Doctoral Thesis

Application-level Fault Tolerance and Resilience in HPC Applications

Nuria Losada

2018

UNIVERSIDADE DA CORUÑA

Application-level Fault Tolerance and Resilience in HPC Applications

---

Nuria Losada

Doctoral Thesis

July 2018

PhD Advisors:

María J. Martín
Patricia González

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dra. María José Martín Santamaría
Profesora Titular de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dra. Patricia González Gómez
Profesora Titular de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Application-level Fault Tolerance and Resilience in HPC Applications*" ha sido realizada por Dña. Nuria Losada López-Valcárcel bajo nuestra dirección en el Departamento de Ingeniería de Computadores de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctora en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 26 de Julio de 2018

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

Fdo.: Patricia González Gómez
Directora de la Tesis Doctoral

Fdo.: Nuria Losada López-Valcárcel
Autora de la Tesis Doctoral

A todos los que lo habéis hecho posible.

Acknowledgments

I would especially like to thank my advisors, María and Patricia, for all their support, hard work, and all the opportunities they've handed me. I consider myself very lucky to have worked with them during these years. I would also like to thank Gabriel and Basilio for their collaboration and valuable contributions to the development of this work. I would like to say thanks to all my past and present colleagues in the Computer Architecture Group and in the Faculty of Informatics for their fellowship, support, and all the coffee breaks and dinners we held together.

Huge thanks to my parents, my brother, and my extended family; and to my closest friends who have always been there for me. A special mention to Jorge and Laura for their kindness, support, and all the good times we've shared during these years.

I would like to thank George Bosilca, Aurélien Bouteiller, Thomas Hérault, Damien Genet, Anthony Danalis, and all their colleagues and friends that made my visit at the ICL (Knoxville, US) a great experience both professionally and personally. I would also like to show my gratitude to Leonardo Bautista, Kai Keller and Osman Unsal for hosting me during my visit at BSC-CNS (Spain).

Last but not least, I want to acknowledge the following funders of this work: the Computer Architecture Group, the Department of Computer Engineering, and the University of A Coruña for the human and material support; the HiPEAC network (EU ICT-287759, 687698, and 779656) and its collaboration grants program; the NESUS network under the COST Action IC1305; the SIAM Student Travel Awards to attend SIAM PP'18; the Galician Government (ref. ED431G/01, ED431C 2017/04, and GRC2013-055); and the Ministry of Economy and Competitiveness of Spain (TIN2016-75845-P, TIN2013-42148-P, FPI grant BES-2014-068066, and mobility grant EEBB-I-17-12005).

Nuria.

Resumo

As necesidades computacionais das distintas ramas da ciencia medraron enormemente nos últimos anos, o que provocou un gran crecemento no rendemento proporcionado polos supercomputadores. Cada vez constrúense sistemas de computación de altas prestacións de maior tamaño, con máis recursos hardware de distintos tipos, o que fai que as taxas de fallo destes sistemas tamén medren. Polo tanto, o estudo de técnicas de tolerancia a fallos eficientes é indispensábel para garantires que os programas científicos poidan completar a súa execución, evitando ademais que se dispare o consumo de enerxía. O *checkpoint/restart* é unha das técnicas máis populares. Sen embargo, a maioría da investigación levada a cabo nas últimas décadas céntrase en estratexias *stop-and-restart* para aplicacións de memoria distribuída tralo acontecemento dun fallo-parada. Esta tese propón técnicas *checkpoint/restart* a nivel de aplicación para os modelos de programación paralela máis populares en supercomputación. Implementáronse protocolos de checkpointing para aplicacións híbridas MPI-OpenMP e aplicacións heteroxéneas baseadas en OpenCL, en ámbolos dous casos prestando especial coidado á portabilidade e maleabilidade da solución. En canto a aplicacións de memoria distribuída, proponse unha solución de resiliencia que pode ser empregada de forma xenérica en aplicacións MPI SPMD, permitindo detectar e reaccionar a fallos-parada sen abortar a execución. Neste caso, os procesos fallidos vólvense a lanzar e o estado da aplicación recupérase cunha volta atrás global. A maiores, esta solución de resiliencia optimizouse implementando unha volta atrás local, na que só os procesos fallidos volven atrás, empregando un protocolo de almacenaxe de mensaxes para garantires a consistencia e o progreso da execución. Por último, proponse a extensión dunha librería de checkpointing para facilitares a implementación de estratexias de recuperación ad hoc ante corrupcións de memoria. En moitas ocasións, estos erros poden ser xestionados a nivel de aplicación, evitando desencadear un fallo-parada e permitindo unha recuperación máis eficiente.

Resumen

El rápido aumento de las necesidades de cómputo de distintas ramas de la ciencia ha provocado un gran crecimiento en el rendimiento ofrecido por los supercomputadores. Cada vez se construyen sistemas de computación de altas prestaciones mayores, con más recursos hardware de distintos tipos, lo que hace que las tasas de fallo del sistema aumenten. Por tanto, el estudio de técnicas de tolerancia a fallos eficientes resulta indispensable para garantizar que los programas científicos puedan completar su ejecución, evitando además que se dispare el consumo de energía. La técnica *checkpoint/restart* es una de las más populares. Sin embargo, la mayor parte de la investigación en este campo se ha centrado en estrategias *stop-and-restart* para aplicaciones de memoria distribuida tras la ocurrencia de fallos-parada. Esta tesis propone técnicas *checkpoint/restart* a nivel de aplicación para los modelos de programación paralela más populares en supercomputación. Se han implementado protocolos de checkpointing para aplicaciones híbridas MPI-OpenMP y aplicaciones heterogéneas basadas en OpenCL, prestando en ambos casos especial atención a la portabilidad y la maleabilidad de la solución. Con respecto a aplicaciones de memoria distribuida, se propone una solución de resiliencia que puede ser usada de forma genérica en aplicaciones MPI SPMD, permitiendo detectar y reaccionar a fallos-parada sin abortar la ejecución. En su lugar, se vuelven a lanzar los procesos fallidos y se recupera el estado de la aplicación con una vuelta atrás global. A mayores, esta solución de resiliencia ha sido optimizada implementando una vuelta atrás local, en la que solo los procesos fallidos vuelven atrás, empleando un protocolo de almacenaje de mensajes para garantizar la consistencia y el progreso de la ejecución. Por último, se propone una extensión de una librería de checkpointing para facilitar la implementación de estrategias de recuperación ad hoc ante corrupciones de memoria. Muchas veces, este tipo de errores puede gestionarse a nivel de aplicación, evitando desencadenar un fallo-parada y permitiendo una recuperación más eficiente.

Abstract

The rapid increase in the computational demands of science has lead to a pronounced growth in the performance offered by supercomputers. As High Performance Computing (HPC) systems grow larger, including more hardware components of different types, the system's failure rate becomes higher. Efficient fault tolerance techniques are essential not only to ensure the execution completion but also to save energy. Checkpoint/restart is one of the most popular fault tolerance techniques. However, most of the research in this field is focused on stop-and-restart strategies for distributed-memory applications in the event of fail-stop failures. This thesis focuses on the implementation of application-level checkpoint/restart solutions for the most popular parallel programming models used in HPC. Hence, we have implemented checkpointing solutions to cope with fail-stop failures in hybrid MPI-OpenMP applications and OpenCL-based programs. Both strategies maximize the restart portability and malleability, i.e., the recovery can take place on machines with different CPU/accelerator architectures, and/or operating systems, and can be adapted to the available resources (number of cores/accelerators). Regarding distributed-memory applications, we propose a resilience solution that can be generally applied to SPMD MPI programs. Resilient applications can detect and react to failures without aborting their execution upon fail-stop failures. Instead, failed processes are re-spawned, and the application state is recovered through a global rollback. Moreover, we have optimized this resilience proposal by implementing a local rollback protocol, in which only failed processes rollback to a previous state, while message logging enables global consistency and further progress of the computation. Finally, we have extended a checkpointing library to facilitate the implementation of ad hoc recovery strategies in the event of soft errors, caused by memory corruptions. Many times, these errors can be handled at the software-level, thus, avoiding fail-stop failures and enabling a more efficient recovery.

Preface

High Performance Computing (HPC) and the use of supercomputers are key in the development of many fields of science. The large calculation capacity of these machines enables the resolution of scientific, engineering, and analytic problems. For more than two decades, the performance offered by supercomputers has grown exponentially as a response to the rapid increase in the computational demands of science. Current HPC systems are clusters of commodity and purpose built processors interconnected by high-speed communication networks. The usage of multicore nodes has dominated the scene since early 2000s, favoring the use of a hybrid programming model (combining distributed-memory and shared-memory parallel programming models). Since mid-2000s, the presence of accelerator devices (e.g. GPUs, Xeon Phi) in supercomputing sites has heavily increased because of the notable improvements in runtime and power consumption they provide, and new programming models to exploit these processors have emerged, such as OpenCL or CUDA.

Heterogeneous, large-scale supercomputers are a great opportunity for HPC applications, however, they are also a hazard for the completion of their execution. As HPC systems continue to grow larger and include more hardware components of different types, the meantime to failure for a given application also shrinks, resulting in a high failure rate. Efficient fault tolerance techniques need to be studied not only to ensure the scientific application completion in these systems, but also to save energy. However, the most popular parallel programming models that HPC applications use to exploit the computation power provided by supercomputers, lack fault tolerance support.

Checkpoint/restart is one of the most popular fault tolerance techniques in HPC.

However, most of the research in this field is focused on stop-and-restart strategies in the event of fail-stop failures, aborting the execution to recover the computation from a past saved state. In addition, most proposals target only distributed-memory applications, for which the Message Passing Interface (MPI) is the de-facto standard.

This thesis focuses on the implementation of application-level checkpoint/restart solutions for the most popular parallel programming models used in HPC. Most of the developments of this thesis have been implemented on top of the checkpointing tool ComPiler for Portable Checkpointing (CPPC) [112] because of the transparent, portable, application-level checkpointing it provides. We provide fault tolerance support to hybrid MPI-OpenMP applications and to OpenCL-based heterogeneous codes, implementing checkpointing strategies to cope with fail-stop failures. Both proposals pay special attention to the portability and malleability of the recovery, enabling the recovery of the applications in machines with different architectures and/or operating systems, and adapting the execution to a different the number and/or type of resources (different number of cores in the nodes, different number/architecture of accelerator devices). This thesis also explores new possibilities in fault tolerance support for distributed-memory applications. We exploit the ULFM interface—the most recent effort to add resilience features in the MPI standard—to transparently obtain resilient applications from generic Single Program, Multiple Data (SPMD) programs. Resilient applications can detect and react to failures without aborting their execution upon fail-stop failures. Instead, the failed processes are re-spawned, and the application state is recovered through a global rollback. Moreover, we have optimized that resilience proposal by implementing a local rollback protocol for generic SPMD codes, in which only failed processes rollback to a previous state. Because failures usually have a localized scope, this technique significantly improves the overhead and the energy consumption introduced by a failure. Finally, this thesis explores how to reduce the impact of soft errors. These types of errors are caused by transiently corrupted bits on the DRAM and SRAM that cannot be corrected by hardware mechanisms, and they are among the most common causes of failures. This thesis extends the Fault Tolerance Interface (FTI) [12], an application-level checkpointing library, to facilitate the implementation of custom recoveries for MPI applications in the event of soft errors. Custom recovery strategies enable the management of soft errors at the application-level, before they cause fail-stop failures, and therefore, these recovery strategies reduce the overall failure overhead.

Main contributions

The main contributions of this thesis are:

- A portable and malleable application-level checkpointing solution to cope with fail-stop failures on hybrid MPI-OpenMP applications [83, 84, 85].
- A portable and malleable application-level checkpointing solution to cope with fail-stop failures on heterogeneous applications [80].
- A global rollback checkpointing solution that can be generally applied to SPMD MPI programs to obtain resilient applications [53, 78, 79, 81, 82].
- A local rollback protocol based on application-level checkpointing and message logging that can be generally applied to SPMD MPI programs to obtain resilient applications [76, 77].
- A set of extensions to an application-level checkpointing library to facilitate the implementation of custom recovery strategies for MPI applications to cope with soft errors [75].

Structure of the thesis

The remainder of the thesis is organized as follows:

- Chapter 1 introduces the thesis. This chapter presents the current trends in HPC systems and programming models. It exposes why these types of machines need fault tolerance support, and it presents one of the most popular fault tolerance techniques: checkpointing. Finally, this chapter presents the CPPC checkpointing tool, which is used as the based infrastructure for the majority of the developments of this thesis.
- Chapter 2 proposes a checkpointing solution to cope with fail-stop failures in hybrid MPI-OpenMP applications. A new checkpointing protocol ensures checkpoint consistency, while the portability features enable the restart on machines with different architectures, operating systems and/or number of cores,

adapting the number of running OpenMP threads for the best exploitation of the available resources.

- Chapter 3 focuses on heterogeneous systems and proposes a fault tolerance solution to tolerate fail-stop failures in the host CPU or in any of the accelerators devices used. The proposal enables applications to be restarted changing the host CPU and/or the architecture of the accelerator devices and adapting the computation to the number of devices available during the recovery.

- Chapter 4 extends the CPPC checkpointing framework to exploit the new features provided by the ULFM interface. This extension transparently obtains resilient MPI applications, i.e., applications that can detect and react to failures without aborting their execution, from generic SPMD programs.

- Chapter 5 extends and optimizes the resilience proposal described in Chapter 4 to implement a local rollback protocol by combining ULFM, the CPPC checkpointing tool, and a two-level message logging protocol. Using this protocol, only failed processes are recovered from the last checkpoint, thus, avoiding the repetition of computation by the survivor processes and reducing both the time and energy consumption of the recovery process.

- Chapter 6 is focused on soft errors, that is, errors originated by transiently corrupted bits on the Dynamic Random Access Memory (DRAM) and Static Random Access Memory (SRAM). In this chapter, the FTI application-level checkpointing library is extended to facilitate the implementation of custom recovery strategies for MPI applications. Custom recovery techniques aim to handle soft errors before they trigger fail-stop failures and to exploit the characteristics of the algorithm, thus, minimizing the recovery overhead both in terms of time and energy.

Funding and Technical Means

The means necessary to carry out this thesis have been the following:

- Working material, human and financial support primarily by the Computer Architecture Group of the University of A Coruña, along with the Fellowship

funded by the Ministry of Economy and Competitiveness of Spain (FPI ref. BES-2014-068066).

- Access to bibliographical material through the library of the University of A Coruña.
- Additional funding through the following research projects and networks:
  - European funding: "High-Performance and Embedded Architecture and Compilation (HiPEAC–5)" ref. 779656, "High-Performance and Embedded Architecture and Compilation (HiPEAC–4)" ref. 687698, "Network For Sustainable Ultrascale Computing (NESUS)" COST Action IC1305, and "High-Performance Embedded Architecture and Compilation Network of Excellence (HiPEAC–3)" ref. ICT-287759.
  - State funding by the Ministry of Economy and Competitiveness of Spain: "New Challenges in High Performance Computing: from Architectures to Applications (II)" ref. TIN2016-75845-P, and "New Challenges in High Performance Computing: from Architectures to Applications" ref. TIN2013-42148-P.
  - Regional funding by the Galician Government (Xunta de Galicia): "Accreditation, Structuring and Improvement of Singular Research Centers: Research Center on Information and Communication Technologies (CITIC) of the University of A Coruña" ref. ED431G/01, and under the Consolidation Program of Competitive Research Groups (Computer Architecture Group, refs. ED431C 2017/04 and GRC2013-055)
- Access to clusters and supercomputers:
  - FinisTerrae-II supercomputer (Galicia Supercomputing Center, CESGA, Spain): 306 nodes with two Intel Xeon E5-2680 v3 @ 2.50GHz processors, with 12 cores per processor and 128 GB of RAM, interconnected via InfiniBand FDR 56Gb/s.
  - Pluton cluster (Computer Architecture Group, the University of A Coruña, Spain):
    - 16 nodes with two Intel Xeon E5-2660 @ 2.20GHz processors, with 8 cores per processor, 64 GB of memory, interconnected via InfiniBand

FDR: eight of the nodes with 1 NVIDIA Tesla Kepler K20m 5 GB, one of them with 2 NVIDIA Tesla Kepler K20m 5 GB, and one of them with one Xeon PHI 5110P 8 GB.

    - 4 nodes with one Intel Xeon hexa-core Westmere-EP processor, 12 GB of memory and 2 GPUs NVIDIA Tesla Fermi 2050 per node, interconnected via InfiniBand QDR.

  - CTE-KNL cluster (Barcelona Supercomputing Center – Centro Nacional de Supercomputación, BSC-CNS, Spain): 16 nodes, each one with Intel(R) Xeon Phi(TM) CPU 7230 @ 1.30GHz 64-core processor, 94 GB of main memory with 16 GB high bandwidth memory (HBM) in cache mode, 120 GB SSD as local storage, and Intel OPA interconnection.

- A one-month research visit at the Barcelona Supercomputing Center – Centro Nacional de Supercomputación (BSC-CNS), Spain, from October 2017 to November 2017. Working on recovery mechanisms for parallel applications in the event of soft errors (memory corruptions) within the FTI checkpointing tool. This research visit was supported by a Mobility Support Grant Severo Ochoa Centre of Excellence Program from the BSC-CNS.

- A five-month research visit at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville, USA, from September 2016 to March 2017. Working on a local recovery protocol based on message logging and the CPPC checkpointing tool. The first three months of the visit were financially supported by a 2016 Collaboration Grant from the European Network on High-Performance and Embedded Architecture and Compilation (HiPEAC–4), and the last two months were supported by the Ministry of Economy and Competitiveness of Spain under the FPI program (ref. EEBB-I-17-12005).

Contents

1. Introduction and Background **1**

1.1. Current Trends in High Performance Computing 1

1.2. Fault Tolerance on HPC Applications 7

1.2.1. Faults, Errors, and Failures 8

1.2.2. Checkpoint/Restart 9

1.3. CPPC Overview 12

2. Application-level Checkpointing for Hybrid MPI-OpenMP Apps. **17**

2.1. Checkpoint/Restart of OpenMP Applications 18

2.2. Checkpoint/Restart of Hybrid MPI-OpenMP Applications 20

2.2.1. Coordination Protocol 20

2.2.2. Restart Portability and Adaptability 24

2.3. Experimental Evaluation 24

2.3.1. Operation Overhead in the Absence of Failures 25

2.3.2. Operation Overhead in the Presence of Failures 28

2.3.3. Portability and Adaptability Benefits 29

2.4. Related Work 30

2.5. Concluding Remarks 31

3. Application-level Checkpointing for Heterogeneous Applications 33

3.1. Heterogeneous Computing using HPL 34

3.2. Portable and Adaptable Checkpoint/Restart of Heterogeneous Applications 37

3.2.1. Design Decisions 37

3.2.2. Implementation Details 40

3.2.3. Restart Portability and Adaptability 42

3.3. Experimental Evaluation 48

3.3.1. Operation Overhead in the Absence of Failures 51

3.3.2. Operation Overhead in the Presence of Failures 54

3.3.3. Portability and Adaptability Benefits 55

3.4. Related Work 58

3.5. Concluding Remarks 62

4. Application-Level Approach for Resilient MPI Applications 63

4.1. Combining CPPC and ULFM to Obtain Resilience 64

4.1.1. Failure Detection 65

4.1.2. Reconfiguration of the MPI Global Communicator 68

4.1.3. Recovery of the Application 69

4.2. Improving Scalability: Multithreaded Multilevel Checkpointing 70

4.3. Experimental Evaluation 71

4.3.1. Operation Overhead in the Absence of Failures 71

4.3.2. Operation Overhead in the Presence of Failures 73

4.3.3. Resilience vs. Stop-and-Restart Global Rollback 77
4.4. Related work 84
4.5. Concluding Remarks 85

5. Local Rollback for Resilient MPI Applications 89
5.1. Local Rollback Protocol Outline 90
5.2. Message Logging 93
5.2.1. Logging Point-to-Point Communications 93
5.2.2. Logging Collective Communications 94
5.2.3. Implications for the Log Size 96
5.3. Communications Interrupted by a Failure 97
5.4. Tracking Messages and Emission Replay 99
5.4.1. Tracking Protocol 100
5.4.2. Ordered Replay 101
5.5. Experimental Evaluation 102
5.5.1. Operation Overhead in the Absence of Failures 105
5.5.2. Operation Overhead in the Presence of Failures 110
5.5.3. Weak scaling experiments 114
5.6. Related Work 117
5.7. Concluding Remarks 118

6. Local Recovery For Soft Errors 121
6.1. Soft Errors 122
6.2. The FTI Checkpointing Library 122
6.3. FTI Extensions to Facilitate Soft Error Recovery 123

6.4. Ad Hoc Local Recovery on HPC Applications 127
6.4.1. Himeno 127
6.4.2. CoMD 127
6.4.3. TeaLeaf 129
6.5. Experimental Evaluation 132
6.5.1. Memory Characterization 133
6.5.2. Overhead in the Absence of Failures 135
6.5.3. Overhead in the Presence of Failures 136
6.6. Related Work 138
6.7. Concluding Remarks 139

7. Conclusions and Future Work **141**

A. Extended summary in Spanish **147**

References **163**

List of Tables

2.1. Hardware platform details. 25

2.2. Configuration parameters of the testbed applications. 25

3.1. Maximum kernel times (seconds) of the most time-consuming applications from popular benchmarks suites for heterogeneous computing (test performed on System#1 from Table 3.2). 39

3.2. Hardware platform details. 49

3.3. Testbed benchmarks description and original runtimes. 49

3.4. Testbed benchmarks characterization. 50

3.5. Instrumentation and checkpoint overhead analysis for the testbed benchmarks. 52

3.6. Related work overview. 61

4.1. Configuration parameters of the testbed applications. 72

4.2. Hardware platform details. 72

4.3. Number of calls to the MPI library per second performed by the process that less calls does, which determines the detection time. . . . 74

4.4. Average size (MB) of data registered by each process (including zero-blocks) and average size (MB) of the checkpoint file generated by each process (excluding zero-blocks). 75

4.5. Configuration parameters of the testbed applications. 78

4.6. Hardware platform details. 78

4.7. Original runtimes (in minutes) and aggregated checkpoint file sizes. . 78

4.8. Testbed checkpointing frequencies and elapsed time (in minutes) between two consecutive checkpoints for different checkpointing frequencies. 80

4.9. Recovery operations in each proposal. 81

5.1. Hardware platform details. 103

5.2. Configuration parameters of the testbed applications. 103

5.3. Original runtimes of the testbed applications in minutes. 104

5.4. Benchmarks characterization by MPI calls and log behavior. 106

5.5. Original runtimes (in minutes) and configuration parameters of the Himeno weak scaling experiments. 115

5.6. Himeno characterization by MPI calls and log behavior (weak scaling).115

6.1. Hardware platform details. 132

6.2. Weak scaling configurations and original application runtimes. 133

6.3. Memory characterization of the tested applications. 134

6.4. Reduction in the overhead when using the local recovery instead of the global rollback upon a soft error: absolute value (in seconds) and percentage value (normalized with respect to the original execution runtime). 138

List of Figures

1.1. Microprocessors trend during the last 42 years (from [64]). 2

1.2. Development of the number of cores per socket over time in the systems included on the TOP500 lists. 3

1.3. Development of architectures over time in the systems included on the TOP500 lists. 4

1.4. Systems presenting accelerators on the TOP500 lists. 4

1.5. Simplified diagram of a hybrid distributed shared memory architecture. 6

1.6. Simplified diagram of a heterogeneous architecture. 6

1.7. Performance development over time: aggregated performance of all the systems, and performance of first (#1) and last (#500) ranked systems on the TOP500 lists. 7

1.8. Relation between faults, errors, and failures. 9

1.9. Inconsistencies caused by communications crossing a recovery line. . . 11

1.10. CPPC global flow. 13

1.11. CPPC instrumentation example. 13

1.12. Spatial coordination protocol. 14

2.1. CPPC on OpenMP applications: coordinated checkpointing across OpenMP threads initiated by fastest thread. 19

2.2. CPPC on hybrid MPI-OpenMP applications: coordinated checkpointing across OpenMP threads and uncoordinated across MPI processes. 21

2.3. Example of a fault-tolerant code: OpenMP coordination protocol can break the spatial coordination between the teams of threads in hybrid applications. 22

2.4. Need for a new coordination protocol for hybrid MPI-OpenMP programs. 23

2.5. Runtimes for the testbed applications varying the number of cores. . 26

2.6. Absolute overheads varying the number of cores. 26

2.7. CPPC checkpointing operations times varying the number of cores. . 27

2.8. CPPC restart operations times varying the number of cores. 29

2.9. Recovery varying the computation nodes. 30

3.1. OpenCL hardware model. 35

3.2. Example of an HPL application where two different kernels are invoked `nIters` times. 36

3.3. Fault tolerance instrumentation of an HPL application. 43

3.4. Adaptability in heterogeneous applications. 45

3.5. Automatic instrumentation of pseudo-malleable applications. 46

3.6. Automatic instrumentation of malleable applications. 47

3.7. Checkpointing runtimes normalized with respect to the original runtimes for the testbed benchmarks. 53

3.8. Original runtimes and restart runtimes on the same hardware for the testbed benchmarks. 55

3.9. Restart runtimes for the testbed benchmarks on different device architectures. 56

3.10. Restart runtimes for the testbed benchmarks using a different number of GPUs. 57

3.11. Restart runtimes for Shwa1ls using a different host, a different number and architecture of devices, and a different operating system. 57

4.1. Instrumentation for resilient MPI applications with CPPC and ULFM. 66

4.2. Global overview of the recovery procedure for resilient MPI applications with CPPC and ULFM. 67

4.3. `CPPC_Check_errors` pseudocode: failure detection and triggering of the recovery. 67

4.4. Runtimes and aggregated checkpoint file size for the testbed benchmarks when varying the number of processes. 73

4.5. Times of the operations performed to obtain resilience. 74

4.6. Runtimes when introducing failures varying the number of processes. The baseline runtime in which no overhead is introduced (apart from recomputation after the failure) is included for comparison purposes. 77

4.7. Checkpointing overhead varying the checkpointing frequency. 79

4.8. Recovery times: addition of the times of all the recovery operations performed by each proposal (the lower, the better). 81

4.9. Time of the different recovery operations. 83

4.10. Reduction in the extra runtime when introducing a failure and using resilience proposal instead of stop-and-restart rollback (higher is better). 84

5.1. Local rollback protocol overview. 91

5.2. Binomial-tree staging of `AllReduce` collective operation. 95

5.3. States of non-blocking communications. 98

5.4. Application level vs. internal point-to-point logging of collective communications: performance, logged data, and number of entries in the log. . . . 108

5.5. Log parameters when checkpointing: maximum log size expressed as the percentage of the total memory available and number of entries. . 108

5.6. Absolute checkpointing overhead with respect to the non fault-tolerant version and aggregated checkpoint file sizes. . . . 110

5.7. Reduction of the recovery times of survivor and failed processes with the local rollback (the higher, the better). . . . 111

5.8. Times of the different operations performed during the recovery. . . . 112

5.9. Percentage that each recovery operation represents over the reduction in the failed processes' recovery times. . . . 112

5.10. Reduction in the extra runtime and energy consumption when introducing a failure and using local rollback instead of global rollback (higher is better). . . . 114

5.11. Results for the Himeno benchmark doing weak scaling (keeping the problem size by process constant). . . . 116

6.1. FTI instrumentation example 123

6.2. Detection of a soft error. . . . 125

6.3. Memory Protection 126

6.4. Himeno simplified pseudocode. . . . 128

6.5. CoMD simplified pseudocode. . . . 130

6.6. TeaLeaf simplified pseudocode. . . . 131

6.7. Relative overheads with respect to application original runtimes in a fault-free execution. . . . 136

6.8. Relative overheads with respect to application original runtimes when introducing a failure. 137

List of Acronyms

ABFT Algorithm-Based Fault Tolerance

CID Communicator ID

CPPC ComPiler for Portable Checkpointing

DCE Detectable Correctable Error

DRAM Dynamic Random Access Memory

DUE Detectable Uncorrectable Error

ECCs Error Correcting Codes

FIFO First In, First Out

FLOPs Floating Point Operations per Second

FTI Fault Tolerance Interface

HPC High Performance Computing

HPL Heterogeneous Programming Library

hwloc Portable Hardware Locality

MPI Message Passing Interface

MPPs Massively Parallel Processing

MTTF Mean Time To Failure

PEs Processing Elements

PMPI Profiling MPI API

SDC Silent Data Corruption

SE Silent Error

SIMD Single Instruction, Multiple Data

SMPs Shared-Memory Multiprocessors

SPMD Single Program,Multiple Data

SRAM Static Random Access Memory

SSID Sender Sequence ID

ULFM User Level Failure Mitigation

Chapter 1

Introduction and Background

This chapter presents the background related to the research carried out in this thesis. It is structured as follows. Section 1.1 briefly introduces current trends in HPC systems and programming models. Section 1.2 exposes the need for fault tolerance support of these type of infrastructures, describing the most popular fault tolerance technique in the last decades, i.e. checkpointing. Section 1.3 describes the main characteristics of the CPPC checkpointing tool, which is used as the base framework on top of which most of the developments of this thesis have been built.

1.1. Current Trends in High Performance Computing

Many fields of science rely on HPC and in supercomputers for their advance. The large computation power these machines provide—today, in the order of 10^{15} floating point operations per second—enables the resolution of scientific, engineering, and analytic problems. However, the computational demands of science keep growing mainly because of two factors: new problems in which the resolution time is critical (such as the design of personalized pharmaceutical drugs, in which patients cannot wait years for the specific molecule they need), and the exponential growing on the amount of data that must be processed (for instance, data originated by large telescopes, particle accelerators and detectors, social networks, or smart cities

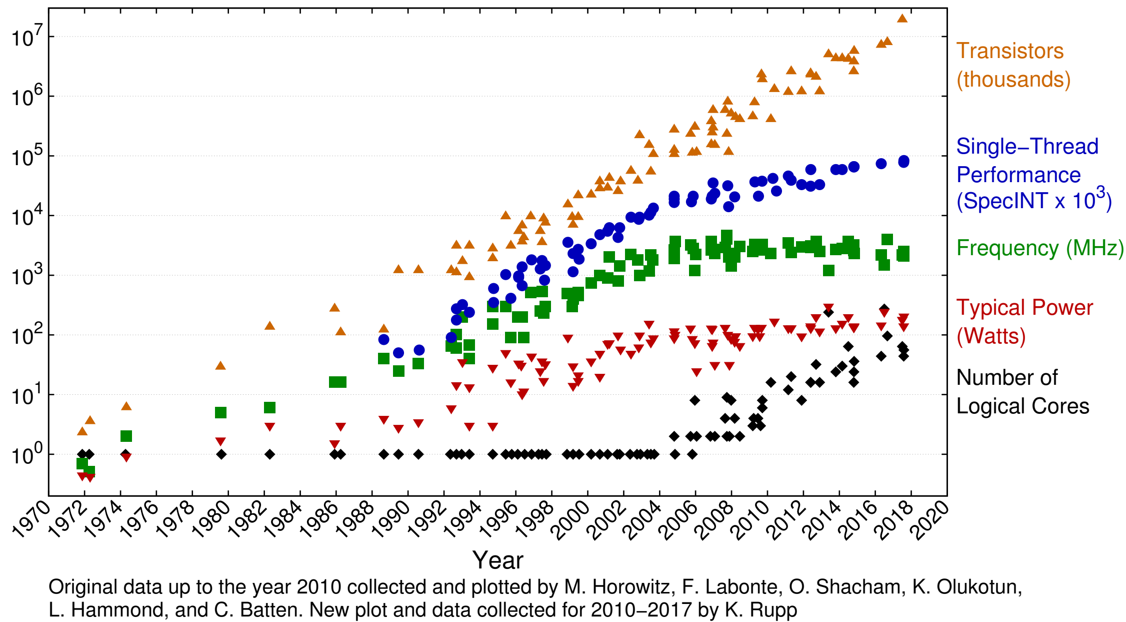


Figure 1.1: Microprocessors trend during the last 42 years (from [64]).

sensors).

Figure 1.1 shows the trends in microprocessors during the last 40 years. Until 2005, the silicon industry responded to the growing computational demand following the Moore's Law [93]. By reducing the size of transistors, their number was doubled in an integrated circuit every two years. This enabled higher clock speeds, bigger cache memories, and more flexible microarchitectures. The existing sequential programming model was preserved, as programs run faster with each new generation of processors. In the early 2000s computer architecture trends switched to multicore scaling as a response to various architectural challenges that severely diminished the gains of further frequency scaling.

The switch within the industry to the multicore era can also be observed in the development of supercomputers over the years. Since 1993, the TOP500 list [44] has gathered, twice a year, the information about the 500 most powerful supercomputers in the world. Using the TOP500's data, Figure 1.2 presents the development over time of the number of cores per socket on supercomputers. The number of cores per socket has increased rapidly and, nowadays, 95% of the entries of the list have between 6 and 24 cores per socket. The supercomputer architectures built up with

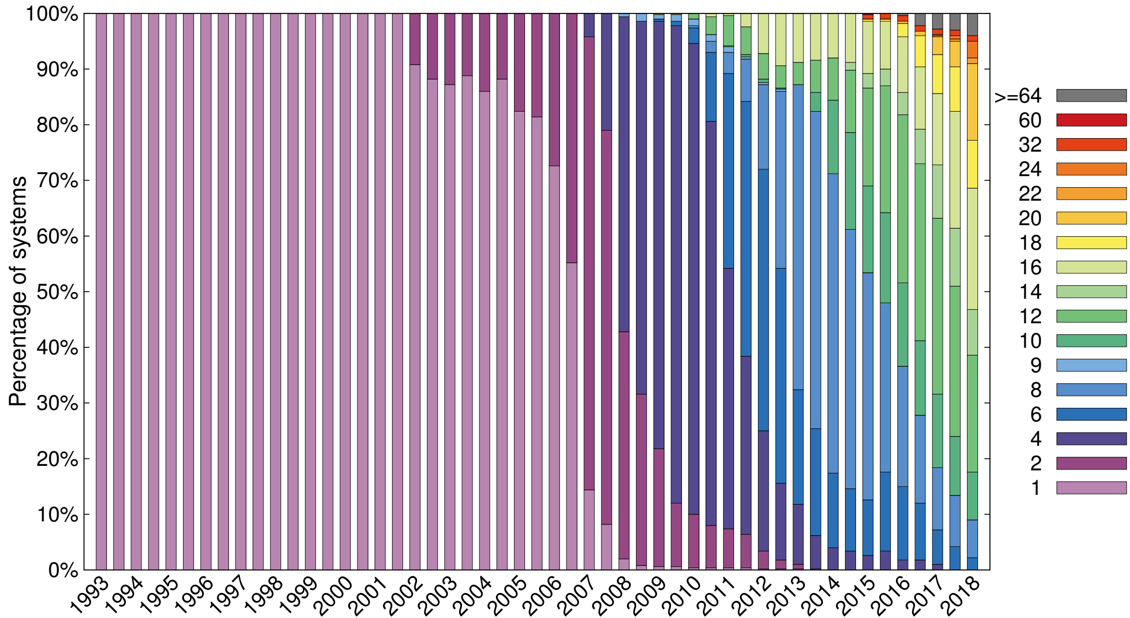


Figure 1.2: Development of the number of cores per socket over time in the systems included on the TOP500 lists.

those microprocessors are studied on Figure 1.3. In the 1980s vector supercomputing dominated HPC. The 1990s saw the rise of Massively Parallel Processing (MPPs) and Shared-Memory Multiprocessors (SMPs). Today, the scene is dominated by clusters of commodity and purpose-built processors interconnected by high-speed communication networks. In addition to this, since 2006 the usage of accelerator devices has become increasingly popular, as they provide notable improvements in runtime and power consumption with respect to approaches solely based on general-purpose CPUs [65]. This trend can be observed in Figure 1.4, which reports the number of systems using accelerators from the TOP500 lists.

Users in HPC sites exploit the computational power provided by supercomputers by means of parallel programming models. Subsequent paragraphs introduce the approaches that have become the most popular ones for the different hardware models that dominate the scene nowadays: distributed-memory systems, shared-memory systems, and heterogeneous systems exploiting accelerators.

Distributed-memory systems present multiple processors, each one of them with its own memory space, and interconnected by a communication network. MPI [132] is the de-facto standard for programming HPC parallel applications in distributed-

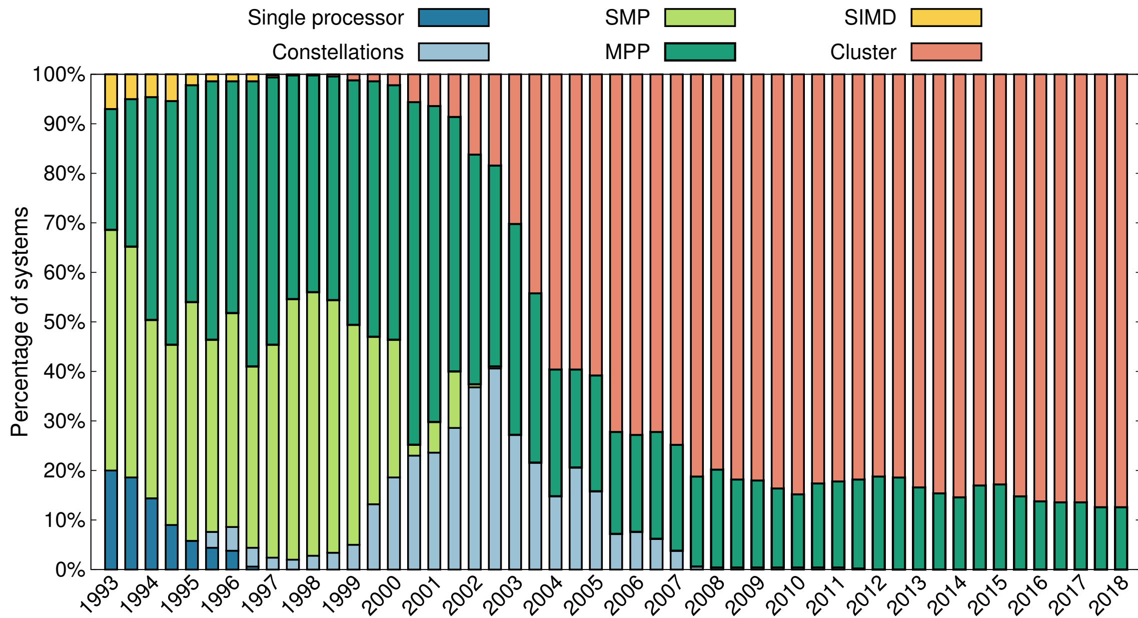


Figure 1.3: Development of architectures over time in the systems included on the TOP500 lists.

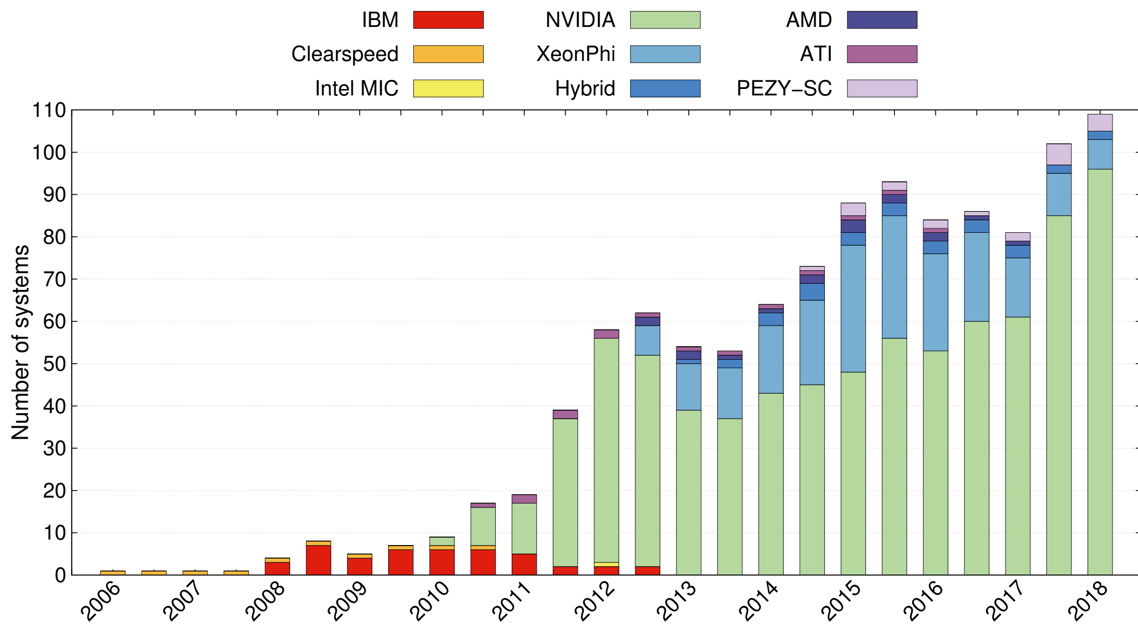


Figure 1.4: Systems presenting accelerators on the TOP500 lists.

memory architectures. MPI provides essential virtual topology, synchronization, and communication functionalities between a set of processes mapped to nodes/servers. Each process has its own address space, and processes communicate and synchronize by exchanging data over the network.

Shared-memory systems present multiple cores/processors that share the same address space. OpenMP [97] is the de-facto standard for parallel programming on these systems. The OpenMP specification defines a collection of compiler directives, library routines and environment variables that implement multithreading with the fork-join model. A main thread runs the sequential parts of the program, while additional threads are forked to execute parallel tasks. Threads communicate and synchronize by using the shared memory.

Current HPC systems are clusters of multicore nodes (architecture illustrated in Figure 1.5) that can benefit from the use of a hybrid programming model, in which MPI is used for the inter-node communications while a shared-memory programming model, such as OpenMP, is used intra-node [62, 127]. Even though programming using a hybrid MPI-OpenMP model requires some effort from application developers, this model provides several advantages such as reducing the communication needs and memory consumption, as well as improving load balance and numerical convergence [62, 106].

Heterogeneous applications are those capable of exploiting more than one type of computing system, gaining performance not only by using CPU cores but also by incorporating specialized accelerator devices such as GPUs or Xeon Phis. Accelerator devices are computing systems that cannot operate on their own and to which the CPUs can offload computations. Among the large number of frameworks for the development of applications that exploit heterogeneous devices, OpenCL [66] is the most widely supported and, thus, the one that provides the largest portability across different device families. Figure 1.6 depicts a general view of the hardware model. It is comprised of a host CPU, where the sequential parts of the execution are run, and a set of attached accelerator devices where the parallel tasks are run. Accelerator devices are comprised of several processing elements. The processing elements execute Single Instruction, Multiple Data (SIMD) operations so that the same instruction is executed simultaneously on different data in several processing elements. The host is responsible for allocating and transferring to the memory of

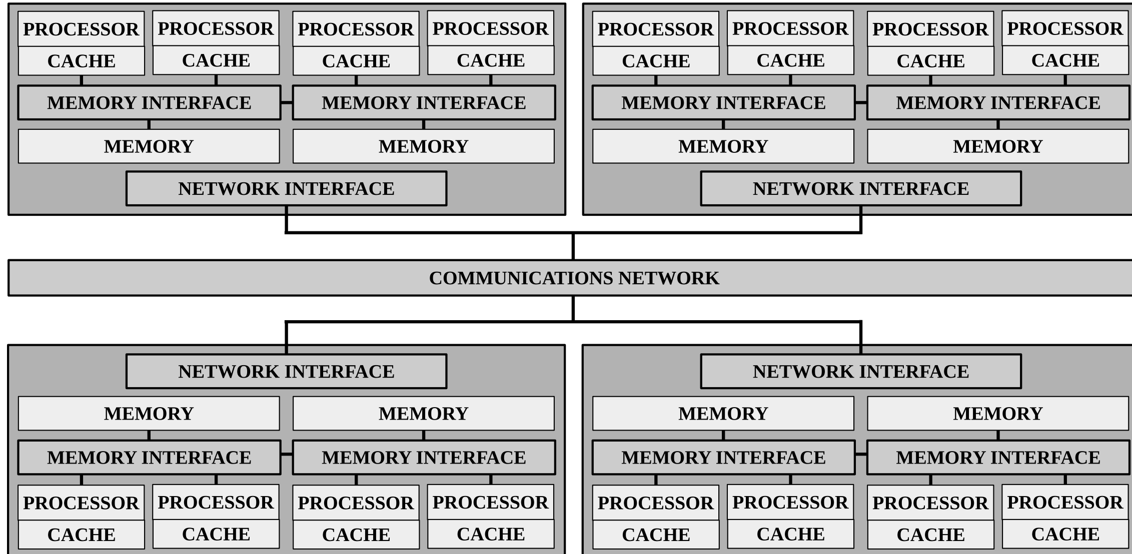


Figure 1.5: Simplified diagram of a hybrid distributed shared memory architecture.

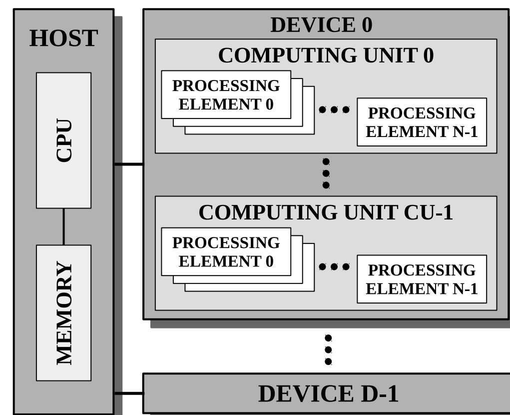


Figure 1.6: Simplified diagram of a heterogeneous architecture.

the devices the data necessary for the offload computations.

The growing popularity of accelerator devices in HPC systems has also contributed to their exploitation cooperatively with other programming models, such as MPI [60, 67], leading to efforts to extending MPI with accelerator awareness [61, 136].

