los procesos, o, cuando no es posible, se proponen algoritmos de consenso ligeros y asíncronos que no afecten negativamente al rendimiento de la aplicación.

Todas las soluciones propuestas se han implementado y evaluado en entornos reales, extendiendo para ello CPPC (ComPiler for Portable Checkpointing), una herramienta de checkpointing a nivel de aplicación que se centra en la inserción de tolerancia a fallos en aplicaciones de pase de mensajes.

CPPC está compuesta por un compilador y una librería. El compilador transforma una aplicación paralela en una aplicación tolerante a fallos insertando llamadas a la librería. Mientras que la librería, ya en tiempo de ejecución, genera los ficheros de checkpoint en función de las llamadas previamente introducidas por el compilador.

Para garantizar la consistencia, CPPC utiliza un protocolo no bloqueante coordinado espacialmente. Se asume un modelo de programación SPMD y todos los procesos generan los checkpoints en los mismos puntos de la aplicación, pero no necesariamente al mismo tiempo. Para evitar los problemas que podrían causar los mensajes entre procesos, los checkpoints se insertan en puntos seguros, es decir, en puntos en los que no existen mensajes en tránsito ni inconsistentes. Estos puntos son detectados automáticamente por el compilador. Los ficheros de checkpoint tienen un tamaño reducido gracias a que CPPC trabaja a nivel de variable, por lo que solo guarda las variables de la aplicación necesarias para el reinicio. Este análisis de variables lo realiza también de forma automática el compilador de CPPC. Por último, los ficheros de checkpoint se guardan en disco utilizando HDF5, una librería y un formato de datos para la transferencia portable de datos gráficos y numéricos entre computadores. Esto proporciona a CPPC la capacidad de guardar y leer los ficheros independientemente de la arquitectura hardware y/o software subyacente, lo que convierte a los ficheros en portables.

A.2. Optimización de la E/S en el checkpointing de aplicaciones paralelas

Aunque el *checkpoint/restart* es la solución más utilizada para proporcionar tolerancia a fallos a aplicaciones científicas, su coste en términos de tiempo de computación, utilización de la red de comunicaciones y ocupación de los recursos de almacenamiento puede ser un serio problema para los sistemas HPC (High Performance Computing) de grandes dimensiones.

El tamaño de los ficheros de checkpoint es el factor que más influye en el rendimiento de la operación de checkpointing. Cuanto más grande sea un fichero, mayor será el tiempo necesario para almacenarlo en disco. Si se consigue disminuir el tamaño de los ficheros de estado, el tiempo para guardarlos y leerlos desde los sistemas de almacenamiento será menor. Los sistemas de checkpointing a nivel de aplicación ya presentan por sí mismos tamaños de ficheros de checkpoint menores a los sistemas que trabajan a nivel de sistema. Sin embargo, estos ficheros siguen siendo demasiado grandes. En esta tesis se proponen técnicas que disminuirán el tamaño de los ficheros, optimizando así las operaciones de E/S y disminuyendo la sobrecarga de la aplicación.

Una técnica que ya aplica inicialmente el compilador de CPPC es el análisis de variables vivas, que se encarga de seleccionar para su almacenamiento únicamente aquellas variables de la aplicación que son necesarias para garantizar un reinicio correcto. Las otras técnicas propuestas e implementadas en CPPC para optimizar la E/S son: checkpointing incremental, exclusión de bloques cero y compresión de datos.

Checkpointing incremental

Es la técnica más utilizada para reducir el tamaño de los ficheros de checkpoint en soluciones SLC. Implica crear dos tipos de checkpoint: completos e incrementales. Los checkpoints completos contienen todos los datos de la aplicación, mientras que los checkpoints incrementales solo contienen aquellos datos que han sido modificados desde el último checkpoint. Los checkpoints incrementales tendrán un tamaño me-

nor que los completos, pues generalmente entre dos checkpoints consecutivos no se modifican todas las variables de una aplicación. Por lo tanto, guardar un checkpoint incremental en disco será más rápido que guardar uno completo. El mayor inconveniente de esta técnica surge a la hora de ejecutar un reinicio, pues será necesario no solo leer el último checkpoint incremental, sino todos los incrementales anteriores hasta el último completo. De esta forma el reinicio se vuelve una operación muy costosa, pues será necesario leer más de un fichero de checkpoint por proceso. Debe tenerse en cuenta que el reinicio será, a priori, una operación muy poco frecuente, por lo que disponer de una técnica que genere checkpoints de forma rápida puede resultar muy conveniente aunque el reinicio sea lento.