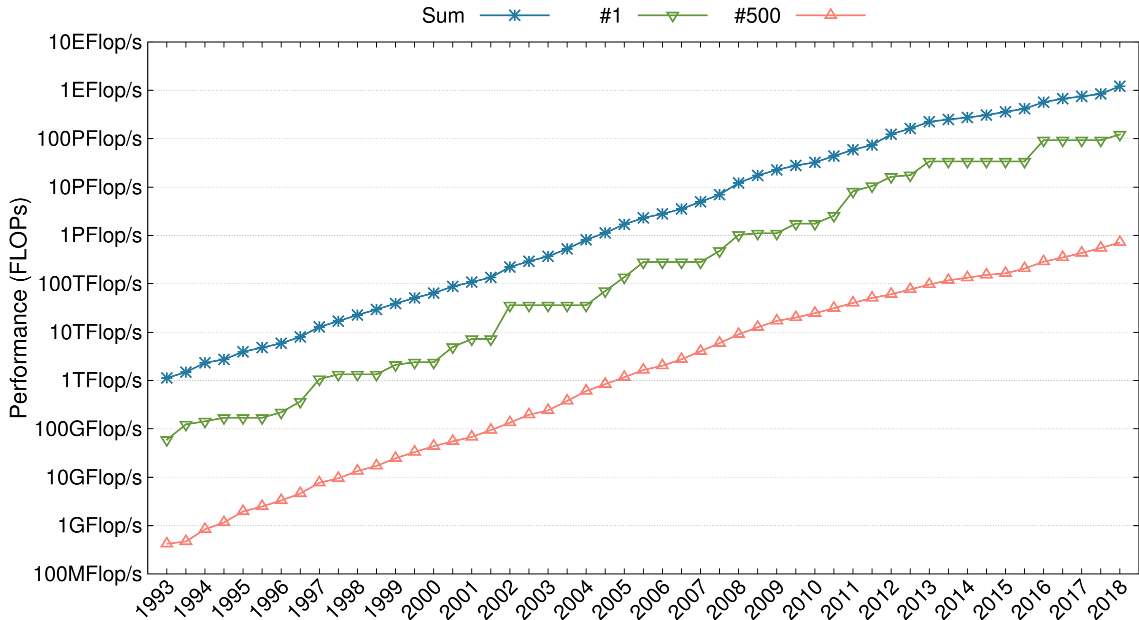


Figure 1.7: Performance development over time: aggregated performance of all the systems, and performance of first (#1) and last (#500) ranked systems on the TOP500 lists.

1.2. Fault Tolerance on HPC Applications

The rapid increase in the computational demands of science has lead to a growth in the performance offered by supercomputers. This trend is shown in Figure 1.7, which depicts the exponential growth of supercomputing power as recorded by the TOP500 list in Floating Point Operations per Second (FLOPs) over time. The performance of these systems is measured using the Linpack Benchmark [43]. In the near future, the exascale era is expected to be reached, building supercomputers comprised of millions of cores and able to perform 10^{18} operations per second. This is a great opportunity for HPC applications, however, it is also a hazard for the completion of their execution. Recent studies show that, as HPC systems continue to grow larger and include more hardware components of different types, the meantime to failure for a given application also shrinks, resulting in a high failure rate overall. Even if one computation node presents a failure every one century, a machine with 100 000 nodes will encounter a failure every 9 hours on average [42]. More alarming, a machine built up with 1 000 000 of those nodes will be hit by a failure every 53 minutes on average. Di Martino et al. [40] have studied the Cray supercomputer Blue

Waters during 261 days, reporting that 1.53% of applications running on the machine failed because of system-related issues. Moreover, applications using heterogeneous nodes displayed a higher percentage of failures due to system errors, which also grew larger when scaling. Overall, failed applications noticeably run for about 9% of the total production node hours. The electricity cost of not using any fault tolerance mechanism in the failed applications was estimated at almost half a million dollars during the studied period of time. Future exascale systems will be formed by several millions of cores, and they will present higher failure rates due to their scale and complexity. Therefore, long-running applications will need to rely on fault tolerance techniques, not only to ensure the completion of their execution in these systems, but also to save energy. However, the most popular parallel programming models that HPC applications use to exploit the computation power provided by supercomputers, lack fault tolerance support.

1.2.1. Faults, Errors, and Failures

The terminology used in this thesis follows the taxonomy of Avižienis and others [7, 26, 118], summarized in Figure 1.8. Faults (e.g. a physical defect in the hardware) can cause system errors, that is, system's incorrect states. Errors may propagate and lead to failures when they cause the incorrect service of the system, i.e., an incorrect system's functionality and/or performance that can be externally perceived. Faults can be active or inactive, depending on whether or not they cause errors; and permanent or transient, depending on whether or not their presence is continuous in time. Resilience is defined as the collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults.

Hardware faults correspond with physical faults, i.e. permanent or transient faults in any of the components of the system and can result in: (1) Detectable Correctable Error (DCE), (2) Detectable Uncorrectable Error (DUE), and (3) Silent Error (SE) or Silent Data Corruption (SDC). DCEs are managed by hardware mechanisms such as Error Correcting Codes (ECCs), parity checks, and Chipkill-Correct ECC, and are oblivious to the applications. DUEs can lead to the interruption of the execution, while SDCs can lead to a scenario in which the application returns

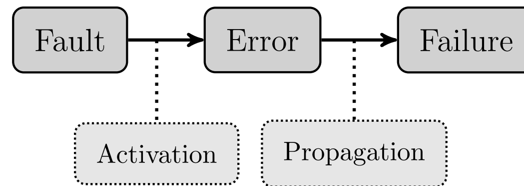


Figure 1.8: Relation between faults, errors, and failures.

incorrect results, although the user might not be aware of it.

Software faults can result in: (1) pure-software errors, (2) hardware problems mishandled by software, and (3) software causing a hardware problem. Pure-software errors correspond with classical correctness issues (such as incorrect control flows), concurrency errors (concurrent code is hard to develop and debug), and performance errors (originated by resource exhaustion that can lead to actual crashes due to timeouts). Examples of the second category correspond with node failures not being handled by software at other nodes, or a disk failure causing a file system failure. Finally, software can trigger an unusual usage pattern for the hardware, causing hardware errors.

Most of the research in this thesis (Chapters 2 to 5) is focused on fail-stop failures, which interrupt the execution of the application, both derived from hardware errors and software errors. In addition, Chapter 6 is focused on handling errors originated by transiently corrupted bits on the DRAM and the SRAM before they cause failures.

1.2.2. Checkpoint/Restart

From all the techniques focused on limiting the impact of fail-stop failures, Checkpoint/restart [42, 46] is the most popular. With this technique, each process in the application periodically saves its state to stable storage into checkpoint files that allow the restart on intermediate states of the execution in case of failure.

A process checkpoint is characterized by the software stack level where it is created, and by how it is generated and stored. At the lowest level, system-level checkpointing tools, such as operating system extensions like BLCR [55], provide a transparent checkpointing of the process. These solutions treat the application as a

black box and need no effort from the user or programmer to obtain fault tolerance support. However, they save the entire state of the process (including CPU registers, the application stack, etc). This detriments portability because the restart needs to take place on the same hardware and software environment. In addition, system-level checkpointing couples the checkpoint overhead to the memory footprint of the process. On the other hand, application-level approaches checkpoint only the critical data in the application. These techniques exploit application-specific knowledge, whether provided by the user or programmer [12] or from compilers that analyse the application code [22, 72, 112]. Application-level checkpointing contributes towards reducing the checkpoint size (and thus, the checkpointing overhead) while it can generate portable checkpoint files that enable the restart on different machines. Besides, with this approach, the user can potentially change the behaviour of the application during the restart. For instance, the application can be adapted for the best exploitation of the restart resources. Alternatively, checkpoint/restart techniques can be used to avoid the repetition of computation, e.g, avoiding a costly initialization step in a simulation by restarting the application to the state after the initialization, enabling running different simulations by changing other parameters. Regarding how checkpoints are generated and stored, different works in the literature study techniques to improve both performance and reliability. This is the case of asynchronous generation of checkpoints [73], which reduces the checkpointing overhead by dumping the state files to stable storage in background, and multilevel checkpointing [12], which exploits different levels of the memory hierarchy. Moreover, different optimization techniques focus on the reduction of the amount of checkpointed data to further reduce the checkpointing cost, such as incremental checkpointing [1, 52], data compression of the checkpoint files [72, 102], and memory exclusion [101].

Parallel applications introduce complexities in the checkpointing protocol. In distributed-memory applications, inter-process dependencies preclude the recovery of individual processes independently. Instead, a successful recovery requires the application state to be restored from a consistent global image. For this purpose, all the processes in the application must identify the most recent consistent set of checkpoint files that correspond with a consistent global image of the application, i.e. a valid recovery line. Coordinated checkpointing [28, 35] guarantees that the last recovery line is consistent by coordinating all processes when checkpointing.

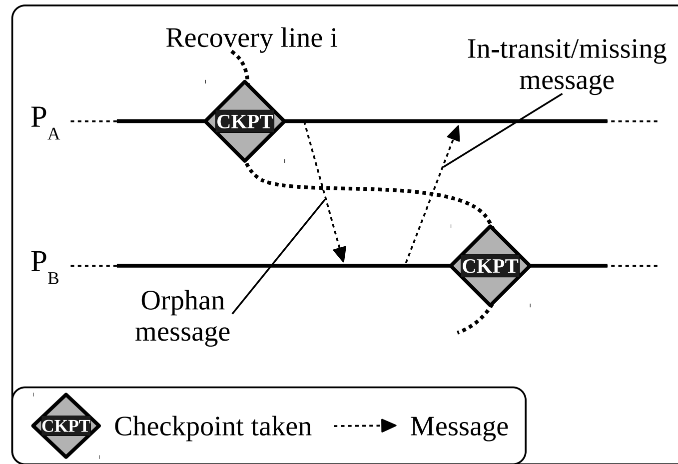


Figure 1.9: Inconsistencies caused by communications crossing a recovery line.

These protocols simplify the recovery and garbage collection, however, they require global coordination among all the application processes and force the rollback of all the processes upon failure. On the other hand, uncoordinated checkpointing allows processes to checkpoint independently. This enables checkpointing when it is more convenient, e.g. when less data needs to be saved. However, the most recent set of checkpoint files generated by each process may not represent a consistent global state. In MPI applications, inconsistencies due to communications crossing a possible recovery line, such as *orphan* or *in-transit* messages illustrated in Figure 1.9, may force a process to rollback to a previous checkpoint file, which might also force other processes to do so. Garbage collection is more complex for uncoordinated protocols, and, even if all the checkpoint files generated during the execution are kept, interprocess dependencies can lead to a situation in which all processes need to restart from the beginning of the execution, i.e. *domino* effect, which poses an unacceptable recovery cost. Combining uncoordinated checkpointing and message logging protocols [18, 20, 88, 89, 113] avoids this problem. Message logging provides a more flexible restart since, potentially, a process can be restarted without forcing the rollback of other processes. However, the memory requirements and overhead introduced by the logging operation can represent a limiting factor.

This thesis proposes new and efficient application-level fault tolerance techniques for the most popular parallel programming models for HPC systems.

1.3. CPPC Overview

Most of the solutions proposed make use of the CPPC [112] application-level checkpointing tool. CPPC is an application-level open-source checkpointing tool for MPI applications available under GPL license at `http://cppc.des.udc.es`. It appears to the final user as a compiler tool and a runtime library.

The original proposal of CPPC provides fault tolerance to MPI applications by applying a stop-and-restart checkpointing strategy [112]: during its execution the application periodically saves its computation state into checkpoint files, so that, in case of failure, the application can be relaunched, and its state recovered using those files. As exemplified in Figure 1.10, the CPPC compiler automatically instruments the application code to obtain an equivalent fault-tolerant version by adding calls to the CPPC library. The resulting fault tolerant code for the stop-and-restart proposal can be seen in Figure 1.11. Instrumentation is added to perform the following actions:

- **Configuration and initialization:** at the beginning of the application the routines `CPPC_Init_configuration` and `CPPC_Init_state` configure and initialize the necessary data structures for the library management.
- **Registration of variables:** the routine `CPPC_Register` explicitly marks for their inclusion in checkpoint files the variables necessary for the successful recovery of the application. During restart, this routine also recovers the values from the checkpoint files to their proper memory location.
- **Checkpoint:** the `CPPC_Do_checkpoint` routine dumps the checkpoint file. At restart time this routine checks restart completion.
- **Shutdown:** the `CPPC_Shutdown` routine is added at the end of the application to ensure the consistent system shutdown.

To allow users to specify an adequate checkpointing frequency, the compiler uses a heuristic evaluation of the computational cost to place the checkpoint calls in the most expensive loops of the application. Checkpoint consistency is guaranteed by locating the checkpoint function in the first *safe point* of these loops. The CPPC

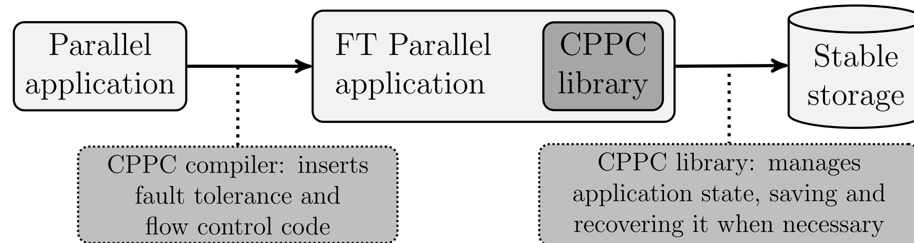


Figure 1.10: CPPC global flow.

```
1  int main( int argc, char* argv[] )
2  {
3     CPPC_Init_configuration();
4     MPI_Init( &argc, &argv );
5     CPPC_Init_state();
6
7     if (CPPC_Jump_next()) goto REGISTER_BLOCK_1;
8     [ ... ]
9
10    REGISTER_BLOCK_1:
11    <CPPC_Register(...)  block>
12    [ ... ]
13    if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1
14    [ ... ]
15
16    for(i = 0; i < nIters; i++){
17       CKPT_BLOCK_1:
18       CPPC_Do_checkpoint();
19       [ ... ]
20
21    }
22    <CPPC_Unregister(...)  block>
23    CPPC_Shutdown();
24    MPI_Finalize();
25 }
```

Figure 1.11: CPPC instrumentation example.

compiler performs a static analysis of inter-process communications and identifies safe points as code locations where it is guaranteed that there are no in-transit, nor inconsistent messages. Safe points allow CPPC to apply a *spatial coordination protocol* [111]: processes perform an uncoordinated checkpointing, generating the checkpoint files independently without the need of inter-process communications or runtime synchronization. Instead, processes are implicitly coordinated: they checkpoint at the same selected safe locations (checkpoint calls) and at the same relative

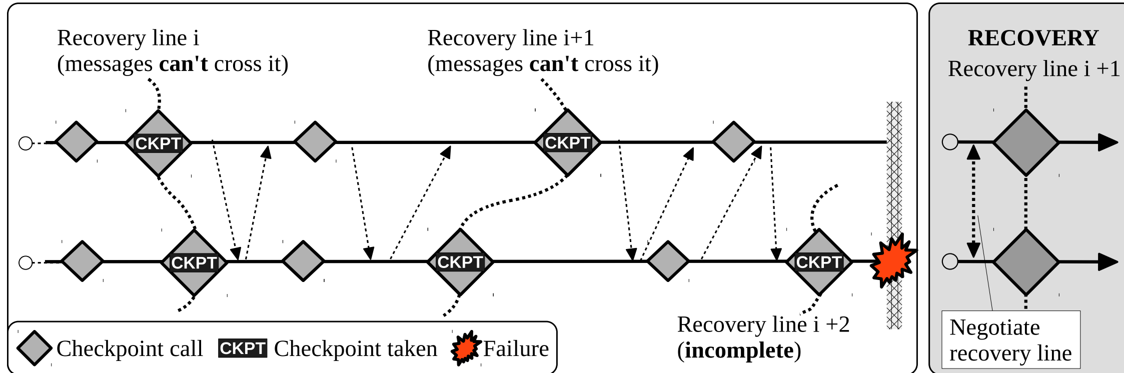


Figure 1.12: Spatial coordination protocol.

moments according to the checkpointing frequency. Figure 1.12 shows an example for a checkpointing frequency $N = 2$. All processes checkpoint at the second, fourth and sixth checkpoint calls, which are invoked by each process at different instants of time. The recovery line is formed by the checkpoint files generated by all the processes at the same safe location and at the same relative moment, thus, no communications can cross the recovery line and no communications need to be replayed during the recovery.

Upon a failure, the application is relaunched, and the restart process takes place. First, the application processes perform a negotiation phase to identify the most recent valid recovery line. Therefore, coordination is delayed until the restart operation, a much less frequent operation. Besides, processes have just been re-spawned and coordination during the recovery imposes minimum overhead. The restart phase has two parts: reading the checkpoint data into memory and reconstructing the application state. The reading is encapsulated inside the routine `CPPC_Init_state`. The reconstruction of the state is achieved through the ordered execution of certain blocks of code called RECs (Required-Execution Code): the configuration and initialization block, variable registration blocks, checkpoint blocks, and non-portable state recovery blocks, such as the creation of communicators. When the execution flow reaches the `CPPC_Do_checkpoint` call where the checkpoint file was generated, the recovery process ends, and the execution resumes normally. The compiler inserts control flow code (labels and conditional jumps using the `CPPC_Jump_next` routine) to ensure an ordered re-execution.

CPPC implements several optimizations to reduce the checkpointing overhead.

The checkpoint file sizes are reduced by using a liveness analysis to save only those user variables indispensable for the application recovery; and by using the zero-blocks exclusion technique, which avoids the storage of memory blocks that contain only zeros [34]. In addition, a multithreaded checkpointing overlaps the checkpoint file writing to disk with the computation of the application.

Also, another CPPC feature is its portability. Applications can be restarted on machines with different architectures and/or operating systems than those in which the checkpoint files were originally generated. Checkpoint files are portable because of the use of a portable storage format (HDF5 `http://www.hdfgroup.org/HDF5/`) and the exclusion of architecture-dependent state from checkpoint files. Such non-portable state is recovered through the re-execution of the code responsible for its creation in the original execution. This is especially useful in heterogeneous clusters, where this feature enables the completion of the applications even when those resources that were being used are no longer available or the waiting time to access them is prohibitive.

For more details about CPPC and its restart protocol the reader is referred to [111, 112].

Chapter 2

Application-level Checkpointing for Hybrid MPI-OpenMP Applications

Most of the current HPC systems are built as clusters of multicores, and the hybrid MPI-OpenMP paradigm provides numerous benefits on these systems. Hybrid applications are based on the use of MPI for the inter-node communications while OpenMP is used for intra-node. During the execution, each MPI process (usually one per node) creates a team of OpenMP threads to exploit the cores in that node. This chapter presents the extensions performed in CPPC to cope with fail-stop failures in hybrid MPI-OpenMP applications.

The chapter is structured as follows. Section 2.1 introduces the checkpointing of OpenMP applications. Section 2.2 describes how CPPC is modified and extended to cope with hybrid MPI-OpenMP codes. Section 2.3 presents the experimental evaluation of the proposal. Section 2.4 covers the related work. Finally, Section 2.5 concludes the chapter.

2.1. Checkpoint/Restart of OpenMP Applications

CPPC was originally developed for its operation on MPI applications. Thus, it needs to be extended to cope with fail-stop failures on OpenMP applications [83, 84]. The CPPC library is modified to allow the management of the necessary data structures for each thread and new functionalities are introduced to deal with OpenMP main features. This extension supports the creation and destruction of threads in multiple parallel regions, parallelized loops with different scheduling types, and reduction operations.

In order to checkpoint OpenMP applications, the registration of variables now distinguishes between private and shared state. For private variables, each thread saves its copy into its own checkpoint file, while only one copy of the shared variables is dumped to disk. Regarding checkpoint consistency, in OpenMP applications safe points are defined as code locations outside or inside of an OpenMP parallel region, as long as they are reachable by all running threads. The shared state in OpenMP applications requires a coordinated checkpointing protocol to ensure the consistency of all the saved variables: all threads must checkpoint at the same time, guaranteeing that none of them modify the shared state when it is being saved, ensuring that both private and shared variables are consistently checkpointed, and allowing the recovery of a consistent global state. In the coordination protocol used, when a thread determines that a checkpoint file must be generated (according to the user-defined checkpointing frequency), it forces all the other threads to do so in the next `CPPC_Do_checkpoint` call. The coordination protocol, illustrated in Figure 2.1, is implemented using OpenMP barrier directives, which both synchronize threads and flush the shared variables, guaranteeing a consistent view of memory when checkpointing. Note that the inclusion of barriers in the checkpoint operation can interfere with other barriers present in the application (including implicit ones present in some OpenMP directives, such as the `for` directive). Those barriers are thus replaced with a call to `CPPC_Barrier`, a library routine that includes an OpenMP barrier and a conditional call to the checkpoint routine. This strategy enables those threads blocked in a `CPPC_Barrier` call to generate checkpoint files when a faster thread forces them to do so, therefore, avoiding possible deadlocks. In contrast with MPI applications, the `CPPC_Do_checkpoint` call neither needs to be placed in the same relative point of the code nor the checkpoint file dumping is

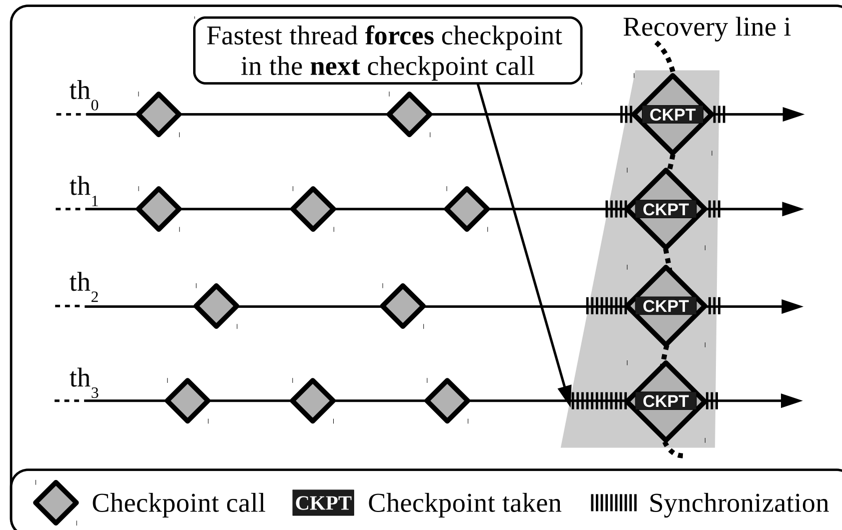


Figure 2.1: CPPC on OpenMP applications: coordinated checkpointing across OpenMP threads initiated by fastest thread.

performed in the same iteration of the loop, e.g. a checkpoint call can be placed in an OpenMP parallelized loop, in which each thread of the team runs a different subset of the iterations in the loop.

CPPC optimizes the checkpoint operation by balancing the load of this operation among the threads running the application. Commonly, the shared state of an OpenMP application corresponds with large variables that need to be included in the checkpoint files, while most private variables are small and/or do not need to be saved. CPPC reduces the checkpointing overhead by distributing both the management and the dumping to stable storage of some of the shared variables among the threads in the application. This strategy avoids bottlenecks due to the checkpointing of the shared variables, therefore, reducing the checkpointing overhead. For this purpose, CPPC applies a heuristic analysis at runtime [84]. The heuristic can be summarized as follows: 20% of the largest shared variables registered (larger than 1 MB) are distributed among the OpenMP threads executing the application only when they represent more than 80% of the checkpointed data in bytes.

Support for parallelized loops with different scheduling types and for the reduction operations is added to the library. For more details about the CPPC extension for OpenMP applications the reader is referred to [83, 84].

2.2. Checkpoint/Restart of Hybrid MPI-OpenMP Applications

This section describes the modifications performed in CPPC to cope with fail-stop failures in hybrid MPI-OpenMP applications [85]. In hybrid codes, checkpoints can be located inside or outside an OpenMP parallel region. Many hybrid codes present the optimal checkpoint location outside an OpenMP parallel region, and thus, they can be adequately checkpointed just by treating them as regular MPI programs. This is the case of those hybrid applications in which the most expensive loop contains several OpenMP parallel regions, allowing the checkpoint call to be placed between two of them. However, there are hybrid codes that do not follow this pattern, and their most expensive loops—those in which a checkpoint should be located to meet an adequate checkpoint frequency—are inside an OpenMP parallel region. Thus, these codes cannot be treated as regular MPI or OpenMP programs. To obtain the most general solution for the hybrid MPI-OpenMP programming model, its specific characteristics must be studied.

As in OpenMP and in MPI applications, in hybrid MPI-OpenMP programs checkpoints must be located at safe points. These safe points correspond with code locations that verify the conditions applied both for MPI and OpenMP applications: (1) safe points must be code locations where it is guaranteed that no inconsistencies due to MPI messages may occur, and (2) safe points placed inside an OpenMP parallel region must be reachable by all the threads in the team. The CPPC compiler automatically detects the most costly loops and inserts a checkpoint call in the first safe point of these loops. Both the relevant shared and private variables in the application are identified by the CPPC liveness analysis for their inclusion in the checkpoint files. Additionally, shared variables are distributed among the running threads for minimizing the checkpointing overhead.

2.2.1. Coordination Protocol

To ensure data consistency when checkpointing hybrid applications, we need to fulfill two conditions: (1) the state shared among the threads in one OpenMP team cannot be modified while it is being saved by any of them, and (2) both the private

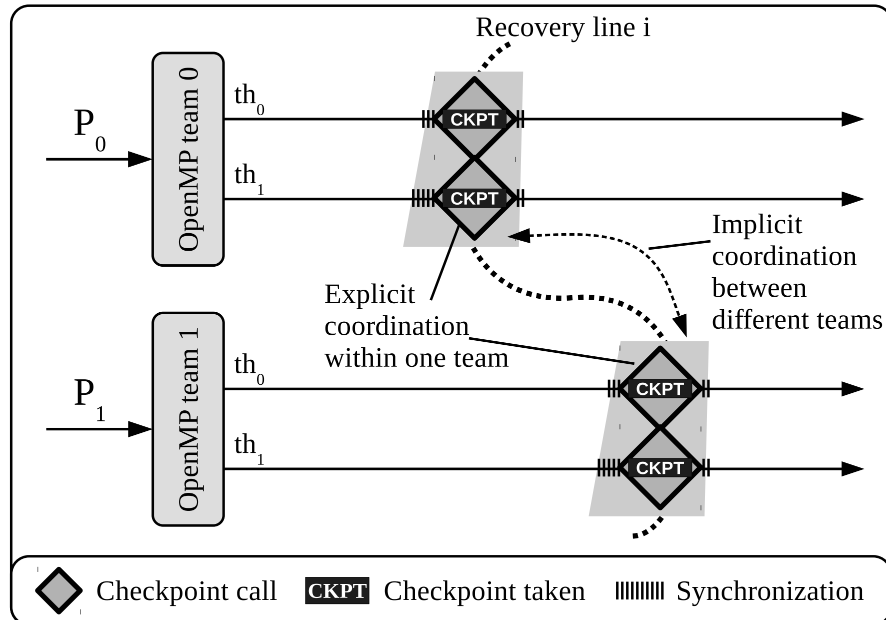


Figure 2.2: CPPC on hybrid MPI-OpenMP applications: coordinated checkpointing across OpenMP threads and uncoordinated across MPI processes.

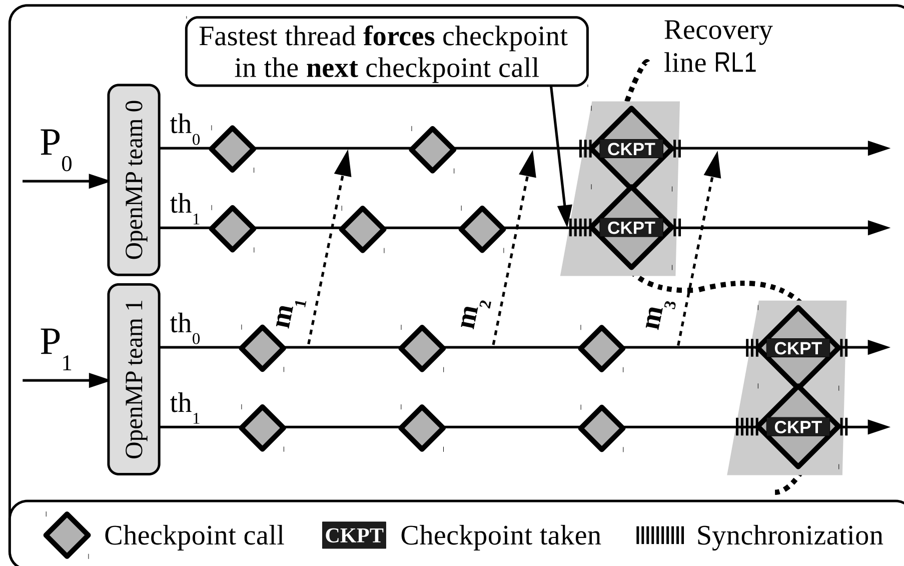
and shared variables need to be consistently checkpointing. Thus, the proposal applies a coordination protocol that ensures all threads in the same team checkpoint at the same time. From the perspective of MPI communications, consistency is guaranteed by the spatial coordination between teams of threads that is obtained by the use of safe points. Therefore, the proposed solution, illustrated in Figure 2.2, applies a coordinated checkpointing among the threads in the same OpenMP team, while an uncoordinated checkpointing is applied among different teams of threads.

The CPPC coordination protocol for OpenMP applications forces all threads in the same team to checkpoint at the same time. However, it cannot be used in hybrid MPI-OpenMP codes because it can transform a safe point into an unsafe one. This is the case of the example code shown in Figure 2.3, in which MPI communications are performed by the master thread of each process. Using a checkpointing frequency of 4 (a checkpoint file will be generated every four checkpoint calls), it is possible to reach the scenario shown in Figure 2.4a, in which the fastest thread forces every other thread within its team to take a checkpoint, breaking the spatial coordination among the different teams of threads. In this case, if the application is recovered

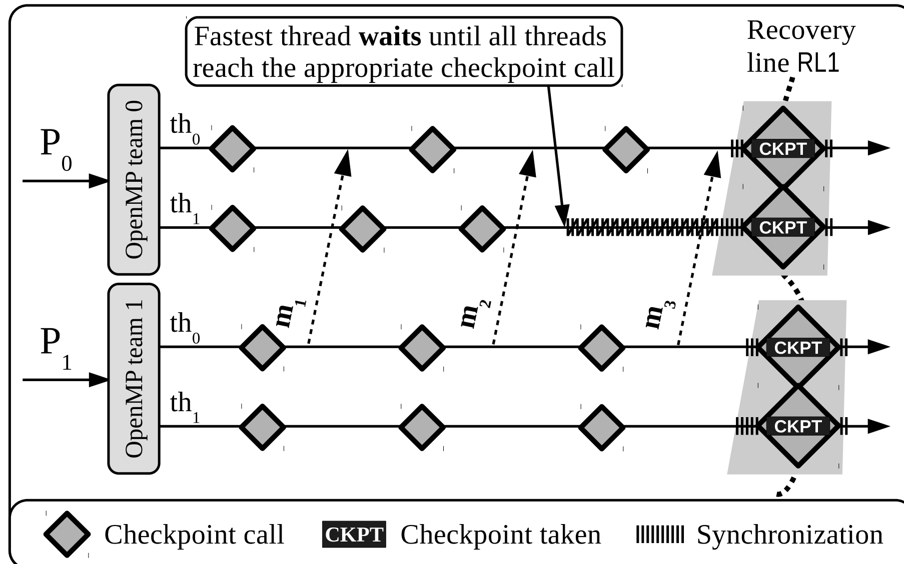
```
1  [ ... ]
2  #pragma omp parallel private(i,j) default(shared)
3  {
4     [ ... ]
5     for(i=0;i<nIters;i++){/*Application main loop*/
6        CPPC_Do_checkpoint();
7        [ ... ]
8        if(processRank==0){
9           #pragma omp master
10          {
11             for(j=1;j<Nproc;j++){
12                MPI_Irecv( [from process j] );
13             }
14             MPI_Waitall( [Irecvs from process 1...Nproc] );
15          }
16       }else{
17          #pragma omp barrier
18          #pragma omp master
19          {
20             MPI_Send( [to processRank 0] );
21          }
22       }
23       [ ... ]
24    }
25 }
26 [ ... ]
```

Figure 2.3: Example of a fault-tolerant code: OpenMP coordination protocol can break the spatial coordination between the teams of threads in hybrid applications.

from checkpoint files in recovery line `RL1`, the master thread of process 0 will expect to receive a message from process 1 that will never be sent, leading to an inconsistent global state. Therefore, to guarantee global consistency when checkpointing hybrid MPI-OpenMP applications, a new coordination protocol is implemented. In this protocol, instead of bringing forward the checkpointing by the fastest thread in the team, it is postponed. When the fastest thread determines that a checkpoint file must be generated, it waits within the `CPPC_Do_checkpoint` routine until all other threads in the same team also determine, according to the checkpointing frequency, that a checkpoint file must be generated (as shown in Figure 2.4b). This strategy may appear costly for those applications in which the threads from the same team run asynchronously, because of the stalls in the fastest threads when checkpointing. However, many times these stalls will not introduce extra overhead in the application, as they only bring forward synchronizations further along in the execution.



(a) OpenMP coordination protocol breaks spatial coordination between teams of threads.



(b) New coordination protocol for hybrid applications preserving spatial coordination.

Figure 2.4: Need for a new coordination protocol for hybrid MPI-OpenMP programs.

2.2.2. Restart Portability and Adaptability

This proposal enhances the portability features of CPPC. It does not only enable applications to be restarted in machines with different architectures and/or operating systems, but it also allows building adaptable applications. The application-level checkpointing and the portability of the state files, preserves the adaptability provided by OpenMP. This feature enables the restart process to adapt the number of OpenMP running threads for the best exploitation of the available resources, i.e., using a number of threads per team different from that of the original run. The CPPC instrumentation only maintains the original number of threads per team on the parallel region in which the checkpoint files were generated (necessary for the recovery of the application state), while further parallel regions run with the updated number of threads per team. This feature will be especially useful on heterogeneous clusters, allowing the adaptation of the application to the available resources.

2.3. Experimental Evaluation

The experimental evaluation of the proposed solution with hybrid MPI-OpenMP applications was performed at CESGA (Galicia Supercomputing Center, Spain) in the FinisTerrae-II supercomputer. The hardware platform is detailed in Table 2.1. The experiments were run spawning one MPI process per node and a team of 24 OpenMP threads per MPI process, thus, using one thread per core. Tests were performed storing the checkpoint files in a remote disk (using Lustre over InfiniBand) or in the local storage of the nodes (1TB SATA3 disk). The average runtimes of 5 executions are reported. The standard deviation is always below 3.5% of the original runtimes.

The application testbed used is comprised of two different programs. The ASC Sequoia Benchmark SPhot [5] is a physics package that implements a Monte Carlo Scalar PHOTon transport code. SNAP is a proxy application from the NERSC-8/Trinity benchmarks [94] to model the performance of a modern discrete ordinates neutral particle transport application. The configuration parameters of the testbed applications are shown in Table 2.2

Table 2.1: Hardware platform details.

Finisterrae II Supercomputer	
Operating System	Red Hat 6.7
Nodes	2x Intel Xeon E5-2680
	2.50 GHz, 12 cores per processor
	128 GB main memory
Network	InfiniBand FDR@56Gb/s & Gigabit Ethernet
Local storage	1 TB SATA3 disk
Remote storage	Lustre over InfiniBand
MPI Version	Open MPI v1.10.1
GNU Compilers	v5.3, optimization level O3

Table 2.2: Configuration parameters of the testbed applications.

Configuration parameters	
SPhot	NRUNS=3×2^{16}
SNAP	2^{19} cells, 32 groups & 400 angles

2.3.1. Operation Overhead in the Absence of Failures

In a failure-free scenario, two main sources of overhead can be distinguished: the instrumentation of the code and the checkpoint operation overhead. The instrumentation overhead corresponds to the CPPC instrumented applications without generating any checkpoint files. The checkpoint overhead is measured in the execution of the CPPC instrumented versions generating one checkpoint file, and it includes the instrumentation, the consistency protocol and the checkpoint file generation overheads.

Figure 2.5 presents the runtimes without checkpointing (but including the instrumentation overhead) and the runtimes when generating a checkpoint file in the remote and in the local disk of the nodes for different number of cores. The aggregated checkpoint file sizes (the addition of the checkpoint file sizes of every thread) are also represented in the figure. In the experiments generating a checkpoint file, the checkpointing frequency for each application is set so that only one checkpoint file is generated when 75% of the computation is completed. Note that the multithreaded dumping implemented by CPPC is being used, thus, the checkpointed data is dumped to disk in background. Both the instrumentation and checkpointing

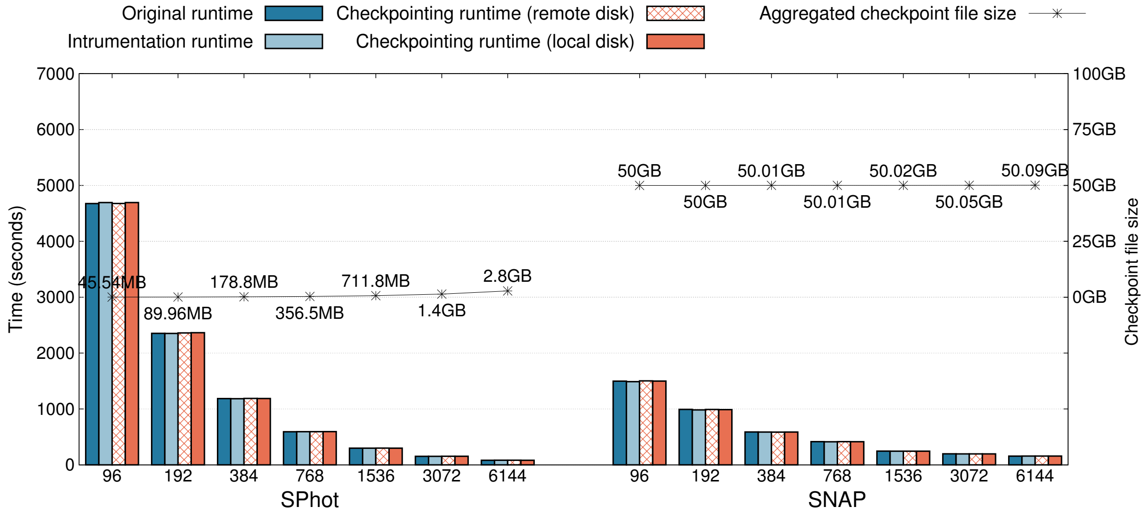


Figure 2.5: Runtimes for the testbed applications varying the number of cores.

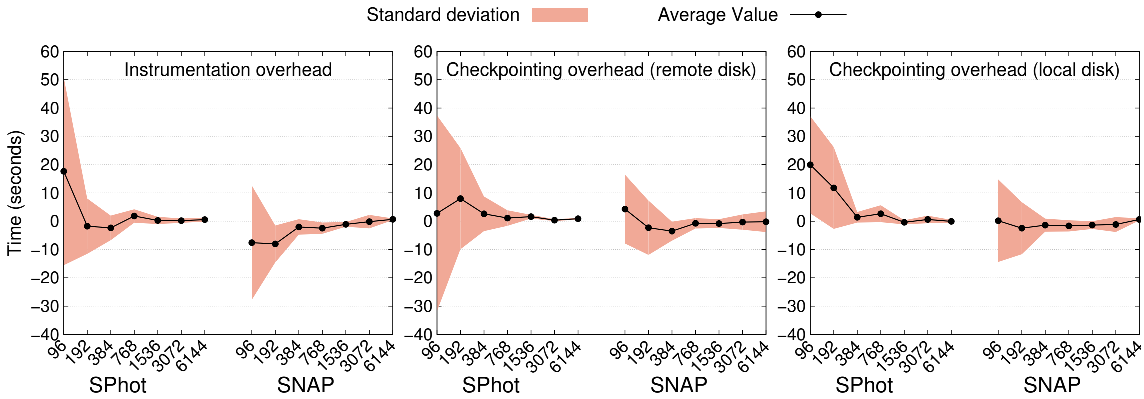


Figure 2.6: Absolute overheads varying the number of cores.

overheads are low. The instrumentation overhead is always below 0.7%. The maximum relative checkpointing overhead is 1.1% when checkpointing in the remote disk, and 0.8% when using the local storage. Figure 2.6 shows the average and standard deviation of the absolute overheads (in seconds), that is, the difference between the original parallel runtimes and the parallel runtimes of each scenario presented in Figure 2.5. Note that, when increasing the number of cores, the overheads do not increase, and the variability of the results decreases. This proves the scalability of the proposal, which applies a massively parallel checkpointing at software and hardware level: a large number of threads checkpoint small contexts and checkpointing is now spread over several processors/nodes/networks/switches/disks. In some experi-

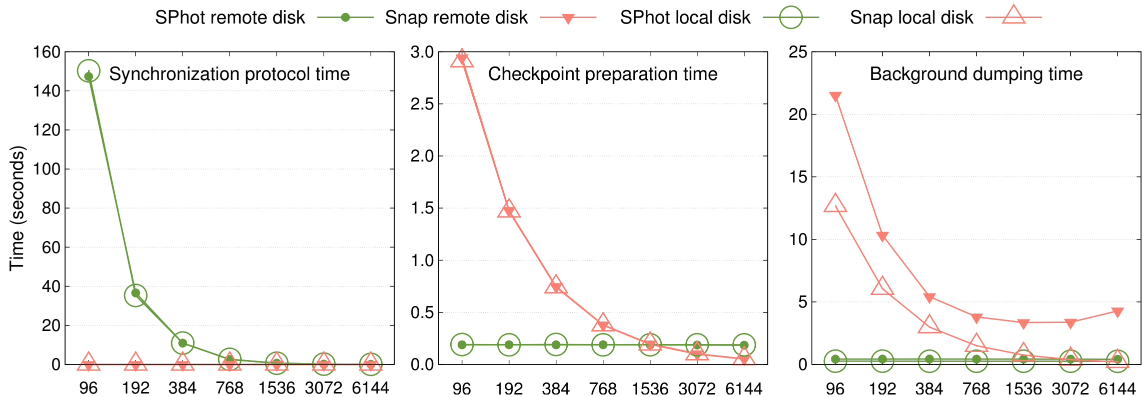


Figure 2.7: CPPC checkpointing operations times varying the number of cores.

ments, the overhead takes negative values. The CPPC instrumentation modifies the application code, thus, the optimizations applied by the compiler (and their benefit) may differ.

Figure 2.7 presents the times of the CPPC checkpointing operations: the coordination protocol, the checkpoint preparation, and the background dumping. First, the coordination protocol time measures the time spent by the synchronization between the threads created by each process to guarantee checkpoint consistency, and it is tightly tied to the application. For SNAP these times are negligible. On the other hand, tests using a small number of processes in SPhot show higher protocol times, due to the synchronization patterns between threads used in this application. However, note that threads in hybrid MPI-OpenMP applications present synchronizations at some point of their execution. In SPhot, the coordination protocol brings forward synchronizations already present in the original application code, and therefore, the protocol time is not translated into overhead. Moreover, in this application, the synchronization times decrease as more cores are used because the distribution of work among the processes and threads is more balanced when scaling out. Secondly, the checkpoint preparation arranges the checkpointed data for its dumping to disk, including the copy in memory of the data and the creation of the auxiliary threads for the data background dumping. The checkpoint preparation times are consistent with the checkpoint files sizes. For SPhot, the aggregated checkpoint file size increases with the number of processes because the individual contribution of each process remains constant when scaling out. Thus, the checkpoint preparation times remain constant for SPhot. On the other hand, for SNAP,

the application data is distributed among the processes. As such, the aggregated checkpoint file size remains almost constant, and each individual checkpoint file size decreases as more processes run the application. Therefore, the checkpoint preparation times for SNAP decrease with the number of processes, as each one of them manages less data. Finally, the checkpointed data is dumped to disk in background by the auxiliary threads. The background dumping times are also explained by the checkpoint file sizes. The use of the multithreaded dumping implemented by CPPC contributes to hide almost completely these dumping times.

2.3.2. Operation Overhead in the Presence of Failures

Figure 2.8 shows the reading and reconstructing times when restarting from the checkpoint files generated when 75% of the computation is completed. The reading phase includes identifying the most recent valid recovery line and reading the checkpoint data into memory. The reconstructing phase includes all the necessary operations to restore the application state and to position the application control flow in the point in which the checkpoint files were generated. Both reading and reconstructing times depend on the aggregated checkpoint file size. For SPhot, the aggregated checkpoint file size increases with the number of processes, and thus both the reading and reconstructing times slightly increase as more processes execute the applications. On the other hand, the aggregated checkpoint file size for SNAP remains almost constant when varying the number of running processes. Thus, as more processes are used, reading and reconstructing times decrease.

Note that the reconstructing times for SPhot are sometimes larger than those of SNAP, especially when scaling out the application. However, the aggregated checkpoint file sizes are significantly larger for the SNAP application. These results are explained by the use of the zero-blocks exclusion technique in CPPC [34]. This technique avoids the dumping to disk of the memory blocks containing only zeros. However, upon restart, these zero-blocks must be rebuilt. In SPhot, the zero-blocks exclusion technique reduces the aggregated checkpoint file size from several tens of gigabytes to less than 3 GB. Therefore, the reconstructing times are higher because of the reconstruction of the zero-blocks.

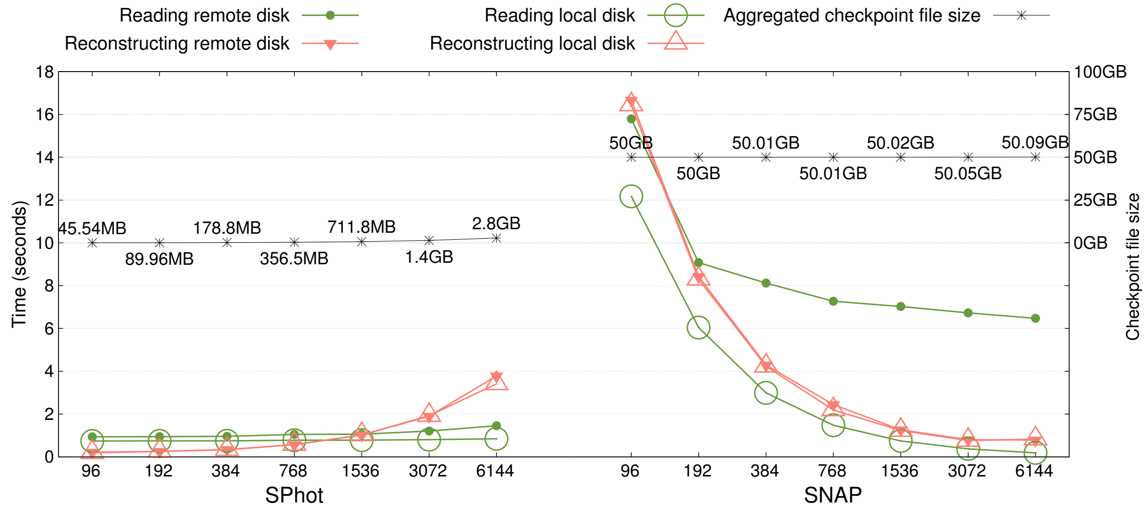
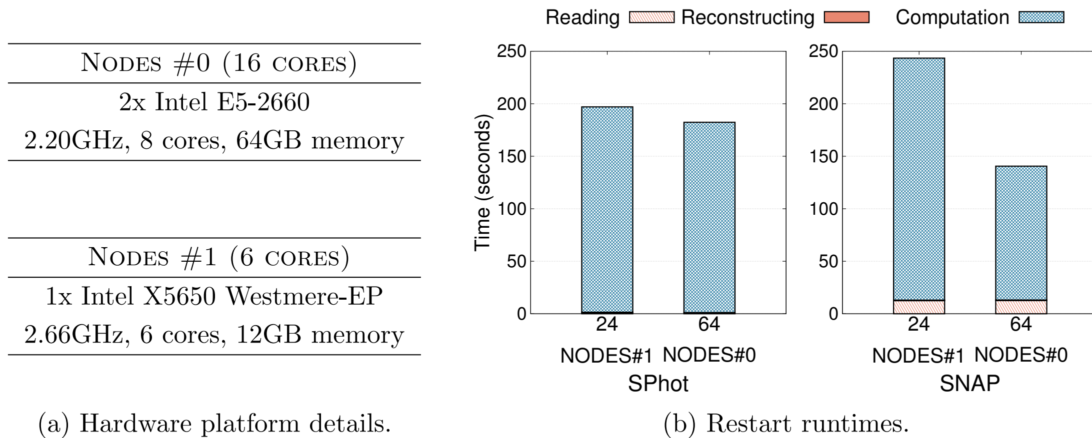


Figure 2.8: CPPC restart operations times varying the number of cores.

2.3.3. Portability and Adaptability Benefits

The independence between the CPPC checkpoint files and the MPI implementation, the OpenMP implementation, the operating system, and the machine architecture allows for restarting the execution in different machines. Furthermore, the application-level approach, enables hybrid MPI-OpenMP applications to be restarted varying the number of threads per process/team for the best exploitation of the available resources. CPPC only maintains the original number of threads during the reading and reconstructing phases, i.e., in the parallel region in which the checkpoint files were generated. On heterogeneous clusters, applications can be started in the available set of resources, to later continue their execution using a more appropriate or powerful set of resources.

The experimental study of the restart on different machines was carried out in a heterogeneous system. Figure 2.9 presents the computation nodes used and the restart runtimes in both configurations. Tests were performed using the optimal number of threads for each compute node (as many threads as cores) and the maximum number of nodes available, i.e., 4 computation nodes. The checkpoint files used for restarting were generated on `NODES#1` when 75% of the computation was completed in an NFS mounted directory. The restart runtimes include the full restart execution, that is, the restart process (reading and reconstructing the application



Nodes #0 (16 cores)
2x Intel E5-2660
2.20GHz, 8 cores, 64GB memory

Nodes #1 (6 cores)
1x Intel X5650 Westmere-EP
2.66GHz, 6 cores, 12GB memory

(a) Hardware platform details.

(b) Restart runtimes.

Figure 2.9: Recovery varying the computation nodes.

state), and the computation from the restart point to the end (25% of the total execution in these experiments). SPhot was run setting the parameter NRUNS to 2^{14} and SNAP processing 2^{15} cells, 24 groups and 192 angles. For SPhot, restarting in `NODES#0` instead of using `NODES#1` shows an improvement of 7.5%. This improvement is low because most of the computation in this application is performed within the parallel region in which the checkpoint is located, and thus CPPC maintains the original number of threads in that parallel region. However, this is not the case for SNAP, which can be fully adapted to the restart nodes and presents an improvement of 43% when changing from `NODES#1` to `NODES#0`.

2.4. Related Work

In the last decades, most of the research on fault tolerance for parallel applications has focused on the message-passing model and the distributed memory systems [13, 19, 30, 120, 121, 135]. Those solutions focused on the checkpointing of shared-memory applications lack of portability, whether code portability (allowing its use on different architectures) or checkpoint files portability (allowing to restart in different machines). Hardware solutions, such as ReVive [104] and SafetyNet [119], are platform dependent, as well as [41], which proposes a checkpointing library for POSIX multithreaded programs. Other solutions generate non-portable checkpointing files that cannot be used for the restart in different machines, e.g., DMTCP [4]

stores and recovers the entire user space, while C^3 [23, 24] forces the checkpointed data recovery at the same virtual address as in the original execution to achieve pointer consistency. Additionally, Martsinkevich et al. [87] have focused on hybrid MPI-OmpSs programs.

This work provides a portable and adaptable application-level checkpointing solution for hybrid programs. The proposal allows the applications to be restarted on different systems, with different architectures, operating systems and/or number of cores, adapting the number of OpenMP threads accordingly.

2.5. Concluding Remarks

This chapter presented the extension of the CPPC checkpointing tool to cope with hybrid MPI-OpenMP applications in supercomputing clusters. The proposal provides a general application-level checkpointing approach for hybrid codes. Checkpoint consistency is guaranteed by using an intra-node coordination protocol, while inter-node coordination is avoided by performing a static analysis of communications. The proposal reduces network utilization and storage resources in order to optimize the I/O cost of fault tolerance, while minimizing the checkpointing overhead. In addition, the portability of the solution and the dynamic parallelism provided by OpenMP enable the restart of the applications on machines with different architectures, operating systems and/or number of cores, adapting the number of running OpenMP threads for the best exploitation of the available resources. The experimental evaluation, using up to 6144 cores, shows the scalability of the proposed approach with a negligible instrumentation overhead (<0.7%), and a low checkpointing overhead (below 1.1% when checkpointing in a remote disk and less than 0.8% when using the local storage of the computation nodes). The restart experiments using a different system architecture with a different number of cores confirm the portability and adaptability of the proposal. This characteristic will allow improving the use of available resources in heterogeneous cluster supercomputers.

Chapter 3

Application-level Checkpointing for Heterogeneous Applications

Current HPC systems frequently include specialized accelerator devices such as Xeon Phis or GPUs. Heterogeneous applications are those capable of exploiting more than one type of computing system, taking advantage both from CPU cores and accelerators. This chapter describes a checkpoint-based fault tolerance solution for heterogeneous applications. The proposed solution allows applications to survive fail-stop failures in the host CPU or in any of the accelerators used. As well, applications can be restarted changing the host CPU and/or the accelerator device architecture, and adapting the computation to the number of devices available during recovery. This proposal is built combining the CPPC application-level checkpointing tool, and the Heterogeneous Programming Library (HPL), a library that facilitates the development of OpenCL-based applications.

This chapter is structured as follows. Section 3.1 comments upon the main characteristics of HPL and heterogeneous computing. The proposed portable and adaptable fault tolerance solution for heterogeneous applications is described in Section 3.2. The experimental results are presented in Section 3.3. Section 3.4 covers the related work. Finally, Section 3.5 concludes the chapter.

3.1. Heterogeneous Computing using HPL

Heterogeneous systems have increased their popularity in recent years due to the high performance and reduced energy consumption capabilities provided by using devices such as GPUs or Xeon Phi accelerators. HPL [129], available under GPL license at `http://hpl.des.udc.es`, is a C++ library for programming heterogeneous systems on top of OpenCL [66]. Among the large number of frameworks for the development of applications that exploit heterogeneous devices, OpenCL is the most widely supported and, thus, the one that provides the largest portability across different device families.

Figure 3.1 depicts the general hardware model for heterogeneous computing provided by OpenCL. HPL supports this model, comprised of a host with a standard CPU and memory, to which a number of computing devices are attached. The sequential portions of the application run on the host and they can only access its memory. The parallel tasks are called kernels and they are expressed as functions that run in the attached devices at the request of the host program. Each device has one or more Processing Elements (PEs). All of the PEs in the same device run the same code and can only operate on data found within the memory of the associated device. PEs in different devices, however, can execute different pieces of code. Thus, both data and task parallelism are supported. Also, in some devices the PEs are organized into groups, called computing units in Figure 3.1, which may share a small and fast scratchpad memory, called local memory.

Regarding the memory model, the devices have four kinds of memory. First, the global memory is the largest one and can be both read and written by the host or by any PE in the device. Second, a device may have a constant memory, which can be set up by the host and it is read-only memory for its PEs. Third, there is a local memory restricted to a single group of PEs. Finally, each PE in an accelerator has private memory that neither the other PEs nor the host can access.

Since the device and host memories are separated, the inputs and outputs of a kernel are specified by means of some of its arguments. The host program is responsible for the memory allocations of these arguments in the memory of the device. In addition, the host program must transfer the data between the host and the device memory for the input arguments, and vice versa for the output arguments.

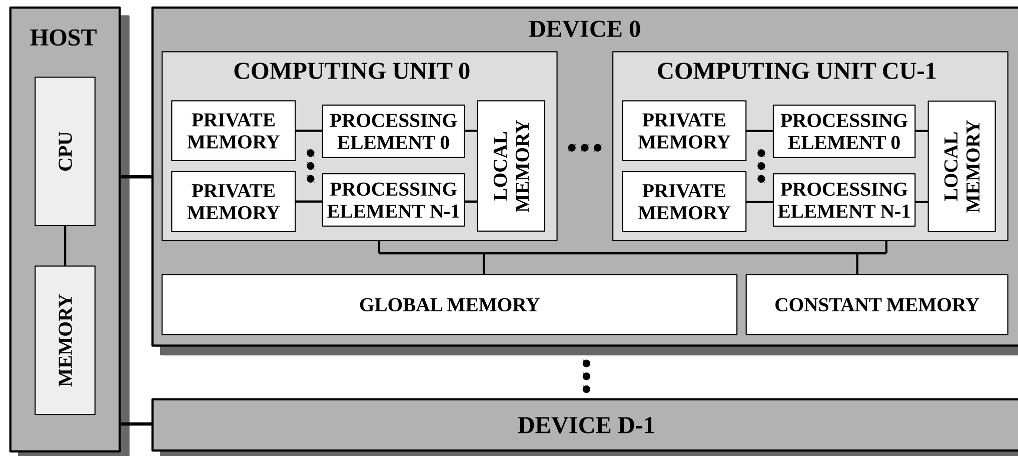


Figure 3.1: OpenCL hardware model.

The HPL library provides three main components to users:

- A template class `Array` that allows the definition of the variables that need to be communicated between the host and the devices.
- An API that allows inspecting the available devices and requesting the execution of kernels.
- An API to express the kernels. Kernels can be written using a language embedded in C++, which allows HPL to capture the computations requested and build a binary for them that can run in the chosen device. Another possibility is to use HPL as an OpenCL wrapper [130], enabling the use of kernels written in native OpenCL C in a string, just as regular OpenCL programs do, and thus, easing code reuse.

The data type `Array`<*type*, *ndim*[, *memoryFlag*]> is used to represent an *ndim*-dimensional array of elements of the C++ type *type*, or a scalar for *ndim*=0. The optional *memoryFlag* specifies one of the kinds of memory supported (Global, Local, Constant, and Private). By default, the memory is global for variables declared in the host and private for those defined inside the kernels. Variables that need to be communicated between the host and the devices are declared as Constant or Global `Arrays` in the host, while those local to the kernels can be declared Private inside the kernels or Local both in the host and inside the kernels.

```
1 void kernel_1(Array<int, 1, Global> a1,/*INPUT*/
2    Array<int, 1, Global> a2,/*INPUT*/
3    Array<float, 1, Global> tmp,/*OUTPUT*/
4    Array<float, 1, Constant> b,/*INPUT*/
5    Array<int, 0, Global> i);/*INPUT*/
6
7 void kernel_2(Array<int, 1, Global> a2,/*OUTPUT*/
8    Array<float, 1, Global> tmp,/*INPUT*/
9    Array<int, 1, Local> c,
10   Array<float, 1, Global> a3);/*INPUT&OUTPUT*/
11
12 Array<int, 1, Global> a1(N), a2(N);
13 Array<float, 1, Global> a3(N), tmp(N);
14 Array<float, 1, Constant> b(M);
15 Array<int, 1, Local> c(M);
16
17 int main( int argc, char* argv[] )
18 {
19    [ ... ]
20    /* kernel_1 and kernel_2 are associated to their
21    OpenCL C kernels using the HPL API (not shown) */
22    [ ... ]
23
24    for(i = 0; i < nIters; i++){
25        eval(kernel_1)( a1, a2, tmp, b, i);
26        eval(kernel_2)( a2, tmp, c, a3);
27    }
28
29    [ ... ]
30 }
31 [ ... ]
```

Figure 3.2: Example of an HPL application where two different kernels are invoked `nIters` times.

Figure 3.2 shows an example of an HPL application. The host code invokes the kernels with the HPL function `eval`, specifying with arguments the kernels inputs and outputs. These arguments can be `Arrays` in global, constant or local memory, as well as scalars. Global and Constant `Arrays` are initially stored only in the host memory. When they are used as kernel arguments, the HPL runtime transparently builds a buffer for each of them in the required device if that buffer does not yet exist. Additionally, the library automatically performs the appropriate data transfers to ensure that all the kernels inputs are in the devices memory before their execution. Local `Arrays` can be also used as kernels arguments. In this case,

the appropriate buffers will be allocated in the devices, however, no data transfers will be performed, as Local `Arrays` are invalidated between kernel runs. As for the output arrays, necessarily in global memory, they are only copied to the host when needed, for instance, when the `Array data` method is used. This method returns a raw pointer to the array data in the host. When this method is called, HPL ensures that the data in the host is consistent by checking if any device has modified the array, and only in that case HPL transfers the newest version of the data from the device to the host. In fact, HPL always applies a lazy copying policy to the kernels arguments that ensures that transfers are only performed when they are actually needed.

3.2. Portable and Adaptable Checkpoint/Restart of Heterogeneous Applications

As mentioned in Chapter 1, applications using heterogeneous nodes display a higher percentage of failures due to system errors, which also grow larger when scaling. Thus, it is necessary to use fault tolerance mechanisms in HPC heterogeneous applications to ensure the completion of their execution. This section describes our proposal combining CPPC and HPL to obtain adaptable fault tolerant HPC heterogeneous applications [80].

3.2.1. Design Decisions

From a fault tolerance perspective, heterogeneous applications can suffer failures both in the main processor (host) or in the accelerators. The first design decision is to determine the optimal location for the checkpoints, i.e., in which points of the application code its state should be saved to stable storage. This proposal is focused on long-running HPC heterogeneous applications to deal with fail-stop failures both in the host CPU and in the accelerator devices. The choice of a host-side checkpointing (placing checkpoints in the host code between kernels invocations) provides several performance, portability and adaptability benefits, further commented in the remain of this section. Moreover, a host-side approach guarantees in

most practical situations an adequate checkpointing frequency because the execution times of kernels are not expected to exceed the Mean Time To Failure (MTTF) of the underlying system. This expectation is based both on physical limitations and on practical observations. Regarding the limitations, a very relevant one is that, as can be seen in Table 3.2, which describes the hardware used in our evaluation, accelerators usually have memories that are considerably smaller than those of regular multi-core servers. As a result, when huge computational loads are executed in them, it is quite common to have to alternate stages of transfers to/from the host memory with stages of kernel computation to be able to process all the data. As for the practical observations, to analyze the typical kernel runtimes, we performed an experimental evaluation of the most popular benchmark suites developed with OpenCL: Rodinia [29], SHOC [37], and SNU-NPB [115]. Table 3.1 shows the results for the four most time-consuming applications in each benchmark suite. The experiments took place in System#1, described in Table 3.2 of Section 3.3. Rodinia benchmarks were executed using the default parameters, while in SHOC the largest problem available (size 4) was used. Finally, in SUN-NPB the configuration used was class B, as it was the largest problem that fits in the device memory. The studied applications execute a large number of kernels whose maximum times range from 0.32 milliseconds in application cfd to 4.1 seconds in CG, thus making host-side checkpointing a very appropriate alternative.

The next step consists in studying which application data should be included in the checkpoint files. The state of a heterogeneous application can be split into three parts: the host private state, the devices private state (data in the local and private memory of the accelerators), and the state shared among the host and the devices (data in the global and constant memory of the accelerators, which may or may not also be in the host memory). By locating checkpoints in the host code, only the host private state and the shared state need to be included in the checkpoint files. The fact that neither private nor local memory data of the devices need to be checkpointed improves the performance of the proposal because smaller checkpoint file sizes are obtained, a key factor for reducing the checkpoint overhead.

Also, the solution must guarantee the consistency of the checkpointed data. During the execution of a heterogeneous application, computations performed in the host and in the devices can overlap, as the host can launch several kernels

Table 3.1: Maximum kernel times (seconds) of the most time-consuming applications from popular benchmarks suites for heterogeneous computing (test performed on System#1 from Table 3.2).

Kernels characterization			
		Number of kernels	Max. time (seconds)
Rodinia benchmarks default parameters	cfd	14 004	0.000 32
	streamcluster	4833	0.000 69
	particlefilter	36	0.917 75
	hybridsort	21	0.103 81
SHOC benchmarks size 4	SCAN	15 360	0.001 38
	Spmv	10 000	0.003 20
	MD	162	0.009 01
	Stencil2D	22 044	0.003 18
SNUNPB BENCHMARKS CLASS B	CG	8076	4.108 87
	FT	111	0.518 84
	SP	5637	0.057 43
	BT	3842	0.105 04

for their execution in the devices and continue with its own computation. Thus, a consistency protocol is needed to ensure a successful restart upon failure. The protocol must include synchronizations so that the kernels that may modify the data included in the checkpoint files are finished before the checkpointing. In addition, those checkpointed shared variables modified by the kernels must be transferred back to the host memory.

Finally, further design decisions aim to improve the portability and adaptability of the proposal in order to obtain a solution completely independent of the machine. Therefore, this proposal can be employed to restart the applications using different hosts and/or devices. The benefits of the migration to a different device architecture may include performance, however, its main advantage is the fact that it enables the execution to be completed when the original resources are no longer available or, for instance, when the waiting time to access them is prohibitive.

As checkpointing back-end, the CPPC tool was chosen because of the portable application-level approach it provides. The checkpoint files generated by CPPC allow the restart on a different host, while their size is reduced by checkpointing only the user-variables necessary for the recovery of the application, thus, reducing

the checkpointing overhead.

When placing the checkpoints in the host code, if a non-vendor specific heterogeneous framework is used, the application can be potentially recovered using a different device architecture. Moreover, if the distribution of data and computation performed by the application is not tied to the number of devices available at runtime, applications could be restarted adapting the computation to a different number of devices. However, decoupling the distribution of data and computation from the number of devices available at runtime can be a difficult task in some applications. The high programmability of HPL simplifies the implementation of programs in which the application data and computation is not tied to the available devices at runtime. This, together with the fact that HPL is not tied to any specific vendor, operating system or hardware platform, facilitates the implementation of a fault tolerance solution that enables applications to be restarted using a different device architecture and/or a different number of devices.

3.2.2. Implementation Details

While no modifications are performed in the HPL library to implement this proposal, the CPPC tool is extended to cope with the particularities of HPL applications. Given an HPL program, the CPPC compiler automatically instruments its code to add fault tolerance support by performing three major actions: inserting checkpoints, registering the necessary host private variables or shared variables, generating the appropriate consistency protocol routines, and identifying the non-portable state recovery blocks.

First, the CPPC compiler identifies the most computationally expensive loops in the host code by performing a heuristic computational load analysis [111] and the `CPPC_Do_checkpoint` call is inserted at the beginning of the body of these loops, in between kernel invocations. The most computationally expensive loops are those that perform the core of the computation, and thus take the longest time to execute, allowing the user to specify an adequate checkpointing frequency.

Once the checkpoints are located, the CPPC compiler automatically registers the necessary data for the successful recovery of the application during restart. As

established by the design of the proposal, by locating checkpoints in the host code, only the host private state and the shared state between host and devices need to be included in the checkpoint files. The CPPC compiler analyses the host code to identify which host private variables are alive when checkpointing, inserting the appropriate calls to CPPC_Register. The compiler is extended to cope with HPL Array objects, identifying which of them correspond to shared variables alive when checkpointing, and using the HPL data method to obtain a raw pointer to the data to be registered.

Regarding data consistency, both the CPPC compiler and the CPPC library have been extended. Now, the CPPC compiler automatically generates a consistency protocol routine (CONSISTENCY_<checkpoint_number>) related to each checkpoint call introduced in the application code. This routine performs the synchronizations and data transfers required to ensure the consistency of the data included in the checkpoint files. The consistency protocol routine works as a callback function: it is passed as an argument to the new routine of the CPPC library CPPC_Consistency_protocol_ref so that, when a checkpoint call triggers a checkpoint generation, the appropriate callback routine for the consistency protocol is invoked. The protocol must ensure the consistency of those registered shared variables that are passed as arguments to the kernels. These potentially inconsistent shared variables necessarily correspond with shared Arrays in global memory. The consistency protocol routine is implemented by invoking the HPL data method on those shared Arrays, which performs the necessary synchronizations and data transfers. Moreover, both synchronizations and data transfers are only performed when the host copy of the variable is inconsistent, otherwise, both operations are avoided, thus, minimizing the consistency protocol overhead.

In order to preserve the portability features of both the checkpointing and the heterogeneous framework back-ends, the CPPC compiler is extended to avoid the inclusion in the checkpoint files of the non-portable state specific to heterogeneous applications. Instead, such non-portable state is recovered by the re-execution of those blocks of code responsible for its creation in the original execution. The CPPC compiler identifies as non-portable state the setup of the available devices at runtime, as well as the kernels' definitions and compilations. As a result, the proposal is completely independent of the machine, and applications can be restarted using

different hosts and/or devices.

Figure 3.3 highlights the instrumentation generated by the compiler for the HPL application used as example in Section 3.1 (Figure 3.2). The CPPC compiler locates the checkpoint at the beginning of the main loop, generates the appropriate registration calls for the live variables, and determines that the consistency protocol must be applied only to the `Arrays` `a2` and `a3`, as no other registered `Array` may be modified by the kernels. HPL simplifies the application of the consistency protocol, as the library automatically tracks at runtime the most recently updated copy of a variable. For example, if OpenCL applications were targeted, there could exist situations in which a compile-time analysis might not be able to detect which copy of a shared variable is the most recent one. This is the case of variables that can be modified both by the host and by the device/s depending on the values of other variables. For example, if both the host and the device code modify a checkpointed variable `v1` depending of the value of another variable `v2` and the value of `v2` is not known at compile-time, the compiler cannot know which copy of `v1` is the valid one. In that situation, to ensure correctness, the compiler would need to register both the copy of the variable in the host memory and the copy in the device memory (assuming for simplicity that it is only used in one device), doubling the state included in the checkpoint files and, thus, introducing more overhead when checkpointing. HPL, instead, ensures that the correct copy of a shared variable is saved.

3.2.3. Restart Portability and Adaptability

As commented previously, the implementation of the proposal pays special attention to the preservation of the portability features of both frameworks, allowing applications to be restarted using different hosts and/or devices. Additionally, by exploiting the high programmability of HPL via the instrumentation introduced by CPPC, applications can be restarted using a different number of devices. Two types of applications can be distinguished: pseudo-malleable applications and malleable applications, the latter being able to fully adapt to a different number of devices at restart time. The CPPC compiler inserts the same instrumentation for both types of applications and it distinguishes one from another by activating a flag in the instrumentation code. The distinction between pseudo-malleability and malleabil-

```
1  void kernel_1(Array<int, 1, Global> a1,/*INPUT*/
2     Array<int, 1, Global> a2,/*INPUT*/
3     Array<float, 1, Global> tmp,/*OUTPUT*/
4     Array<float, 1, Constant> b,/*INPUT*/
5     Array<int, 0, Global> i);/*INPUT*/
6  void kernel_2(Array<int, 1, Global> a2,/*OUTPUT*/
7     Array<float, 1, Global> tmp,/*INPUT*/
8     Array<int, 1, Local> c,
9     Array<float, 1, Global> a3);/*INPUT&OUTPUT*/
10
11 void CONSISTENCY_1( ){
12    a1.data();a2.data();a3.data();
13 }
14
15 Array<int, 1, Global> a1(N), a2(N);
16 Array<float, 1, Global> a3(N), tmp(N);
17 Array<float, 1, Constant> b(M);
18 Array<int, 1, Local> c(M);
19
20 int main( int argc, char* argv[] )
21 {
22    CPPC_Init_configuration();
23    CPPC_Init_state();
24    if (CPPC_Jump_next()) goto REGISTER_BLOCK_1;
25    [ ... ]
26    REGISTER_BLOCK_1:
27    CPPC_Register(&i, ...);
28    CPPC_Register(a1.data(), ...);CPPC_Register(a2.data(), ...);
29    CPPC_Register(a3.data(), ...);CPPC_Register(b.data(), ...);
30    if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1
31    [ ... ]
32    RECOVERY_BLOCK_1:
33    /* kernel_1 and kernel_2 are associated to their
34    OpenCL C kernels using the HPL API (not shown) */
35    CPPC_Consistency_protocol_ref(&CONSISTENCY_1);
36    if (CPPC_Jump_next()) goto CKPT_BLOCK_1;
37    [ ... ]
38    for(i = 0; i < nIters; i++){
39       CKPT_BLOCK_1:
40       CPPC_Do_checkpoint();
41       eval(kernel_1)( a1, a2, tmp, b, i);
42       eval(kernel_2)( a2, tmp, c, a3);
43    }
44    [ ... ]
45    CPPC_Shutdown();
46 }
```

Figure 3.3: Fault tolerance instrumentation of an HPL application.

ity depends on the dependencies between the application data and the number of devices used, and whether or not they can be overcome. Figure 3.4 presents the different cases that are studied in the following paragraphs.

Pseudo-malleable applications are those in which the distribution of data and/or computation must be preserved when restarting, otherwise, the restart will not be successful. The CPPC compiler determines that an application is pseudo-malleable when any of the checkpointed variables has a dependency with the number of devices available at runtime. For instance, in the example code shown in Figure 3.4a the registered variable `d` has such a dependency. These applications can be restarted using a larger or smaller number of physical devices but preserving the same number of virtual devices. When fewer physical devices are used to recover the application, some of them will have to perform extra computations. When using a larger number of physical devices, some of them will not perform any computation. Thus, the load balancing between devices is not optimal. As shown in Figure 3.5, the CPPC compiler inserts the instrumentation to perform the following actions:

- Saving and recovering the number of devices originally used by the application (with the variable `orig_devices`).
- Setting the number of devices used to the original number when activating the `pseudomalleable` flag.
- Modifying all the references to particular devices to ensure they correspond with a physical device by using the `real_devices` variable.

At runtime, HPL transparently manages the data allocations and transfers into the devices memory, releasing the CPPC instrumentation of this duty.

On the other hand, malleable applications are those which can be adapted to a different number of devices during the restart, obtaining an optimal data and computation distribution while ensuring the correctness of the results. The CPPC compiler identifies malleable applications when none of the registered variables presents dependencies with the number of devices available at runtime. The high programmability of HPL simplifies the implementation of malleable applications. A typical pattern is shown in Figure 3.4b, in which an array of references `v_d` is built from a single unified image of the data, the array `d`. These references represent a particular

```
1  [ ... ]
2  int num_devices = /*Available number of devices*/
3  [ ... ]
4  Array<int, 1, Global> d(num_devices*N);
5  [ ... ]
6  for(j = 0; j < T; j++){
7    for(i = 0; i < num_devices; i++){
8      device = HPL::Device(DEV_TYPE, i);
9      eval(kernel_1).device(device)(
10                     d(Tuple(i*N, (i+1)*N-1)));
11   }
12   [ ... ]
13 }
14 [ ... ]
```

(a) Example code that can be instrumented for pseudo-malleability.

```
1  [ ... ]
2  int num_devices = /*Available number of devices*/
3  [ ... ]
4  Array<int, 1, Global> d(N);
5  Array<int, 1, Global> * v_d[MAX_GPU_COUNT];
6  [ ... ]
7  for(i = 0; i < num_devices; ++i){
8    /* Tuple builds HPL subarrays references */
9    v_d[i] = &d(Tuple(ini_p,end_p));
10 }
11 [ ... ]
12 for(j = 0; j < T; j++){
13   eval(kernel_1).device(v_devices)(v_d);
14   [ ... ]
15 }
16 [ ... ]
```

(b) Example code that can be instrumented for malleability.

Figure 3.4: Adaptability in heterogeneous applications.

distribution of data among the devices, which is actually performed by the `eval` routine. As shown in Figure 3.6, the CPPC compiler registers the single unified image of each array, which can then be split among an arbitrary different number of devices when the application is restarted. Thus, the load balancing between devices is optimal. The compiler includes the instrumentation used for pseudomalleable applications, but now the `pseudomalleable` flag is deactivated.

To achieve both pseudo-malleability or malleability, the focus is only on check-

```
1  [ ... ]
2  int pseudomalleable=1;
3  int num_devices = /*Available number of devices*/
4  int real_devices=num_devices;
5  int orig_devices=num_devices;
6  [ ... ]
7
8  REGISTER_BLOCK_1:
9  /* Register/Recover original number of devices */
10 CPPC_Register(&orig_devices,[...]);
11 if(CPPC_Jump_next() && (pseudomalleable==1)){
12   /* Force same number of devices during restart */
13   num_devices=orig_devices;
14 }
15
16 Array<int, 1, Global> d(num_devices*N);
17
18 CPPC_Register(&j, ...);
19 CPPC_Register(d.data(),[...]);
20 if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1
21 [ ... ]
22
23 RECOVERY_BLOCK_1:
24 /* kernel_1 is associated to its OpenCL C
25   kernel using the HPL API (not shown) */
26
27 CPPC_Consistency_protocol_ref(&CONSISTENCY_ROUTINE);
28
29 /* Array data has been recovered and its size */
30 /* is dependent on num_devices*/
31
32 if (CPPC_Jump_next()) goto CKPT_BLOCK_1;
33 [ ... ]
34 for(j = 0; j < T; j++){
35   CKPT_BLOCK_1:
36   CPPC_Do_checkpoint();
37   for(i = 0; i < num_devices; i++){
38     device = HPL::Device(DEV_TYPE, i%real_devices );
39     eval(kernel_1).device(device)(
40                 d(Tuple(i*N, (i+1)*N-1)));
41   }
42   [ ... ]
43 }
44 [ ... ]
```

Figure 3.5: Automatic instrumentation of pseudo-malleable applications.

```
1   [ ... ]
2   int pseudomalleable=0;
3   int num_devices = /*Available number of devices*/
4   int real_devices=num_devices;
5   int orig_devices=num_devices;
6   [ ... ]
7
8   REGISTER_BLOCK_1:
9   /* Register/Recover original number of devices */
10  CPPC_Register(&orig_devices,[...]);
11  if(CPPC_Jump_next() && (pseudomalleable==1)){
12    /* Force same number of devices during restart */
13    num_devices=orig_devices;
14  }
15
16  Array<int, 1, Global> d(N);
17  Array<int, 1, Global> * v_d[MAX_GPU_COUNT];
18  CPPC_Register(&j, ...);
19  CPPC_Register(d.data(),[...]);
20  if (CPPC_Jump_next()) goto RECOVERY_BLOCK_1
21  [ ... ]
22
23  RECOVERY_BLOCK_1:
24  /* kernel_1 is associated to its OpenCL C
25    kernel using the HPL API (not shown) */
26
27  CPPC_Consistency_protocol_ref(&CONSISTENCY_ROUTINE);
28
29  /* Block of code re-executed upon restart */
30  for(i = 0; i < num_devices; ++i){
31    /* Tuple builds HPL subarrays references */
32    v_d[i%real_devices ] = &d(Tuple(ini_p,end_p));
33  }
34
35  if (CPPC_Jump_next()) goto CKPT_BLOCK_1;
36  [ ... ]
37
38  for(j = 0; j < T; j++){
39    CKPT_BLOCK_1:
40    CPPC_Do_checkpoint();
41    eval(kernel_1).device(v_devices)(v_d);
42    [ ... ]
43  }
44  [ ... ]
```

Figure 3.6: Automatic instrumentation of malleable applications.

pointed variables, as any other data dependency with the number of available devices can be solved through regeneration by means of re-execution of code. Note that, although the adaptability of an application when its execution starts is a feature of HPL, the adaptability achieved during the restart operation is exclusively enabled by the design and implementation decisions presented in this section. The contributions to support adaptability can be summarized in three main points. First, applications are analyzed to determine how they should be modified to be restarted in different device architectures and different numbers of devices. Second, CPPC is extended to make the required changes in the source codes. Third, as the restarted code must be executed in a different way depending whether the application is pseudo-malleable or malleable, an algorithm to detect the kind of adaptability of the application is designed and implemented in the CPPC compiler.

3.3. Experimental Evaluation

The experiments presented in this section took place in the hardware described in Table 3.2. The first system has two GPUs and it is used for the experiments in Sections 3.3.1 and 3.3.2, while all the systems are used in Section 3.3.3 to show the portability and adaptability of the solution. The tests were performed writing and reading the checkpoint files from the local storage of the node (SATA magnetic disks). The CPPC version used was 0.8.1, working in tandem with HDF5 v1.8.11. The GNU compiler v4.7.2 was used with optimization level O3 in systems #1 and #2, while Apple LLVM version 7.3.0 (clang-703.0.31) is used in system#3. Each result presented in this section corresponds to the average of 15 executions.

The application testbed used, summarized in Table 3.3, is comprised of seven applications already implemented in HPL by our research group: three single-device applications (FT, Floyd, Spmv) and four multi-device applications (FTMD, Summa, MGMD, and Shwa1ls). Spmv is a benchmark of the SHOC Benchmarks suite [37]. Floyd is from the AMD APP SDK. FT, FTMD, and MGMD are benchmarks of the SNU NPB suite [115]. Summa implements the algorithm for matrix multiplication described in [128]. Finally, Shwa1ls is a real application that performs a shallow water simulation parallelized for multiple GPUs in [131]. Most of the experimental results shown in this section were carried out using Shwa1ls-Short configuration,

Table 3.2: Hardware platform details.

		System#1	System#2	System#3
Host	Operating system	CentOS 6.7	CentOS 6.7	MacOS X 10.11
	Processor	2x Intel E5-2660	2x Intel E5-2660	Intel I7-3770
	Frequency GHz	2.20	2.20	3.4
	#Cores	8 (16 HT)	8 (16 HT)	4 (8 HT)
	Mem. capacity GB	64	64	16
	Mem. bandwidthGB/s	51.2	51.2	25.6
	Compilers	GNU v4.7.2, optimization O3	GNU v4.7.2, optimization O3	LLVM v7.3.0 (clang-703.0.31)
Device	Processor	Nvidia K20m	Xeon PHI 5110P	Nvidia GeForce GTX 675MX
	Frequency GHz	0.705	1.053	0.6
	#Cores	2496	60 (240 HT)	960
	Mem. capacity GB	5	8	1
	Mem. bandwidth GB/s	208	320	115.2
	Driver	NVIDIA 325.15	Intel OpenCL 4.5.0.8	NVIDIA 310.42

Table 3.3: Testbed benchmarks description and original runtimes.

		Benchmark description	N.GPUs	Runtime (seconds)
Singledevice	FT	Fourier Transform Class B	1	43.31
	Floyd	Floyd-Warshall algorithm on 2^{14} nodes	1	260.64
	Spmv	Sparse matrix-vector product 2^{15}rows, 1e4 iters	1	153.14
Multidevice	FTMD	Fourier Transform Class B	1	51.27
			2	34.58
	Summa	Matrix multiplication NxN, N=2^{13}	1	45.50
			2	26.20
	MGMD	Multi-Grid Class B	1	26.45
			2	20.06
	Shwa1ls Short	Simulation of a contaminant 1 week, 400x400 cell mesh	1	271.48
			2	238.03
	Shwa1ls Large	Simulation of a contaminant 6 week, 800x800 cell mesh	1	10 203.89
			2	6515.67

since it presents a reasonable execution time to carry out exhaustive experiments. However, some of the experiments have also been conducted using the Shwa1ls-Large configuration, so as to show the impact of the checkpointing operation in a long-running application. The original runtimes of the testbed benchmarks on system#1

Table 3.4: Testbed benchmarks characterization.

		#GPU	Kernels			Global buff.		Local buff.		App.* data ratio
			#	Number of calls	Longest Kernel (s)	#	Size (MB)	#	Size (B)	
Singledevice	FT	1	8	111	0.637 57	10	2307	1	512	0.64
	Floyd	1	1	16 384	0.019 59	3	2048	0	0	0.60
	Spmv	1	1	10 000	0.013 69	6	410	1	512	1.00
Multidevice	FTMD	1	17	156	0.491 46	22	2560	1	736	0.80
		2	17	312	0.246 30	22	3072	1	736	0.80
	Summa	1	1	8	4.973 66	14	1536	2	256	0.92
		2	1	64	0.619 72	50	1536	8	512	0.92
	MGMD	1	43	5623	6.508 12	25	456	2	16	0.57
		2	43	17 182	3.255 74	25	466	4	32	0.57
	Shwa1ls Short	1	3	3 356 465	0.000 24	17	17	1	256	0.68
		2	3	6 712 930	0.000 25	23	17	2	256	0.81
	Shwa1ls Large	1	3	40 327 770	0.000 83	17	68	1	256	0.68
		2	3	80 655 540	0.000 72	23	68	2	256	0.81

*Application data ratio calculated as $\frac{InputData+OutputData}{InputData+OutputData+IntermediateData}$

are shown in Table 3.3, and, on average, they are only 0.4% slower than their native OpenCL equivalents. Table 3.4 further characterizes the testbed heterogeneous applications. Regarding the kernels, it shows the number of kernel functions, the total number of invocations to the kernels, and the execution time of the longest kernel in the application. Additionally, the table presents how many buffers in global and local memory are used (showing both the number and their total size), and the overall ratio between the application data inputs and outputs and the intermediate results. As can be seen in the table, the testbed applications cover a wide range of scenarios.

3.3.1. Operation Overhead in the Absence of Failures

Table 3.5 analyses the instrumentation and checkpoint overheads. First, the original runtimes (in seconds) are presented. Then, both the instrumentation overhead absolute value (the difference with the original runtimes, in seconds) and relative value (that difference normalized with respect to the original times, in percentage) are shown. The instrumentation overhead is negligible, always below a few seconds. In the application Summa running on two GPUs, the instrumentation overhead is negative, explained by the optimizations applied by the GNU compiler. Regarding the checkpoint overhead, in all the experiments the dumping frequency is set to take a single checkpoint when 75% of the computation has been completed, as this accounts for an adequate checkpointing frequency given the testbed applications runtimes. In order to assess the performance as well in long-running applications where more checkpoints would have to be done during a normal execution, we also show results for the Shwa1ls-Large application, where 10 checkpoint files (one every 17 minutes when using one GPU and one every 11 minutes when using 2 GPUs) are performed.

Table 3.5 also presents, for each experiment, the number of checkpoints taken, and their frequency, i.e., every how many iterations of the most computationally expensive loop a checkpoint file is generated, as well as the absolute and relative values of the checkpoint overhead. Note that the checkpointing overhead includes both the cost of the instrumentation and the cost of taking as many checkpoints as specified in the table. In addition to this, the times of the actions performed when a single checkpoint is taken, as well as the checkpoint file size, are included in the table under the title "Checkpoint operation analysis". Figure 3.7 presents a summary of this information: the runtimes when generating one checkpoint file are normalized with respect to the original runtimes of each application. In addition, both the consistency protocol and the checkpoint file generation times are highlighted.

The total overhead introduced in the applications when checkpointing is small. Its absolute value ranges from a minimum of 0.78 seconds for Shwa1ls running on one GPU and generating a 5.96 MB checkpoint file, to a maximum value of 6.03 seconds for Floyd when saving 2 GB of data. The checkpoint file generation overhead includes the state management operations, the copy in memory of the data, and the

Table 3.5: Instrumentation and checkpoint overhead analysis for the testbed benchmarks.

		N.GPU	Original runtime (s)	Instrum. overhead$*_1$		Checkpoints taken$*_2$		Checkpoint overhead$*_1$		Checkpoint operation analysis		
				Δ(s)	[%]	#	N	Δ(s)	[%]	Consistency protocol(s)	Checkpoint generat.(s)	File size (MB)
Singledevice	FT	1	43.31	0.65	1.5	1	15	2.56	5.9	0.140	1.835	769.54
	Floyd	1	260.64	0.31	0.12	1	12 288	6.03	2.32	4.323	4.878	2048.02
	Spmv	1	153.14	0.18	0.12	1	7500	1.95	1.27	3.320	0.980	410.15
Multidevice	FTMD	1	51.27	0.98	1.91	1	15	2.74	5.34	0.138	1.823	768.04
		2	34.58	0.58	1.67	1	15	1.79	5.17	0.145	1.827	768.04
	Summa	1	45.50	0.26	0.57	1	3	4.08	8.97	5.101	3.665	1536.07
		2	26.20	−0.09	−0.35	1	12	4.21	16.06	0.768	3.660	1536.10
	MGMD	1	26.45	0.55	2.08	1	15	1.77	6.71	0.513	1.072	300.78
		2	20.06	0.74	3.7	1	15	1.8	8.98	0.130	1.085	303.53
	Shwa1ls Short	1	271.48	0.78	0.29	1	503 470	0.78	0.29	0.003	0.022	5.96
		2	238.03	0.98	0.41	1	503 470	1.04	0.44	0.003	0.022	5.96
	Shwa1ls Large	1	10 203.89	1.18	0.01	10	733 236	2.61	0.03	0.010	0.077	21.42
		2	6567.82	1.39	0.02	10	733 236	2.33	0.04	0.013	0.082	21.41

$*_1$) Δ(s): absolute overhead in seconds. [%]: relative overhead with respect to the original runtimes.

$*_2$) #: Total number of Checkpoints taken. N: Checkpointing frequency, iterations between checkpoints.

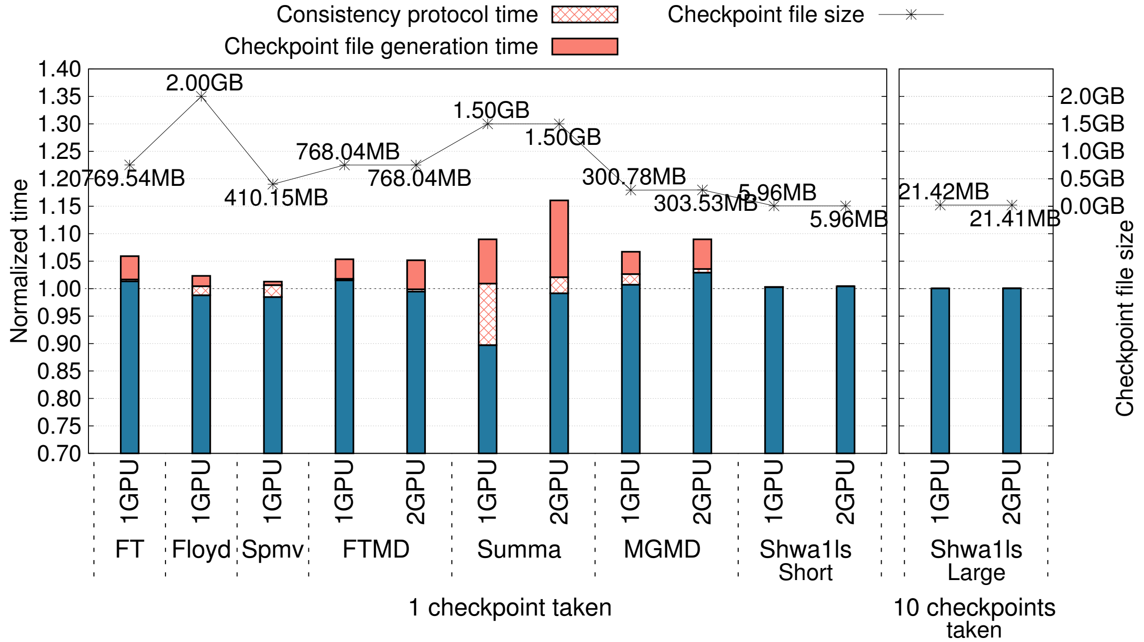


Figure 3.7: Checkpointing runtimes normalized with respect to the original runtimes for the testbed benchmarks.

creation of a thread to dump the data to disk. The actual dumping to disk is performed by the multithreaded dumping of CPPC in background, overlapping the checkpoint file writing to disk with the computation of the application. As can be observed, the checkpoint file generation overhead heavily depends on the size of the checkpoint files. Besides, the impact of this overhead obviously depends on the original application runtime.

Regarding the consistency protocol times, they are inherently dependent on the application. These times are the addition of the time spent in the synchronizations and the data transfers performed by the protocol. For the applications Floyd, Spmv and Summa, the absolute checkpoint overhead is lower than the addition of the consistency protocol and checkpoint file generation times. This situation can also be observed in Figure 3.7, where the consistency protocol is represented below the value 1 for those applications. This occurs because some operations in the original application take slightly less time when a checkpoint file is generated, due to the synchronizations performed by the consistency protocol. In Floyd, experimental results show that these synchronizations reduce the time spent by OpenCL, used as back-

end by HPL, in some inner operations. In Summa and Spmv these synchronizations reduce the time spent in synchronizations that exist further along in the original application code. This situation can happen quite frequently, since heterogeneous applications present synchronizations at some point of their execution.

3.3.2. Operation Overhead in the Presence of Failures

The restart overhead plays a fundamental role in the global execution time when failures arise. The restart process includes all the operations required to reach the point where the checkpoint file was generated. It can be broken down into two parts: reading the checkpoint file and positioning in the application. In heterogeneous applications, the positioning overhead can be split in the host and the devices positioning. The host positioning is determined by the operations that must be re-executed in the host during the restart and by the state that must be moved to the proper memory location. The devices positioning is the set-up of the devices, including the kernels compilation and the transfers of the recovery data to their memory. Figure 3.8 shows the original runtimes without fault tolerance support (left bars) and the restart runtimes when the applications are recovered from a failure using the checkpoint files generated when the application has completed 75% of its computation (right bars). These times include all the costs: the restart overhead (reading and positioning) and the application computation from the restart point to the end (25% of the total execution in these experiments).