Hasta donde nosotros conocemos, no existe ningún sistema ALC que utilice checkpointing incremental. A diferencia de los sistemas SLC, en las soluciones ALC no podemos trabajar a nivel de página de memoria. En esta tesis se propone una solución basada en el uso de funciones *hash*. Esta propuesta divide las variables a guardar en bloques de un tamaño constante, y genera un valor único que identifica a cada bloque en función de su contenido con una función *hash*. Para saber si un bloque debe ser guardado en un checkpoint, es suficiente con calcular el nuevo valor *hash* de ese bloque y compararlo con el antiguo. Si el *hash* varía, es que los datos han sido modificados y se debe guardar el bloque. Si el *hash* no varía, el bloque no ha sido modificado y no se guardará en el nuevo checkpoint incremental.

Exclusion de bloques cero

En aplicaciones científicas reales es común que muchos de los elementos en matrices y estructuras tengan valores nulos o ceros, lo que conlleva grandes trozos de memoria sin información útil. Por lo tanto, una optimización para reducir el tamaño de los checkpoints es evitar que estos grandes bloques de memoria que solo contienen ceros se guarden en disco.

Partiendo del checkpoint incremental descrito anteriormente es fácil identificar los bloques cero. Antes de decidir si un bloque de datos debe ser guardado en el checkpoint, tan solo hay que comparar el *hash* de dicho bloque con el valor *hash* de un bloque de referencia que contiene todo ceros. Si los valores coinciden, el bloque que está siendo analizado contendrá todo ceros y se descarta su almacenamiento

en el checkpoint. En este caso se almacenará como metadatos una indicación para que en el reinicio se identifique ese bloque como un bloque con ceros, y se pueda regenerar esa sección de la memoria de la aplicación.

Compresión de datos

La última técnica que ayuda a reducir el tamaño de los ficheros de estado es la compresión de datos. En este caso, se comprimen todos los datos a medida que se van volcando a disco, consiguiendo disminuir el tamaño final del fichero de checkpoint. En contrapartida, se pierde tiempo en la propia compresión. En el reinicio hay que leer el fichero de checkpoint y descomprimirlo (gastando tiempo de CPU) antes de guardar los datos en memoria.

Existen en la bibliografía un gran número de algoritmos para llevar a cabo la compresión de ficheros de datos. En esta tesis se propone un nuevo algoritmo basado en las características particulares observadas en los ficheros de checkpoint, que representa un compromiso entre eficiencia de la compresión y sobrecarga. Esta técnica resultará más atractiva cuanto más lentos sean la red de comunicaciones y los medios de almacenamiento permanente. En estas condiciones, generar ficheros más pequeños compensará la posible sobrecarga causada por la pérdida de ciclos de CPU empleados en la compresión.

A.3. Migración de procesos basada en checkpointing

En un escenario que realmente quiera sacar partido a aplicaciones con checkpointing en caso de fallo, estos deben ser generados con una cierta frecuencia, lo que implica una gran sobrecarga debido a la E/S. Además, tras un fallo, todos los procesos tienen que reiniciarse desde el último checkpoint generado. Sin embargo, un reinicio completo es innecesario, ya que muchos procesos se estarán ejecutando en nodos de cómputo que no han sufrido ningún fallo. Por otra parte, reiniciar trabajos conlleva otras desventajas. Primero, el trabajo debe ser reenviado a colas de ejecución, con la consecuente pérdida de tiempo. Y segundo, como la asignación de nodos

puede variar en la nueva ejecución, los ficheros de checkpoint deberán reenviarse a estos nuevos nodos, sobrecargando la red de comunicaciones.