Note that the restart overhead is always below 25 seconds, the positioning overhead being at most 3 seconds. Thus, the restart overhead is mainly determined by the reading phase, which is related to the checkpoint file sizes. Only in Shwa1ls the positioning times represent a larger percentage of the restart overhead, due to the small checkpoint file size of this application. In some applications the reading phase has a high impact due to the short runtimes, making the restart runtime close to the original runtime. However, restarting the application from a previous checkpoint is always better than starting it from the beginning of the computation, and the benefits of including fault tolerance mechanisms in long-running HPC heterogeneous applications will be unquestionable.

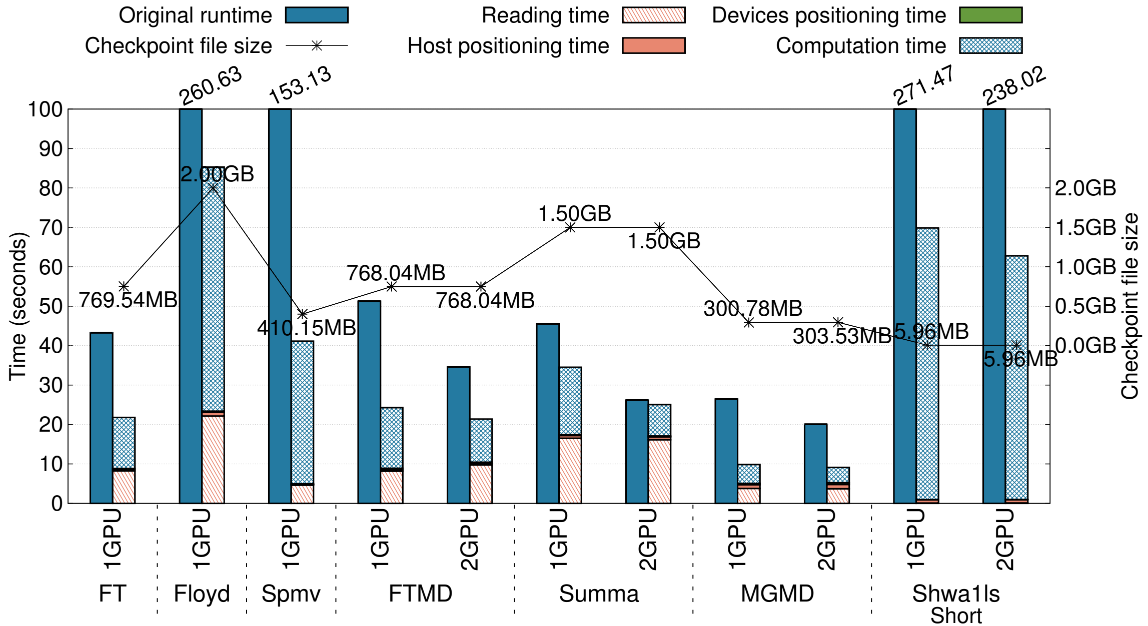


Figure 3.8: Original runtimes and restart runtimes on the same hardware for the testbed benchmarks.

3.3.3. Portability and Adaptability Benefits

The restart experiments presented in the previous subsection recovered the application using the same host and devices available during the original execution. This subsection presents the results when restarting using a different host or device architecture and/or a different number of devices. All the systems described in Table 3.2 will now be used.

Figure 3.9 shows the restart runtimes when recovering the applications using the same host and different devices: the same GPUs, Xeon Phi accelerators, and the CPU. The checkpoint file sizes are also shown. The devices positioning times vary with the device architecture, as the kernels compilation times are larger when using the Intel OpenCL driver in the Xeon Phi and CPU experiments. The computation times on the different devices are consistent with the original runtimes in the same device. For instance, in Spmv the original runtime is larger when using a GPU than when using a Xeon Phi accelerator, thus, the same tendency can be observed in the restart runtimes.

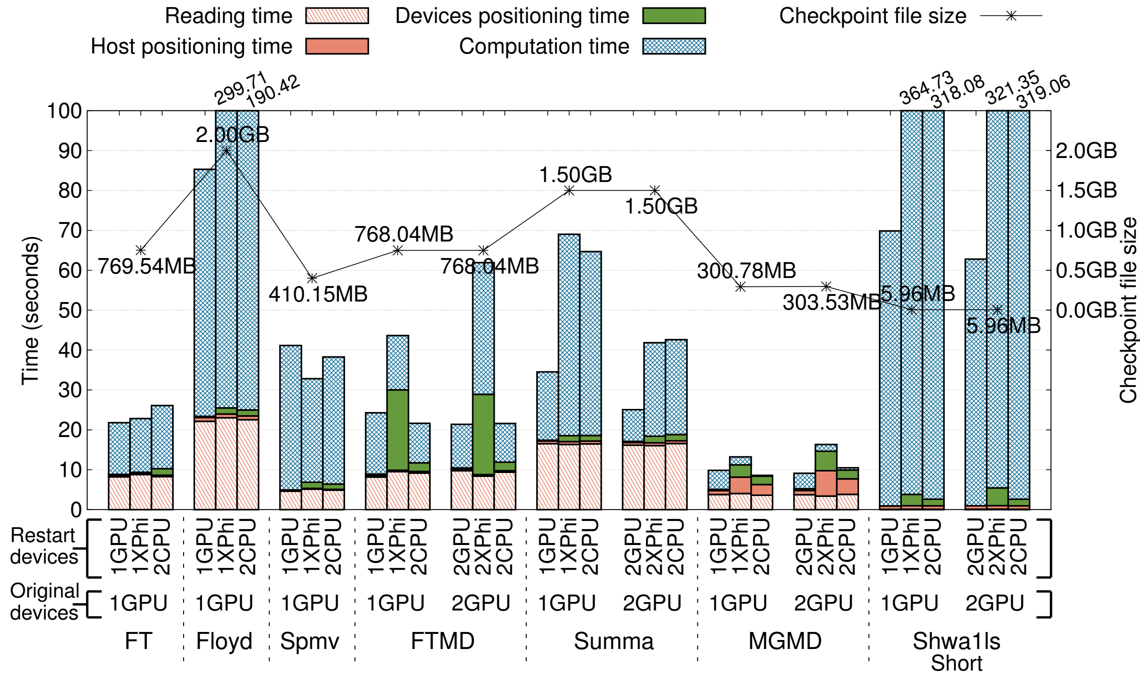


Figure 3.9: Restart runtimes for the testbed benchmarks on different device architectures.

Figure 3.10 presents the restart runtimes when using a number of GPUs that is different from the one used in the execution where the checkpoint files were generated. In our testbed benchmarks, Summa and MGMD are pseudo-malleable applications, while FTMD and Shwa1ls are fully malleable applications. As can be observed, it is possible to restart all the applications using a larger or a smaller number of devices. Besides, the restart runtimes of the malleable applications (FTMD and Shwa1ls) are not conditioned by the number of devices used for the checkpoint file generation, and, instead, these times are only influenced by the number of devices used during the restart execution, as an optimal distribution among them is performed.

Finally, Figure 3.11 shows the restart times when the application Shwa1ls is recovered in system#3 and system#1 from the checkpoint files generated in system#3. In this scenario, the application is restarted using a different host with a different operating system, a different device architecture and a different number of devices, demonstrating the portability and adaptability benefits of the proposal. The combination of CPPC and HPL allows applications to continue the execution

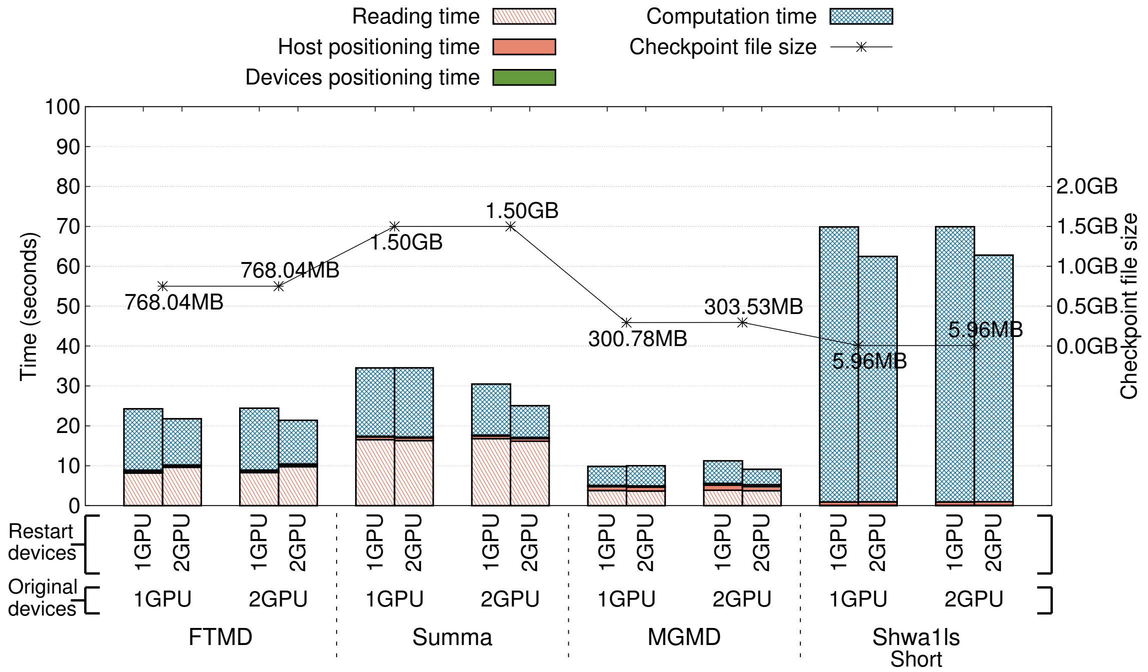


Figure 3.10: Restart runtimes for the testbed benchmarks using a different number of GPUs.

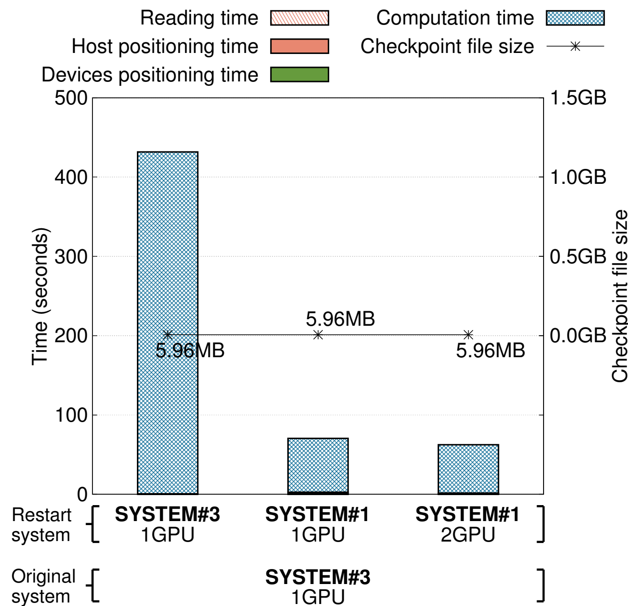


Figure 3.11: Restart runtimes for Shwa1ls using a different host, a different number and architecture of devices, and a different operating system.

after a failure occurs in one or several of the running devices, and/or in the host CPUs. In addition, in heterogeneous cluster systems, the proposed solution allows any application to start its execution using the available devices, even without using any accelerators at all, and to later continue its execution using a more appropriate set of devices, or vice-versa, depending on the availability of resources in the cluster.

3.4. Related Work

Fault tolerance for heterogeneous parallel applications is a very active research topic with many approaches published in the last years. Approaches exist based on Algorithm-Based Fault Tolerance (ABFT), in which extra information in the application is used to check the results for errors. Such a solution is presented in A-ABFT [21], which describes an ABFT proposal for matrix operations on GPUs. However, these solutions are highly specific for each particular application and algorithm.

More generic solutions are based on checkpointing. On the one hand, solutions exist that have focused on detecting soft errors, which usually do not result in a fail-stop error but cause a data corruption. A solution of this kind is VOCL-FT [100], which combines ECC error checking, logging of OpenCL inputs and commands, and checkpointing for correcting those ECC errors in the device memory that cannot be corrected by the device. On the other hand, the following paragraphs comment on other proposals that, like the one presented in this paper, are focused on fail-stop failures.

Bautista et al. [11] propose a user-level diskless checkpointing using Reed-Solomon encoding for CPU-GPU applications. The checkpointing frequency is determined by the data transfers in the host code. When the CUDA kernels and the data transfers are finished, an application-level strategy checkpoints the host memory. The main drawback of diskless checkpointing is its large memory requirements. As such, this scheme is only adequate for applications with a relatively small memory footprint at checkpoint. Besides, some GPU applications postpone the data transfers until the end of the execution, which will be translated in an unsuitable checkpointing frequency.

CheCUDA [124] and CheCL [123] are checkpointing tools for CUDA and OpenCL applications, respectively. Both are implemented as add-on packages of BLCR [55], a system-level checkpoint/restart implementation for Linux clusters. Checkpointing is triggered by POSIX signals: after receiving the signal, in the next synchronization between the host and the devices, the user data from the device memory is transferred to the host memory, and the host memory is checkpointed using BLCR. Also, both use wrapper functions to log the CUDA or OpenCL calls, in order to enable their re-execution during the restart process. CheCUDA requires no CUDA context to exist when checkpointing, as otherwise BLCR fails to restart. For this reason, the context is destroyed before every checkpoint and recreated afterwards, as well as during the restart process, using the log of CUDA calls. In CheCL a different strategy is used. The OpenCL application is executed by at least two processes: an application process and an API proxy. The API proxy is an OpenCL process and the devices are mapped to its memory space, allowing the application process to be safely checkpointable.

NVCR [95] uses a protocol similar to CheCUDA, however, it supports CUDA applications developed using the CUDA driver API and CUDA runtime API , without the need to recompile the application code. NVCR uses wrapper functions to log the CUDA calls. It is also based in BLCR, thus, as in CheCUDA, all CUDA resources have to be released before every checkpoint and recreated afterwards, as well as during the restart process, using a replay strategy to re-execute the CUDA calls from the log. However, the replay during the restart process relays on the reallocation of the memory chunks at the same address as before checkpointing, which is not guaranteed by NVIDIA and could not work correctly in certain environments.

Laosooksathit et al. [71] model and perform simulations to estimate the performance of checkpoints relying on virtualization and CUDA streams that are applied at synchronization points under the control of a model, but they offer no actual implementations.

HeteroCheckpoint [63] presents a CPU-GPU checkpointing mechanism using non-volatile memory (NVM). The application is instrumented by the programmer, explicitly indicating where and when a checkpoint is taken and which CUDA variables need to be checkpointed. CUDA streams are used to enable parallel data movement and the programmer can specify which CUDA variables are not modi-

fied in the kernels executed before a checkpoint, allowing variables to be pre-copied before a checkpoint starts. Also, redundant data between two checkpoints, as in an incremental checkpointing, do not cause unnecessary data transfers. In this proposal, when checkpointing, the host and the device are synchronized, and data is transferred from the device memory to the NVM via the host memory.

Snapify [108] is a specific solution for Xeon Phi accelerators. It is a transparent, coordinated approach to take consistent snapshots of the host and coprocessor processes of Xeon Phi offload applications. It is based on BLCR and it applies a device-side checkpointing taking into account the data private to an offload process and dealing with the distributed states of the processes that conform the offload application. When the host receives a checkpoint signal, it pauses the offload application, drains all the communication channels, captures a snapshot of the offload processes and the host, and resumes the execution. Snapify can be used for checkpoint and restart, process migration, and process swapping.

Table 3.6 summarizes the main features of the fail-stop checkpoint-based solutions commented above, specifying: the supported devices, the checkpointing granularity (application level vs. system-level) and frequency, and whether the restart process is portable (can take place in a different machine, using both different host and devices) and adaptable (can take place using a different number of accelerator devices). Most of the proposals are focused on CUDA applications, which restricts them to GPUs from a specific vendor. Five of them use a system-level approach. System-level checkpointing simplifies the implementation of a transparent solution, in which no effort from users is needed to obtain fault tolerance support, however, it results in larger checkpoint files and, thus, in larger overhead introduced in the application. Moreover, system-level checkpointing binds the restart process to the same host, thus, the restart will not be possible on different host architectures and/or operating systems, as the checkpoint files may contain non-portable host state. Furthermore, to allow the successful checkpointing of the application, the system-level strategy forces the use of an API proxy in CheCL, or the destruction and reconstruction of the devices context every time a checkpoint is taken in CheCUDA and NVCR, which have a negative influence in the checkpointing overhead.

The solution presented in this work targets HPL applications, based on OpenCL. Thus, this proposal is not tied to a specific device architecture or vendor. By com-

Table 3.6: Related work overview.

	Suported devices			Checkpointing strategy		Restart	
	Nvidia	XeonPhi	Generic	Granularity	Frequency	Portable	Adaptable
Bautista et al. [11]	✓			Application level	Timer decides when data transfer originates ckpt		
CheCUDA [124]	✓			System level (BLCR [55])	Signal triggers ckpt in next host-devs synchro.		
CheCL [123]			✓	System level (BLCR [55])	Signal triggers ckpt in next host-devs synchro.		
NVCR [95]	✓			System level (BLCR [55])	Signal triggers checkpoint		
Laosooksathit et al. [71]	✓			System level (VCCP [96])	In kernels at synchros. chosen by a model		
HeteroCkpt [63]	✓			Application level	Indicated by the user in the host code		
Snapify [108]		✓		System level (BLCR [55])	Signal triggers checkpoint		
Proposal: CPPC+HPL			✓	Application level	User-defined freq. at points chosen by the tool	✓	✓

bining an OpenCL back-end and a host-side checkpointing strategy, the approach provides several advantages to the checkpoint files: their size is reduced, and they are decoupled from the particular characteristics and number of devices used during their generation. Moreover, the application-level portable checkpointing further reduces the checkpoint files size and also decouples them from the host machine. Therefore, applications can be recovered not only using a different device architecture and/or a different number of devices, but also using a host with a different architecture and/or operating system. To the best of our knowledge, no other work provides such portability and adaptability benefits to heterogeneous applications.

3.5. Concluding Remarks

This chapter presented a fault tolerance solution for heterogeneous applications. The proposal is implemented by combining CPPC, a portable and transparent checkpointing infrastructure, and HPL, a C++ library for programming heterogeneous systems on top of OpenCL, thus, it is not tied to any particular accelerator vendor.

Fault tolerance support is obtained by using a host-side application-level checkpointing. The host-side approach avoids the inclusion in the checkpoint files of the device's private state, while the application-level strategy avoids the inclusion of non-relevant host data, thus minimizing the checkpoint files size. This approach provides portability and efficiency, while ensuring an adequate checkpointing frequency, as most of HPC heterogeneous applications execute a large number of short kernels. A consistency protocol, based on synchronizations and data transfers, ensures the correctness of the saved data. The protocol overhead is minimized by the HPL lazy copying policy for the data transfers. The host-side application-level strategy and the combination of CPPC and HPL maximizes portability and adaptability, allowing failed executions to be resumed using a different number of heterogeneous computing resources and/or different resource architectures. The ability of applications to adapt to the available resources is particularly useful for heterogeneous cluster systems.

The experimental evaluation shows the low overhead of the proposed solution, which is mainly determined by the saved state size. The restart experiments using hosts with different operating systems, different device architectures and different numbers of devices demonstrate the portability and adaptability of the proposal.

Chapter 4

Application-Level Approach for Resilient MPI Applications

Traditionally, MPI failures are addressed with stop-and-restart checkpointing solutions. The proposal of the ULFM interface for the inclusion of resilience capabilities in the MPI standard provides new opportunities in this field, allowing the implementation of resilient MPI applications, i.e., applications that are able to detect and react to failures without aborting their execution. This chapter describes how the CPPC checkpointing framework is extended to exploit the new ULFM functionalities. The proposed solution transparently obtains resilient MPI applications by instrumenting the original application code.

This chapter is structured as follows. Section 4.1 details the extension of CPPC to obtain resilient MPI applications. An optimization to improve scalability, the multithreaded multilevel checkpointing technique, is described in Section 4.2. The experimental results are presented in Section 4.3, while Section 4.4 covers the related work. Finally, Section 4.5 concludes the chapter.

4.1. Combining CPPC and ULFM to Obtain Resilience

Even though the MPI standard is the most popular parallel programming model in distributed-memory systems, it lacks fault tolerance support. Upon a single process failure, the state of MPI will be undefined, and there are no guarantees that the MPI program can successfully continue its execution. Thus, the default behavior is to abort the entire MPI application. However, when a failure arises it frequently has a limited impact and affects only a subset of the cores or computation nodes in which the application is being run. Thus, most of the nodes will still be alive. In this context, aborting the MPI application to relaunch it again introduces unnecessary recovery overheads and more efficient solutions need to be explored.

In recent years new methods have emerged to provide fault tolerance to MPI applications, such as failure avoidance approaches [33, 133] that preemptively migrate processes from processors that are about to fail. Unfortunately, these solutions are not able to cope with already happened failures.

In line with previous works [6, 48, 58], the ULFM interface [14], under discussion in the MPI Forum, proposes to extend the MPI standard with resilience capabilities to make MPI more suitable for fault-prone environments (e.g., future exascale systems). Resilient MPI programs are able to detect and react to failures without stopping their execution, thus avoiding re-spawning the entire application. ULFM includes new semantics for process failure detection, communicator revocation, and reconfiguration, but it does not include any specialized mechanism to recover the application state at failed processes. This leaves flexibility to the application developers to implement the most optimal checkpoint methodology, taking into account the properties of the target application.

The CPPC checkpointing tool is extended to use the new functionalities provided by ULFM to transparently obtain resilient MPI applications from generic SPMD codes [79]. To maximize the applicability, this proposal is non-shrinking, and it uses a global backward recovery based on checkpointing:

- Non-shrinking recovery: the number of running processes is preserved after a failure. MPI SPMD applications generally base their distribution of data

and computation on the number of running processes. Thus, shrinking solutions are restricted to applications which tolerate a redistribution of data and workload among the processes during runtime.

- Backward recovery based on checkpoint: after a failure the application is restarted from a previous saved state. Forward recovery solutions attempt to find a new state to successfully continue the execution of the application. Unfortunately, forward recovery solutions are application-dependent, and, thus, unsuitable to be applied in a general approach.

- Global recovery: the application repairs a global state to survive the failure. In MPI SPMD applications that means restoring the state of all application processes to a saved state, in order to obtain the necessary global consistency to resume the execution. Local recovery solutions attempt to repair failures by restoring a small part of the application, e.g., a single process. However, due to interprocess communication dependencies, these solutions require the use of message logging techniques for its general application.

Subsequent subsections describe the strategy upon failure: all the survivor processes need to detect the failure, so that the global communicator is reconfigured and a global consistent application state is recovered, allowing the execution to be resumed. Figure 4.1 shows an example of the new CPPC instrumentation added to perform these tasks, while Figure 4.2 illustrates the whole procedure.

4.1.1. Failure Detection

By default, when a process fails, the MPI application is aborted. The routine `MPI_Comm_set_errhandler` is used to set `MPI_ERRORS_RETURN` as the default error handler on each communicator, so that ULFM defined error codes are returned after a failure. Each MPI function call is instrumented with a call to the `CPPC_Check_errors` routine to check whether the returned value corresponds with a failure, as shown in Figure 4.1. Within the `CPPC_Check_errors` routine, whose pseudocode is shown in Figure 4.3, the survivor processes detect failures and trigger the recovery process.

```
1  int error;
2  int GLOBAL_COMM;
3
4  void function1(){
5    [ ... ]
6    error = MPI_...( ..., GLOBAL_COMM, ...);
7    if(CPPC_Check_errors(error)) return;
8    function2();/* MPI calls inside */
9    if(CPPC_Go_init()) return;
10   [ ... ]
11 }
12
13 void function2(){ /* MPI calls inside */ }
14
15 int main( int argc, char* argv[] )
16   CPPC_GOBACK_REC_0:
17   CPPC_Init_configuration();
18   if(!CPPC_Go_init()) MPI_Init()
19   CPPC_Init_state();
20
21   GLOCAL_COMM=CPPC_Get_comm();
22   error = MPI_Comm_split(GLOBAL_COMM, ..., NEW_COMM);
23   if(CPPC_Check_errors(error)) goto CPPC_GOBACK_REC_0;
24   CPPC_Register_comm(NEW_COMM);
25   if (CPPC_Jump_next()) goto CPPC_REC_1;
26   [ ... ]
27
28   CPPC_REC_1:
29   <CPPC_Register() block>
30   if (CPPC_Jump_next()) goto CPPC_REC_2;
31   [ ... ]
32
33   for(i=0;i<niters;i++){
34     CPPC_REC_2:
35     CPPC_Do_Checkpoint();
36     if(CPPC_Go_init()) goto CPPC_GOBACK_REC_0;
37     [ ... ]
38     error = MPI_...( ..., GLOBAL_COMM, ...);
39     if(CPPC_Check_errors(error)) goto CPPC_GOBACK_REC_0;
40
41     function1(); /*MPI calls inside*/
42     if(CPPC_Go_init()) goto CPPC_GOBACK_REC_0;
43   }
44   [ ... ]
45   <CPPC_Unregister() block>
46   CPPC_Shutdown();
47 }
```

Figure 4.1: Instrumentation for resilient MPI applications with CPPC and ULFM.

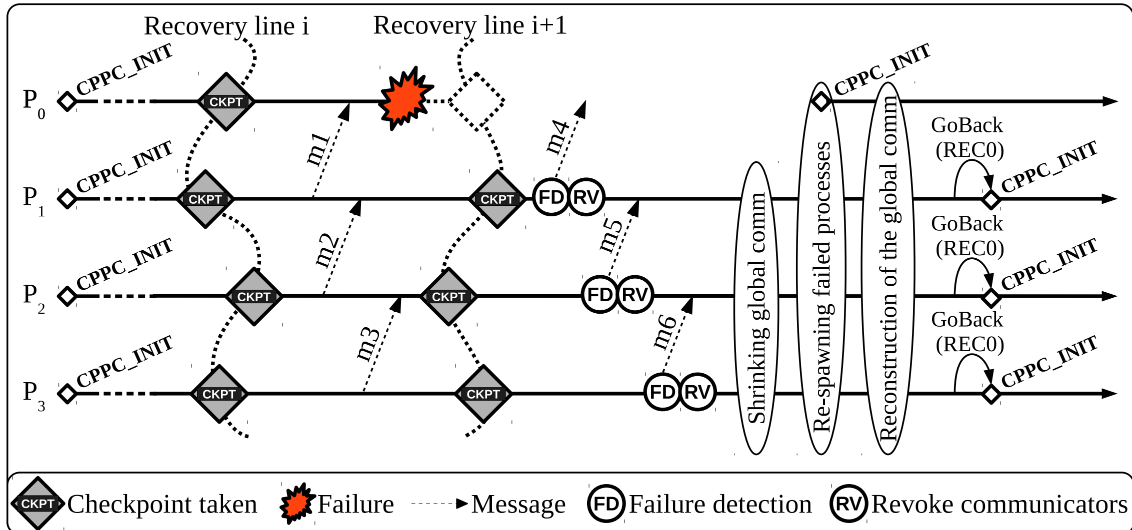


Figure 4.2: Global overview of the recovery procedure for resilient MPI applications with CPPC and ULFM.

```
1  bool CPPC_Check_errors(int error_code){
2     if( error_code == process failure ){
3        /* Revoke communicators */
4        for comm in application_communicators {
5            MPI_Comm_revoke(..., comm, ...);
6        }
7
8        /* Shrink global communicator */
9        MPI_Comm_shrink(...);
10       /* Re-spawn failed processes */
11       MPI_Comm_spawn_multiple(...);
12       /* Reconstruct communicator */
13       MPI_Intercomm_merge(...);
14       MPI_Comm_group(...);
15       MPI_Group_incl(...);
16       MPI_Comm_create(...);
17
18       /* Start the recovery */
19       return true;
20
21    }
22    return false; /* No error detected */
23 }
```

Failure detection (lines 4–6)

Reconfiguration of the global communicator (lines 8–16)

Start the recovery (lines 18–20)

Figure 4.3: `CPPC_Check_errors` pseudocode: failure detection and triggering of the recovery.

However, the failure is only detected locally by those survivor processes involved in communications with failed ones. To reach a global detection, the failure must be propagated to all other survivors by revoking all the communicators used by the application. CPPC keeps a reference to the new communicators created by the application by means of the `CPPC_Register_comm` function. When a failure is locally detected, the ULFM `MPI_Comm_revoke` function is invoked over the global (`MPI_COMM_WORLD`) and the registered communicators, assuring a global failure detection. In the example shown in Figure 4.2, process $P1$ detects that $P0$ failed because they are involved in a communication, while processes $P2$ and $P3$ detect the failure after the communicators are revoked by $P1$.

Lastly, in order to guarantee failure detection in reasonable time, the function `MPI_Iprobe` is invoked within the `CPPC_Do_checkpoint` routine. This avoids excessive delays in failure detection in absence of communications.

4.1.2. Reconfiguration of the MPI Global Communicator

In the SPMD model, the processes ranks distinguish their role in the execution, therefore, ranks must be preserved after a failure. During the recovery, not only the failed processes need to be re-spawned and the communicators need to be reconstruct, but also, the processes ranks in the communicators need to be restored.

Those communicators created by the application, which derive from the global communicator (`MPI_COMM_WORLD`), will be reconstructed by re-executing the MPI calls used for creating them in the original execution. On the other hand, the global communicator has to be reconfigured after failure detection. First, the failed processes are excluded from the global communicator using the ULFM function `MPI_Comm_shrink` and they are re-spawned by means of the MPI function `MPI_Comm_spawn_multiple`. Then, the dynamic communicator management facilities provided in MPI-2 are used to reconstruct the global communicator, so that, after a failure, the survivor processes keep their original ranks, while each one of the re-spawned ones takes over a failed process. To ensure that the correct global communicator is used, the application obtains it by means of the new `CPPC_Get_comm` function, and stores it in a global variable, as shown in Figure 4.1. The application uses this global variable instead of the named constant `MPI_COMM_WORLD`, allowing

CPPC to transparently replace it with the reconfigured communicator after a failure.

4.1.3. Recovery of the Application

A crucial step in ensuring the success of the restart is to be able to conduct the application to a consistent global state before resuming the execution. Solutions based on checkpoint, like this proposal, recover the application state rolling back to the most recent valid recovery line. However, CPPC only dumps to checkpoint files portable state, while non-portable state is recovered through the ordered re-execution of certain blocks of code (RECs). Therefore, to achieve a consistent global state, all processes must go back to the beginning of the application code so that they can re-execute the necessary RECs (including those for the creation of the derived communicators). The re-spawned processes are already in this function after the reconfiguration of the communicators. However, survivor processes are located at the point of the application code where they have detected the failure and they need to go back in the application control flow. This is performed by reversed conditional jumps introduced in two instrumentation blocks: the `CPPC_Check_errors` and the `CPPC_Go_init` blocks. While the `CPPC_Check_errors` blocks are placed after MPI calls, the `CPPC_Go_init` blocks are located after those application functions containing MPI calls. As shown in Figure 4.1, the reversed conditional jumps consist in a `goto` with a label or a `return` instruction, depending if it is placed in the main program or in an internal function. Although this strategy increases the CPPC instrumentation of the application's code, it can be directly applied both in C and Fortran programs. Alternatives, such as the use of a MPI custom handler and non-local jumps [68] would require workarounds for its usage in Fortran codes.

Once all the processes reach the beginning of the application code, a regular CPPC restart takes place. First, processes negotiate the most recent valid recovery line. In the example shown in Figure 4.2, all processes will negotiate to recover the application state from recovery line i. Then, the negotiated checkpoint files containing portable state are read and the actual reconstruction of the application state is achieved through the ordered re-execution of RECs, recovering also the non-portable state. Finally, the application resumes the regular execution when the point where the checkpoint files were generated is reached.

4.2. Improving Scalability: Multithreaded Multi-level Checkpointing

CPPC uses a multithreaded dumping to minimize the overhead introduced when checkpointing. The multithreaded dumping overlaps the checkpoint file writing with the computation of the application. When a MPI process determines that a checkpoint file must be generated (according to the checkpointing frequency), it prepares the checkpointed data, performs a copy in memory of that data, and creates an auxiliary thread that builds and dumps to disk the checkpoint file in background, while the application processes continue their execution.

With the extension of CPPC to exploit the ULFM functionalities, a multilevel checkpointing is implemented to minimize the amount of data to be moved across the cluster, thus, reducing the recovery overhead when failures arise and increasing the scalability of the proposal. As in a memory hierarchy, in which higher levels present lower access time but less capacity and larger miss rates, this technique stores the checkpoint files in three different locations:

- Memory of the compute node: each process maintains a copy in memory of the last checkpoint file generated until a new checkpoint file is built.
- Local disk: processes also save their checkpoint files in local storage.
- Remote disk: all the checkpoint files generated by the application are stored in a remote disk.

The multilevel checkpointing is performed in background by the auxiliary threads, thus, the cost of checkpointing is not increased. During recovery, the application processes perform a negotiation phase to identify the most recent valid recovery line, formed by the newest checkpoint file available simultaneously to all processes. When using the multilevel technique, the three checkpointing levels are inspected during the negotiation, and the application processes choose the closest copy of the negotiated checkpoint files.

4.3. Experimental Evaluation

The experimental evaluation of the resilience proposal is split in two parts. First, a thorough study of the overheads introduced by the operations necessary to obtain resilience is presented using up to 128 processes. Secondly, the proposal is evaluated on a different machine up to 3072 processes and comparing the benefits of the resilience global rollback versus the stop-and-restart global rollback.

The application testbed used is comprised of three benchmarks with different checkpoint file sizes and communication patterns. The ASC Sequoia Benchmark SPhot [5] is a physics package that implements a Monte Carlo Scalar PHOTon transport code. The Himeno benchmark [56] is a Poisson equation solver using the Jacobi iteration method. Finally, MOCFE-Bone [134] simulates the main procedures in a 3D method of characteristics (MOC) code for numerical solution of the steady state neutron transport equation. The CPPC version used was 0.8.1, working along with HDF5 v1.8.11 and GCC v4.4.7. The Portable Hardware Locality (hwloc) [25] is used for binding the processes to the cores. Experiments are always run using one process per core, and applications were compiled with optimization level O3.

The configuration parameters of the testbed applications and the hardware platform are detailed in Table 4.1 and Table 4.2, respectively. For these experiments, the ULFM commit 67d6cc5b9cfa beta 4 was used with the default configuration parameters and the agreement algorithm number 1. Each node of the cluster used for the experiments consists of two Intel Xeon E5-2660 Sandy Bridge-EP 2.2 GHz processors with Hyper-Threading support, with 8 cores per processor and 64 GB of RAM, interconnected to an InfiniBand FDR and a Gigabit Ethernet networks. The experiments are run spawning 16 MPI process per node, one per core. When checkpointing, each MPI process creates an auxiliary thread to dump data to disk and uses the multithreaded multilevel checkpointing technique described in Section 4.2.

4.3.1. Operation Overhead in the Absence of Failures

This section analyzes the instrumentation and checkpointing overheads. The instrumentation overhead is measured in the execution of the CPPC instrumented applications without generating any checkpoint files. On the other hand, the check-

Table 4.1: Configuration parameters of the testbed applications.

Configuration parameters	
SPhot	NRUNS=6144
Himeno	Gridsize: 512x256x256, NN=24000
Mocfe	Energy groups: 4, angles: 8, mesh: 19^3, strong scaling in space, trajectory spacing = $0.5cm^2$

Table 4.2: Hardware platform details.

Pluton cluster details	
Operating system	CentOS 6.7
Nodes	2x Intel E5-2660
	2.20 GHz, 8 cores per processor (16 HT)
	64 GB main memory
Network	InfiniBand FDR@56Gb/s & Gigabit Ethernet
Local storage	800GB HDD
Remote storage	NFS over Gigabit Ethernet
MPI version	ULFM commit 67d6cc5b9cfa (beta 4)
GNU Compilers	v4.4.7, optimization level O3

pointing runtime corresponds with the execution in which only one checkpoint is taken when 50% of the computation has completed. Figure 4.4 shows the original, the instrumentation, and the checkpointing runtimes varying the number of processes. The aggregated checkpoint file size saved to disk in each application (the addition of the checkpoint files generated by each process) is also represented in the figure. For SPhot the checkpoint file of each individual process is constant, no matter how many processes run the application, thus, the aggregated checkpoint file size increases with the number of processes. On the other hand, for Himeno and MOCFE-Bone, the application data is distributed among the processes, therefore, each individual checkpoint file size decreases as more processes run the application, and the aggregated checkpoint file size remains almost constant.

The instrumentation overhead is always very low, below 1.7 seconds. As regards the checkpointing, the maximum overhead is 7.9 seconds. The two main sources of overhead in the checkpointing operation are: the copy in memory of the checkpointed data (including the application of the CPPC zero-blocks exclusion technique), and

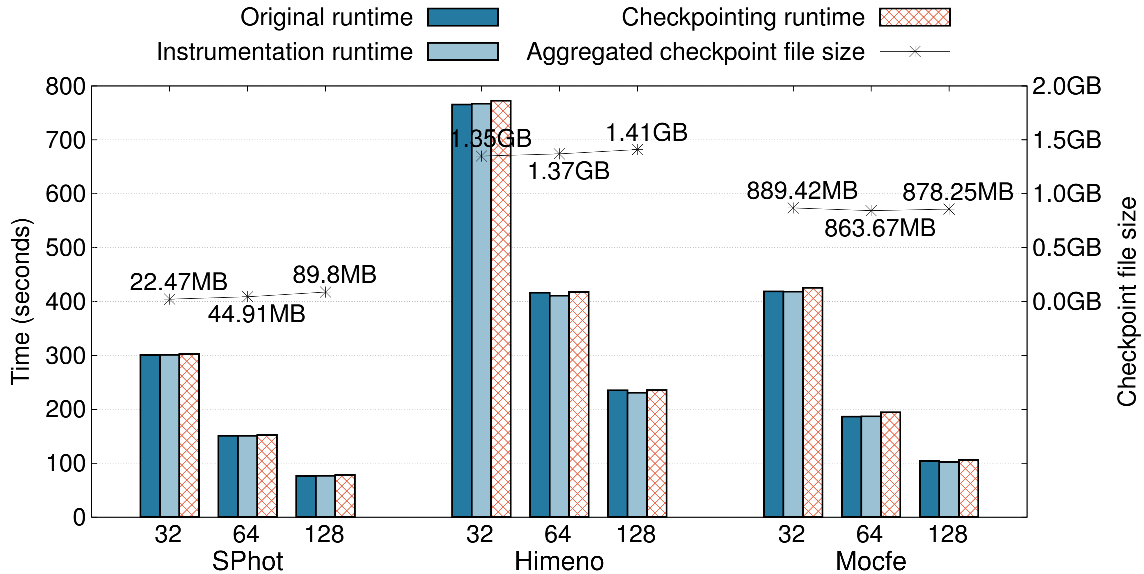


Figure 4.4: Runtimes and aggregated checkpoint file size for the testbed benchmarks when varying the number of processes.

the dumping to disk. Thanks to the use of the multithreaded checkpointing technique, the overhead of dumping the checkpoint files to disk is significantly reduced.

4.3.2. Operation Overhead in the Presence of Failures

The performance of the proposal is evaluated inserting one-process or full-node failures by killing one or sixteen MPI processes, respectively. Failures are introduced when 75% of the application has completed and the applications are recovered using the checkpoint files generated at 50% of the execution. In all the experiments, the survivor processes recover from the copy in memory of the checkpoint files. When one process fails, it is re-spawned in the same compute node, thus, it uses the checkpoint file in local storage. When a node fails, the failed processes are re-spawned in a different compute node: an already in use node, overloading it; or an spare node, pre-allocated for this purpose when scheduling the MPI job. In both cases, the failed processes use the checkpoint files located in the remote disk.

The analysis of the overhead introduced by the proposal requires the study of the different operations it involves. Figure 4.5 shows the times, in seconds, of each operation performed to obtain resilient MPI applications, indicating whether one-

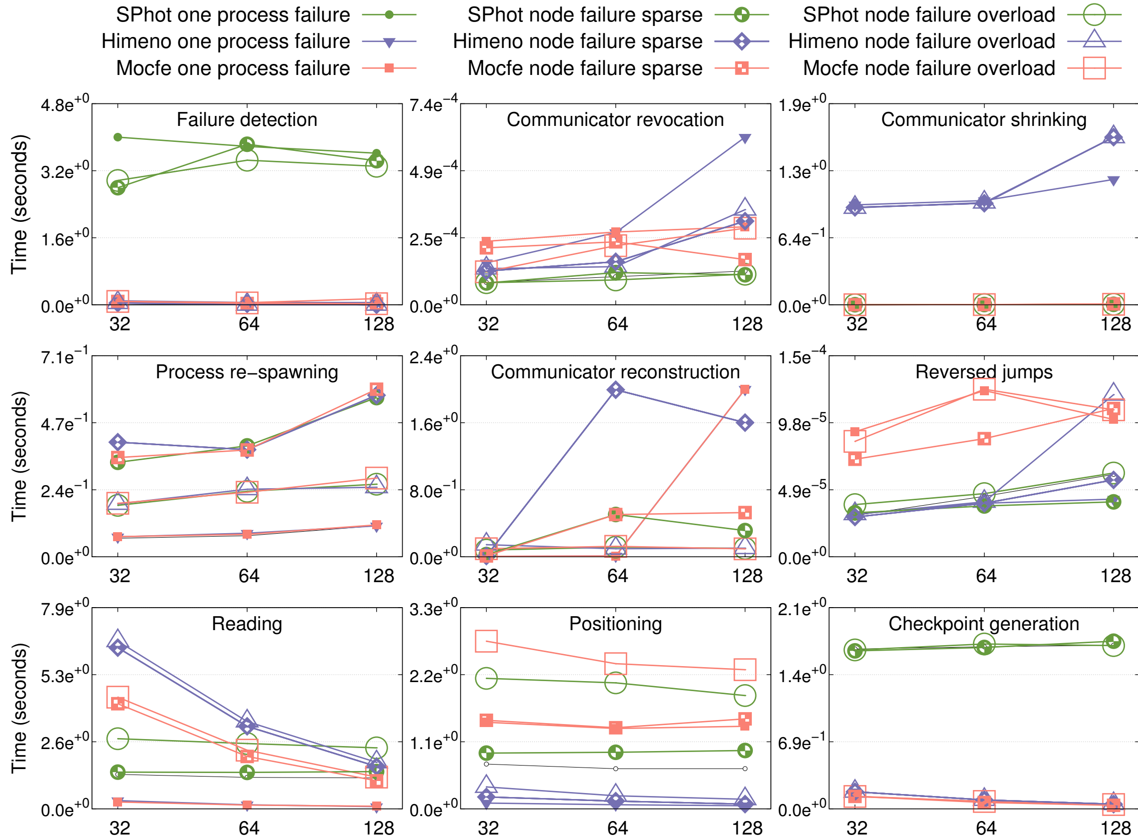


Figure 4.5: Times of the operations performed to obtain resilience.

Table 4.3: Number of calls to the MPI library per second performed by the process that less calls does, which determines the detection time.

	# MPI calls per second done by the process with fewest calls		
	32 Processes	64 Processes	128 Processes
SPhot	1.36	1.43	1.57
Himeno	815.18	1498.38	2652.24
Mocfe	1352.21	3034.68	5431.90

process or full-node failures are introduced, and for the late one, if a compute node is overloaded or a spare one is used for the recovery. Failure detection times are deeply dependent on the application, as they depend on the frequency of MPI calls. Table 4.3 shows, for each application, the average number of MPI calls per second in the process that performs the fewest MPI calls. The more the application invokes

Table 4.4: Average size (MB) of data registered by each process (including zero-blocks) and average size (MB) of the checkpoint file generated by each process (excluding zero-blocks).

Average size per process (MB): Registered data → ckpt file excluding zero-blocks			
	32 Processes	64 Processes	128 Processes
SPhot	586.23 → 0.70	586.23 → 0.70	586.23 → 0.70
Himeno	61.49 → 43.25	31.49 → 21.97	16.15 → 11.32
Mocfe	44.90 → 27.79	22.89 → 13.49	12.30 → 6.86

the MPI library, the sooner the failure will be detected. SPhot is the application with the smallest number and it also presents the largest detection time. The communicator revocation times are low, and they slightly increase with the number of processes. The same tendency can be observed in the reconfiguration operations: the communicator shrinking, the process spawning and the communicator reconstruction. Even though the processes spawning times remain low, they increase with the number of failed processes. Furthermore, spawning times are larger when processes are launched in a spare node than when the target is an already-in-use node, because all the MPI environment (including a new ORTE daemon) has to be launched in the spare one.

Regarding the CPPC operations, the time spent in the reversed conditional jumps for the recovery upon failure is negligible in all cases. The checkpoint, reading, and positioning times are consistent with the state registered by each process for its inclusion in the checkpoint files. Note that, the size of the registered state can be different from the size of the generated checkpoint file, as CPPC applies the zero-blocks exclusion technique [34], which avoids the dumping to disk of the memory blocks containing only zeros. During the checkpoint operation, all the registered data is inspected to identify zero-blocks. Also, during the reading and positioning operations, zero-blocks are reconstructed and moved to their proper memory location, respectively. Thus, checkpoint, reading, and positioning are influenced by the size of registered data. Table 4.4 shows, for each application, the average size of state registered by each process, and the average checkpoint file size that each process generates after applying the zero-blocks exclusion technique. For instance, in SPhot even though the aggregated checkpoint file size is inferior to 100 MB, the to-

tal registered state corresponds with several gigabytes before the application of the zero-blocks exclusion technique. Therefore, the CPPC operations are more costly for SPhot when comparing with applications that generate larger checkpoint files. Besides, due to the use of the multilevel checkpointing, the reading phase depends on the location of the negotiated checkpoint files. Consequently, the reading times for the experiments introducing one process failures present the lowest reading time, as the failed process reads the checkpoint file from the local storage of the compute node. On the other hand, in the experiments introducing node failures, the reading operation presents a higher cost because the failed processes use the copy of the checkpoint files in the remote disk.

Finally, the total overhead introduced by the proposal is studied. The failure-introduced runtimes are measured, including the execution until the failure, the detection and recovery from it, and the completion of the execution afterwards. As checkpointing takes place at 50% of the execution and the failure is introduced at 75%, the minimum runtime of an execution tolerating a failure will be 125% of the original runtime (75% until failure plus 50% from the recovery point until the end of the execution). Therefore, we consider this time as the baseline runtime, and the overhead of the proposal is measured as the difference between the baseline and runtime when a failure is introduced.