Con los avances recientes en monitorización de sistemas y, por lo tanto, en la predicción de fallos hardware, han surgido soluciones que utilizan checkpointing para implementar políticas proactivas. Esto significa que los procesos se migran de forma preventiva a nuevos recursos en caso de detectar algún problema en los nodos en los que las aplicaciones se están ejecutando. Una de las principales ventajas de estas aproximaciones es que las aplicaciones aún no han fallado cuando se detecta el problema, y solo los procesos en esos nodos necesitan almacenar su estado y reiniciarse en otro nodo. Todo esto implica una disminución del volumen de datos almacenados y/o volcados a disco, provocando una disminución de las operaciones de E/S.

En esta tesis se ha extendido CPPC para permitir la migración de procesos cuando se notifica que un fallo está a punto de ocurrir. La migración será iniciada por una señal externa. Cuando los procesos que deben migrar reciben la señal, guardan su estado a disco. A continuación se generan los procesos encargados de suplir a los originales. Estos, gracias a los mecanismos de reinicio propios de CPPC, leerán los ficheros de checkpoint y continuarán la ejecución. Un punto a destacar es que, justo después de generar los nuevos procesos y antes de iniciar el reinicio, los comunicadores globales encargados de gestionar las comunicaciones tienen que ser reconfigurados. Esta reconfiguración es obligatoria porque, a nivel de comunicadores, los procesos nuevos deben ser exactamente iguales a los que substituyen para poder continuar comunicándose con el resto de procesos.

Para evitar problemas de consistencia, la reconfiguración de los comunicadores se llevará a cabo en puntos seguros del código. El compilador de CPPC ya detecta de forma automática estos puntos, facilitando por tanto la implementación de esta aproximación. Sin embargo, no basta con llevar a cabo la reconfiguración en puntos seguros, sino que también habrá que garantizar que todos los procesos realizan dicha reconfiguración en el mismo punto seguro. Se necesita, por tanto, un protocolo de negociación que asegure que todos los procesos convergen a un único punto de checkpoint para llevar a cabo la migración. En este trabajo se propone un algoritmo de negociación hacia adelante de forma que todos los procesos se coordinan en la siguiente llamada a la función de checkpointing común a todos. Desde este punto

se podrá llevar a cabo el volcado a disco por parte de los procesos que migran, la generación de los nuevos procesos y la reconfiguración de los comunicadores. Tras esta reconfiguración, los procesos que no migran podrán continuar libremente su ejecución, por lo menos hasta que lleguen a alguna comunicación bloqueante (en caso de existir).

Se utilizaron funciones *one-sided* y ventanas MPI para el algoritmo de negociación, de forma que los procesos pueden continuar su ejecución de forma asíncrona durante el proceso de negociación. La generación de nuevos procesos se realiza mediante la función *MPI_Comm_spawn_multiple*, que es colectiva y bloqueante. Además, la reconfiguración de los comunicadores necesita utilizar las funciones de gestión de grupos y comunicadores propias de MPI-2.

A.4. Mejorando el rendimiento en la migración de procesos

La migración de procesos basada en checkpointing hereda uno de los mayores problemas del checkpointing, que es su dependencia de la velocidad de los sistemas de almacenamiento. Dicho de otro modo, el tiempo de migración se ve limitado por la escritura y lectura de los ficheros de checkpoint en disco. Esto puede tener consecuencias graves, pues no se debe olvidar que la migración se realiza generalmente cuando se recibe una señal indicando que un recurso hardware está próximo a fallar, por lo que la migración deberá ser lo más rápida posible. Para acelerarla, se proponen dos optimizaciones: migración memoria a memoria y particionado de los ficheros de checkpoint.

Migración memoria a memoria

Para reducir la sobrecarga de E/S en las migraciones, CPPC ha sido modificado para almacenar los ficheros de checkpoint en memoria en lugar de en disco. La mayor diferencia será que, ahora, en lugar de que los nuevos procesos lean los ficheros directamente de disco, los procesos que van a migrar deberán enviar los ficheros de checkpoint desde su memoria a la memoria del proceso que lo va a substituir. Este

envío se realizará mediante mensajes MPI.

Implementar este nuevo tipo de migración en CPPC requirió realizar cambios en la capa de escritura.