Figure 4.6 shows the original runtimes, as well as the baseline and the testbed failure-introduced runtimes. In the experiments, when only one MPI process is killed, the total cost of tolerating the failure is, on average, 6.6 seconds, introducing 3.6% of relative overhead with respect to the baseline runtimes. Similarly, when a full-node failure is recovered using a spare node, the absolute overhead is 8.7 seconds on average, introducing 3.7% of relative overhead. However, when an already in use node is overloaded, runtimes are larger, as both the computation after the failure and the operations to recover the applications are slower.

Regarding the relation between the number of running processes and the total overhead, we observed that there are not significant differences for the experiments introducing one process failures or node failures using a spare node. However, in the experiments overloading a computation node, the total overhead decreases with the number of processes. As more processes execute the applications, less work corresponds to each one of them, and the impact of the overloading is reduced.

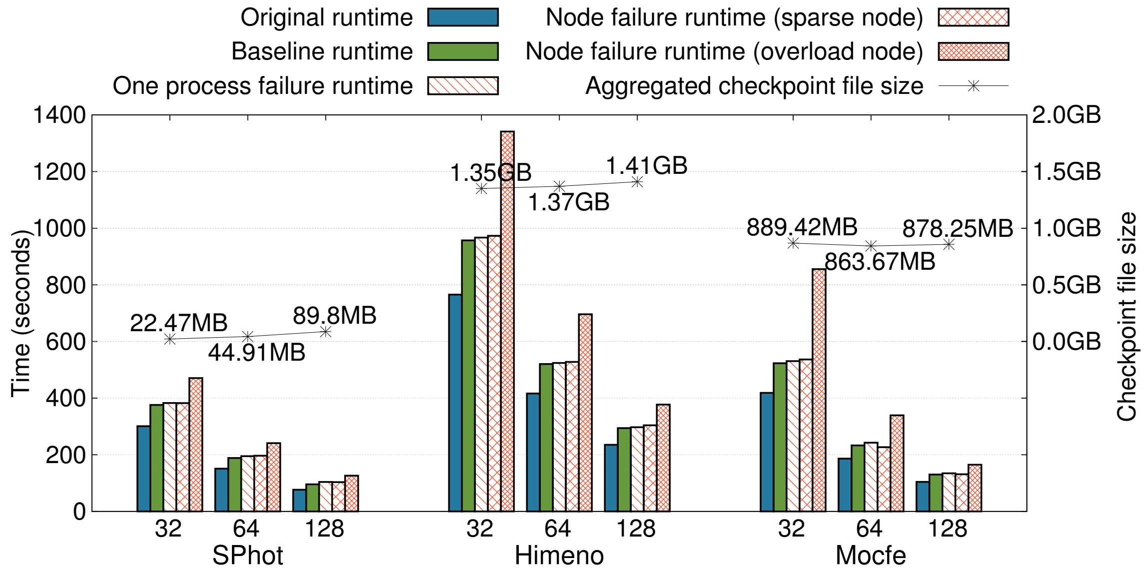


Figure 4.6: Runtimes when introducing failures varying the number of processes. The baseline runtime in which no overhead is introduced (apart from recomputation after the failure) is included for comparison purposes.

4.3.3. Resilience vs. Stop-and-Restart Global Rollback

In order to further study the behavior of the resilience proposal, this section compares its performance with an equivalent traditional stop-and-restart checkpointing solution [82]. The same applications are tested on a larger machine and using larger problem sizes. The configuration parameters of the testbed applications and the hardware platform are detailed in Table 4.5 and Table 4.6, respectively. The experiments presented in this section were performed at CESGA (Galicia Supercomputing Center) in the "FinisTerrae-II" supercomputer, comprised of nodes with two Intel Haswell E5-2680 v3 @ 2.50GHz processors, with 12 cores per processor and 128 GB of RAM, interconnected to an InfiniBand FDR 56Gb/s. The experiments were run spawning 24 MPI process per node (one per core).

For each application and varying the number of MPI processes, Table 4.7 shows the original runtime (without fault tolerance support), and the aggregated checkpoint file size generated when one checkpoint is taken, that is, the addition of the individual checkpoint file size generated by each process. The reminder of this section evaluates and compares the resilience protocol described in this chapter,

Table 4.5: Configuration parameters of the testbed applications.

Configuration parameters	
SPhot	NRUNS=24×2^{16}
Himeno	Gridsize: 2048x2048x1024, NN=24000
Mocfe	Energy groups: 4, angles: 8, mesh: 28^3, strong scaling in space, trajectory spacing = $0.01cm^2$

Table 4.6: Hardware platform details.

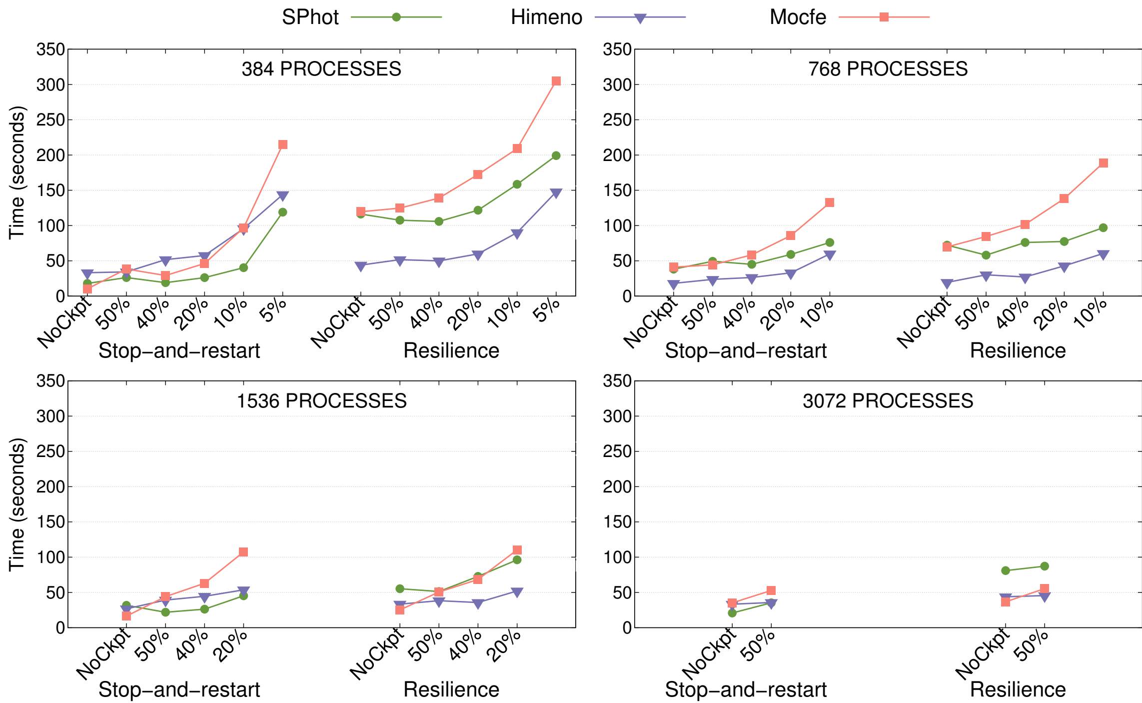
Finisterrae II Supercomputer	
Operating system	Red Hat 6.7
Nodes	2x Intel Haswell E5-2680 v3 2.50 GHz, 12 cores per processor 128 GB main memory
Network	InfiniBand FDR@56Gb/s & Gigabit Ethernet
Local storage	1 TB HDD
Remote storage	Lustre over InfiniBand
MPI version	ULFM commit a1e241f816d7 (release 1.0)
GNU Compilers	v4.4.7, optimization level O3

Table 4.7: Original runtimes (in minutes) and aggregated checkpoint file sizes.

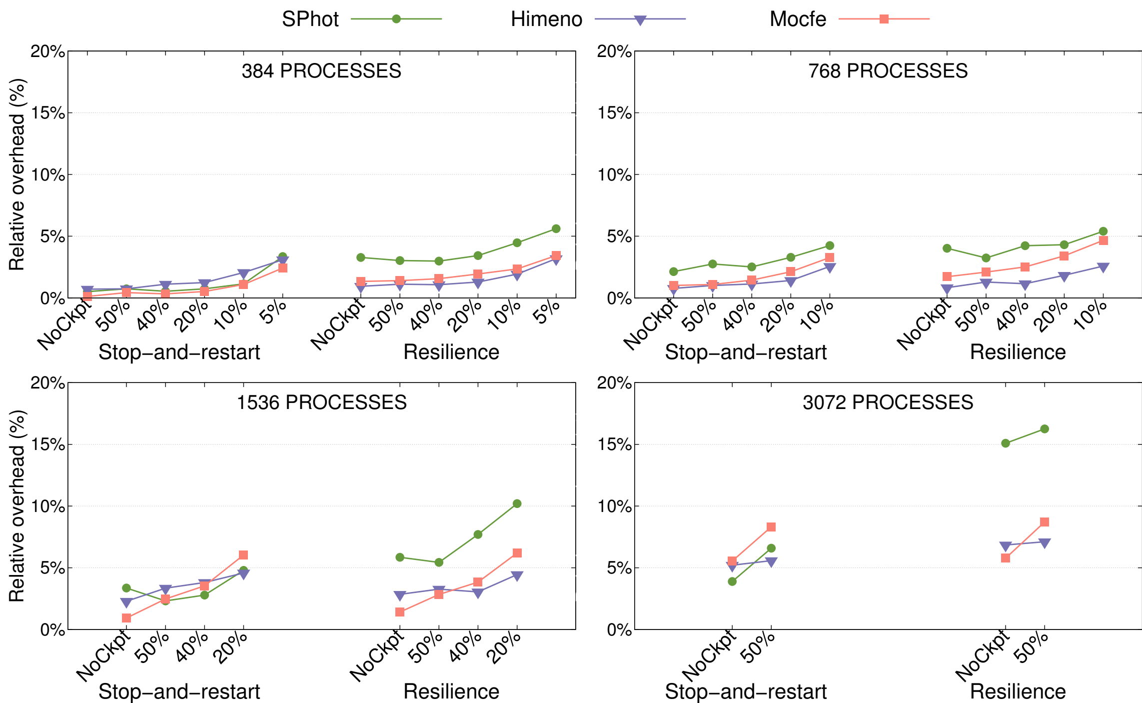
	Original runtimes (minutes)				Aggregated checkpoint file size (GB)			
	# of MPI processes				# of MPI processes			
	384	768	1536	3072	384	768	1536	3072
SPhot	60.3	30.4	15.6	8.9	0.3	0.5	1.0	2.0
Himeno	77.4	39.1	19.5	10.7	165.1	166.2	168.6	170.6
Mocfe	155.9	64.5	29.7	10.6	160.2	146.4	153.2	136.0

with the equivalent stop-and-restart strategy provided by CPPC and described in [110, 112]. For a fair comparison, the same Open MPI version was used in all tests. In both proposals, checkpoint files are stored in a remote disk using the Lustre parallel file system over InfiniBand. In the experiments of this section the multilevel checkpointing technique is not used.

The instrumentation overhead corresponds to the CPPC instrumented applications without generating any checkpoint files. It is tagged as "NoCkpt" in Figure 4.7,



(a) Absolute checkpointing overhead (seconds).



(b) Relative checkpointing overhead (normalized with repect to the original runtimes).

Figure 4.7: Checkpointing overhead varying the checkpointing frequency.

Table 4.8: Testbed checkpointing frequencies and elapsed time (in minutes) between two consecutive checkpoints for different checkpointing frequencies.

	50% 1 ckpt taken			40% 2 ckpts taken			20% 4 ckpts taken			10% 8 ckpts taken			5% 16 ckpts taken		
	SPhot	Himeno	Mocfe	SPhot	Himeno	Mocfe	SPhot	Himeno	Mocfe	SPhot	Himeno	Mocfe	SPhot	Himeno	Mocfe
384 Processes	31	39	80	25	32	64	13	16	33	7	9	17	4	5	9
768 Processes	16	20	33	12	16	26	6	8	14	3	4	7	–	–	–
1536 Processes	8	10	15	6	8	12	3	4	6	–	–	–	–	–	–
3072 Processes	5	5	5	–	–	–	–	–	–	–	–	–	–	–	–

which reports the absolute instrumentation overhead (in seconds), and the relative value (normalized with respect to the original runtimes), respectively. As observed, the instrumentation overhead is larger when using the resilience proposal, which relies on a more extensive instrumentation, adding blocks of code around every MPI call for failure detection and backwards conditional jumping during the recovery.

Figure 4.7 also reports the absolute and relative checkpointing overheads for different checkpointing frequencies. Table 4.8 shows frequencies used (e.g. 20% means checkpointing every time 20% of the computation has been completed). The table also specifies the number of checkpoint files generated and the time elapsed between two consecutives checkpoints in each case. Note that the checkpointing frequency is increased until checkpoints are generated every 3-5 minutes with each number of processes. The checkpointing operation presents no differences whether using the stop-and-restart or the resilience proposals. However, the checkpointing overhead is larger for the resilience proposal. This is explained because the checkpointing overhead also includes the instrumentation cost, which, as commented previously, is larger in the resilience proposal. As observed, when increasing the checkpointing frequency, more checkpoint calls are taken, and thus, the checkpointing overhead increases. All in all, the absolute overhead does not increase with the number of cores, while the relative overhead, which in general is below 5%, increases when scaling out the applications because the original runtimes decrease.

The performance of both the stop-and-restart and the resilience proposal is eval-

Table 4.9: Recovery operations in each proposal.

		Stop-and-restart	Resilience
Failure Detection		Until application aborted due to failure	Until global knowledge of the failure (includes comm. revoke)
Resilience operations (A)		–	Agreement about failed proc. and comm. shrinking (MPI_Comm_shrink)
Re-spawning		Application is relaunched, all processes re-spawned	Failed processes are re-spawned & initialized
Resilience operations (B)		–	Global comm. reconstruction & backwards conditional jumps
Restart	Reading	Find recovery line and read checkpoint files	
	Positioning	Recover application state and positioning in the code	

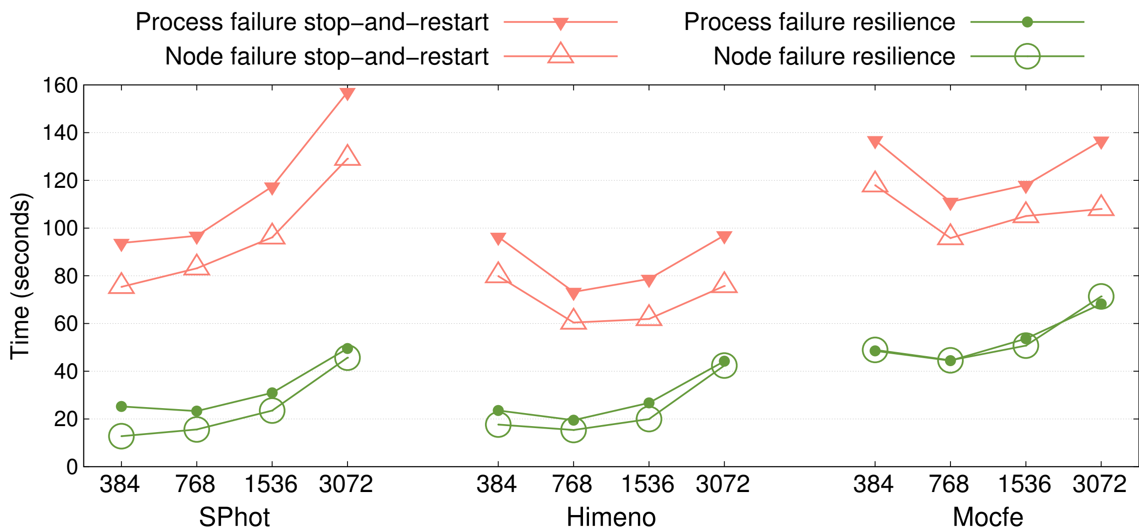


Figure 4.8: Recovery times: addition of the times of all the recovery operations performed by each proposal (the lower, the better).

uated inserting one-process or full-node failures by killing the last ranked MPI process or twenty-four of them, respectively. Failures are introduced when 75% of the application has completed and the applications are recovered using the checkpoint files generated at 50% of the execution. Table 4.9 summarizes the recovery operations performed in each proposal to tolerate the failure. The times that these operations consume in each proposal is represented in Figure 4.8, showing the times spent since the failure is triggered (by killing the process/es) until the end of the

positioning. OOn average, the resilience proposal reduces the recovery times of the stop-and-restart solution by 65%.

Figure 4.9 breaks down the recovery operation times for each application. Failure detection times measure the time spent from the introduction of the failure until its detection. In the resilience proposal, it includes the time spent revoking all the communicators in the application, which is inferior to 5 milliseconds in all the experiments. Detection times are better in the resilience proposal because of the detection mechanisms provided by ULFM. On average, detection is twice faster for one process failures and 6 times faster in the presence of node failures than when using the traditional stop-and-restart solution. In both proposals, as the number of failed processes increases, the time to detect the failure decreases.

In both proposals, the re-spawning times also include the initialization of the failed processes (time spent in the `MPI_Init` routine). The backwards conditional jumps, with a maximum value of 0.5 milliseconds in all the tests, were not included in the figures. Note that, relaunching the entire application is always more costly than relaunching only the processes that have actually failed. However, this difference decreases as the number of failed processes increases, and, more important, when scaling out the application. As a result, the cost of re-spawning 24 processes when there are 3048 surviving processes, is close to the cost of relaunching 3072 processes from scratch.

The reading of the checkpoint files is faster in the resilience proposal, because the surviving processes benefit from the use of the page cache in which the checkpoint files from the most recent recovery line will frequently be present. In other scenarios, reading times can be reduced by using optimizations techniques, such as diskless checkpointing [103, 126] in which copies of the checkpoint files are stored in the memory of neighbor nodes, or multilevel checkpointing [79, 92], which saves those copies in different levels of the memory hierarchy. The restart positioning times are tight to the particular applications and the re-execution of the non-portable state recovery blocks. In SPhot positioning times are lower in the resilience proposal because the re-execution of these blocks benefits from the usage of page cache. Finally, the shrinking and the global communicator reconstruction are also represented in the figures. In both cases, these times increases with the number of processes running the applications. Shrinking times are larger when more survivors participate in the

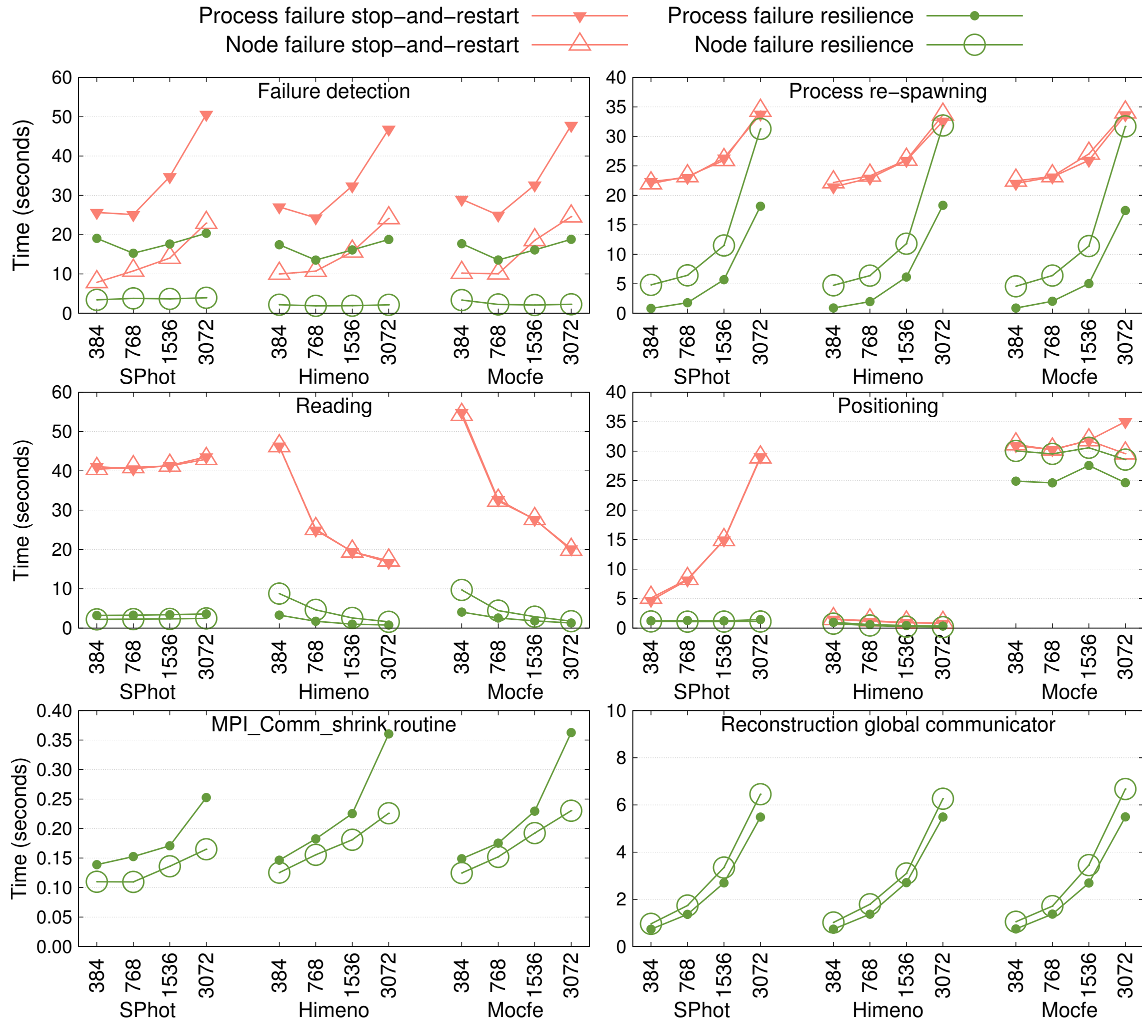


Figure 4.9: Time of the different recovery operations.

operation, as more survivors must agree about the subset of failed processes.

Note that, in both proposals the restart overhead would also include the re-execution of the computation done from the point in which checkpoint files were generated until the failure occurrence, an overhead that will be tight to the selected checkpointing frequency (more frequent checkpoints imply less re-execution overhead in the event of a failure, although more overhead is introduced during the fault free execution). Additionally, in some systems, the stop-and-restart proposal would also imply the re-queueing of a new job to the scheduling system, introducing an overhead dependent of the availability of the cluster resources. All in all, the

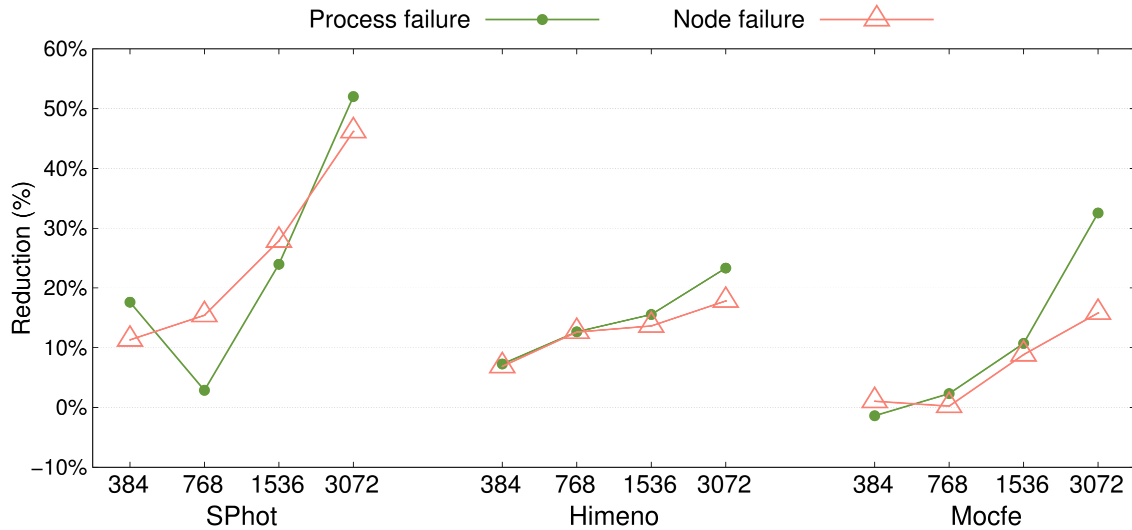


Figure 4.10: Reduction in the extra runtime when introducing a failure and using resilience proposal instead of stop-and-restart rollback (higher is better).

improvement in overall execution runtime when tolerating a failure is represented in Figure 4.10, showing the reduction in the extra runtime achieved when using the resilience proposal instead of the stop-and-restart technique to tolerate the failure. The extra runtime is calculated as the difference between the original runtime and the runtime when a failure occurs. On average, the overhead is reduced by 15.72%, and the percentage reduction increases as the applications scale out.

4.4. Related work

Some works in the literature have focused on implementing resilient applications using ULFM [2, 15, 50, 51, 68, 98, 109, 116, 126]. Most of those proposals are specific to one or a set of applications [2, 15, 68, 98, 109]. Bland et al. [15] and Pauli et al. [98] focused on Monte Carlo methods. Laguna et al. evaluate ULFM on a massively scalar molecular dynamics code [68]. Partial Differential Equation (PDE) codes are targeted by Ali et al. [2] and by Rizzi et al. [109]. All these proposals consider the particular characteristics of the applications to simplify the recovery process. A customized solution allows reducing the recovery overhead upon failure, e.g., simplifying the detection of failures by checking the status of the execution in

specific points; avoiding the re-spawning of the failed processes when the algorithm tolerates shrinking the number of the MPI processes; or recovering the application data by means of its properties as an alternative to checkpointing. In contrast, unlike our proposal, they cannot be generally applied to any SPMD application.

Other alternatives to ULFM to build resilient applications are NR-MPI [122], FMI [114], or Reinit [69]. In contrast to ULFM, which proposes a low-level API that supports a variety of fault tolerance models, these alternatives propose a simplified interface towards a non-shrinking model, repairing the MPI inner state upon failure, and re-spawning the failed processes. Reinit [69] proposes a prototype fault-tolerance interface for MPI, suitable for global, backward, non-shrinking recovery. FMI [114] is a prototype programming model with a similar semantic to MPI that handles fault tolerance, including checkpointing application state, restarting failed processes, and allocating additional nodes when needed. Finally, NR-MPI [122] is a non-stop and fault resilient MPI built on top of MPICH that implements the semantics of FT-MPI [48]. These proposals hide the complexities of repairing the MPI state, however, they still rely on the programmers to instrument and modify the application code to obtain fault-tolerance support, including the responsibility of identifying which application data should be saved and in which points of the program.

In contrast, the CPPC resilience proposal provides a transparent solution in which the application code is automatically instrumented by the CPPC compiler adding full fault tolerance support, both for detecting failures and repairing the MPI inner state as well as for checkpointing and recovering the application data. The fact that this proposal provides a transparent solution is especially useful for those scientific applications already developed over the years in HPC centers, in which manually adding fault tolerance support by programmers is, in general, a complex and time-consuming task.

4.5. Concluding Remarks

In this chapter the CPPC checkpointing tool is extended to exploit the new ULFM functionalities to transparently obtain resilient applications from general MPI SPMD programs. By means of the CPPC instrumentation of the original

application code, failures in one or several MPI processes are tolerated using a non-shrinking backwards recovery based on checkpointing. Besides, a multithreaded multilevel checkpointing stores a copy of the checkpoint files in different levels of storage, minimizing the amount of data to be moved upon failure, and thus, reducing the recovery overhead without increasing the checkpointing overhead.

The experimental evaluation analyzes the behavior when one process or an entire node fails. Furthermore, in case of a node failure, two different scenarios are considered: the failed processes are re-spawned in a spare node or in an already in use one (overloading it). Results show the low overhead of the solution.

In addition, the evaluation assesses the performance as well as compares two application-level fault-tolerant solutions for MPI programs: a traditional stop-and-restart approach and its equivalent resilience proposal using ULFM capabilities. The resilience solution clearly outperforms the stop-and-restart approach, reducing the time consumed in the recovery operations between 1.6x and 4x, and avoiding the resubmission of the job. During the recovery, the most costly steps are the failure detection and the re-spawning of failed processes. In the resilience proposal, the failure detection times are between 2x and 6x faster and the re-spawning times are also notably smaller. However, the re-spawning times significantly increase when the number of failed processes grow and when scaling out the application. Thus, optimizations to minimize the re-spawning cost should be studied, such as the use of spare processes initialized at the beginning of the execution that can take over the failed ranks upon failure [126].

The evaluation performed in this work is done on the basis of a general solution that can be applied to any SPMD code. However, ULFM allows for the implementation of different fault-tolerant strategies, depending on the nature of the applications at hand. Ad-hoc solutions could reduce the failure-free or the recovery overhead upon a failure. For instance, simplifying the detection of failures by checking the status of the execution in specific points, or avoiding the re-spawning of the failed processes in those applications that tolerate the shrinking of MPI processes.

Finally, in the evaluated resilience solution, all the application processes roll back to the last valid recovery line, thus, all processes re-execute the computation done from the checkpoint until the point where the failure have occurred. We believe

that a global recovery should be avoided to improve the application performance both in time and energy consumption. Thus, in the next chapter, we will explore this direction further considering the development of a message-logging protocol to avoid the roll back of the surviving processes.

Chapter 5

Local Rollback for Resilient MPI Applications

The resilience approach presented in the previous chapter relies on a global rollback checkpoint/restart, rolling back all processes in the application after a failure. However, in many instances, the failure has a localized scope and its impact is usually restricted to a subset of the resources being used. The ULFM interface enables the deployment of more flexible recovery strategies, including localized recovery. This chapter proposes a local rollback approach that can be generally applied to SPMD applications by combining ULFM, the CPPC checkpointing tool, and message logging.

This chapter is structured as follows. Section 5.1 gives a global overview of the local rollback protocol. Section 5.2 explains the message logging strategy. Section 5.3 presents the management of the communications interrupted by the failure, and Section 5.4 describes the tracking protocol developed to ensure the consistency of the replay process. The experimental results are presented in Section 5.5. Section 5.6 covers the related work. Finally, Section 5.7 concludes this chapter.

5.1. Local Rollback Protocol Outline

Traditional checkpointing solutions for MPI applications force all processes running the application, disregarding their statuses, to restart from the last checkpoint. This implies rolling back to the last recovery line and repeating a computation that has already been done. In many instances a complete restart is unnecessary, since most of the computation nodes used by a parallel job will still be alive. Thus, a global rollback introduces unnecessary overheads and energy consumption, and more efficient solutions need to be explored.

In order to restrict the number of processes rolling back to those that have failed, this chapter proposes a local rollback protocol. The goal is to reach, in the event of a failure, a consistent global state from which the application can resume the execution by rolling back only the failed processes. Figure 5.1 shows a global overview of the operation. In the left part of the figure, the application is executed normally until a failure occurs. The point of the execution where the failure takes place, from the survivors' perspective, is called the "failure line". The right part of the figure shows the recovery using the local rollback protocol, which is split into two phases: (1) the "processes recovery" phase detects the failure and re-spawns failed processes, and (2) the "consistency recovery" phase leads the application to a consistent state from which the execution can resume.

During the processes recovery, CPPC exploits the ULFM functionalities as explained in the previous chapter to avoid terminating the application in the event of a failure. The default MPI error handler is replaced with a custom one, which is invoked upon process failure. Within the error handler, survivors revoke all their communicators to ensure global knowledge of the failure, agree about the failed processes, and then re-spawn them. Together, all processes reconstruct the global communicator of the application (conceptually similar to MPI_COMM_WORLD), and then rebuild all revoked communicators. The fact that all communicators are revoked to ensure failure detection implies that all communicators need to be reconstructed, by substituting all failed processes with their new replacement processes. CPPC tracks all communicators used by the application at compile time, and the CPPC compiler replaces the communicators in the application with a custom CPPC communicator that contains a pointer to the underlying MPI communicator actually

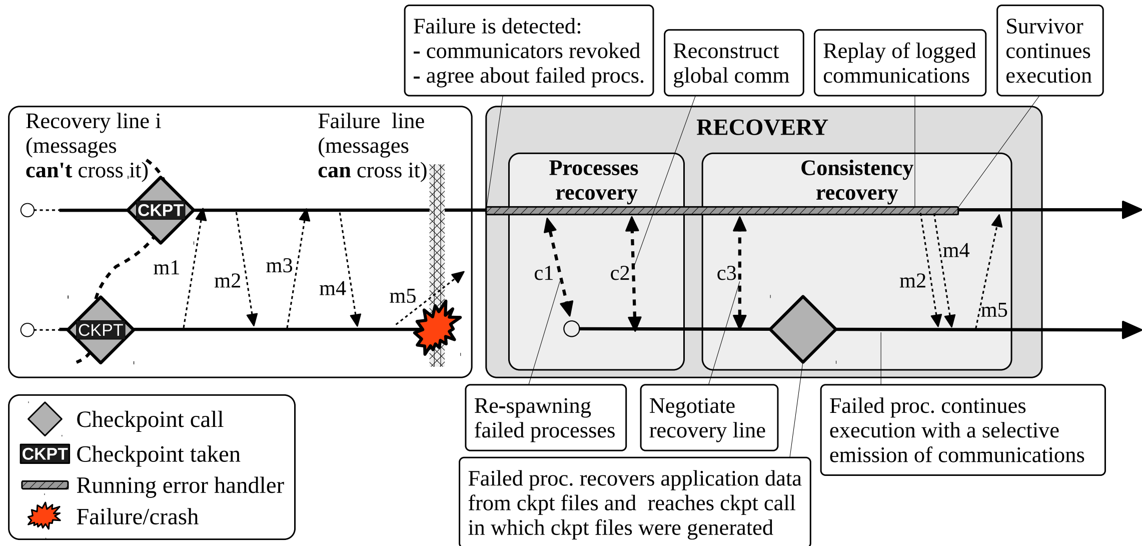


Figure 5.1: Local rollback protocol overview.

used by MPI. With this approach, all communicators used by the application are known to CPPC, and they can be revoked and transparently substituted with their repaired replacement.

In the consistency recovery phase, the state of a failed process is recovered using the checkpoint file from the last valid recovery line. Then, to reach a consistent global state, failed processes need to progress between that recovery line and the failure line. In order to reach the same state as before the failure (a known consistent state), this progress needs to occur exactly as in the original execution. For this purpose, we use a message logging protocol detailed in Section 5.2.

Message logging protocols have two fundamental parts: (1) the logging of events, and (2) the logging of the content of the messages. A particular piece of the execution of a process is a sequence of states and events. An event corresponds with a computational or communication step of a process that, given a preceding state, leads the process to a new state. By repeating the same events, the progress of the rolled back processes will lead to the same state as before the failure. As the system is basically asynchronous, there is no direct time relationship between events occurring on different processes; however, events are partially ordered by the Lamport "happened before" relationship [70]. Events can be deterministic or non-deterministic, depending on whether or not, from a given state, the same outcome state would always

be obtained. Deterministic events follow the code flow (e.g., message emission or computations), while non-deterministic events, such as message receptions, depend on the time constraints of message deliveries. Processes are considered "piecewise deterministic": only sparse non-deterministic events occur, separating large parts of deterministic computation. To progress the failed processes from the recovery line to the failure line and reach the same state, all events in that part of the execution need to be replayed in the exact same order as the initial execution. Deterministic events will be naturally replayed as the processes execute the application code. However, the same outcome must be ensured for non-deterministic events, and thus they must be logged in the original execution. Event logging must be done reliably, and different techniques (pessimistic, optimistic, and causal), providing different levels of synchronicity and reliability, have been studied [3]. In addition, failed processes replay any message reception that impacted their state in the original execution, and thus the content of the messages (i.e. the message payload) needs to be available without restarting the sender process. A variety of protocols have been proposed for this purpose. Receiver-based approaches [91] perform the local copy of the message contents in the receiver side. The main advantage here is that the log can be locally available upon recovery; however, messages need to be committed to stable storage or to a remote repository to make them available after the process fails. On the other hand, in sender-based strategies [18, 20, 88, 89, 113], the logging is performed on the sender process. This is the most-used approach, as it provides better performance [107] than other approaches. In a sender-based approach, the local copy can be made in parallel with the network transfer, and processes can keep the log in their memory; if the process fails, the log is lost, but it will be regenerated during the recovery. As a drawback, during the recovery, failed processes need to request the replay of messages by the survivor processes. The message logging protocol proposed in this chapter applies a sender-based payload logging and a pessimistic event logging.

As illustrated in Figure 5.1, some communications need to be replayed during the consistency recovery phase: those to be received by a failed process to enable its progress (messages m_2 and m_4 in the figure) and those that were interrupted by the failure (message m_5). Other communications need to be skipped during the recovery: those that were successfully received by a survivor process (messages m_1 and m_3 in the figure). The success of the recovery is predicated on correctly

identifying the communications belonging to each subset. Section 5.4 explains how this identification is performed and how the replay process takes place.

In addition, the reconstruction of communicators has an important implication for replaying communication messages interrupted by failures: all communications initiated but not completed before the failure are lost. We assume this includes the communication call in which the failure is detected, because—as stated in the ULFM specification—there are no guarantees regarding the outcome of that call. The management of communications interrupted by failure(s) is explained in Section 5.3.

5.2. Message Logging

This section describes the message logging protocol that combines system-level logging and application-level logging. Point-to-point communications are logged using the Open MPI Vprotocol component. Collective communications are logged by CPPC, at the application level. The spatial coordination protocol used by CPPC contributes to a reduction of the log size by enabling processes to identify when a log will never be used for future replays.

5.2.1. Logging Point-to-Point Communications

Point-to-point communications are logged using a pessimistic, sender-based message logging, which saves every outgoing message in the senders' volatile memory. Sender-based logging enables copying the messages, in parallel, while the network transfers the data. Pessimistic event logging ensures that all previous non-deterministic events of a process are logged before a process is allowed to impact the rest of the system. In MPI, non-deterministic events to be logged correspond to any-source receptions and non-deterministic deliveries (i.e., `iProbe`, `WaitSome`, `WaitAny`, `Test`, `TestAll`, `TestSome`, and `TestAny` functions). Because MPI communication channels are First In, First Out (FIFO), replaying message emissions in order—and guaranteeing the same outcome for any-source receptions and non-deterministic deliveries—will lead to a consistent global execution state.

The method proposed here uses the Vprotocol [18] message logging component,

which provides sender-based message logging and pessimist event logging. The sender-based logging is integrated into the data-type engine of Open MPI and copies the data in a `mmap`-ed memory segment as it is packed [16] and, thus, moves the memory copies out of the critical path of the application. While log of the content of the messages is kept in the memory of the sender processes, for event logging, the outcome of non-deterministic events is stored on a stable remote server.

After a failure, communications must be replayed using the appropriate communicator. Because our approach is a hybrid between application-level and library-level recovery, failures cannot be masked completely at the application level, as is customary in pure, system-level message logging. Instead, the communication capability is restored at the application level. The ULFM communicator reconstruction implies new MPI communicators, and—in our work—the Vprotocol component has been extended to use translation tables between the old and new communicators, which are identified by their internal Communicator ID (CID). The CID is included in the log, and—during the recovery—hash tables are built with the correspondence between old and new CIDs. This approach ensures the replay through the correct communicator and a consistent log in the event of additional failures.

5.2.2. Logging Collective Communications

The original Vprotocol component, due to its location in the Open MPI software stack, sees only point-to-point communications. Instead of noticing a collective communication as such, it sees them unfolding as a set of point-to-point communications according to the collective algorithm implementation. It therefore logs collective operations by logging all of the corresponding point-to-point communications. This has two detrimental effects: (1) it prevents operating with hardware-based or architecture-aware collective communications, and (2) can result in a significant log volume that is not semantically necessary. To overcome these limitations, the method proposed here logs collective communications at the application level, thereby enabling the use of different collective communication implementations and potentially reducing the total log size, as the buffers sent in the intermediate steps of the collective are not logged. Conversely, when logging at the application level, the memory copies of the logged data are in the critical path of the application.

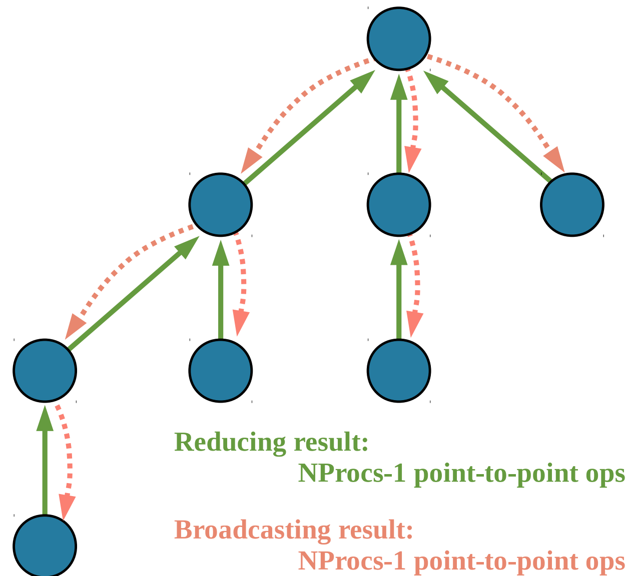


Figure 5.2: Binomial-tree staging of `AllReduce` collective operation.

The benefits of this technique are tied to the implementation of the collective operations, and they will appear when intermediary copies are performed. For instance, Figure 5.2 presents a common binomial-tree staging of the AllReduce collective operation. A first wave of point-to-point communications is performed to reduce the result, and a second one broadcasts this result to all processes. Therefore, the internal point-to-point implementation of this collective operation performs $2 \times (\texttt{NProcs} - 1)$ send operations, which implies logging $2 \times (\texttt{NProcs} - 1) \times \texttt{BuffSize}$ bytes of data. On the other hand, the application-level logging of the `AllReduce` collective operation logs the contribution of each process involved in the collective call, i.e., $\texttt{NProcs} \times \texttt{BuffSize}$ bytes of data. Therefore, both the number of entries that are appended to the log and the total logged data are divided by a $\frac{2 \times (\texttt{NProcs} - 1)}{\texttt{NProcs}}$ factor when logging at the application level.

An MPI wrapper implemented on top of CPPC performs the application-level logging. Instead of using the traditional Profiling MPI API (PMPI), we decided to implement our own wrappers around MPI functions, leaving the PMPI layer available for other usages, and therefore allowing CPPC-transformed applications to benefit from any PMPI-enabled tools. The CPPC wrappers around MPI function calls perform the logging of the pertinent data and then calls the actual MPI routine. Each process logs the minimum data necessary to be able to re-execute the collective

after a failure in a `mmap`-ed memory segment.

During the recovery, collective operations will be re-executed by all processes in the communicator, including survivors of the previous faults and replacement processes. Although survivor processes do not roll back, they will re-execute the collective communications during the recovery procedure, taking their inputs directly from their log. To ensure consistency, point-to-point and collective calls must be replayed in the same order as in the original execution. Thus, when logging a collective, Vprotocol is notified, and it introduces a mark within its log. During the recovery, when a survivor process encounters a collective mark, the Vprotocol component transfers control to CPPC to insert the collective re-execution call.

5.2.3. Implications for the Log Size

Traditional message logging, used in combination with uncoordinated checkpointing, treats all messages in the application as possible in-transit or orphan messages, which can be requested at any time by a failed process. In contrast, in the method proposed here—thanks to the spatially coordinated checkpoints provided by CPPC—the recovery lines cannot be crossed by any communication. Thus, only the messages from the last recovery line need to be available. Recovery lines, therefore, correspond with safe locations in which obsolete logs can be cleared, which means we can avoid keeping the entire log of the application or including it in the checkpoint files. However, with CPPC, processes checkpoint independently; the only way to ensure that a recovery line is completed would be to introduce a global acknowledgement between processes, which would add a synchronization point (and corresponding overhead) during the checkpoint phase. Instead, the logs from the l latest recovery lines are kept in the memory of the processes, l being a user-defined parameter. After a failure, the appropriate log will be chosen depending on which line is the selected recovery line. In the improbable case where an even older recovery line needs to be used, the log would not be available. However, in this case, a global rollback is always possible. This approach reduces the overhead and the memory usage introduced by the message logging. Furthermore, in most application patterns, safe points are separated by semantically synchronizing collective communications that prevent a rollback going further than the last recovery line.

Note that communicator creation calls correspond with a particular type of collective operation. In many cases, derived communicators are created at the beginning of the application code, and they are used during the whole execution. Thus, these log entries are not cleared when checkpointing, as they will always be necessary for the failure recovery procedure to recreate the necessary communicators.

5.3. Communications Interrupted by a Failure

Revoking and reconstructing communicators implies that all ongoing communications at the time of the failure are purged at all survivor processes. The incomplete communications correspond to the communication call in which the failure was detected and to all non-blocking communication calls that were not completed when the failure hit. A communication crossing the failure line (including between survivor processes) would then be lost and it would need to be reposted to ensure that the execution resumes successfully. Note that this implies not only replaying emissions (as in traditional system-level message logging) but also reposting, at the application level, those receptions and collective communications that were interrupted by the failure.

There are no guarantees regarding the outcome of the data transfer related to an MPI call in which a failure is detected (i.e., output buffers may contain garbage data). Therefore, to ensure consistency, the MPI calls mentioned above have to be re-executed. As commented earlier, all MPI calls in the application are performed through the MPI wrapper implemented on top of CPPC. Thus, within the MPI function wrappers, the code returned by the MPI call is checked for errors, and corrective actions are initiated when necessary. When an error is returned, the call is re-executed with its original arguments. Note, however, that some of those arguments are updated, such as the reconstructed communicators or—in the case of non-blocking communications—the requests that were reposted during the recovery replay.

For non-blocking communication calls, as stated in the MPI standard, a non-blocking send call (e.g., `MPI_Isend`) initiates the send operation but does not finish it. A separate completion call (e.g., `MPI_Wait`) is needed to finish the communication

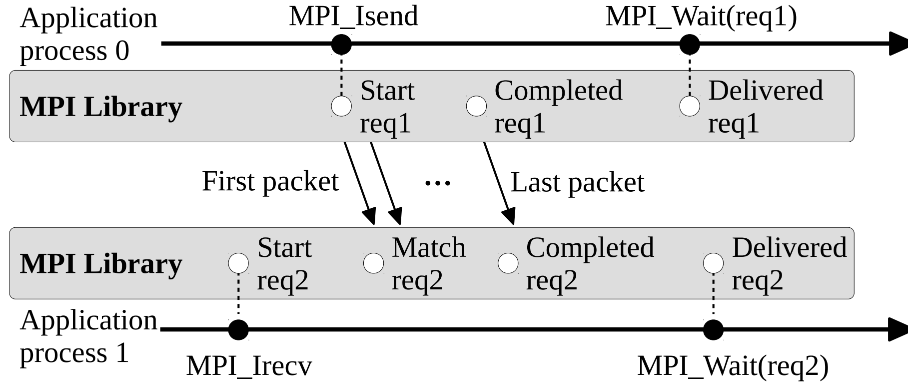


Figure 5.3: States of non-blocking communications.

i.e., to verify that the data has been copied out of the send buffer. Similarly, a non-blocking receive call (e.g., `MPI_Irecv`) initiates the receive operation, but a separate completion call (e.g., `MPI_Wait`) is needed to finish the operation and to verify that the data has been received into the reception buffer.