Particionado de los ficheros de checkpoint

Incluso con la migración de memoria a memoria, la escritura y lectura de los ficheros de checkpoint sigue siendo la causa principal de sobrecarga en la migración. Para reducir esta sobrecarga, en lugar de escribir todo el fichero para luego transferirlo a la memoria remota, la solución propuesta particiona el fichero de checkpoint en múltiples ficheros de menor tamaño, de forma que mientras el primer fichero está siendo transferido, el siguiente fichero ya puede comenzar a ser creado en memoria. De esta forma se solapan la escritura de los ficheros de estado con la transferencia y lectura por parte del proceso nuevo.

La solución propuesta en esta tesis utiliza la librería HDF5 y respeta la jerarquía de los ficheros de checkpoint utilizada en la versión original de CPPC, preservando una de las características más importantes de la herramienta de checkpoiting, su portabilidad.

La técnica propuesta no solo permite disminuir el tiempo de migración, sino que también permite reducir el consumo de memoria. En la versión previa de migración memoria a memoria era necesario generar todo el fichero HDF5 en memoria para luego enviarlo como un buffer de memoria. Al aplicar la técnica de particionado, primero se ocupa un bloque de memoria con el primer fragmento del checkpoint para, al mismo tiempo que este primer bloque se está enviando, ocupar otro bloque en memoria para ir generando el segundo fragmento de checkpoint. Una vez que el primer bloque es enviado, se libera la memoria para ser utilizada por sucesivos fragmentos y se procede a enviar el segundo. De este modo la memoria total por proceso nunca se incrementará en más de dos veces el tamaño de bloque, haciendo posible migrar procesos en sistemas que no disponen de mucha memoria. Sin esta técnica la memoria se incrementaría en el mismo tamaño que ocupan los ficheros de checkpoint, siendo en algunos casos aumentos del orden de GB.

A.5. Maleabilidad virtual basada en checkpointing

Las aplicaciones maleables son aquellos programas paralelos que son capaces de adaptar su ejecución al número de procesadores disponibles en tiempo de ejecución. Existen en la literatura dos aproximaciones diferentes para proporcionar maleabilidad a las aplicaciones MPI. *Maleabilidad Virtual*, donde se preserva el número de procesos y la aplicación se adapta a los cambios en el número de recursos repartiendo los procesos entre los procesadores disponibles; y *Maleabilidad Real*, donde el número de procesos cambia para adaptarse al número de procesadores disponibles. La solución propuesta en esta tesis proporciona maleabilidad virtual.

Una vez que las aplicaciones tienen la capacidad de migrar procesos, lo único que las separa de obtener maleabilidad virtual es la capacidad de decisión autónoma, esto es, decidir cuándo migrar y a qué recursos deben ir los procesos que abandonan su actual ubicación. Para esta tarea se introducirá en CPPC un algoritmo de planificación, que analizará la localización actual de los procesos y los recursos disponibles, para decidir si es necesario migrar y, en caso afirmativo, qué procesos migran y a dónde.

La información de monitorización del entorno está fuera de los objetivos de esta tesis, por lo que, a efectos prácticos, supondremos que los cambios en la disponibilidad se verán reflejados en un fichero de texto. Este fichero de texto contendrá todos los nodos disponibles junto con el número de núcleos de cómputo que ofrece cada uno. Por lo tanto, cada vez que CPPC vea un cambio en este fichero iniciará el proceso de migración de forma automática, sin intervención por parte de los usuarios.

Una vez que los procesos advierten que deben migrar, se utiliza el algoritmo de negociación explicado anteriormente para decidir en que punto seguro se llevará a cabo dicha migración. A continuación será necesario ejecutar el algoritmo de planificación para averiguar qué procesos tienen que migrar y los nodos de cómputo de destino.

Un proceso puede migrar por dos motivos: que esté en un nodo que desaparece del fichero de nodos disponibles, lo que significa que todos los procesos en ese nodo deben abandonarlo; que su nodo esté sobrecargado y aparezcan nuevos nodos libres en el

fichero de disponibilidad. En este último caso, algunos procesos deberían migrar a los recursos libres para equilibrar la carga, y habrá que decidir cuáles son los procesos ideales para migrar y cuáles deberían quedarse. En nuestra propuesta se realiza una distribución de forma que los procesos que más comunicaciones intercambian permanezcan en el mismo nodo, minimizando así el número de mensajes que se intercambian entre nodos y aliviando la carga en la red de comunicaciones. Para ello se utilizó un componente de monitorización de las comunicaciones ya existente en OpenMPI y las herramientas TreeMatch y Hwloc para optimizar la asignación de procesos a núcleos de ejecución.

TreeMatch es un algoritmo que obtiene el mejor emplazamiento de procesos a núcleos basándose en una matriz de comunicaciones de la aplicación y en una representación de la topología hardware subyacente. TreeMatch trata de minimizar las comunicaciones en todos los niveles, incluyendo red, memoria y jerarquía de caché. Por otro lado, Hwloc permite obtener fácilmente una representación de la topología hardware de todo el sistema de cómputo, facilitando también la asignación de procesos a núcleos.

El algoritmo de planificación se ha incorporado a CPPC, proporcionando una solución completa para alcanzar maleabilidad virtual en aplicaciones MPI de forma totalmente automática y transparente al usuario.

Conclusiones y trabajo futuro

Los sistemas de computación de alto rendimiento incrementan el número de recursos hardware año a año. Con esta tendencia, la probabilidad de fallos también crece, pues es proporcional al número de procesadores del sistema. Por lo tanto, es necesario aplicar técnicas de tolerancia a fallos a las aplicaciones que se ejecutan en este tipo de sistemas para garantizar que los programas finalicen.

La técnica de checkpointing y recuperación es el método de tolerancia a fallos para aplicaciones paralelas más popular en la bibliografía. Sin embargo, la dimensión, heterogeneidad y naturaleza dinámica de las grandes infraestructuras de computación actuales abren nuevos desafíos que todavía deben ser resueltos, requiriéndose propuestas: escalables, para ser ejecutadas en cientos de núcleos de computo; por-

tables, para que puedan adaptarse a la heterogeneidad de los sistemas; y maleables, para adaptarse a la disponibilidad de los recursos. Para este fin, esta tesis hace las siguientes contribuciones:

- Diferentes técnicas de optimización para reducir el coste de E/S del checkpointing a nivel de aplicación: checkpointing incremental, exclusión de bloques cero y compresión de datos. La técnica de checkpointing incremental almacena solamente la información que ha sido modificada desde el último checkpoint. La exclusión de bloques cero evita almacenar bloques de memoria que contengan solo ceros. Finalmente, el algoritmo de compresión de datos comprime los datos antes de volcarlos a disco. Todas las técnicas propuestas reducen el volumen total de datos a guardar, que será especialmente beneficioso en sistemas con un gran número de procesos paralelos, donde almacenar toda la información de todos los procesos puede saturar la red y provocar una caída en el rendimiento de las aplicaciones.

- Una propuesta que permite la migración transparente de procesos MPI cuando se notifican fallos inminentes, sin tener que reiniciar por completo la aplicación. La solución propuesta tiene una sobrecarga casi despreciable en ejecuciones libres de migraciones, ya que evita generar checkpoints con la alta frecuencia que sería necesaria en las soluciones tradicionales de checkpoint y recuperación. Además, la sobrecarga de esta técnica en caso de fallo también es mínima, gracias a un protocolo de coordinación entre procesos ligero y asíncrono.

- Una propuesta para reducir el coste de E/S de la migración de procesos basada en el checkpoint *memoria a memoria*. La migración *memoria a memoria* evita almacenar los ficheros de checkpoint en almacenamiento estable, transfiriéndolos directamente de la memoria del nodo inicial a la del nodo destino. Por lo tanto, el cuello de botella se traslada a la red de comunicaciones, que, en sistemas de computación de alto rendimiento, suele tener alta velocidad y baja latencia.

- Una técnica que permite solapar las diferentes fases del proceso de migración, ocultando el tiempo de transferencia. La propuesta consiste en dividir los ficheros de checkpoint en múltiples ficheros más pequeños, de forma que se solapa

la escritura en memoria de los ficheros, con el envío y recepción por parte de los nuevos procesos.

- Una solución para transformar automáticamente aplicaciones MPI en trabajos maleables. La solución utiliza la técnica de checkpointing y la migración, junto con un algoritmo de planificación basado en TreeMatch, para mover los procesos seleccionados a los nodos destino de forma automática y transparente.