We consider the following states for a non-blocking communication request, illustrated in Figure 5.3. When the completion call is invoked over a request, the request is considered delivered to the application. However, at any point between the initiation of the request to its delivery to the application, the request might be completed internally by CPPC; that is, the data has been received into the reception buffer even though the application has not yet acknowledged it.

When a failure strikes, there could be a number of pending non-blocking send and receive communications whose completion calls have not been invoked yet. Their requests, even when internally completed (i.e. the correct result is available in the output buffers), will not be delivered to the application after the recovery. The reason is that these requests will be associated with an old communicator, and a later invocation of a completion call will generate an error. On the other hand, those that did not complete internally will be lost upon failure, and they will need to be re-posted before the execution continues.

CPPC maintains a temporary log for non-blocking communication calls for both emissions and receptions, which is discarded upon delivery of a request. Note that a correctly designed MPI application will not modify the send buffers, nor will it use the receive buffers until a completion routine (e.g., `MPI_Wait`) has been invoked

over the associated requests. Thus, this temporary log avoids doing memory copies of the send buffers and instead keeps a reference to them. For each non-blocking call, CPPC creates a log entry that permits the re-execution of the call, to again initiate the send/receive operation, and it keeps a reference to the associated request it generated. Due to the temporary characteristics of this log, a pool of log entries is used to avoid the overhead from allocating and freeing these items.

The consistency of the temporary non-blocking log is maintained as follows. First, when a request is internally completed by CPPC the associated log entry is removed. However, the request becomes a non-delivered request until the application layer is informed, and CPPC maintains a reference to it. Eventually, when a completion routine (e.g., `MPI_Wait`) is invoked and the request is delivered, the CPPC reference is finally removed. The management described here applies for all types of non-blocking requests and is purely local, we will detail in Section 5.4 the distributed management including the ordering of communication reposts.

After a failure, the first step consists of freeing the resources associated with the incomplete and non-delivered requests. The incomplete request will be reposted within the survivors' replay, as explained in Section 5.4. Note that when reposting a non-blocking communication call, its associated request in the application needs to be updated. This is solved using the same approach as is used with the communicators: a custom CPPC request is used by the application, which actually contains a pointer to the real MPI request, thereby enabling the MPI request to be updated transparently after it has been reposted.

5.4. Tracking Messages and Emission Replay

In MPI, observing a communication completion at one peer does not imply that it has also completed at other peers. For example, an `MPI_Send` can complete as soon as the send buffer can be reused at the local process. Meanwhile, the message may be buffered, and the corresponding receive may still be pending. During the replay, survivors have to resend the messages from their log to the restarting processes (as in traditional message logging), but they also have to send some messages from their log to other survivor processes whose receptions have been interrupted by the

failure (i.e., those receptions that have been reposted by the protocol described in Section 5.3). Said another way, the success of the replay relies on the receivers' capability to inform the senders which communications have been received and on the ability of the senders to distinguish which communications need to be replayed. Since this list of completed or incomplete communications depends on post order at the receiver, and not post order at the sender, the messages that need to be replayed are not necessarily contiguous in the sender-based log. Thus, it is critical that a given communication can be unequivocally identified by both peers involved.

Our strategy for identifying messages relies on sequence numbers. Every time a message is sent over a particular communicator to a particular receiver, a per-channel counter is increased, and its value is piggybacked in the message as the Sender Sequence ID (SSID). This SSID is then used to implement a tracking protocol for point-to-point communications to identify the messages that are expected by other peers and need to be replayed and to identify those that were completed and need to be skipped.

5.4.1. Tracking Protocol

During the execution, processes track the SSIDs for each send and receive operation they have completed over each communication channel. For the emissions on a given channel, SSIDs grow sequentially. Thus, the sender only needs to keep the most recent SSID for that communication channel. However, receiving a message with a particular SSID does not ensure that all previous messages in that channel have been received, because a receiver can enforce a different reception order (e.g., by using different tags). Thus, processes maintain the highest SSID (h_{ssid}) they have received and a list of pending SSIDs ranges. Each range is represented with a pair $[a_{ssid}, b_{ssid}]$, meaning that messages with SSIDs in that range are pending. When, in a channel, a non-consecutive SSID is received:

- If it is larger than the highest SSID received, a new range $[h_{ssid}+1, current_{ssid}-1]$ is added to the pending reception list.
- Otherwise, it is a pending SSID, and it must be removed from the pending list. Note that the removal can imply splitting a pending range in two.

Each checkpoint file includes the latest sent and the highest-received SSIDs. Note that, when using CPPC, messages cannot cross the recovery line, and therefore there are no pending ranges when checkpointing.

SSIDs tracking is only used for point-to-point communication replay. When using Open MPI, the header of the message already contains a sequence number for MPI point-to-point ordering. To avoid the extra cost of duplicate tracking and piggybacking, we reuse that existing sequence number in the SSID tracking algorithm. To prevent the collective communications—implemented using point-to-point communications—from impacting the SSID tracking, they are run through a different communicator. Additionally, when communicators are reconstructed with ULFM, Open MPI SSIDs are reset. The tracking protocol deals with this issue by calculating the SSID offsets. During the recovery, the value of the SSID before the failure is restored, and the SSID tracking continues using the saved value as a baseline and then adding the current value of the Open MPI sequence number. This absolute indexing of SSIDs allows for tolerating future failures after the first recovery and, notably, failures hitting the same recovery line multiple times.

5.4.2. Ordered Replay

Once the failed processes are recovered using the checkpoint files, all processes exchange the tracking information. For each pair of processes that have exchanged messages in the past, and for each particular communication channel they have used, the receiver notifies the sender of the highest SSID it has received and of the pending ranges (if any). With that remote information on board, if the sender is:

- A failed process: it knows which emissions can be skipped because they were successfully received during the previous execution, and which ones must be re-emitted.
- A survivor process: it can determine which messages from its log must be replayed because other peers require them.

Then, failed processes continue their execution, skipping the communications already received by survivors, and emitting those that the peers expect to receive.

Additionally, the information from the event logger is used by the failed processes to ensure that non-deterministic events are replayed exactly as they were in the original execution (e.g., an any-source and/or any-tag reception will be regenerated as a named reception, with a well specified source and tag to prevent any potential communication mismatch and deliver a deterministic re-execution in which the data is received in exactly the same order as in the original execution). All collective communications are normally executed; the collective protocol, detailed in Section 5.2.2, guarantees both correctness and native collective performance.

Meanwhile, survivors start the replay of logged communications and invoke Vprotocol's replay—replaying the necessary point-to-point communications from the log. When a survivor encounters a collective log mark in its log, it transfers control to CPPC to re-execute the appropriate collective. Towards the end of the log, the survivor can encounter gaps originated from non-blocking emissions that were interrupted by the failure, which—again—results in control transfers to CPPC to re-execute the pending emissions. As mentioned in Section 5.3, there can also be pending receptions interrupted by a failure that need to be reposted. However, there are no marks for non-blocking receptions interrupted by the failure in Vprotocol's log (the only logged information for receptions relates to the ordering on non-deterministic events to the remote event logging server, when applicable). Pending receptions that were interrupted (i.e. produced an error code) are tracked by the CPPC component and are re-posted and ordered with respect to the collective communications and pending non-blocking emissions, thereby ensuring consistency. Once a survivor finishes processing its log, it continues the execution.

5.5. Experimental Evaluation

The experimental evaluation was performed at the Galicia Supercomputing Center using the "FinisTerrae-II" supercomputer. The hardware platform is detailed in Table 5.1. The experiments spawned 24 MPI process per node (one per core). Checkpoint files are dumped on a remote parallel file system using Lustre over InfiniBand. For our testing, we used CPPC version 0.8.1, working with HDF5 version 1.8.11 and GCC version 4.4.7. The Open MPI version used corresponds with ULFM 1.1, modified for the integration of Vprotocol and CPPC. The hwloc package

Table 5.1: Hardware platform details.

Finisterrae II Supercomputer	
Operating system	Red Hat 6.7
Nodes	2x Intel Haswell E5-2680 v3
	2.50 GHz, 12 cores per processor
	128 GB main memory
Network	InfiniBand FDR@56Gb/s & Gigabit Ethernet
Local storage	1 TB HDD
Remote storage	Lustre over InfiniBand
MPI version	ULFM commit a1e241f816d7
GNU Compilers	v4.4.7, optimization level O3

Table 5.2: Configuration parameters of the testbed applications.

Configuration parameters	
Himeno	Gridsize: $1024 \times 512 \times 512$, NN=12 000
Mocfe	Energy groups: 4, angles: 8, mesh: 57^3, strong scaling in space, trajectory spacing = $0.01 cm^2$
SPhot	NRUNS=24×2^{16}
TeaLeaf	`x_cells`=4096, `y_cells`=4096, 100 time steps

was used for binding the processes to the cores. Applications were compiled with optimization level O3. We report the average times of 20 executions.

We used an application testbed comprised of four domain science MPI applications with different checkpoint file sizes and communication patterns. The configuration parameters of the testbed applications are detailed in Table 5.2. The application TeaLeaf is tested in addition to the applications presented in the previous chapter (see Section 4.3 for more details about the other applications). TeaLeaf [125] is a mini-app, originally part of the Mantevo project, that solves a linear heat conduction equation on a spatially decomposed regular grid using a five-point stencil with implicit solvers. The original runtimes of the applications, in minutes, are reported in Table 5.3. Most of the experiments were run doing strong scaling (i.e., maintaining the global problem size constant as the application scales out), with the exception of the tests reported in Section 5.5.3, which study the behavior of the proposal when doing weak scaling (i.e., maintaining the problem size by process constant as the

Table 5.3: Original runtimes of the testbed applications in minutes.

	48P	96P	192P	384P	768P
HIMENO	18.48	9.22	4.76	2.45	1.56
MOCFE	20.49	8.26	4.75	2.41	1.48
SPHOT	16.38	8.25	3.78	2.25	1.48
TEALEAF	17.50	9.31	4.28	2.35	1.79

application scales out) in the Himeno application.

In order to measure the benefits of the proposed solution in recovery scenarios, the experiments compare the local rollback with the global rollback strategy presented in Chapter 4. In both cases, the checkpointing frequency (N) is fixed so that two checkpoint files are generated during the execution of the applications—the first one at 40% of the execution progress and the second one at 80%. The N value for each application is shown in Table 5.4. In all experiments, a failure is introduced by killing the last rank in the application when 75% of the computation is completed. Once the failure is detected, communicators are revoked, survivors agree about the failed process, and a replacement process is spawned. In the global rollback, all processes roll back to the checkpoint generated at 40% of the execution. In the local rollback, the replacement processes continue from the last checkpoint; meanwhile, the survivors use their logs to provide the restarting process with messages that enable the progress of the failed process until it reaches a consistent state. In the global rollback, no extra overhead besides CPPC instrumentation and checkpointing is introduced. On the other hand, the local rollback proposal introduces extra operations to maintain the logs. In these experiments, only the logs from the last recovery line are kept in memory ($l = 1$). Below, Section 5.5.1 describes the extra fault-free overhead, and Section 5.5.2 studies the benefits (upon recovery) of the local rollback proposal.

The settings for the experiments (i.e., checkpointing and failure frequencies) establish homogeneous parameters across the different tests, simplifying a thorough study of the performance of the local rollback protocol. In realistic scenarios, applications runtimes would be in the order of days, thus, multiple failures would hit the execution. Therefore, checkpoints would need to be taken with a higher frequency to ensure the execution completion. In this scenario, the local rollback would provide more efficient recoveries each time a failure strikes, thus, improving

the overall execution time under those conditions. Even though a logging overhead is introduced during the execution, the fact that the log can be cleared upon checkpointing provides an important advantage over traditional message logging techniques. More precisely, this characteristic can enable the operation of the proposal on communication-bound applications that otherwise, will exceed the memory limitations of the running environment.

5.5.1. Operation Overhead in the Absence of Failures

The overhead introduced by the local rollback is influenced by the communication pattern of the applications and how the logs evolve during the execution. Table 5.4 characterizes the applications in terms of their log volume. It shows the MPI routines that are called in the main loop of the application (where the checkpoint call is located). The table also shows the number of processes running the experiment, the total number of iterations run by the application, and the checkpointing frequency (N) indicating the number of iterations between two consecutive checkpoints. Regarding event logging, only SPhot generates non-deterministic events—precisely $(num_procs - 1) \times 5$ per iteration. Finally, the aggregated average log behavior per iteration is reported, that is, the average among all iterations, where the aggregated value for each iteration is computed as the addition of the log from all processes (i.e. the total log generated by the application in one iteration). The number of entries and the size of the log generated by the proposal are reported for both point-to-point and collective calls. For the latter, the table also shows the reduction, in percentage, that the application-level logging provides over the internal point-to-point logging.

In the target applications, one can see a very significant reduction in the collective communications log volume thanks to application-level logging of collective operations. Even though the log size of the testbed applications is dominated by point-to-point operations, other scenarios may present a larger contribution from collective communications. The results from a profiling study performed by Rabenseifner [105] show that nearly every MPI production job uses collective operations and that they consume almost 60% of the MPI time, `AllReduce` being the most called collective operation. In traditional HPC MPI applications, `AllReduce` oper-

Table 5.4: Benchmarks characterization by MPI calls and log behavior.

	MPI calls in main loop	#Procs	#Iters	N	Aggregated average log behavior per iteration			
					Point-to-Point		Collective Communications	
					Entries	Size	Entries (%↓)	Size (%↓)
Himeno	MPI_Irecv, MPI_Isend, MPI_Wait, MPI_AllReduce	48	12K	5K	208	28.5MB	48(↓75.00%)	1.5KB(↓81.82%)
		96	12K	5K	448	34.8MB	96(↓78.57%)	3.0KB(↓84.42%)
		192	12K	5K	944	51.5MB	192(↓81.25%)	6.0KB(↓86.36%)
		384	12K	5K	2.0K	68.7MB	384(↓83.33%)	12.0KB(↓87.88%)
		768	12K	5K	4.1K	82.2MB	768(↓85.00%)	24.0KB(↓89.09%)
Mocfe	MPI_Irecv,	48	10	4	69.9K	1.7GB	2.0K(↓75.00%)	140.4KB(↓78.53%)
	MPI_Isend,	96	10	4	150.5K	2.3GB	3.9K(↓78.57%)	280.9KB(↓81.59%)
	MPI_Waitall,	192	10	4	317.2K	3.0GB	7.9K(↓81.25%)	561.8KB(↓83.89%)
	MPI_Allreduce,	384	10	4	666.6K	3.9GB	15.7K(↓83.33%)	1.1MB(↓85.68%)
	MPI_Reduce	768	10	4	1376.3K	5.0GB	31.5K(↓85.00%)	2.2MB(↓87.12%)
SPhot	MPI_Irecv, MPI_Send, MPI_Waitall, MPI_Barrier	48	1K	400	235	29.3KB	192(↓83.33%)	1.5KB(↓96.67%)
		96	500	200	475	59.2KB	384(↓85.71%)	3.0KB(↓97.14%)
		192	250	100	955	118.8KB	768(↓87.50%)	6.0KB(↓97.50%)
		384	125	50	1.9K	238.0KB	1.5K(↓88.89%)	12.0KB(↓97.78%)
		768	62	25	3.8K	476.5KB	3.1K(↓90.00%)	24.0KB(↓98.00%)
Tealeaf	MPI_Irecv, MPI_Isend, MPI_Waitall, MPI_AllReduce	48	100	40	184.6K	1.7GB	107.9K(↓75.00%)	3.7MB(↓81.25%)
		96	100	40	387.4K	2.5GB	216.0K(↓78.57%)	7.4MB(↓83.93%)
		192	100	40	800.7K	3.7GB	431.3K(↓81.25%)	14.8MB(↓85.94%)
		384	100	40	1638.6K	5.4GB	863.3K(↓83.33%)	29.6MB(↓87.50%)
		768	100	40	3331.1K	7.8GB	1726.6K(↓85.00%)	59.3MB(↓88.75%)

ations frequently work with relatively small message sizes, however, emerging disciplines, such as deep learning, usually rely on medium and large messages sizes [8]. To illustrate the performance effect of this technique with different message sizes, we refine the analysis with the behavior of relevant collective communications in the Intel MPI Benchmarks [59] (IMB). Figure 5.4 compares the two logging methods on 48 processes (24 per node), showing—for different message sizes—the overhead induced over a non-fault-tolerant deployment (100% means that the logging method imparts no overhead). The volume of log data and the number of log entries are reported for the collective operations `Allgather`, `AllReduce`, and `Bcast`. For these collective operations, application logging shows notable reductions in the size and number of entries in the log, which in turn translates to a notable reduction in collective communication latency. Note that sudden changes in the logged data and the number of entries in the log correspond with Open MPI choosing a different implementation of collective communications depending on message size. In the general case, the reduction in the logging overhead when logging collective communications at the application level depends on the operation's semantic requirements and on the communication pattern in the point-to-point implementation of the collective communication. In the collective operations not presented here, application logging yields minor advantages over point-to-point logging in terms of log volume. This, in turn, translates to smaller performance differences between the two approaches. Note that, in any case, point-to-point collective logging is not compatible with the use of hardware-accelerated collective communication. Thus, point-to-point collective logging is expected to impart a significant overhead from which application logging is immune, independent of the log volume.

Figure 5.5 shows the memory consumption overhead introduced by the log for the testbed applications. To provide a better overview of the impact of this memory consumption, the maximum total size of the log has been normalized with respect to the available memory (number of nodes $\times$ 128 GB per node). Using CPPC and its spatial coordination protocol allows the log to be cleared upon checkpointing; thus, the maximum log size corresponds to the product of the number of iterations between checkpoints (N) and the addition of both point-to-point and collective communications log sizes per iteration (values shown in Table 5.4). Equivalently, the maximum number of entries in the log corresponds to the product of N and the total number of entries in the log. Figure 5.5 also presents this maximum. Both

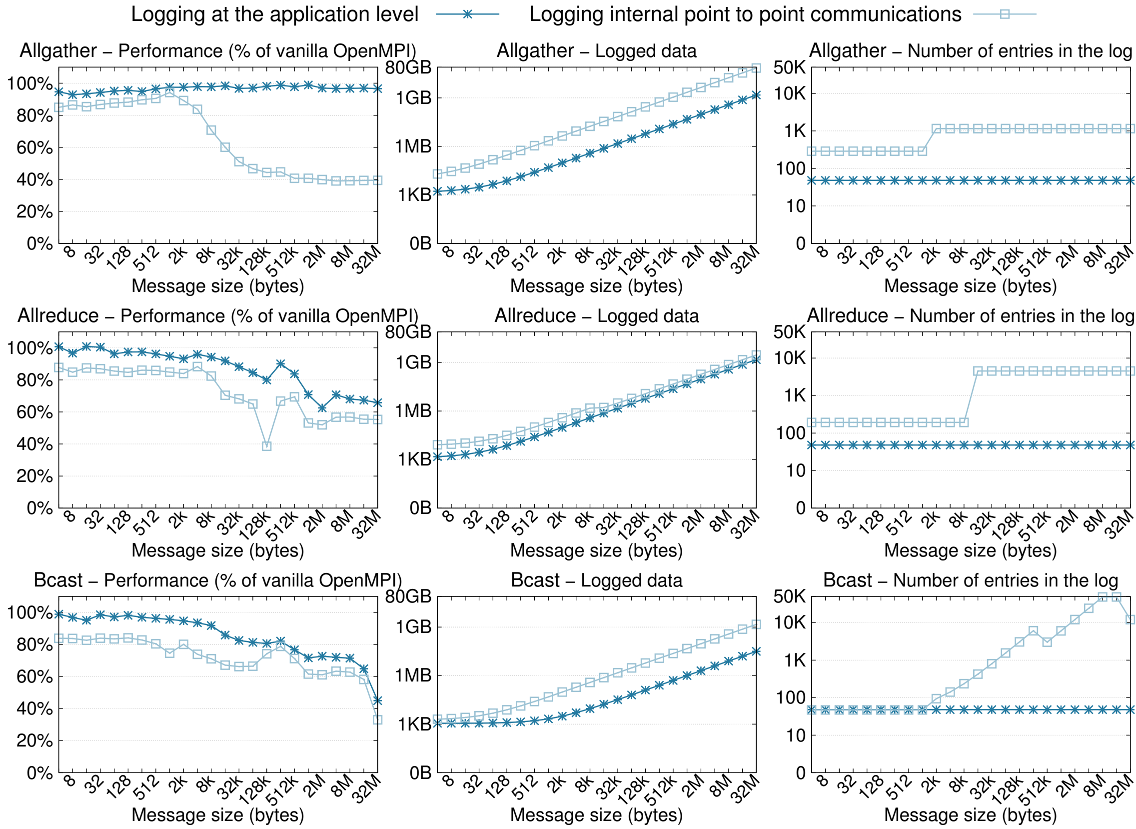


Figure 5.4: Application level vs. internal point-to-point logging of collective communications: performance, logged data, and number of entries in the log.

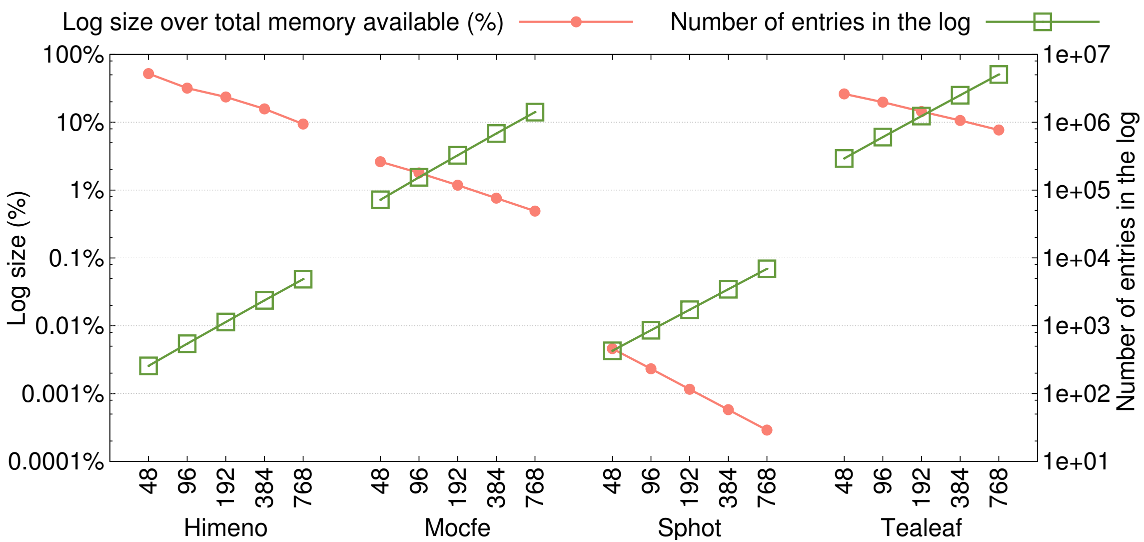


Figure 5.5: Log parameters when checkpointing: maximum log size expressed as the percentage of the total memory available and number of entries.

the number of entries in the log and the log size are represented in log scale. In all applications tested here, we see that the number of log entries increases as more processes run the applications, while the percentage of the total memory occupied by the log decreases. The Himeno application has the largest log size, ranging from 52% of the available memory when using 48 processes to 9% of the available memory when using 768 processes. Tealeaf has the second largest log, with a maximum log size ranging from 26.2% to 7.7% of the available memory when scaling up the application. In these applications, the bulk of the communication employs point-to-point calls, and the collective operations do not account for a significant portion of the logged data.

In fault-free executions, the local rollback protocol also introduces overhead in the form of communication latency and checkpoint volume. Figure 5.6 compares the overhead of global rollback and local rollback resilience in the absence of failures. It reports the absolute overhead in seconds, with respect to the original runtime (shown in Table 5.3), and reports the aggregated checkpoint file size (i.e., the total checkpoint volume from all processes). First, the amount of log management data that needs to be added to the checkpoint data is negligible. Therefore, checkpoint file sizes do not present a relevant difference between the local and global rollback solutions. Second, the overhead introduced by the local rollback is close to the overhead introduced by the global rollback solution. Most of the logging latency overhead is hidden, and its contribution to the total overhead of the local rollback approach is small compared to the contribution of the checkpoint cost. The overhead grows with the size of the checkpoint file. The SPhot and Tealeaf applications present the smallest checkpoint file sizes, with SPhot having checkpoint file sizes of 46–742 MB and TeaLeaf having checkpoint file sizes of 261–302 MB, with the upper ranges representing an increased number of processes. Given the very small overhead imparted by checkpointing on Tealeaf, for some experiments the overhead for the global rollback for Tealeaf is very slightly negative (less than 0.5% of the original runtime), presumably because of the optimizations applied by the compiler when the code is instrumented with CPPC routines. The overhead of the logging latency, while minimal in the overall runtime of this application, comes to dominate the cost of checkpointing in the failure-free overhead breakdown.

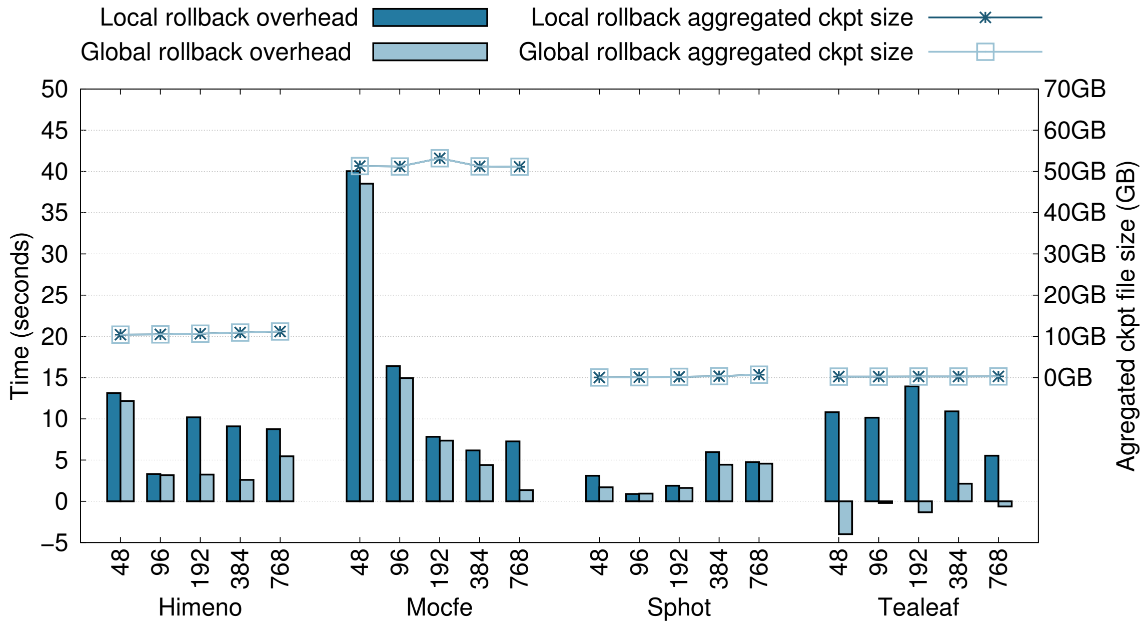


Figure 5.6: Absolute checkpointing overhead with respect to the non fault-tolerant version and aggregated checkpoint file sizes.

5.5.2. Operation Overhead in the Presence of Failures

The local rollback solution reduces the time required for recovering the applications when a failure occurs. The application is considered to have fully recovered when all processes (failed and survivors) have reached the execution step at which the failure originally interrupted the computation. In both the global and local recovery approaches, for a failed process this point is attained when it has finished re-executing the lost work. A survivor process is considered fully recovered when it has either re-executed all lost work in the global recovery scheme, or when it has served all necessary parts of the log to the failed (restarting) processes, and to other survivors that require it.

Figure 5.7 presents the reduction percentage of the local rollback recovery time over the global rollback recovery for both the survivor and failed processes. The improvement in the recovery times is very similar for all processes: these applications perform collective operations in the main loop, and all survivor processes that originally participated in the collective communications are also involved in their replay during recovery.

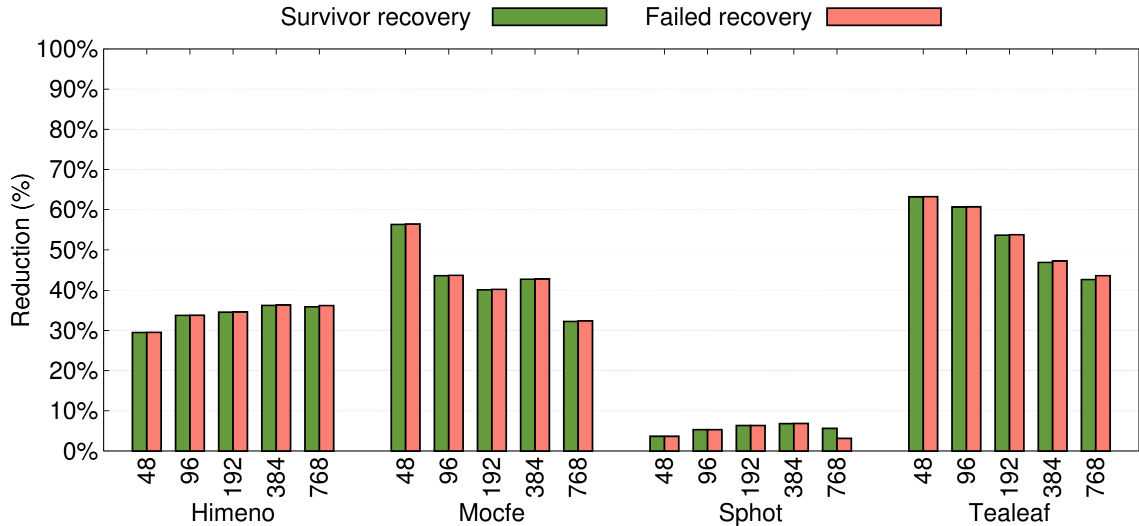


Figure 5.7: Reduction of the recovery times of survivor and failed processes with the local rollback (the higher, the better).

Figure 5.8 shows the times (in seconds) of the different operations performed during the recovery. The ULFM recovery times include the failure detection, the re-spawning of the replacement processes and the entire reconstruction of the MPI environment, including the communicators' revocation, shrinking and reconfiguration. The CPPC reading times measure the time spent during the negotiation of all processes about the recovery line to be used, and the reading of the selected checkpoint files by the failed ones. The CPPC positioning times include the time to recover the application's state of the rolled back processes, including the reconstruction of the application data (moving it to the proper memory location, i.e., the application variables), and the re-execution of non-portable recovery blocks (such as the creation of communicators). The CPPC positioning finishes when the failed processes reach the checkpoint call in which the checkpoint file was originally generated. Finally, we included the failed processes re-computation time which corresponds with the time spent from the checkpoint call in which the recovery files where generated until the failed processes have reached the execution step at which the failure originally interrupted the computation. The results in Figure 5.8 are summarized in Figure 5.9, which represents the percentage of the reduction in the recovery times that is due to each recovery operation in the failed processes.

Both approaches, local and global, perform the same ULFM operations during

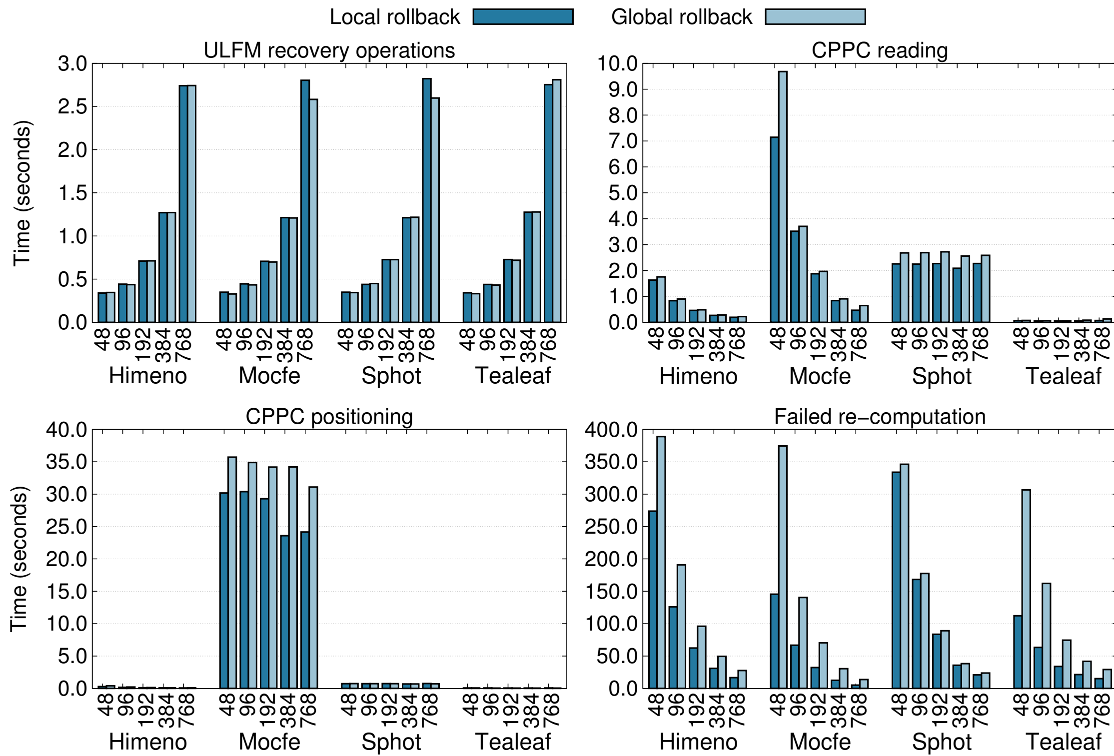


Figure 5.8: Times of the different operations performed during the recovery.

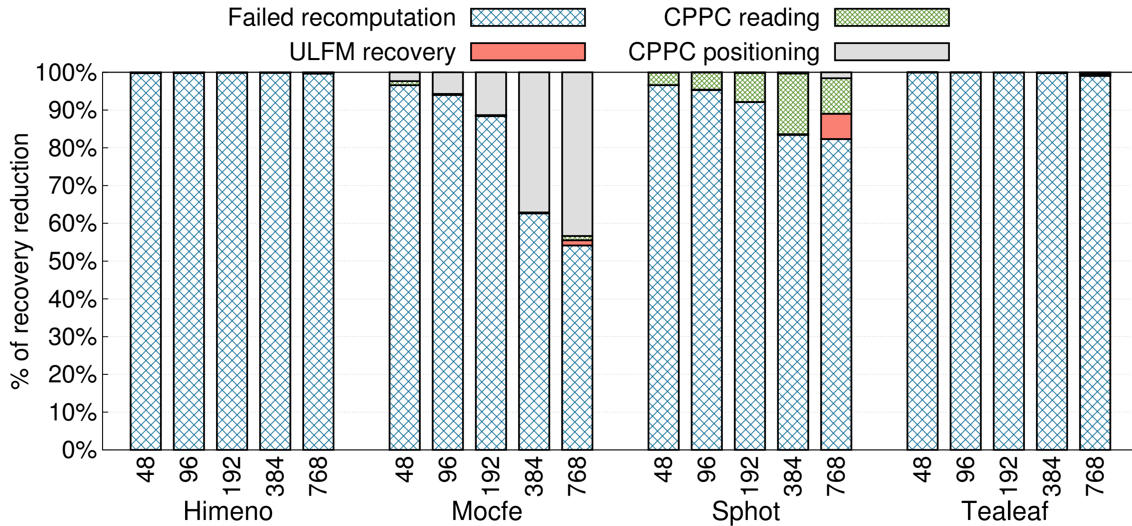


Figure 5.9: Percentage that each recovery operation represents over the reduction in the failed processes' recovery times.

the recovery, thus, no relevant differences arise. In the general case, the CPPC reading and positioning operations benefit from the local rollback strategy, as the number of processes reading checkpoint files and moving data in memory decreases. As can be observed in both figures, the re-computation of failed processes is the recovery operation with the largest weight in the reduction of the failed processes' recovery times. This happens because the failed processes perform a more efficient execution of the computation: no communications waits are introduced, received messages are rapidly available, and unnecessary message emissions from past states of the computation are skipped (although failed processes log them). Note that, in the testbed applications, the collective operations synchronize the failed and survivor processes recovery, thus, the time spent by the survivors in the replay of communications is almost the same as the failed processes re-computation times. In applications that do not present collective communications in the main loop, and where the applications are less tightly coupled, i.e the communications patterns between survivor and failed processes are less synchronizing, a larger improvement is expected.

The reduction in recovery time has a direct impact on the improvement of the overall execution time when a failure occurs. Figure 5.10 shows the reduction in runtime and energy consumption achieved when using local rollback instead of global rollback when faced with a failure. The extra runtime and the energy consumption are calculated as the difference between the original runtime/energy use and the runtime/energy use when a failure occurs. Energy consumption data is obtained using the `sacct` Slurm command. As an example, in the 48 processes execution, the overhead added (upon failure) to MOCFE's runtime is reduced by 53.09% when using the local rollback instead of the global rollback, and the extra energy consumed by failure management is reduced by 74%.

The SPhot application shows the lowest performance benefit. In this application, all point-to-point communications consist of *sends* from non-root processes to the root process. Thus, during the recovery, survivor processes do not replay any point-to-point communications to the failed rank, although they do participate in the re-execution of the collective communications. For this reason, the performance benefit for SPhot during the recovery is mainly due to the execution of the failed process computation without synchronization with other ranks. The *receives* are served from the message log, and most of its emissions have already been delivered

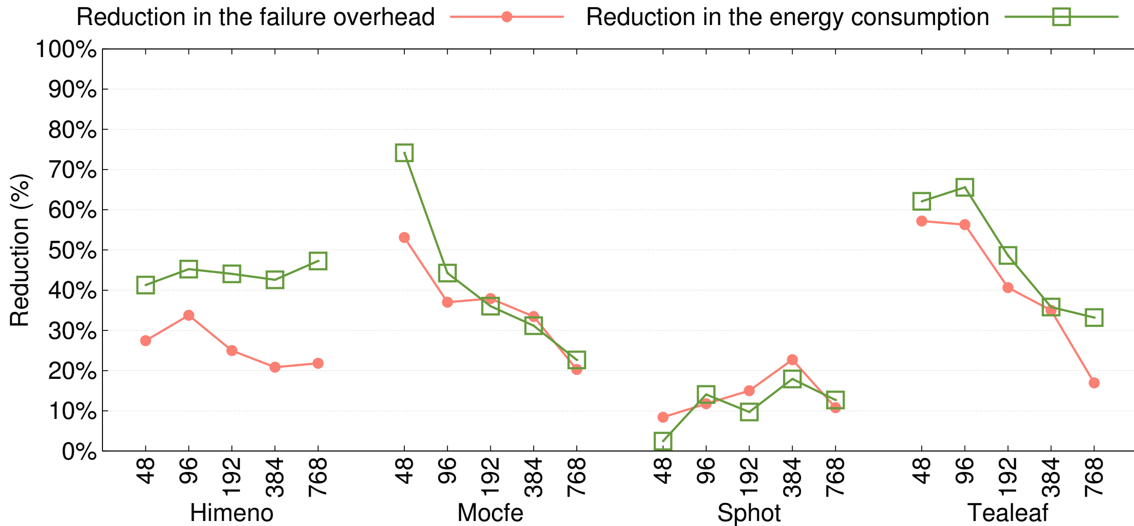


Figure 5.10: Reduction in the extra runtime and energy consumption when introducing a failure and using local rollback instead of global rollback (higher is better).

at the root process and are avoided. Even though these messages are not sent, they are still being logged to enable the recovery from future failures. In the other applications, where the communication pattern is more favorable, the local recovery permits a significant reduction in the extra time when compared to the global restart strategy.

5.5.3. Weak scaling experiments

In addition to the previous experiments, this section reports the results when doing weak scaling on the Himeno application, i.e., maintaining the problem size by process constant as the application scales out. These experiments compare the local rollback and global rollback under the same conditions as in the previous experiments (checkpointing frequency, failure introduction, etc.). Table 5.5 reports the configuration parameters and the original runtimes of the application, in minutes.

Table 5.6 (equivalent to Table 5.4 in the strong scaling experiments) characterizes these tests in terms of their log volume, reporting the MPI routines that are called in the main loop of the application (where the checkpoint call is located), the number of processes running the experiment, the total number of iterations run

Table 5.5: Original runtimes (in minutes) and configuration parameters of the Himeno weak scaling experiments.

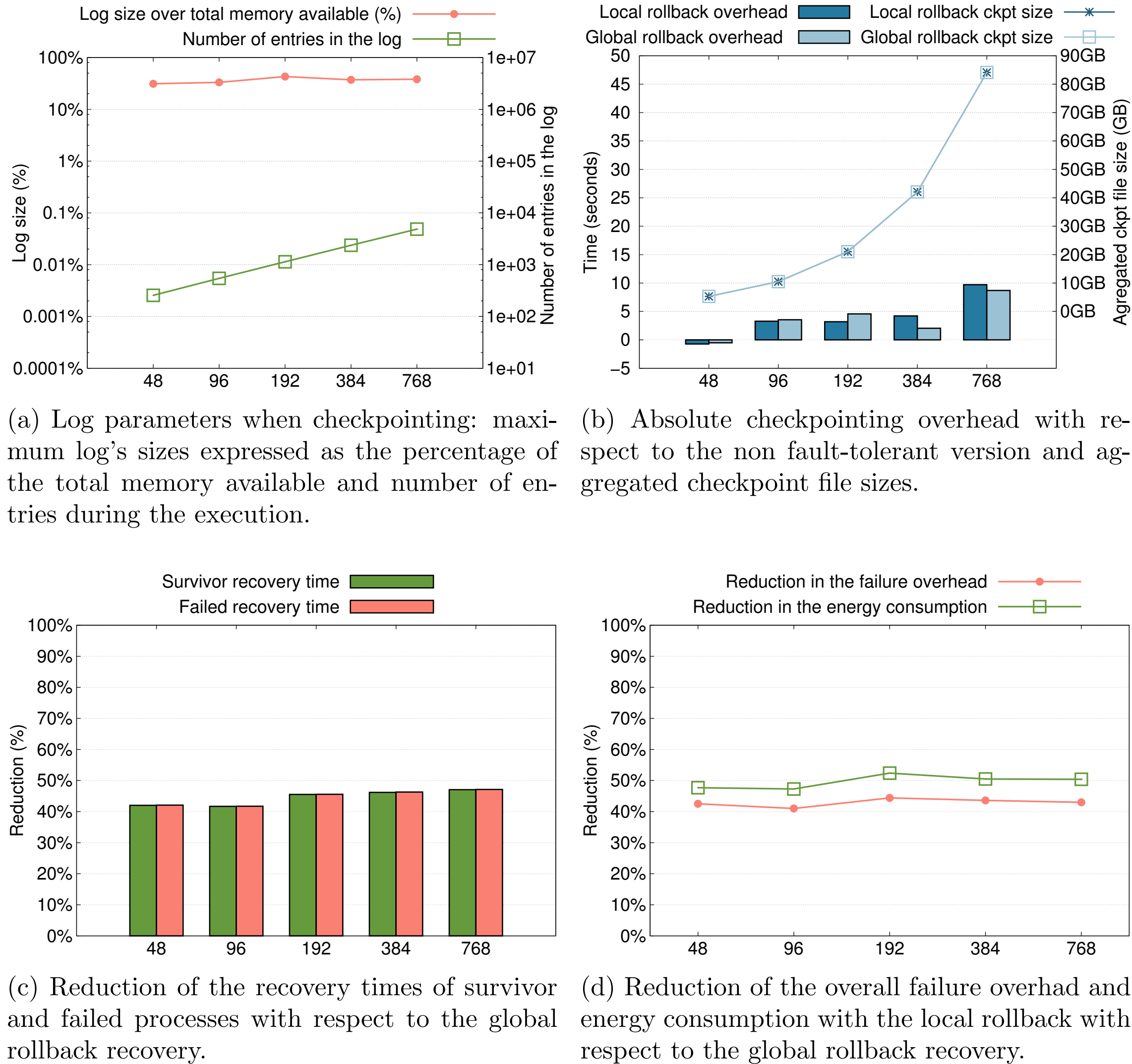
Himeno - Weak Scaling		
	Configuration parameters	Run time (minutes)
48 processes	Gridsize: 512 × 512 × 512, NN=12 000	9.16
96 processes	Gridsize: 1024 × 512 × 512, NN=12 000	9.14
192 processes	Gridsize: 1024 × 1024 × 512, NN=12 000	9.72
384 processes	Gridsize: 1024 × 1024 × 1024, NN=12 000	10.11
768 processes	Gridsize: 2048 × 1024 × 1024, NN=12 000	10.24

Table 5.6: Himeno characterization by MPI calls and log behavior (weak scaling).

	MPI calls in main loop	#Procs	#Iters	N	Aggregated average log behavior per iteration			
					Point-to-Point		Collective Communications	
					Entries	Size	Entries (%↓)	Size (%↓)
Himeno	`MPI_Irecv`, `MPI_Isend`, `MPI_Wait`, `AllReduce`	48	12K	5K	208	16.3MB	48(↓75.00%)	1.5KB(↓81.82%)
		96	12K	5K	448	34.8MB	96(↓78.57%)	3.0KB(↓84.42%)
		192	12K	5K	944	90.0MB	192(↓81.25%)	6.0KB(↓86.36%)
		384	12K	5K	2.0K	155.8MB	384(↓83.33%)	12.0KB(↓87.88%)
		768	12K	5K	4.1K	320.1MB	768(↓85.00%)	24.0KB(↓89.09%)

by the application, and the checkpointing frequency (N) indicating the number of iterations between two consecutive checkpoints. Table 5.6 also presents the aggregated average log behavior per iteration for the point-to-point and collective calls in terms of number of entries and size of the log. In comparison with the strong scaling experiments, there are no changes in the number of collective communications performed during the execution. On the other hand, even though the same number of point-to-point communications are performed in one iteration, the data transmitted presents a more abrupt increase when scaling out. Figure 5.11a shows, in log scale, the memory consumption overhead introduced by the log (equivalent to Figure 5.5 in the strong scaling experiments). As can be observed, in these experiments the increase in the data transmitted by the point-to-point communications results in a more steady maximum percentage of the total memory that is occupied by the log (between 31-42% of the available memory) when scaling out.