Todas las propuestas fueron implementadas a nivel de aplicación utilizando CPPC y manteniendo las principales características de esta herramienta: transparencia para el usuario y portabilidad, al no estar asociada a ninguna arquitectura, sistema operativo o implementación MPI.

Los resultados de este trabajo de investigación han sido publicados en las siguientes revistas y conferencias:

- Artículos en revistas (4):
  - Iván Cores, Mónica Rodríguez, Patricia González y María J. Martín. Reducing the overhead of an MPI application-level migration approach. *Parallel Computing.* Enviado, actualmente bajo revisión menor.
  - Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. In-memory application-level checkpoint-based migration for MPI programs. *The Journal of Supercomputing.* 70(2): 660–670, 2014.
  - Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. Failure avoidance in MPI applications using an application-level approach. *The Computer Journal.* 57(1): 100–114, 2014.
  - Iván Cores, Gabriel Rodríguez, María J. Martín, Patricia González y Roberto R. Osorio. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Computing.* 31(3): 163–185, 2013.
- Conferencias internacionales (6):
  - Iván Cores, Patricia González, Emmanuel Jeannot, María J. Martín y Gabriel Rodríguez. Checkpoint-based virtual malleability of MPI applications. Enviado a una conferencia internacional. 2015.

- Mónica Rodríguez, Iván Cores, Patricia González y María J. Martín. Improving an MPI application-level migration approach through checkpoint file splitting. En *Proc. of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2014)*. Páginas 33–40. Paris, Francia. 2014.
- Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. High-performance process-level migration of MPI applications. En *Proc. of the 2013 International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE'13)*. Páginas 456–466. Cabo de Gata, Almería, España. 2013.
- Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. Achieving checkpointing global consistency through a hybrid compile time and runtime protocol. En *Proc. of the 13th International Conference on Computational Science (ICCS'13)*. Páginas 169–178. Barcelona, España. 2013.
- Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. Reducing application-level checkpoint file sizes: towards scalable fault tolerance solutions. En *Proc. of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'12)*. Páginas 371–378. Leganes, Madrid, España. 2012.
- Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. An application level approach for proactive process migration in MPI applications. En *Proc. of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11)*. Páginas 400–405. Gwangju, Corea del Sur. 2011.

▪ Conferencias nacionales (1):

- Iván Cores, Gabriel Rodríguez, María J. Martín y Patricia González. Checkpoint size reduction in application-level fault-tolerant solutions. En *Actas de las XII Jornadas de Paralelismo*. Páginas 713–718. La Laguna, Tenerife, España. 2011.

Trabajo futuro

La solución de tolerancia a fallos proactiva propuesta en esta tesis consigue reducir la sobrecarga de las soluciones de checkpointing y recuperación tradicionales. Por desgracia, esta propuesta no puede manejar fallos que ya han ocurrido. Recientemente, el grupo de trabajo encargado de la tolerancia a fallos (Fault Tolerance Working Group) dentro del *MPI forum* propuso una interfaz denominada ULFM (User Level Failure Mitigation) para dotar a MPI 4.0 con la capacidad de adaptarse y sobrevivir a determinados fallos. Esta interfaz incluye una nueva semántica para la detección de fallos en procesos y la reconfiguración y anulación de comunicadores. Por lo tanto, habilita la implementación de aplicaciones MPI resistentes a fallos, esto es, aplicaciones que tienen la habilidad de recuperarse a sí mismas tras un fallo. El principal problema de ULFM es que la incorporación de sus características a los códigos existentes es un proceso complejo y arduo.

Como trabajo futuro, la técnica de migración descrita en esta tesis se puede extender y mejorar para utilizar las funciones proporcionadas por ULFM y obtener aplicaciones MPI resistentes a fallos a partir de programas MPI genéricos. La solución podría usar ULFM para detectar fallos en uno o más procesos, mientras que el algoritmo de migración sería utilizado para generar nuevos procesos con los que continuar la ejecución. Esto dotaría a los programas MPI de la capacidad de recuperarse de fallos sin parar la ejecución y sin la necesidad de re-encolar los trabajos MPI en el sistema.