Figure 5.11b reports the absolute overheads introduce by the local rollback and global rollback in the absence of failures, and aggregated checkpoint file size (equiv-



(a) Log parameters when checkpointing: maximum log's sizes expressed as the percentage of the total memory available and number of entries during the execution.

(b) Absolute checkpointing overhead with respect to the non fault-tolerant version and aggregated checkpoint file sizes.

(c) Reduction of the recovery times of survivor and failed processes with respect to the global rollback recovery.

(d) Reduction of the overall failure overhad and energy consumption with the local rollback with respect to the global rollback recovery.

Figure 5.11: Results for the Himeno benchmark doing weak scaling (keeping the problem size by process constant).

alent to Figure 5.6). The overhead introduced by both proposals is low, and the aggregated checkpoint file size increases when scaling out, as the contribution from each process remains constant.

Figure 5.11c reports the reduction in the failed and survivors processes recovery times, while Figure 5.11d shows the reduction in run time and energy consumption achieved when using local rollback instead of global rollback when faced with a failure (equivalent to Figures 5.7 and 5.10). On average, the reduction on the total run time

is 42.9%, while the reduction in the energy consumption corresponds with a 49.7% on average. As in Himeno strong scaling experiments, most of the improvement is due to a more efficient execution of the failed processes.

5.6. Related Work

Message logging has been a very active research topic in the past decades because the flexibility it provides to the restart operations [3, 18, 20, 88, 89, 91, 107, 113]. Main drawbacks of this technique are related to the overhead introduced by the payload logging and its memory requirements. Sender-based logging is the most popular approach because it provides better performance during the fault-free execution. However, hybrid solutions have also been studied [90] to join the benefits of receiver-based strategies, i.e., lower restart overhead because the log can be locally available upon recovery. In any case, the memory requirements to maintain the log can represent a limiting factor. Strategies to reduce the memory consumption of the payload logging have been studied and are available in the literature. For instance, hierarchical checkpointing reduces the memory cost of logging by combining it with coordinated checkpointing [20, 89, 113]. Coordination is applied within a group of processes, while only messages between different groups are logged. When a process fails, all processes within its group have to roll back. This idea is also combined in [113] with "send-determinism" [27], to also reduce the number of logged events by assuming that processes send the same sequence of messages in any correct execution. In [86], dedicated resources (logger nodes) cooperate with the compute nodes by storing part of their message logs in the memory. Other strategies offer special consideration when logging collective communications—in order to decrease the memory footprint—by reducing the number of internal point-to-point messages of the logged collective [88] or by logging their result on a remote logger [74].

Our approach implements a local rollback protocol where only the failed processes are recovered from the last checkpoint, while consistency and further progress of the computation are enabled using ULFM and the message logging capabilities. A two-level message logging protocol is implemented: (1) at the MPI-level the Vprotocol [18] message logging component is used for sender-based message logging and pessimist event logging of point-to-point communications, while (2) collective

communications are optimally logged at the application level by CPPC. This strategy dissociates the collective communications from their underlying point-to-point expression, allowing the use of architecture-aware collective communications and reducing the memory footprint for their message logging. Additionally, the spatially coordinated protocol used for checkpointing by CPPC further contributes to this reduction, as checkpoints provide locations in which the log can be cleared. The proposal solves the issues that arise when a hard failure terminates one or several processes in the application by dealing with the communications that were interrupted by the failure and ensuring a consistent ordered replay so that the application can successfully resume the execution.

5.7. Concluding Remarks

This chapter proposed a novel local rollback solution for SPMD MPI applications that combines several methods to provide efficient resilience support to applications. ULFM fault mitigation constructs, together with a compiler-driven application-level checkpointing tool, CPPC, and the message logging capabilities of the Open MPI library-level Vprotocol combine to significantly reduce the resilience overhead of checkpoint and recovery. The ULFM resilience features are used to detect failures of one or several processes, maintain communication capabilities among survivor processes, re-spawn replacement processes for the failed processes, and reconstruct the communicators. Failed processes are recovered from the last checkpoint, while global consistency and further progress of the computation is enabled by means of message logging capabilities. Fine tracking of message sequences and partial message completion after failures permit the deployment of message logging over ULFM, and an original two-level logging protocol permits alternating the recovery level from library-level message logging to application-directed semantic-aware replay. Collective communications are logged at the application level, thereby reducing the log size and enabling the use with architecture-aware collective communications even after faults. The resultant local rollback protocol avoids the unnecessary re-execution overheads and energy consumption introduced by a global rollback, as survivor processes do not roll back to repeat computation already done; yet, the spatially coordinated protocol used by CPPC helps reduce the volume of stall logs

carried from past checkpoints. This combination of protocols proves singularly symbiotic in alleviating the shortcomings of each individual strategy.

The experimental evaluation was carried out using four real MPI applications with different checkpoint file sizes and communication patterns. The performance of the local rollback protocol has been compared with an equivalent global rollback solution. While in a failure-free execution, the required operations to maintain the logs imply a small increase in the overhead compared to the global rollback solution, in the presence of failures, the recovery times of both failed and survivor processes are noticeably improved. This improvement translates to a considerable reduction in the extra runtime and energy consumption introduced by a failure when using local rollback instead of global rollback.

Chapter 6

Local Recovery For Soft Errors

Detectable soft errors can be corrected by replacing corrupted data with correct data. Handling these types of errors at the software level before they are translated into failures enable the use of more efficient recovery techniques. In this chapter, we extend the FTI application-level checkpointing library to allow the developers of MPI applications the handling of soft errors during runtime. The new routines enable the construction of an ad hoc local recovery mechanism that considers the characteristics of the applications to minimize the overhead introduced when coping with soft errors.

This chapter is structured as follows. Section 6.1 introduces soft errors. The FTI library is briefly described in Section 6.2, and its extensions to support ad hoc local recovery are presented in Section 6.3. The integration of these new functionalities on three HPC applications is described in Section 6.4. The experimental evaluation of the tested applications is presented in Section 6.5. Section 6.6 provides an insight into related work. Finally, Section 6.7 comments on the main conclusions of this chapter.

6.1. Soft Errors

Corrupted memory regions are among the most common causes of failures. Memory faults lead to *hard errors*, when bits are repeatedly corrupted due to a physical defect (e.g., a short-circuit in a device that make a bit to be permanently at 1), and *soft errors*, when bits are transiently corrupted [9]. Most modern systems employ hardware mechanisms such as ECCs, parity checks or Chipkill-Correct ECC against data corruption to prevent SDCs and reduce DUEs [39]. Software techniques can be used to handle the remaining DUEs, reducing the number of fail-stop failures caused by them. Thus, reducing the recovery overhead, as it would prevent the overheads related to the re-spawning, the repairing of the communication environment, and even the repetition of computation originated by these failures.

Generally, in order to tolerate fail-stop failures, checkpointing APIs enable programmers to save a subset of the application variables in different levels of the memory hierarchy at a specific frequency. Such frequency is usually calculated as defined by Young [138] and Daly [36], taking into account the checkpoint cost and the MTTF of the underlying system. Recovery mechanisms for soft errors, may require access to past values of variables that differ from the ones checkpointed, and/or require a different frequency.

6.2. The FTI Checkpointing Library

FTI [12] is an application-level checkpointing library that provides programmers with a fault tolerance API. It is available at `https://github.com/leobago/fti`. Figure 6.1 shows an example of a fault tolerant code for the stop-and-restart checkpointing. The user is on charge of instrumenting the application code by: (1) marking those variables necessary for the recovery for their inclusion in the checkpoint files using the `FTI_Protect` routine, and (2) inserting the `FTI_Snapshot` in the computationally most expensive loops of the application. The `FTI_Snapshot` routine will generate checkpoint files (according to the checkpointing frequency specified by the user) and it will recover the application data during the restart operation. FTI applies a multi-level checkpointing protocol, which leverages local storage plus data replication and erasure codes to provide several levels of reliability and performance.

```
1  int main( int argc, char* argv[] )
2  {
3     MPI_Init( &argc, &argv );
4     FTI_Init();
5
6     [ ... ]
7
8     REGISTER_BLOCK_1:
9     <FTI_Protect(...) block>
10    [ ... ]
11
12    for(i = 0; i < nIters; i++){
13       FTI_Snapshot();
14       [ ... ]
15    }
16    [ ... ]
17
18    FTI_Finalize();
19    MPI_Finalize();
20 }
```

Figure 6.1: FTI instrumentation example

The user specifies a checkpoint interval (in minutes), for each one of the checkpointing levels. Consistency is guaranteed by using a coordinated checkpointing protocol (i.e., synchronizing processes during checkpointing), and locating checkpoints at safe points (i.e., code locations where no pending communications may exist). For more details about FTI and its operation the reader is referred to [11, 12].

6.3. FTI Extensions to Facilitate Soft Error Recovery

Fault tolerance techniques for HPC applications usually focus on fail-stop failures. One of the most widely used technique to cope with these failures is checkpoint/restart, which enforces a rollback to the last checkpoint upon a failure. Soft errors (i.e., errors affecting/corrupting part of the data in a non-permanent fashion) are usually handled using the same fault tolerance mechanisms. When a soft error arises, the recovery process is determined by how the application has been protected against failures. No protection will force the complete re-execution of the application

from the beginning. Protecting the application with traditional checkpoint/restart enables the recovery from the last checkpoint. This is inefficient because many of those soft errors could be handled locally in a much more time and energy efficient way.

In this work, we provide a framework that allows developers to exploit the characteristics of their MPI applications in order to achieve a more efficient resilience strategy against soft errors. We leverage the FTI library [12] with an extended API for this purpose. Two types of soft errors are considered: (1) DUEs, and (2) SDCs that are detected by software mechanisms [10, 99]. We implement a signal handler to inform the local process about the occurrence of a soft error. This is achieved by the handling of the signal sent to the affected process by the operating system when a DUE occurs, or by means of software error/data-corruption detection (e.g., online data monitoring). The OS can provide information of the affected memory page which allows to derive the affected variables. In any case, the MPI process handles the error and triggers a recovery (example code shown in Figure 6.2). Once the soft error is locally detected, the process will notify FTI of the error event by means of the extended API function `FTI_RankAffectedBySoftError`. The call to this function is non-collective and merely sets a flag in the local address space.

The recovery strategy relies on protecting as much application data as possible against soft errors by implementing an ad hoc mechanism that allows the regeneration of the protected variables to correct values when corrupted. Two different scenarios are possible: (1) the process affected by the soft error can regenerate the affected data using only local information, and (2) the process affected by the soft error needs the neighbor processes to be involved in the recovery. The second scenario requires knowledge of the soft error by those unaffected peers that need to participate in the recovery. In this work we focus on the first scenario, in which data can be recovered locally.

The `FTI_Snapshot` function, which is inserted in the computationally most expensive loop of the application, has been modified to check for soft error status at a user defined interval. The default soft error detection mechanism is global, involving all processes running the application. However, the programmer may also implement a custom detection mechanism in which only a subset of the application processes detect the soft error, i.e., scenarios in which the application processes are

```
1  void sig_handler_sigdue(int signo){
2      FTI_RankAffectedBySoftError();
3      /* Algorithm recovery code */
4      [ ...  ]
5  }
6
7  int main( int argc, char* argv[] )
8      /* SIGDUE: signal reporting a DUE */
9      signal(SIGDUE, sig_handler_sigdue);
10     /* Application initialization */
11     [ ... ]
12     for(i=0; i<N; i++){ /* Main loop */
13         int ret=FTI_Snapshot();
14         if (ret==FTI_SB_FAIL) {
15             int* status_array;
16             FTI_RecoverLocalVars( { ...  } )
17             /* Algorithm recovery code */
18             [...]
19         }
20         [...]
21     }
22     [...]
23 }
```

Figure 6.2: Detection of a soft error.

divided in groups, and only those in the group of the failed peer need to participate in the recovery of the failed process.

In most situations the regeneration of the protected variables will imply accessing a subset of data from a past state of the computation. The extended API function `FTI_RecoverLocalVars` allows the calling process to recover only a subset of the checkpointed variables from its last checkpoint. This is a non-collective function, thus, for completely local recoveries, it is called only by the processes actually affected by the soft error. Relying on the data from the last checkpoint to implement an ad hoc local recovery is useful for some applications, however, this is not the general case. For instance, let's consider an iterative application in which a relevant amount of the data used by the algorithm can be protected against soft errors by using the values of some variables at the beginning of the last iteration. Those variables may not be needed to perform a rollback when a fail-stop failure occurs. Thus, checkpointing them will lead to larger checkpoint files, and therefore, higher overheads. In addition to this, the ad hoc soft error recovery would imply

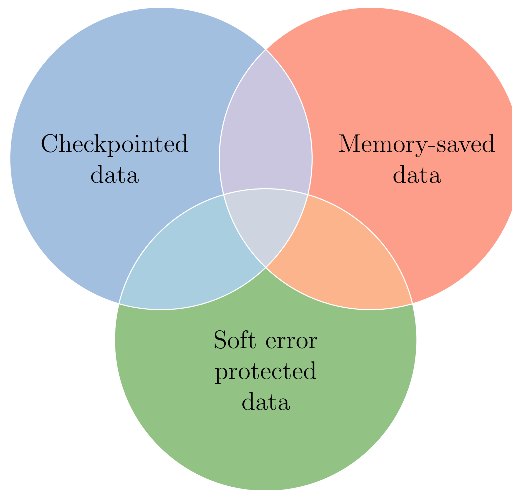


Figure 6.3: Memory Protection

checkpointing at every iteration of the execution, which will be translated in most cases into an inadequate checkpointing frequency for the application. To prevent this situation, a fast memory-saving mechanism is added to FTI. The `FTI_MemSave` and `FTI_MemLoad` routines are added to the library to perform a copy in-memory of a given variable and restore its contents, respectively. These routines allow the programmer to save only the subset of variables that are mandatory for the ad hoc soft error recovery procedure, and to do so with the required frequency.

The flexibility of these new extensions to the FTI API enables developers to implement a more efficient recovery strategy exploiting the particular characteristics of the application over a wide range of scenarios. As commented before, some variables can be checkpointed and some variables can be memory-saved using a different frequency in each case. Figure 6.3 shows a schematic view of the memory used by the application. Note that there is no restriction regarding the checkpointed, memory-saved, and protected data: the intersections between these three sets may or may not be empty. Protected data is defined as data that can be regenerated by using the checkpointed variables, the memory-saved ones, both, or neither of them (e.g., read-only data initialized to constant values can be recovered by reinitialization). The fact that a variable is checkpointed and/or memory-saved depends only on the application requirements when doing a rollback from a fail-stop failure or when doing an ad hoc soft error recovery.

6.4. Ad Hoc Local Recovery on HPC Applications

In order to demonstrate the value of the new extensions to the FTI library, this section describes the implementation of an ad hoc local recovery on three different HPC applications. All applications are instrumented with FTI to obtain checkpoint/restart fault tolerance support. Then, the new functionalities are used to protect a relevant part of the application data against soft errors.

6.4.1. Himeno

The Himeno benchmark [56] solves a 3D Poisson equation in generalized coordinates on a structured curvilinear mesh. A simplified overview of the application code instrumented with FTI is shown in Figure 6.4a. The call to FTI_Snapshot is located in the main loop of the application, i.e. the computationally most expensive loop. The only variables that need to be checkpointed by FTI are the array `p` (pressure) and the loop index `n`. The other variables used inside Himeno are either initialized at the beginning of the execution and read-only (`a`, `b`, `c`, `wrk1`, `bnd`), initialized in each iteration to constant values, or they merely depend on `p`(`n`). When a DUE hits, the failed process can re-initialize the read-only variables locally at any point of the execution, as shown in Figure 6.4b.

For Himeno, data protected against soft errors correspond with the read-only variables, which account for 85% of the memory used by the application, as it will be presented in Section 6.5. Himeno does not need to memory-save any variable to tolerate soft errors. Also, none of the checkpointed variables (i.e., pressure) can tolerate a soft error through local recovery. The data protected against soft errors are read-only variables that can be regenerated without being saved or checkpointed.

6.4.2. CoMD

CoMD [32] is a reference implementation of classical molecular dynamics algorithms and workloads as used in materials science. It is created and maintained by

```
1  [ ... ]
2
3  FTI_Protect { n, p }
4  signal(SIGDUE, sig_handler_sigdue);
5
6  for(n=0 ; n<NN ; ++n){
7    /* Application main loop */
8
9    int checkpointed = FTI_Snapshot();
10
11   for i=1:imax-1, j=1:imax-1, k=1:imax-1{
12         /*  Read only data:  { a,b,c,p,wrk1,bnd } */
13         /*  Write only data:  { s0,ss,gosa,wrk2 } */
14   }
15
16   for i=1:imax-1, j=1:imax-1, k=1:imax-1{
17         /*  Read only data:  { wrk2 } */
18         /*  Write only data:  { p }    */
19   }
20
21 }
22
23 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2
3    FTI_RankAffectedBySoftError();
4
5    /* Algorithm recovery code */
6
7    /* Re-initialize the       */
8    /* read-only data:         */
9    /*  { a, b, c, wrk1, bnd } */
10
11 }
```

(b) Handler for local recovery.

Figure 6.4: Himeno simplified pseudocode.

the Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). Figure 6.5a presents the code instrumented with FTI. To tolerate fail-stop failures, FTI checkpoints the variables `nAtoms`, `gid`, `iSpecies`, `r`, `p`, `f`, and `iStep`.

In contrast to the Himeno application, there are no read-only variables within each timestep. However, some variables are read-only within each invocation of the `computeForce` method, which consumes around 93% of the total runtime. These read-only variables (`gid`, `iSpecies`, `r`, and `p`) can be protected against soft errors by memory-saving their contents before the invocation of the `computeForce` method and replacing the default error handler with a custom one within the scope of this function. Within the error handler (shown in Figure 6.5b), their values will be set to those valid when the method was invoked. In CoMD, the protected variables that can be regenerated are both memory-saved and checkpointed.

6.4.3. TeaLeaf

TeaLeaf [125] is a mini-app from the Mantevo project that solves the linear heat conduction equation. A simplified overview of the application FTI instrumented code is shown in Figure 6.6a. A CG (Conjugate Gradient) solver is performed within each step of the main loop. Most of the variables are initialized for each CG solver run. Therefore, locating the checkpoint call in the most external main loop avoids checkpointing all the internal data of the CG solver, reducing the checkpointed data by 87% (and thus, the checkpointing overhead) while providing an adequate checkpointing frequency. To tolerate fail-stop failures, only the main loop index `tt` and the `density` and `energy` arrays need to be checkpointed.

There are read-only variables within a CG solver, initialized for each solver using local `density` arrays. These read-only variables can be protected against soft errors: they can be regenerated using the version of the `density` array when the CG solver was invoked. Figure 6.6b shows the error handler for TeaLeaf to perform the local recovery. Note that the handler can be invoked at any time during the CG solver execution, and the `density` array may have been modified. Thus, the current value of the `density` array is preserved in a temporal variable, while its memory-saved version is used for the regeneration of the read-only protected variables. In TeaLeaf none of the variables protected against soft errors are checkpointed or memory-saved.

```
1  [ ... ]
2
3  FTI_Protect { nAtoms, gid, iSpecies, r, p, f, iStep }
4
5  for (iStep=0; iStep<nSteps; iStep++){
6    /* Application main loop */
7
8    int checkpointed = FTI_Snapshot();
9
10   if(iStep%printRate==0)sumAtoms(s);
11   advanceVelocity( ... );
12   advancePosition( ... );
13   redistributeAtoms( ... );
14
15   signal(SIGDUE, sig_handler_sigdue);
16   FTI_MemSave { gid, iSpecies, r, p }
17   computeForce( ... );
18   signal(SIGDUE, SIG_DFL);
19
20   advanceVelocity( ... );
21   kineticEnergy(s);
22
23 }
24
25 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2
3    FTI_RankAffectedBySoftError();
4
5    /* Algorithm recovery code */
6
7    /* Recover values of read-only     */
8    /* variables within computeForce: */
9     FTI_MemLoad { gid, iSpecies, r, p }
10
11 }
```

(b) Handler for local recovery.

Figure 6.5: CoMD simplified pseudocode.

```
1  [ ... ]
2
3  FTI_Protect { tt, density, energy }
4  signal(SIGDUE, sig_handler_sigdue);
5
6  for(tt = 0; tt < end_step; ++tt){
7    /* Application main loop: diffuse method */
8
9    int checkpointed = FTI_Snapshot();
10
11   FTI_MemSave { density }
12
13   /* CG Solver */
14   cg_init_driver(chunks, settings, rx, ry, &rro);
15   for(t = 0; t < max_iters; ++t){
16     cg_main_step_driver(chunks, settings,
17                         t, &rro, error);
18     halo_update_driver(chunks, settings, 1);
19     if(fabs(*error) < eps) break;
20   }
21
22   solve_finished_driver(chunks, settings);
23
24 }
25
26 [ ... ]
```

(a) Instrumenting for checkpointing and soft error correction.

```
1  void sig_handler_sigdue(int signo){
2
3    FTI_RankAffectedBySoftError();
4
5    /* Algorithm recovery code */
6
7    memcpy(aux_density, density, sizeof(double)*sizeD);
8    FTI_MemLoad { density }
9    /* Regenerate read-only variables using */
10   /* the density values previous to the   */
11   /* CG solver invocation                  */
12   [...]
13   memcpy(density, aux_density, sizeof(double)*sizeD);
14
15 }
```

(b) Handler for local recovery.

Figure 6.6: TeaLeaf simplified pseudocode.

Table 6.1: Hardware platform details.

	CTE-KNL cluster
Operating system	Linux Operating System
Nodes	1x Intel Xeon Phi CPU 7230
	1.30 GHz, 64 cores
	96 GB (@90 GB/s) main memory
	16GB (@480 GB/s) High Bandwidth Memory (HBM) in cache mode
Network	Intel OPA 100 Gbit/s Omni-Path interface
Local storage	120 GB SSD
Remote storage	GPFS via ethernet 1 GBit/s
MPI version	Intel MPI v2017.1.132

6.5. Experimental Evaluation

The experimental evaluation was performed on 4 nodes of the CTE-KNL cluster at the Barcelona Supercomputing Center (BSC-CNS). Each node is composed by an Intel Xeon Phi Knights Landing processor, described in Table 6.1 The technique proposed in this work leverages all four storages levels efficiently: HBM in cache mode is used to store the computation variables, the main memory is used to store the extra data necessary for the local recovery, the SSDs are used to store the multilevel checkpoint files and the file system is used to store checkpoints required to comply with batch scheduler limitations (i.e., 24-48 hours jobs).

In order to quantify the results, we determined the relative overheads introduced by our modifications with respect to the original code. The measurements are performed by increasing the number of processes while keeping the problem size per process constant (i.e., *weak scaling*). Experiments were run using 64 processes per node, one per core, as hyperthreading is disabled on the CTE-KNL. The parameters used for each experiment are given in Table 6.2, together with the original completion runtimes without any FTI instrumentation. For statistical robustness, we performed the experiments multiple times, thus both in the table and in the rest of the section, each reported measurement corresponds to the mean of ten executions.

Table 6.2: Weak scaling configurations and original application runtimes.

	NProcs	Parameters	Runtime (seconds)
Himeno	64	gridsize:513x2049x1025	498.54
	128	gridsize:1025x1025x2049	501.92
	256	gridsize:2049x2049x1025	502.61
CoMD	64	x=128, y=128, z=256, N=100	561.69
	128	x=128, y=256, z=256, N=100	582.10
	256	x=256, y=256, z=256, N=100	591.66
TeaLeaf	64	cells:3000x3000, timesteps=20	256.71
	128	cells:6000x3000, timesteps=20	482.70
	256	cells:6000x6000, timesteps=20	765.25

6.5.1. Memory Characterization

As observed, Himeno is the application with the largest amount of used memory (82.03% of the total memory that is available for each experiment) which was expected as Himeno is a memory bounded application. In contrast to that, CoMD and TeaLeaf only use a small fraction of the available memory. Most molecular dynamic applications such as CoMD have small memory footprint. On the other hand, TeaLeaf is a sparse-matrix computationally-bounded application, which explains its reduced memory consumption. All in all, these three applications give us a wide spectrum with significantly different memory usages to analyze the impact of the proposed resilience schemes.

First, we characterize the memory footprint of the applications in Table 6.3. The table reports as Used Memory the percentage of memory used by the application in comparison to the total memory available in the running nodes. Additionally, the table presents (1) the percentage of the used memory that needs to be Checkpointed, (2) the percentage of the used memory that needs to be Memory-saved using the new FTI extensions, and (3) the percentage of the used memory that is Protected against soft error.

The checkpointed datasets are also heterogeneous across the different testbed applications. Himeno checkpoints 7.14% of the used memory which corresponds to aggregated checkpoint file sizes varying between 5.51GB and 22.03GB when increasing the number of processes. CoMD generates checkpoint file sizes ranging between

Table 6.3: Memory characterization of the tested applications.

	Himeno			CoMD			TeaLeaf		
	64p	128p	256p	64p	128p	256p	64p	128p	256p
Used Memory (%avail.)	82.03	82.03	82.03	10.92	10.90	10.92	1.05	1.05	1.05
Checkpointed (%used)	7.14	7.14	7.14	89.24	89.29	89.24	13.88	13.88	13.88
Memory-saved (%used)	0.00	0.00	0.00	62.42	62.46	62.42	6.94	6.94	6.94
Protected (%used)	85.71	85.71	85.71	62.42	62.46	62.42	51.37	51.37	51.37

9.16GB and 36.65GB as the applications scale out, which correspond to checkpointing around 89.3% of the used memory. Finally, TeaLeaf checkpoints 13.88% of the used memory, and generates the smallest checkpoint file sizes of the testbed applications, which range between 0.14GB and 0.55GB. This shows that not all the data used by the applications needs to be checkpointed to perform a successful global rollback after a failure, and accounts for significantly different checkpointing overheads.

Another relevant difference between the three applications is the memory footprint of the recovery strategy (the data that is memory-saved) and the protection coverage that is achieved by doing so. The table reports both quantities (Memory-saved and Protected data) as the percentage of the used memory. As commented before, a memory corruption on the protected data is tolerated using the local recovery strategies described in Section 6.4, while corruptions on other datasets will lead to a global rollback from the last checkpoint. The potential benefits of this technique will be defined by the amount of protected, memory-saved, and checkpointed data; as well as by the amount of computation that needs to be repeated when doing a global rollback. Scenarios with a large protection coverage and small amounts of memory-saved data (i.e. at a low cost) can introduce important performance benefits in the recovery upon a soft error, and, thus, in the total execution runtime. In the ideal scenario, a large amount of data can be regenerated from a small portion of memory-saved data, however it is not always the case. The testbed applications cover very different situations regarding protected and memory-saved data. For the

Himeno application, we can protect 85.71% of the used data without memory-saving any variables, thus, at no cost. In the case of CoMD, we can protect 62.42% of the application data from soft errors by memory-saving the same amount of data, which accounts for datasets 30% smaller than the checkpointed data. Finally, for TeaLeaf, 51.37% of the used data is protected by memory-saving only 6.94% of the datasets, which corresponds to half of the checkpointed data.

The following sections evaluate the overheads that the implementation of the local recovery introduces in the applications during the fault-free execution and study the benefits that are obtained when this technique is used to recover the datasets affected by a soft error.

6.5.2. Overhead in the Absence of Failures

This section studies the overheads introduced in the fault-free execution. Figure 6.7 shows the relative overheads with respect to the original execution runtimes (reported in Table 6.2) for three different experiments. Firstly, *FTI-instrumented* experiments measure the performance penalty that is introduced by the calls to the FTI library added to the application code, but neither checkpointing nor memory-saving any dataset. The relative instrumentation overhead is low, on average, 0.66% and always below 1.77%.

Secondly, we study the overhead introduced when checkpointing for a global rollback. The *Checkpointing* experiments generate checkpoint files at the optimal frequency, calculated as defined by Young [138] and Daly [36]. These experiments show the overhead that is introduced when checkpointing to tolerate fail-stop failures, enabling a global rollback recovery after a failure. The relative checkpointing overhead is, on average, 5.92% and never exceeds 14.88%. This overhead is tied to the checkpoint file size (i.e., the time writing the data) and the synchronization cost that is introduced by the coordinated checkpointing provided by FTI.

Lastly, *Checkpointing+Memory-saving* experiments study the overhead introduced when not only checkpoints are taken but also the necessary variables are memory-saved. Thus, allowing the recovery using a global rollback upon a fail-stop failure and enabling the usage of the local recovery when a soft error hits any of the

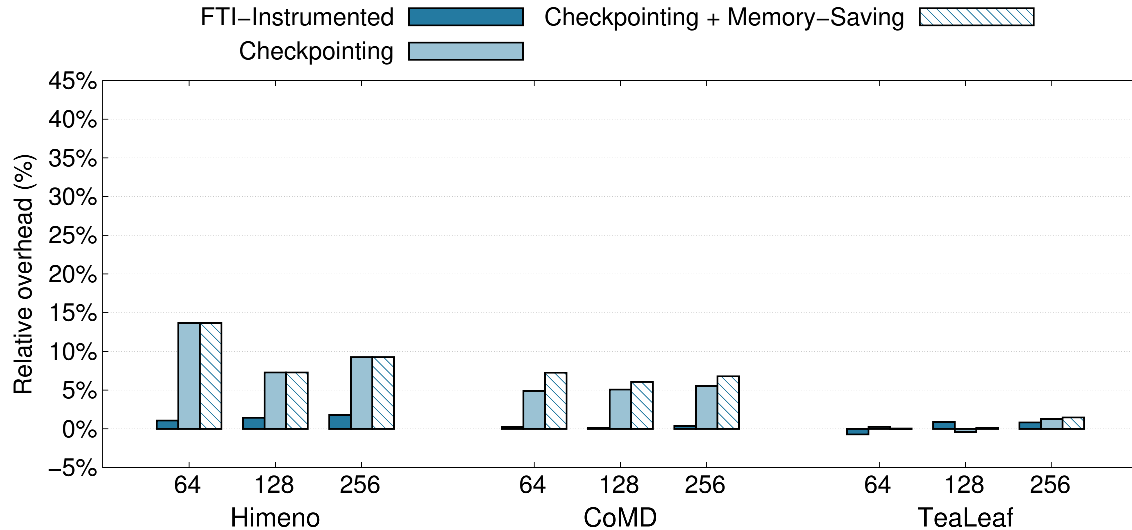


Figure 6.7: Relative overheads with respect to application original runtimes in a fault-free execution.

protected variables. In these experiments, checkpointing is performed at the optimal checkpointing frequency, while memory-saving data is performed at every iteration of the main loop of the application. The overhead introduced by the memory-saving operations is negligible, and the cost of the in-memory copy accounts for an increase over the checkpointing overhead of, on average, 0.57%, and always below 2.36%. As we see in Figure 6.7, Himeno does not incur any extra overhead because no data is memory-saved. On the other hand, for CoMD and TeaLeaf the local recovery strategies described in Section 6.4 require a memory copy of a subset of the application data in every iteration for the regeneration of the protected variables upon corruption.

6.5.3. Overhead in the Presence of Failures

This section compares the performance upon errors of the global rollback recovery and the local recovery. In these experiments, a soft error is introduced when approximately 75% of the execution has been completed by signaling one of the processes running the application. In the local recovery, the affected process handles the signal and recovers from the soft error to continue the execution, as depicted in Section 6.4. On the other hand, in the global rollback the signal aborts the execution

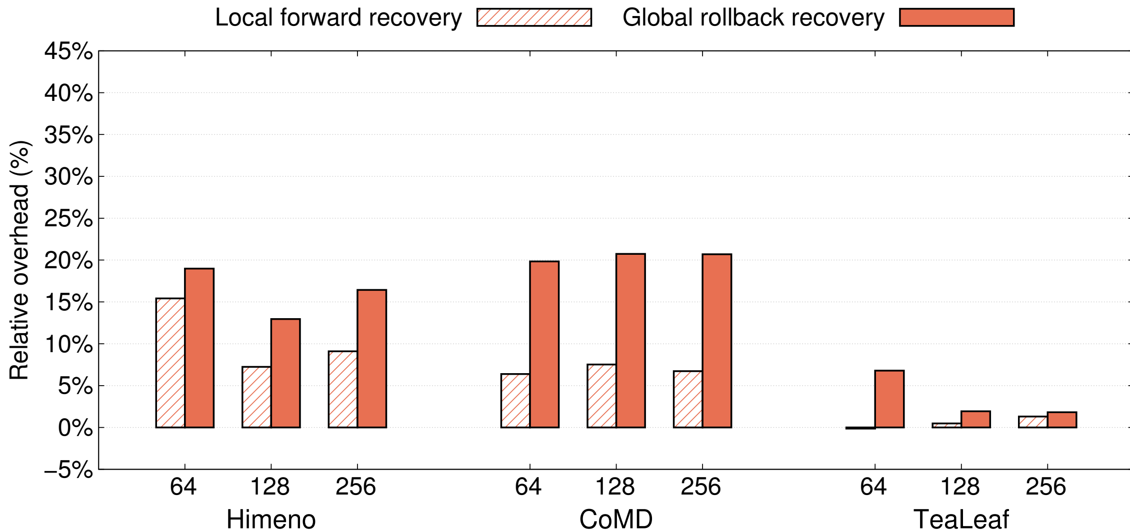


Figure 6.8: Relative overheads with respect to application original runtimes when introducing a failure.

and a global restart takes place, recovering all the processes running the application from the most recent available checkpoint file.

Figure 6.8 presents the overheads introduced when handling the failure with each strategy. The overheads are calculated as the difference between the original execution runtimes (reported in Table 6.2) and the runtimes when introducing a soft error. Thus, the overheads correspond to the extra time consumed by each proposal in order to tolerate the error. Additionally, Table 6.4 details the reduction in the failure overhead that the local recovery achieves over the global rollback, both in absolute and relative terms.

The local recovery avoids that all processes read the checkpoint files at the same instant, as it occurs in a global rollback recovery. More importantly, it avoids rolling back the processes running the application, avoiding the repetition of computation already done, and, therefore, reducing the failure overhead and energy consumption. On average, the reduction corresponds to 7.38% of the execution time, with a maximum of 14.01% for CoMD. Note that this reduction is determined by two factors: the size of the checkpoint files and the amount of re-computations that has to be performed. For Himeno and CoMD, the reduction in the overhead increases as more processes run the applications. On the other hand, for TeaLeaf, the reduction de-

Table 6.4: Reduction in the overhead when using the local recovery instead of the global rollback upon a soft error: absolute value (in seconds) and percentage value (normalized with respect to the original execution runtime).

NProcs	Reduction in the overhead: seconds [%]		
	Himeno	CoMD	TeaLeaf
64	17.93 [3.60%]	75.75 [13.49%]	17.65 [6.88%]
128	29.06 [5.79%]	77.01 [13.23%]	7.07 [1.46%]
256	37.51 [7.46%]	82.91 [14.01%]	3.93 [0.51%]

creases when scaling out, because when using the optimal checkpointing frequency, the amount of computation to be repeated in the global rollback decreases, e.g. in the 256 processes experiment a checkpoint is taken in every iteration and barely no computation is repeated when rolling back.

6.6. Related Work

Detectable soft errors related to memory corruption may be handled by the application before triggering a failure, and hence, avoiding the interruption of the execution when they arise. Some of the techniques presented in the literature to do that are based on redundancy [45, 47, 49]. Comparing the results of the replicas enables error detection, and error correction in the case of triple-redundancy. However, replication requires substantially more hardware resources which rapidly becomes prohibitively expensive.

Other approaches exploit the characteristics of the particular application/algorithm by implementing ad hoc recovery techniques, which can introduce significant performance benefits in the recovery process. Traditionally, checkpoint/restart relies on a global backward recovery, in which all processes in the application rollback to a previous committed state and repeat the computation done from that point on. Increasing the checkpointing frequency reduces the amount of computation to be repeated, decreasing the failure overhead but increasing the checkpointing overhead in terms of performance, storage and bandwidth. Forward recovery strategies attempt to build a new application state from which the execution can resume,

without rolling back to a past checkpointed state and repeating all the computation already done, thus, significantly reducing the recovery overhead. This is the case of partial re-computation [117] which is focused on limiting the scope of the recomputation after a failure, and ABFT techniques. ABFT was originally introduced by Huang and Abraham [57] to detect and correct permanent and transient errors on matrix operations. The method is based on the encoding of data at a high level and the design of algorithms to operate on the encoded data. ABFT has been used in combination with disk-less checkpointing for its usage in matrix operations [17, 31], and it has been implemented on algorithms such as the High Performance Linpack (HPL) benchmark [38], Cholesky factorization [54], algorithms using sparse matrices and dense vectors [99], and tasks-based applications [137]. Strategies such as the proposed by Pawelczak et al. have implemented ABFT for TeaLeaf [99] as an alternative method for protecting sparse matrices and dense vectors from data corruptions, which can be combined with this proposal to obtain a fast local recovery upon a soft error corrupting the protected variables. Another key aspect of the recovery process is the locality, i.e. the number of processes not affected by the error that need to be involved in the recovery. Restricting the recovery actions to only those processes affected by the error, or a subset of the processes (i.e., in those scenarios that failed processes cannot recover on their own and require the participation of neighbor processes) also contributes towards the efficiency of the fault tolerance solution.

This work builds on top of those strategies exploiting the particularities of the application to boost the recovery and aims to provide a generic and intuitive way for applications to implement and leverage ad hoc recovery strategies. This new extensions to the FTI library provide the necessary flexibility to facilitate the implementation of custom recovery mechanisms.

6.7. Concluding Remarks

This work presents the extension of the FTI checkpointing library to facilitate the implementation of ad hoc recovery strategies for HPC applications, to provide protection against those soft errors that cannot be corrected by hardware mechanisms. The usage of these new extensions does not disturb the operation of traditional

checkpoint/restart to handle any other type of failures by means of a global rollback to the recovery line established by the last valid set of checkpoint files.

The evaluation of these new functionalities was achieved by implementing local recovery strategies on three different applications. In all of them, the protected data account for an important part of the memory used by the application, thus, providing a good coverage ratio. In order to provide this protection, one application does not need to memory-save any data, while the other two do: in one, the amount of memory-saved and protected data is the same, while in the other the protected data is 7.4 times larger than the memory-saved data. In all cases, any process can recover locally from the soft error that corrupted the protected data by re-initialize/regenerate it. The experimental evaluation demonstrates the low overhead introduced by the proposal, while it has been proved to provide important performance benefits when recovering from memory corruptions.

Although in the experimental evaluation the protected variables are identified by the user, they could be automatically detected by a compiler as it only involves a state analyses of the application code.

Chapter 7

Conclusions and Future Work

As HPC systems continue to grow larger and include more hardware components of different types, the meantime to failure for a given application also shrinks, resulting in a high failure rate overall. Long-running applications will need to rely on fault tolerance techniques not only to ensure the completion of their execution in these systems but also to save energy. However, the most popular parallel programming models HPC applications use to exploit the computation power provided by supercomputers lack fault tolerance support.

Checkpoint/restart is one of the most popular fault tolerance techniques, however, most of the research present in the literature is focused on stop-and-restart strategies for distributed-memory applications in the event of fail-stop failures. In this context, this thesis makes the following contributions:

- **An application-level checkpointing solution to cope with fail-stop failures on hybrid MPI-OpenMP applications.** A new consistency protocol applies coordination intra-node (within each OpenMP team of threads), while no inter-node coordination is needed (between different MPI processes). The proposal reduces network utilization and storage resources in order to optimize the I/O cost of fault tolerance, while minimizing the checkpointing overhead. As well, the portability of the solution and the dynamic parallelism provided by OpenMP enable the restart of the applications on machines with different architectures, operating systems and/or number of cores, adapting

the number of running OpenMP threads for the best exploitation of the available resources.

- **An application-level checkpointing solution to cope with fail-stop failures on heterogeneous applications.** This proposal is built upon HPL, a library that facilitates the development of OpenCL-based applications, thus, it is not tied to any specific vendor, operating system or hardware platform. A host-side application-level checkpointing is implemented for fault tolerance support. The host-side approach (placing checkpoints in the host code between kernels invocations) avoids the inclusion in the checkpoint files of the device/s private state, while the application-level strategy avoids the inclusion of not relevant host data, thus minimizing the checkpoint files size. In addition, the design decisions maximize portability and adaptability, allowing failed executions to be resumed using a different number of heterogeneous computing resources and/or different resource architecture. The ability of applications to adapt to the available resources will be particularly useful for heterogeneous cluster systems.

- **A resilience checkpointing solution that can be generally applied to SPMD MPI applications.** Traditional stop-and-restart recoveries strategies are avoided by exploiting the new functionalities provided by the ULFM interface, which proposes resilience features for their inclusion in the MPI standard. The resulting resilient applications are able to detect and react to failures without stopping their execution, instead, the failed processes are re-spawned, and the application state is recovered through a global rollback.

- **A local rollback protocol that can be generally applied to SPMD MPI applications by combining ULFM, application-level checkpointing and message logging.** This proposal prevents processes not affected by a fail-stop failure from discarding their state, performing a rollback to a previous checkpoint, and repeating computations already done. Instead, the rollback is restricted to those processes that have failed. Failed processes are recovered from the last checkpoint, while global consistency and further progress of the computation is enabled by means of an original two-level logging protocol. Point-to-point communications are logged by the Open MPI Vprotocol component, while collective communications are optimally logged

at the application level, thereby reducing the log size and enabling the use with architecture-aware collective communications.

- **A set of extensions to an application-level checkpointing library, FTI, to facilitate the implementation of custom recovery strategies for MPI applications to cope with soft errors.** These types of errors correspond with data corruptions in DRAM or SRAM that cannot be corrected by hardware mechanisms and, most of the time, they are translated into fail-stop failures. Handling these types of errors at the software level before they are translated into failures enables the use of more efficient recovery techniques. The new functionalities provide programmers different mechanisms to save and access data from a past state of the computation, without requiring it to be checkpointed nor imposing restrictions on the saving frequency. The flexibility of the extensions enables to exploit the particular characteristics of the application over a wide range of scenarios, facilitating the implementation of ABFT techniques, and local forward recoveries.

All these proposals were implemented on the CPPC checkpointing tool because it provides a transparent, portable, application-level checkpointing solution, which is based on the automatic instrumentation of code. The only exception corresponds with the extensions to facilitate the implementation of custom recoveries upon soft errors. These extensions require the participation from the programmer, and they were implemented on the FTI application-level checkpointing library, whose operation is based on the instrumentation of code by the programmer.

To further reduce the cost of tolerating a failure, future work includes the optimization of the local rollback protocol presented in Chapter 5 to perform a more efficient replay of the communications needed for the progress of the failed processes. In the proposed strategy, all processes involved in the original execution of a collective operation are also involved in its replay. A custom replay of collective communications, e.g., replacing the collective operation by one or a set of point-to-point operations, has the potential to further improve the performance of the recovery and reduce its synchronicity. At the next level, the replay process can benefit of using remote memory access operations provided by MPI to enable the implementation of a receiver-driven replay. This strategy will allow failed processes to obtain the message log from survivor processes without their active involvement

during the recovery procedure.

In addition, future work includes the study of ad hoc recovery strategies to handle soft errors on Monte Carlo applications and Genetic Algorithms, in which memory corruptions over random data can be regenerated on the fly.

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

- International Journals (5)

  - N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín. Local Rollback for Resilient MPI Applications with Application-Level Checkpointing and Message Logging. **Under review in an international journal**, 2018
  - N. Losada, B. B. Fraguela, P. González, and M. J. Martín. A portable and adaptable fault tolerance solution for heterogeneous applications. Journal of Parallel and Distributed Computing, 104:146–158, 2017
  - N. Losada, M. J. Martín, and P. González. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications. The Journal of Supercomputing, 73(1):316–329, 2017
  - N. Losada, I. Cores, M. J. Martín, and P. González. Resilient MPI applications using an application-level checkpointing framework and ULFM. The Journal of Supercomputing, 73(1):100–113, 2017
  - N. Losada, M. J. Martín, G. Rodríguez, and P. González. Extending an Application-Level Checkpointing Tool to Provide Fault Tolerance Support to OpenMP Applications. Journal of Universal Computer Science, 20(9):1352–1372, 2014

- International Conferences (7)

  - P. González, N. Losada, and M. J. Martín. Insights into application-level solutions towards resilient MPI applications. In International Conference on High Performance Computing & Simulation (HPCS 2018), pages 610–613. IEEE, 2018

- N. Losada, L. Bautista-Gomez, K. Keller, and O. Unsal. Towards Ad Hoc Recovery For Soft Errors. **Under review in an international conference**, 2018
- N. Losada, G. Bosilca, A. Bouteiller, P. González, T. Hérault, and M. J. Martín. Local Rollback of MPI Applications by Combining Application-Level Checkpointing and Message Logging. In SIAM Conference on Parallel Processing for Scientific Computing (PP'18), 2018
- N. Losada, M. J. Martín, and P. González. Stop&Restart vs Resilient MPI applications. In International Conference on Computational and Mathematical Methods in Science and Engineering, pages 817–820, 2016
- N. Losada, M. J. Martín, G. Rodríguez, and P. González. Portable Application-level Checkpointing for Hybrid MPI-OpenMP Applications. Procedia Computer Science, 80:19–29, 2016
- N. Losada, I. Cores, M. J. Martín, and P. González. Towards resilience in MPI applications using an application-level checkpointing framework. In International Conference on Computational and Mathematical Methods in Science and Engineering, pages 717–728, 2015
- N. Losada, M. J. Martín, G. Rodríguez, and P. González. I/O optimization in the checkpointing of OpenMP parallel applications. In International Conference on Parallel, Distributed and Network-Based Processing, pages 222–229. IEEE, 2015

Appendix A

Extended summary in Spanish

Este apéndice contiene un resumen extendido de la tesis. Tras una introducción que describe brevemente el contexto y la motivación de esta tesis, se presenta un resumen de cada uno de los capítulos que la componen. Finalmente, se enumeran las principales conclusiones de esta investigación así como las principales líneas de trabajo futuro.

A.1. Introducción

La computación de altas prestaciones (High Performance Computing, HPC) y el uso de supercomputadores se han convertido en factores clave para el avance de muchas ramas de la ciencia. La gran capacidad de cálculo de estas máquinas, hoy en día del orden de 10^{15} operaciones por segundo en punto flotante, permiten la resolución de problemas científicos, de ingeniería o analíticos. Sin embargo, la demanda computacional de la ciencia continúa creciendo, principalmente debido a dos motivos: la aparición nuevos problemas en los que el tiempo de resolución es crítico (e.g. el diseño de fármacos personalizados, donde los pacientes no pueden esperar años por la molécula específica que necesitan), y el crecimiento exponencial del volumen de datos que deben ser procesados (e.g. aquellos originados por grandes telescopios, aceleradores de partículas, redes sociales, redes de sensores en *smart cities*, etc.). Para satisfacer esta creciente demanda cada vez se construyen

clústers de supercomputación más grandes, en los que una red de comunicaciones interconecta un gran número de nodos, cada uno de ellos con uno o varios procesadores multinúcleo de propósito general y que, en muchos casos, incluyen también procesadores especializados o aceleradores tales como GPUs o Xeon Phis. En los próximos años está previsto que se construya el primer supercomputador *exascale*, formado por millones de núcleos de procesamiento y capaz de realizar 10^{18} cálculos por segundo.

Las máquinas exascale suponen una gran oportunidad para las aplicaciones HPC, sin embargo, también representan una amenaza para la correcta finalización de la ejecución de estos programas. Estudios recientes muestran como, a medida que los sistemas HPC aumentan su tamaño e incluyen recursos hardware de diferentes tipos, el tiempo medio entre fallos de una aplicación disminuye. Esto se traduce en un considerable aumento de la tasa de fallos global del sistema. Incluso en el caso de que un nodo de cómputo presentara un fallo cada siglo, una máquina formada por 100 000 de esos nodos se verá afectada por un fallo, de media, cada 9 horas [42]. Es más, una máquina construida con 1 000 000 de esos nodos, presentará un fallo, de media, cada 53 minutos. La tolerancia a fallos se centra en estudiar técnicas que permitan que las applicaciones alcancen una solución correcta, en un tiempo finito y de una forma eficiente a pesar de los fallos subyacentes del sistema.

Las aplicaciones HPC científicas presentan tiempos de ejecución largos, generalmente del orden de horas o días. En esta situación, resulta imprescindible el uso de técnicas de tolerancia a fallos para que la ejecución de las aplicaciones se complete de forma satisfactoria, y para evitar además que se dispare el consumo de energía (ya que, de no utilizar ningún mecanismo de tolerancia a fallos, la ejecución tendría que volver a comenzar desde el principio). A pesar de esto, los modelos de programación paralela más populares que las aplicaciones HPC utilizan para explotar el poder de cómputo proporcionado por supercomputadores carecen de soporte de tolerancia a fallos. Por ello, en las últimas décadas muchos trabajos se han centrado en el estudio de técnicas y herramientas que permitan dotar de tolerancia a fallos a estas aplicaciones. Una de las técnicas más populares es el checkpoint/restart [42, 46]. Esta técnica consiste en salvar periódicamente el estado de la aplicación a almacenamiento estable en ficheros de checkpoint, que permiten reiniciar la ejecución desde estados intermedios en caso de fallo. Debido a la gran popularidad de los sistemas

de memoria distribuida (típicamente clústers de nodos, cada uno con su memoria privada, e interconectados por una red de comunicaciones), la mayor parte de la investigación en este campo se ha centrado en aplicaciones de memoria distribuida. En concreto, la mayoría de las propuestas se centran en aplicaciones paralelas de pase de mensajes, ya que es el modelo de programación más usado en sistemas de memoria distribuida, siendo MPI (Message Passing Interface, MPI) [132] el estándar de facto para este modelo de programación. Además, la inmensa mayoría de estas técnicas se corresponden con estrategias *stop-and-restart*. Es decir, tras un fallo que aborta la ejecución de la aplicación, esta se recupera a un estado salvado previamente a partir del cual puede continúar la computación. Sin embargo, con la popularización de otras arquitecturas hardware, otros paradigmas de programación paralela han cobrado relevancia en los últimos años. Tal es el caso de los modelos de programación híbridos, que combinan pase de mensajes con modelos de programación de memoria compartida, además de modelos de programación heterogéneos, en los que se hace uso de procesadores especializados (e.g. GPUs o Xeon Phis) para cómputo de propósito general. Además, cuando se produce un fallo, frecuentemente está limitado a un subconjunto de los nodos en los que la aplicación está ejecutándose. En este contexto abortar la aplicación y volver a enviarla a ejecución introduce sobrecargas innecesarias, por lo que se necesitan explorar soluciones más eficientes que permitan la implementación de aplicaciones resilientes.

Esta tesis propone técnicas checkpoint/restart a nivel de aplicación para los modelos de programación paralela más populares en supercomputación. La investigación llevada a cabo se centra en fallos-parada derivados de errores hardware y software en aplicaciones MPI, híbridas y heterogéneas; además de propuestas para la gestión de corrupciones transitorias de bits en memorias DRAM (*Dynamic Random Access Memory*) y SRAM (*Static Random Access Memory*), evitando que deriven en fallos-parada. La mayor parte de los desarrollos de esta tesis se han implementado en la herramienta de checkpointing CPPC [112] (*ComPiler for Portable Checkpointing*). Esta herramienta está formada por un compilador y una librería. CPPC opera a nivel de aplicación y se basa en la instrumentación automática del código por el compilador, que incluye llamadas a la librería para el registro de las variables relevantes en la aplicación (para que sean salvadas/recuperadas en/de los ficheros de checkpoint) y llamadas a funciones de checkpoint, que generarán los ficheros de checkpoint con la frecuencia que el usuario especifique.

A.2. Checkpointing en Aplicaciones Híbridas MPI-OpenMP

Los sistemas HPC actuales son clústeres de nodos multinúcleo, con múltiples núcleos de cómputo por nodo, que se pueden beneficiar del uso de un paradigma de programación híbrido, que emplea paradigmas de pase de mensajes, como MPI, para la comunicación inter-nodo; mientras que un modelo de programación de memoria compartida, tal como OpenMP, se usa intra-nodo [62, 127]. De esta forma, se permite el uso de N hilos de ejecución que se comunican empleando la memoria compartida dentro de un mismo nodo, mientras que la comunicación entre hilos de distintos nodos se realiza utilizando pase de mensajes. A pesar de que el uso de un modelo híbrido MPI-OpenMP requiere cierto esfuerzo por parte de los desarrolladores software, este modelo genera importantes ventajas tales como reducir las necesidades de comunicación y el consumo de memoria, a la vez que mejora el balanceo de carga y la convergencia numérica [62, 106].

Esta tesis propone un protocolo de checkpointing para aplicaciones híbridas MPI-OpenMP, implementado en la herramienta CPPC. Para poder operar en aplicaciones híbridas, en primer lugar se ha extendido CPPC para su uso en aplicaciones de memoria compartida OpenMP. El estado de estas aplicaciones está formado por variables privadas (para las cuales cada hilo de ejecución presenta su propia copia) y variables compartidas (para las que existe una única copia accesible por todos los hilos de ejecución del nodo). Por tanto, se han propuesto mecanismos que permitan incluir en los ficheros de checkpoint ambos tipos de variables, así como gestionar las funcionalidades proporcionadas por OpenMP tales como los bucles paralelizados o las operaciones de reducción [83, 84]. Se ha implementado un protocolo de checkpointing en CPPC que garantiza la consistencia de los ficheros generados basándose en la coordinación intra-nodo, mientras que se evita el uso de sincronizaciones entre distintos nodos [85]. Además, la portabilidad y adaptabilidad de esta propuesta en combinación con el paralelismo dinámico que proporciona OpenMP, permiten el reinicio no solo en máquinas con distintas arquitecturas o sistemas operativos, sino también en aquellas máquinas con un número distinto de núcleos de cómputo, permitiendo adaptar el número de hilos de ejecución para aprovechar de la forma más óptima los recursos disponibles.

La evaluación experimental de esta propuesta se ha llevado a cabo utilizando aplicaciones de las suites ASC Sequoia Benchmarks [5] y NERSC-8/Trinity Benchmarks [94]. Se ha estudiado el rendimiento y escalabilidad de la propuesta empleando hasta 6144 núcleos de cómputo. Los resultados obtenidos muestran una sobrecarga despreciable debido a la instrumentación del código original (menos de un 0.7 %) y una sobrecarga de checkpointing muy baja (menor a un 1.1 % cuando se almacenan los ficheros en discos remotos y menor a un 0.8 % cuando se utilizan discos locales). Los experimentos de reinicio en máquinas con distinta arquitectura y número de núcleos de cómputo confirman la portabilidad y adaptabilidad de esta propuesta, cualidades que permitirán mejorar el uso de los recursos disponibles en supercomputadores formados por nodos con arquitecturas diferentes.

A.3. Checkpointing en Aplicaciones Heterogéneas

El uso de procesadores especializados o aceleradores tales como GPUs o Xeon Phis se ha popularizado en los últimos años debido a su gran rendimiento y bajo consumo de energía. Las listas del TOP500 [44] han recopilado la información de los 500 supercomputadores más potentes del mundo durante los últimos 25 años. Atendiendo a los datos disponibles, hace una década menos del 1 % de las máquinas presentaban aceleradores, mientras que hoy en día más del 20 % de los sistemas cuentan con ellos. Debido a su incipiente presencia en los supercomputadores actuales, resulta de especial interés dotar de tolerancia a fallos a las aplicaciones que los utilizan.

Las aplicaciones heterogéneas son aquellas capaces de explotar más de un tipo de sistema de cómputo, obteniendo rendimiento no solo de los núcleos de cómputo de la CPU, sino también de recursos especilizados tales como aceleradores (GPUs, Xeon Phi, etc.). En general, estas aplicaciones ejecutan las partes secuenciales del programa principal en la CPU o host, mientras que las tareas paralelas se ejecutan en los acelaradores a medida que el programa principal lo solicita. La comunicación de datos entre host y aceleradores se hace a través de una memoria especial en la que ambos pueden escribir y leer, mientras que el resto de la memoria únicamente puede ser accedida por su propietario (o bien el host, o bien el acelerador). Existe un gran número de *frameworks* para el desarrollo de aplicaciones heterogéneas, entre los

cuales OpenCL es el más ampliamente soportado, y, por tanto, el que proporciona la mayor portabilidad entre dispositivos de diferentes arquitecturas y fabricantes. Por ello, el protocolo de checkpointing que esta tesis propone para aplicaciones heterogéneas se ha centrado en aplicaciones basadas en OpenCL. En particular, nos centramos en aplicaciones implementadas en HPL (Heterogeneous Programming Library, HPL) [129], una librería que facilita el desarrollo de aplicaciones basadas en OpenCL. Esta propuesta de tolerancia a fallos implementa un checkpointing *host-side* (ubicando los checkpoints en el código del *host* entre invocaciones a kernels) a nivel de aplicación utilizando la herramienta de checkpointing CPPC. El enfoque *host-side* evita incluir el estado privado de los aceleradores en los ficheros de checkpoint, mientras que la estrategia a nivel de aplicación salva únicamente los datos relevantes del host. Ambas estrategias contribuyen a minimizar el tamaño de los ficheros de checkpoint, minimizando así también la sobrecarga sin perjudicar la frecuencia de checkpointing. Se ha implementado un protocolo de consistencia, basado en sincronizaciones y transferencias de datos entre host y aceleradores, que garantiza la consistencia de los datos salvados. Además, HPL aplica una política de copia vaga (*lazy*) que minimiza la sobrecarga introducida por las transferencias de datos.

La evaluación experimental de esta propuesta se ha llevado a cabo utilizando aplicaciones heterogéneas de las suites SHOC Benchmarks [37] y SNU NPB [115], además de una simulación real de la evolución de un contaminante en la Ría de Arousa [131]. Las pruebas han demostrado la baja sobrecarga que introduce esta propuesta, que está principalmente determinada por el estado que debe ser salvado en los ficheros de checkpoint. Por otra parte, la elección de HPL y un host-side checkpointing maximiza la portabilidad y adaptabilidad de esta propuesta, permitiendo que las ejecuciones fallidas continúen en máquinas con diferente arquitectura y/o número de aceleradores. Los experimentos reiniciando las aplicaciones en máquinas con hosts de diferentes arquitecturas y/o sistemas operativos, y utilizando además distinto número de aceleradores con arquitecturas diferentes, demuestran la portabilidad y adaptabilidad de la propuesta, lo que constituye una funcionalidad muy interesante en supercomputadores heterogéneos.

A.4. Aplicaciones MPI Resilientes

Debido a que los supercomputadores actuales son clústeres de nodos interconectados con una red de comunicaciones, MPI continúa siendo el modelo programación predominante para estos sistemas de memoria distribuida. A pesar de ello, MPI carece de soporte de tolerancia a fallos. Un fallo en uno de los procesos que ejecutan la aplicación causa la terminación de ese proceso y provoca un estado indefinido en MPI, en el que no existe ninguna garantía de que el programa pueda continuar con éxito la ejecución. Por tanto, el comportamiento por defecto es abortar la ejecución de todos y cada uno de los procesos que ejecutan la aplicación, motivo por el que tradicionalmente se han empleado soluciones de checkpointing stop-and-restart. A gran escala, esto puede significar abortar la ejecución de cientos de miles de procesos cuando únicamente uno de ellos falla, lo cual resulta sumamente ineficiente. En este contexto, la interfaz ULFM (User Level Failure Mitigation) [14] es el último esfuerzo para incluir características de resiliencia en el estándar MPI. Las nuevas extensiones que propone permiten detectar y reaccionar a fallos sin interrumpir la ejecución de toda la aplicación. Se incluyen semánticas para detectar fallos, revocar y reconfigurar comunicadores, pero no se incluye ningún mecanismo para recuperar el estado de los procesos fallidos. Por tanto, deja flexibilidad a los desarrolladores de la applicación para implementar la metodología de checkpoint más óptima en cada caso.

Esta tesis aprovecha estas nuevas funcionalidades para implementar una solución de resiliencia que puede ser aplicada de forma genérica y transparente a aplicaciones SPMD (Single Program, Multiple Data) dentro de la herramienta de checkpointing CPPC. Para ello, se utilizan nuevos bloques de instrumentación, que permiten a las aplicaciones resilientes detectar y reaccionar a fallos de forma autónoma y sin abortar la ejecución cuando uno o varios de los procesos que la ejecutan se ven afectados por fallos-parada. En lugar de abortar la ejecución, se vuelven a lanzar los procesos fallidos y se recupera el estado de la aplicación con una vuelta atrás global, es decir, el estado de todos los procesos de la aplicación se recupera utilizando el último checkpoint generado. Además, se ha implementado un checkpointing multi-nivel y multi-hilo, que guarda una copia de los ficheros de checkpoint en distintos niveles de almacenamiento, minimizando la cantidad de datos que se deben mover a través del sistema ante un fallo, reduciendo por tanto la sobrecarga de la recuperación sin

aumentar la sobrecarga del checkpoining.

La evaluación experimental, utilizando aplicaciones con diferentes tamaños de checkpoint y patrones de comunicaciones, pone de manifiesto la baja sobrecarga introducida por esta propuesta, analizando para ello el comportamiento cuando uno o todos los procesos de un nodo fallan, considerando además dos escenarios para la recuperación: (1) utilizar un nuevo nodo libre para recuperar los procesos fallidos, o (2) sobrecargar uno de los nodos ya en uso con los procesos fallidos. Además, se han llevado a cabo experimentos utilizando hasta 3072 núcleos de cómputo para evaluar el rendimiento y la escalabilidad de la solución, comparando esta propuesta para aplicaciones resilientes MPI con una propuesta equivalente que realice un stop-and-restart tradicional. La propuesta de resiliencia claramente supera a la solución stop-and-restart, reduciendo el tiempo consumido por el proceso de recuperación entre 1.6x y 4x; evitando además el reenvío del trabajo al gestor de colas, que introduciría a mayores en la solución stop-and-restart una sobrecarga dependiente de la carga del sistema.

A.5. Vuelta Atrás Local en Aplicaciones MPI Resilientes

La propuesta para obtener aplicaciones MPI resilientes descrita anteriormente mejora notablemente el rendimiento frente a una solución stop-and-restart. Sin embargo, en muchos casos, un fallo tiene un efecto localizado y su impacto está restringido a un subconjunto de los recursos en uso. Una vuelta atrás global implica, por tanto, un gasto de tiempo y energía innecesarios, ya que todos los procesos de la aplicación (incluyendo aquellos no afectados por el fallo) descartan su estado y vuelven atrás recuperando su estado del último checkpoint para repetir un cómputo que ya han realizado anteriormente.

Para mejorar el rendimiento de la recuperación ante fallos-parada en aplicaciones MPI resilientes, esta tesis propone un protocolo de vuelta atrás local en el que sean únicamente los procesos afectados por un fallo los que vuelvan a un estado anterior y repitan cómputo. Este protocolo de recuperación local combina ULFM, la herramienta de checkpointing CPPC, y registro de mensajes (*message-logging*). Las

funcionalidades proporcionadas por ULFM se utilizan para detectar fallos en uno o múltiples procesos, mantener las capacidades de comunicación entre aquellos procesos supervivientes, levantar procesos que reemplacen a los fallidos, y reconstruir los comunicadores de la aplicación. Los procesos fallidos se recuperan del último checkpoint, mientras que la consistencia global y el progreso del cómputo se consiguen gracias al *message-logging*. Se ha implementado un protocolo de *message-logging* en dos niveles que permite mantener un registro de las comunicaciones punto a punto dentro de la librería Open MPI (utilizando el componente Vprotocol), mientras que el registro de las comunicaciones colectivas se mantiene a nivel de aplicación en CPPC, lo cual reduce el tamaño del registro y permite el uso de colectivas conscientes de la arquitectura (*architecture-aware collectives*) incluso después de producirse fallos. Además, se lleva a cabo un minucioso seguimiento de los números de secuencia y de los mensajes completados y no completados tras un fallo para permitir el despliegue del protocolo de *message-logging* sobre ULFM y el correcto reenvío de las comunicaciones necesarias para la recuperación. El protocolo resultante evita la sobrecarga introducida por la re-ejecución y el consumo de energía innecesarios introducidos por una vuelta atrás global. Además, el protocolo de checkpointing de CPPC permite reducir el impacto del protocolo de almacenaje de mensajes, ya que las operaciones de checkpoint se corresponden con puntos seguros para limpiar los registros de mensajes anteriores.

La evaluación experimental de este protocolo se ha llevado a cabo utilizando aplicaciones MPI reales con differentes tamaños de registros de mensajes y patrones de comunicación, y se ha comparado el rendimiento del protocolo de recuperación local con uno equivalente que realice una vuelta atrás global. A pesar de que en la operación sin fallos el protocolo de recuperación local implica operaciones adicionales para gestión del registro de mensajes, que hacen que aumente ligeramente la sobrecarga cuando se introducen fallos, los tiempos de recuperación de procesos fallidos y supervivientes mejoran notablemente. Las mejoras en los tiempos de recuperación de los procesos se traducen en mejoras importantes, tanto en el tiempo como en la energía consumidos a mayores en ejecuciones en las que se introducen fallos.

A.6. Recuperación Ad Hoc Ante Corrupciones de Memoria Transitorias

Las corrupciones en memoria son responsables de un alto porcentaje de los fallos en supercomputadores. Existen mecanismos hardware, tales como códigos de corrección de errores (*Error Correcting Codes*, ECCs), que permiten detectar y corregir errors en un bit, y que pueden además detectar algunos errores en múltiples bits. Sin embargo, los errores en múltiples bits detectados se traducen en la mayoría de los casos en fallos-parada. Gestionar este tipo de errores a nivel software, antes de que se traduzcan en fallos, permite el uso de técnicas de recuperación más eficientes.

Esta tesis propone un conjunto de extensiones a la librería de checkpointing a nivel de aplicación FTI (*Fault Tolerance Interface*) [12] para facilitar la implementación de estrategias de recuperación ad hoc para aplicaciones HPC. Estas estrategias proporcionan mecanismos de protección ante esas corrupciones de memoria que no pueden ser corregidas mediante mecanismos hardware. Las nuevas funcionalidades proporcionan a los programadores differentes mecanismos para salvar y acceder a los datos de un estado anterior del cómputo, sin necesidad de que se incluya en los ficheros de checkpoint, ni imponiendo restricciones en la frecuencia de salvado. La flexibilidad de estas extensiones permite la implementación de protocolos de recuperación más eficientes, en los que se aprovechan las características particulares de las aplicaciones en un amplio rango de escenarios. Además, el uso de estas nuevas extensiones es compatible con el uso de un checkpoint/restart tradicional.

Estas nuevas extensiones han sido evaluadas en tres aplicaciones HPC diferentes, que cubren situaciones muy distintas respecto a uso de memoria y datos salvados en los ficheros de checkpoint. En todas ellas, un porcentaje relevante de los datos de la aplicación se protege frente a corrupciones de memoria, teniendo en cuenta las particularidades de cada aplicación. La evaluación experimental demuestra la baja sobrecarga introducida por esta propuesta, a la vez que pone de manifiesto los importantes beneficios en el rendimiento que pueden obtenerse con la recuperación ante corrupciones de memoria, alcanzando hasta un 14 % de reducción en los tiempos de recuperación con respecto a una vuelta atrás global.

A.7. Conclusiones y Trabajo Futuro

A medida que los sistemas HPC aumentan su tamaño e incluyen más componentes hardware de diferentes tipos, el tiempo medio entre fallos para una aplicación concreta disminuye, lo cual resulta en una alta tasa de fallos. Las aplicaciones con tiempos de ejecución largos necesitan usar técnicas de tolerancia a fallos no solo para asegurar que completan su ejecución, sino también para ahorrar energía. Sin embargo, los modelos de programación paralela más populares carecen de soporte de tolerancia a fallos.

Checkpoint/restart es una de las técnicas de tolerancia a fallos más populares, sin embargo, la mayor parte de las investigaciones utilizando esta técnica se centran en estrategias *stop-and-restart* para aplicaciones de memoria distribuida ante fallos-parada. En este contexto, esta tesis hace las siguientes contribuciones:

- **Una solución de checkpointing a nivel de aplicación para hacer frente a fallos-parada en aplicaciones híbridas MPI-OpenMP.** Un nuevo protocolo de consistencia aplica coordinación dentro del nodo (dentro de cada equipo de hilos OpenMP) mientras que no utiliza coordinación entre distintos nodos (entre differentes procesos MPI). Esta propuesta reduce el uso de la red y recursos de almacenamiento para optimizar el coste de la entrada/-salida introducido por el uso de tolerancia a fallos, mientras que se minimiza la sobrecarga de checkpointing. Además, la portabilidad de la solución y el paralelismo dinámico de OpenMP permiten el reinicio de las aplicaciones en máquinas con diferentes arquitecturas, sistemas operativos y/o número de núcleos de cómputo, adaptando el número de hilos OpenMP para obtener un mejor aprovechamiento de los recursos disponibles.
- **Una solución de checkpointing a nivel de aplicación para hacer frente a fallos-parada en aplicaciones heterogéneas.** Esta propuesta está construida sobre HPL, una librería que facilita el desarrollo de aplicaciones basadas en OpenCL, por tanto, no está supeditada a ningún fabricante específico, sistema operativo, o plataforma hardware. La propuesta sigue una aproximación *host-side* a nivel de aplicación (ubicando los checkpoints en el código del *host* entre invocaciones a kernels). La estrategia *host-side* evita incluir en los ficheros de checkpoint el estado privado de los aceleradores. Además, la estrategia

a nivel de aplicación evita salvar los datos no relevantes del host. Ambos enfoques contribuyen a minimizar el tamaño de los ficheros de checkpoint, además de maximizar la portabilidad y adaptabilidad, permitiendo que las ejecuciones fallidas pueden continuar usando un número diferente de aceleradores y/o aceleradores con diferentes arquitecturas. La capacidad de las aplicaciones para adaptarse a los recursos disponibles será particularmente útil en clústeres heterogéneos

- **Una solución de resiliencia basada en checkpointing que puede ser aplicada de forma genérica a aplicaciones MPI SPMD.** Esta propuesta evita la recuperación tradicional *stop-and-restart* en aplicaciones MPI, evitando abortar la ejecución de todos los procesos de la aplicación ante un fallo. Para ello, se utilizan las nuevas funcionalidades proporcionadas por la interfaz ULFM, la cual propone características resiliencia para su inclusión en el estándar MPI. Las aplicaciones resilientes obtenidas son capaces de detectar y reaccionar a fallos sin detener su ejecución; en su lugar, se vuelven a lanzar los procesos fallidos y el estado de la aplicación se recupera con una vuelta atrás global.

- **Un protocolo de vuelta atrás local que se puede aplicar de forma genérica a aplicaciones MPI SPMD y que combina checkpointing a nivel de aplicación, ULFM, y *message-logging*.** Esta propuesta evita que los procesos no afectados por el fallo descarten su estado, vuelvan atrás y repitan cálculos que ya han hecho. En su lugar, la vuelta atrás está restringida a esos procesos que han fallado. Los procesos fallidos se recuperan utilizando el último checkpoint, mientras que la consistencia global y el progreso del cómputo se consiguen mediante un protocolo de *message-logging* en dos niveles. Las comunicaciones punto-a-punto son registradas por el componente VProtocol de Open MPI, mientras que las comunicaciones colectivas se registran a nivel de aplicación, reduciendo de esta forma el tamaño de los registros y permitiendo el uso de comunicaciones *architecture-aware*.

- **Un conjunto de extensiones en una librería de checkpointing a nivel de aplicación que facilita la implemetación de estrategias de recuperación personalizadas en aplicaciones MPI ante corrupciones de memoria.** Este tipo de errores se corresponde con corrupciones de memoria

en DRAM o SRAM que no se pueden corregir con mecanismos hardware y que, la mayoría de las veces, se traducen en errores fallo-parada. Gestionar estos errores a nivel software, antes de que se traduzcan en fallos, permite el uso de técnicas de recuperación más eficientes. Las nuevas funcionalidades proporcionan a los programadores diferentes mecanismos para salvar y acceder a datos correspondientes a estados anteriores de la computación, sin necesidad de que se incluyan en los ficheros de checkpoint, ni imponiendo restricciones en la frecuencia de salvado. La flexibilidad de estas extensiones permite aprovechar las caracterísiticas particulares de la aplicación en un amplio rango de escenarios, facilitando la implementación de técnicas ABFT (*Algorithm-Based Fault Tolerance*) y recuperaciones locales sin vuelta atrás.

La mayoría de estas propuestas han sido implementadas en la herramienta de checkpointing CPPC porque proporciona un checkpointing transparente, portable y a nivel de aplicación que está basado en la instrumentación automática del código. La única excepción son las extensiones para facilitar la implementación de recuperaciones personalizadas ante corrupciones de memoria. Estas extensiones requieren de la participación del programador, y han sido implementatadas en FTI, una librería de checkpointing cuya operación se basa en la instrumentación del código por parte del programador.

Para continuar reduciendo el coste de tolerar un fallo, el trabajo futuro incluye la optimización del protocolo de vuelta atrás local para realizar una re-ejecución de comunicaciones más eficiente. En la estrategia propuesta, todos los procesos que participan en la ejecución original de una operación colectiva también deben de participar en su re-ejecución durante la recuperación. Sin embargo, se puede optimizar esta re-ejecución de comunicaciones colectivas, por ejemplo, sustituyendo la operación colectiva por una o un conjunto de comunicaciones punto-a-punto. Este tipo de optimizaciones tienen el potencial de mejorar todavía más el rendimiento de la recuperación y de reducir el nivel de sincronización entre procesos durante esta operación. En el siguiente nivel, la re-ejecución de communicaciones también se puede beneficiar del uso de las operaciones de acceso a memoria remota proporcionadas por MPI para implementar una re-ejecución de comunicaciones dirigida por el receptor de los mensajes. Esta estrategia permitiría a los procesos fallidos obtener los mensajes del registro de mensajes de los procesos supervivientes sin su participación

activa durante el proceso de recuperación.

A mayores, el trabajo futuro incluye también el estudio de estrategias de recuperación ad hoc para gestionar corrupciones de memoria en aplicaciones Monte Carlo y algoritmos genéticos, en los que se pueden gestionar corrupciones de memoria en datos aleatorios sin apenas coste alguno.

Los resultados de estas investigaciones se han publicado en las siguientes revistas y congresos:

- International Journals (5)
  - N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín. Local Rollback for Resilient MPI Applications with Application-Level Checkpointing and Message Logging. **En revisión en una revista internacional**, 2018
  - N. Losada, B. B. Fraguela, P. González, and M. J. Martín. A portable and adaptable fault tolerance solution for heterogeneous applications. Journal of Parallel and Distributed Computing, 104:146–158, 2017
  - N. Losada, M. J. Martín, and P. González. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications. The Journal of Supercomputing, 73(1):316–329, 2017
  - N. Losada, I. Cores, M. J. Martín, and P. González. Resilient MPI applications using an application-level checkpointing framework and ULFM. The Journal of Supercomputing, 73(1):100–113, 2017
  - N. Losada, M. J. Martín, G. Rodríguez, and P. González. Extending an Application-Level Checkpointing Tool to Provide Fault Tolerance Support to OpenMP Applications. Journal of Universal Computer Science, 20(9):1352–1372, 2014
- International Conferences (7)
  - P. González, N. Losada, and M. J. Martín. Insights into application-level solutions towards resilient MPI applications. In International Conference on High Performance Computing & Simulation (HPCS 2018), pages 610–613. IEEE, 2018

- N. Losada, L. Bautista-Gomez, K. Keller, and O. Unsal. Towards Ad Hoc Recovery For Soft Errors. **En revisión en una conferencia internacional**, 2018
- N. Losada, G. Bosilca, A. Bouteiller, P. González, T. Hérault, and M. J. Martín. Local Rollback of MPI Applications by Combining Application-Level Checkpointing and Message Logging. In SIAM Conference on Parallel Processing for Scientific Computing (PP'18), 2018
- N. Losada, M. J. Martín, and P. González. Stop&Restart vs Resilient MPI applications. In International Conference on Computational and Mathematical Methods in Science and Engineering, pages 817–820, 2016
- N. Losada, M. J. Martín, G. Rodríguez, and P. González. Portable Application-level Checkpointing for Hybrid MPI-OpenMP Applications. Procedia Computer Science, 80:19–29, 2016
- N. Losada, I. Cores, M. J. Martín, and P. González. Towards resilience in MPI applications using an application-level checkpointing framework. In International Conference on Computational and Mathematical Methods in Science and Engineering, pages 717–728, 2015
- N. Losada, M. J. Martín, G. Rodríguez, and P. González. I/O optimization in the checkpointing of OpenMP parallel applications. In International Conference on Parallel, Distributed and Network-Based Processing, pages 222–229. IEEE, 2015

Bibliography

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *International Conference on Supercomputing (ICS)*, pages 277–286, 2004.

[2] M. M. Ali, P. E. Strazdins, B. Harding, and M. Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *The International Journal of High Performance Computing Applications*, 30(3):335–359, 2016.

[3] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *International Conference on Distributed Computing Systems*, pages 229–236. IEEE, 1995.

[4] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *International Parallel and Distributed Processing Symposium*, pages 1–12. IEEE, 2009.

[5] ASC Sequoia Benchmark Codes. `https://asc.llnl.gov/sequoia/benchmarks/`. Last accessed: July 2018.

[6] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *International Parallel and Distributed Processing Symposium*, page 15. IEEE, 2004.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[8] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *23rd European MPI Users' Group Meeting*, pages 15–22. ACM, 2016.

[9] R. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

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

[11] L. Bautista-Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Low-overhead diskless checkpoint for hybrid computing systems. In *International Conference on High Performance Computing*, pages 1–10. IEEE, 2010.

[12] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2011.

[13] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, 1994.

[14] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[15] W. Bland, K. Raffenetti, and P. Balaji. Simplifying the recovery model of user-level failure mitigation. In *Workshop on Exascale MPI at International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 20–25. IEEE, 2014.

[16] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, and J. J. Dongarra. Dodging the cost of unavoidable memory copies in message logging protocols. In *European MPI Users' Group Meeting*, pages 189–197. Springer, 2010.

[17] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[18] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[19] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: A Fault Tolerant MPI for Volatile Nodes Based on Pessimistic Sender Based Message Logging. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 25. ACM, 2003.

[20] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *European Conference on Parallel Processing*, pages 51–64. Springer, 2011.

[21] C. Braun, S. Halder, and H. J. Wunderlich. A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In *International Conference on Dependable Systems and Networks*, pages 443–454. IEEE, 2014.

[22] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C 3: A system for automating application-level checkpointing of MPI programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 357–373. Springer, 2003.

[23] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. *ACM SIGARCH Computer Architecture News*, 32(5):235–247, 2004.

[24] G. Bronevetsky, K. Pingali, and P. Stodghill. Experimental evaluation of application-level checkpointing for OpenMP programs. In *International Conference on Supercomputing (ICS)*, pages 2–13. ACM, 2006.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hard-

ware affinities in HPC applications. In *International Conference on Parallel, Distributed and Network-Based Processing*, pages 180–186. IEEE, 2010.

[26] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[27] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8, 2010.

[28] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, 1985.

[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization*, pages 44–54. IEEE, 2009.

[30] Y. Chen, K. Li, and J. Plank. CLIP: A Checkpointing Tool for Message Passing Parallel Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 33–33. IEEE Computer Society, 1997.

[31] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[32] CoMD website. `http://proxyapps.exascaleproject.org/apps/comd/`. Last accessed: July 2018.

[33] 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, 2012.

[34] 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.

[35] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 127. ACM, 2006.

[36] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[37] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[38] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *International conference on Supercomputing (ICS)*, pages 162–171. ACM, 2011.

[39] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division*, 11, 1997.

[40] C. Di Martino, Z. Kalbarczyk, and R. Iyer. Measuring the Resiliency of Extreme-Scale Computing Environments. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 609–655. Springer, 2016.

[41] W. R. Dieter and J. E. Lumpp. A user-level checkpointing library for POSIX threads programs. In *International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 224–227. IEEE, 1999.

[42] J. Dongarra, T. Herault, and Y. Robert. Fault tolerance techniques for high-performance computing. In *Fault-Tolerance Techniques for High-Performance Computing*, pages 3–85. Springer, 2015.

[43] J. J. Dongarra. *Performance of various computers using standard linear equations software*. University of Tennessee, Computer Science Department, 1993.

[44] J. J. Dongarra, H. W. Meuer, E. Strohmaier, et al. Top500 supercomputer sites, 1994.

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

[46] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *Computing Surveys*, 34(3):375–408, 2002.

[47] C. Engelmann, H. Ong, and S. L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *IASTED International Conference on Parallel and Distributed Computing and Networks*, volume 641, page 046, 2009.

[48] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent advances in parallel virtual machine and message passing interface*, pages 346–353, 2000.

[49] K. Ferreira, J. Stearley, J. H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2011.

[50] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar. Evaluating online global recovery with Fenix using application-aware in-memory checkpointing techniques. In *International Conference on Parallel Processing Workshops (ICPPW)*, pages 346–355. IEEE, 2016.

[51] M. Gamell, K. Teranishi, J. Mayo, H. Kolla, M. Heroux, J. Chen, and M. Parashar. Modeling and Simulating Multiple Failure Masking Enabled by Local Recovery for Stencil-based Applications at Extreme Scales. *Transactions on Parallel and Distributed Systems*, 28(10):2881–2895, 2017.

[52] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *International Conference for High Performance*

Computing, Networking, Storage and Analysis (SC), page 9. IEEE Computer Society, 2005.

[53] P. González, N. Losada, and M. J. Martín. Insights into application-level solutions towards resilient MPI applications. In *International Conference on High Performance Computing & Simulation (HPCS 2018)*, pages 610–613. IEEE, 2018.

[54] D. Hakkarinen and Z. Chen. Algorithmic Cholesky factorization fault recovery. In *International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2010.

[55] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, pages 494–499, 2006.

[56] Himeno Benchmark. `http://accc.riken.jp/en/supercom/himenobmt/`. Last accessed: July 2018.

[57] K.-H. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *Transactions on Computers*, 33(6):518–528, 1984.

[58] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. *Recent advances in the message passing interface*, pages 329–332, 2011.

[59] Intel MPI Benchmarks. `https://software.intel.com/en-us/imb-user-guide`. Last accessed: July 2018.

[60] D. Jacobsen, J. Thibault, and I. Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010.

[61] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, and X. Ma. Efficient intranode communication in GPU-accelerated systems. In *IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1838–1847. IEEE, 2012.

[62] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575, 2011.

[63] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan. HeteroCheckpoint: Efficient checkpointing for accelerator-based systems. In *International Conference on Dependable Systems and Networks*, pages 738–743. IEEE, 2014.

[64] Karl Rupp, Freelance Computational Scientist. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`. Last accessed: July 2018.

[65] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro*, 31(5):7–17, 2011.

[66] Khronos OpenCL Working Group. The OpenCL especification. Version 2.0, 2014.

[67] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20):7692–7714, 2010.

[68] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating user-level fault tolerance for MPI applications. In *European MPI Users' Group Meeting*, page 57. ACM, 2014.

[69] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 2016.

[70] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[71] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin. Lightweight checkpoint mechanism and modeling in GPGPU environment. *Computing (HPC Syst)*, 12:13–20, 2010.

[72] C.-C. Li and W. K. Fuchs. CATCH-compiler-assisted techniques for checkpointing. In *International Symposium on Fault-Tolerant Computing*, pages 74–81. IEEE, 1990.

[73] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, 1994.

[74] X. Liu, X. Xu, X. Ren, Y. Tang, and Z. Dai. A message logging protocol based on user level failure mitigation. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 312–323. Springer, 2013.

[75] N. Losada, L. Bautista-Gomez, K. Keller, and O. Unsal. Towards Ad Hoc Recovery For Soft Errors. In ***Under review in an international conference***, 2018.

[76] N. Losada, G. Bosilca, A. Bouteiller, P. González, T. Hérault, and M. J. Martín. Local Rollback of MPI Applications by Combining Application-Level Checkpointing and Message Logging. In *SIAM Conference on Parallel Processing for Scientific Computing (PP'18)*, 2018.

[77] N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín. Local Rollback for Resilient MPI Applications with Application-Level Checkpointing and Message Logging. ***Under review in an international journal***, 2018.

[78] N. Losada, I. Cores, M. J. Martín, and P. González. Towards resilience in MPI applications using an application-level checkpointing framework. In *International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 717–728, 2015.

[79] N. Losada, I. Cores, M. J. Martín, and P. González. Resilient MPI applications using an application-level checkpointing framework and ULFM. *The Journal of Supercomputing*, 73(1):100–113, 2017.

[80] N. Losada, B. B. Fraguela, P. González, and M. J. Martín. A portable and adaptable fault tolerance solution for heterogeneous applications. *Journal of Parallel and Distributed Computing*, 104:146–158, 2017.

[81] N. Losada, M. J. Martín, and P. González. Stop&Restart vs Resilient MPI applications. In *International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 817–820, 2016.

[82] N. Losada, M. J. Martín, and P. González. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications. *The Journal of Supercomputing*, 73(1):316–329, 2017.

[83] N. Losada, M. J. Martín, G. Rodríguez, and P. González. Extending an Application-Level Checkpointing Tool to Provide Fault Tolerance Support to OpenMP Applications. *Journal of Universal Computer Science*, 20(9):1352–1372, 2014.

[84] N. Losada, M. J. Martín, G. Rodríguez, and P. González. I/O optimization in the checkpointing of OpenMP parallel applications. In *International Conference on Parallel, Distributed and Network-Based Processing*, pages 222–229. IEEE, 2015.

[85] N. Losada, M. J. Martín, G. Rodríguez, and P. González. Portable Application-level Checkpointing for Hybrid MPI-OpenMP Applications. *Procedia Computer Science*, 80:19–29, 2016.

[86] T. Martsinkevich, T. Ropars, and F. Cappello. Addressing the Last Roadblock for Message Logging in HPC: Alleviating the Memory Requirement Using Dedicated Resources. In *European Conference on Parallel Processing*, pages 644–655. Springer, 2015.

[87] T. Martsinkevich, O. Subasi, O. Unsal, F. Cappello, and J. Labarta. Fault-tolerant protocol for hybrid task-parallel message-passing applications. In *International Conference on Cluster Computing*, pages 563–570. IEEE, 2015.

[88] E. Meneses and L. V. Kalé. CAMEL: collective-aware message logging. *The Journal of Supercomputing*, 71(7):2516–2538, 2015.

[89] E. Meneses, C. L. Mendes, and L. V. Kalé. Team-based message logging: Preliminary results. In *International Conference on Cluster, Cloud and Grid Computing*, pages 697–702. IEEE, 2010.

[90] H. Meyer, R. Muresano, M. Castro-León, D. Rexachs, and E. Luque. Hybrid Message Pessimistic Logging. Improving current pessimistic message logging protocols. *Journal of Parallel and Distributed Computing*, 104:206–222, 2017.

[91] H. Meyer, D. Rexachs, and E. Luque. RADIC: A FaultTolerant Middleware with Automatic Management of Spare Nodes. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[92] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2010.

[93] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[94] NERSC-8 / Trinity Benchmarks website: `https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/`. Last accessed: July 2018.

[95] A. Nukada, H. Takizawa, and S. Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 104–113. IEEE, 2011.

[96] H. Ong, N. Saragol, K. Chanchio, and C. Leangsuksun. VCCP: A transparent, coordinated checkpointing system for virtualization-based cluster computing. In *International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[97] OpenMP website: `http://openmp.org/`. Last accessed: July 2018.

[98] S. Pauli, M. Kohler, and P. Arbenz. A fault tolerant implementation of multi-level Monte Carlo methods. *Parallel Computing: Accelerating Computational Science and Engineering*, 25:471, 2014.

[99] G. Pawelczak, S. McIntosh-Smith, J. Price, and M. Martineau. Application-Based Fault Tolerance Techniques for Fully Protecting Sparse Matrix Solvers. In *IEEE International Conference on Cluster Computing*, pages 733–740. IEEE, 2017.

[100] A. J. Peña, W. Bland, and P. Balaji. VOCL-FT: introducing techniques for efficient soft error coprocessor recovery. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2015.

[101] J. S. 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.

[102] J. S. Plank and K. Li. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology: Systems & Applications*, 2(2):62–67, 1994.

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

[104] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 111–122. IEEE, 2002.

[105] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Message Passing Interface Developers and Users Conference (MPIDC99)*, pages 77–85, 1999.

[106] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436. IEEE, 2009.

[107] S. Rao, L. Alvisi, and H. M. Vin. The cost of recovery in message logging protocols. *Transactions on Knowledge and Data Engineering*, 12(2):160–173, 2000.

[108] A. Rezaei, G. Coviello, C.-H. Li, S. Chakradhar, and F. Mueller. Snapify: Capturing snapshots of offload applications on Xeon Phi manycore processors. In *International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–12. ACM, 2014.

[109] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B. Debusschere, O. LeMaitre, and O. Knio. ULFM-MPI implementation of a resilient task-based partial differential equations preconditioner. In *Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 19–26. ACM, 2016.

[110] G. Rodríguez, M. J. Martín, and P. González. Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 371–378. IEEE, 2012.

[111] G. Rodríguez, M. J. 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.

[112] 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.

[113] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello. SPBC: Leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 8. ACM, 2013.

[114] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka. FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *International Parallel and Distributed Processing Symposium*, pages 1225–1234. IEEE, 2014.

[115] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *International Symposium on Workload Characterization*, pages 137–148. IEEE, 2011.

[116] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance. *CoRR*, abs/1708.02030, 2017.

[117] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2013.

[118] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, et al. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[119] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, pages 123–134. IEEE, 2002.

[120] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *International Parallel Processing Symposium*, (IPPS'96), pages 526–531. IEEE Computer Society, 1996.

[121] A. Sunil, K. Jungwhan, and H. Sagyong. PC/MPI: Design and Implementation of a Portable MPI Checkpointer. In *European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 302–308. Springer Verlag, 2003.

[122] G. Suo, Y. Lu, X. Liao, M. Xie, and H. Cao. NR-MPI: a Non-stop and Fault Resilient MPI. In *International Conference on Parallel and Distributed Systems*, pages 190–199. IEEE, 2013.

[123] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent checkpointing and process migration of OpenCL applications. In *International Parallel & Distributed Processing Symposium*, pages 864–876. IEEE, 2011.

[124] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413. IEEE, 2009.

[125] TeaLeaf website. `https://github.com/UoB-HPC/TeaLeaf`. Last accessed: July 2018.

[126] K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using MPI-ULFM. In *European MPI Users' Group Meeting*, page 51. ACM, 2014.

[127] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff. MPI at Exascale. *Scientific Discovery through Advanced Computing (SciDAC)*, 2:14–35, 2010.

[128] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[129] M. Viñas, Z. Bozkus, and B. B. Fraguela. Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. *Journal of Parallel and Distributed Computing*, 73(12):1627–1638, 2013.

[130] M. Viñas, B. B. Fraguela, Z. Bozkus, and D. Andrade. Improving OpenCL programmability with the heterogeneous programming library. *Procedia Computer Science*, 51:110–119, 2015.

[131] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. A. García, M. J. Castro, and R. Doallo. A multi-GPU shallow-water simulation with transport of contaminants. *Concurrency and Computation: Practice and Experience*, 25(8):1153–1169, 2013.

[132] D. W. Walker and J. J. Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[133] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in HPC environments. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 43. IEEE Press, 2008.

[134] E. Wolters and M. Smith. MOCFE-Bone: the 3D MOC mini-application for exascale research. Technical Report ANL/NE-12/59, Argonne National Laboratory (ANL), Argonne, IL (United States), 2012.

[135] N. Woo, H. Jung, H. Yeom, T. Park, and H. Park. MPICH-GF: Transparent checkpointing and rollback-recovery for grid-enabled MPI processes. *IEICE Transactions on Information and Systems*, 87(7):1820–1828, 2004.

[136] W. Wu, G. Bosilca, R. Vandevaart, S. Jeaugey, and J. Dongarra. GPU-Aware Non-contiguous Data Movement In Open MPI. In *ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 231–242. ACM, 2016.

[137] J. Yeh, G. Pawelczak, J. Sewart, J. Price, A. A. Ibarra, S. McIntosh-Smith, F. Zyulkyarov, L. Bautista-Gomez, and O. Unsal. Software-level Fault Tolerant Framework for Task-based Applications. 2016.

[138] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.