High Performance Fault-Tolerant Solution of PDEs using the Sparse Grid Combination Technique

Md Mohsin Ali

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The Australian National University

October 2016

© Md Mohsin Ali 2016

Some of the work in this thesis has been accepted for publication, or published jointly with others, see for example [Strazdins et al., 2016a,b; Ali et al., 2016, 2015, 2014]. Except where otherwise indicated, this thesis is my own original work.

Md Mohsin Ali
12 October 2016

to my parents, mother Manik Zan Bibi and father Md Sefatullah Pk

Acknowledgments

This thesis, in fact, is a result of many people's contributions. I may not be able to mention all of them, but my heartfelt appreciation goes to them for their valuable support and inspiration towards the accomplishment of my degree.

First and foremost I offer my sincerest gratitude to my supervisor Peter Strazdins. This thesis would not have been possible without his immense support, knowledge, guidance and patience. Thanks Peter for helping me happily sail through this academic journey. I feel very lucky to have such an awesome supervisor like Peter.

I would also like to thank other members of my supervisory panel Alistair Rendell and Markus Hegland for helping me throughout my doctoral study.

I had the privilege of being able to contribute to a collaborative project (project LP110200410 under the Australian Research Council's Linkage Projects funding scheme) and would like to thank all members of the project for the many interesting discussions. Specifically thanks to Brendan Harding for helping me during all the way of my doctoral study. He made things understand very easily and his informative discussions were very helpful. I would also like to thank Jay Larson for valuable suggestion and discussion about the research, and the remaining members of the project.

Fujitsu Laboratories of Europe Ltd was the collaborative partner in this project and I would also like to thank James Southern, Nick Wilson, and Ross Nobes for the interactions I had with them throughout the project and for hosting me for 7 weeks from 8 July to 31 August in the Hayes office in 2013 as a research intern.

I would also like to thank Christoph Kowitz, Ethan Coon, George Bosilca, Aurélien Bouteiller, and Wesley Bland for valuable discussions.

All computations relating to this work were performed on Raijin, the peak system at the NCI Facility in Canberra, Australia, which is supported by the Australian Commonwealth Government. Thanks to the NCI Facility and its helpful staff.

I would also like to thank the Australian National University for helping me financially throughout the duration of my studies and stay in Australia.

I would also like to express my gratitude to all my fellow students for rendering their friendliness and support, which made my stay in Australia very pleasant. Thank you Brian Lee and Sara Salem Hamouda for the helpful discussions.

I would also like to thank my friends in Australia especially Rifat Shahriyar, Wayes Tushar, Fazlul Hasan Siddiqui, Nevis Wadia, Sakib Hasan Siddiqui, little Afeef Ayman, little Abeed Ayman, Tofazzal Hossain, Presila Israt, little Taseen, Masud Rahman, Falguni Biswas, Adnan Anwar, Shama Naz Islam, little Arisha, Mirza Adnan Hossain, Ashika Basher, Shampa Shahriyar, SM Abdullah, Abdullah Al Mamun, Samia Israt Ronee, little Aaeedah Samreen, Rakib Ahmed, Surovi Sultana, Saikat

Islam, Shaila Pervin, Md Zakir Hossain, Imun Ruby, Hossain Jalal Uddin, Sajeda Sultana, Md Masud Rana, Naruttam Kumar Roy, and the others. They care about me and are always there to provide assistance when I need it. My life in Australia is much more colorful with their company.

Last, but certainly not the least, I would like to express my deepest gratitude to my parents and the other family members for their unconditional love and support. My special thanks to my beloved wife Shakila Khan Rumi for her ceaseless sacrifice and mental support, and my adorable son little Aariz Ehan for always cheering me up with his sparkling smiles.

Abstract

The data volume of Partial Differential Equation (PDE) based ultra-large-scale scientific simulations is increasing at a higher rate than that of the system's processing power. To process the increased amount of simulation data within a reasonable amount of time, the evolution of computation is expected to reach the exascale level. One of several key challenges to overcome in these exascale systems is to handle the high rate of component failure arising due to having millions of cores working together with high power consumption and clock frequencies. Studies show that even the highly tuned widely used checkpointing technique is unable to handle the failures efficiently in exascale systems. The Sparse Grid Combination Technique (SGCT) is proved to be a cost-effective method for computing high-dimensional PDE based simulations with only small loss of accuracy, which can be easily modified to provide an Algorithm-Based Fault Tolerance (ABFT) for these applications. Additionally, the recently introduced User Level Failure Mitigation (ULFM) MPI library provides the ability to detect and identify application process failures, and reconstruct the failed processes. However, there is a gap of the research how these could be integrated together to develop fault-tolerant applications, and the range of issues that may arise in the process are yet to be revealed.

My thesis is that with suitable infrastructural support an integration of ULFM MPI and a modified form of the SGCT can be used to create high performance robust PDE based applications.

The key contributions of my thesis are: (1) An evaluation of the effectiveness of applying the modified version of the SGCT on three existing and complex applications (including a general advection solver) to make them highly fault-tolerant. (2) An evaluation of the capabilities of ULFM MPI to recover from a single or multiple real process/node failures for a range of complex applications computed with the modified form of the SGCT. (3) A detailed experimental evaluation of the fault-tolerant work including the time and space requirements, and parallelization on the non-SGCT dimensions. (4) An analysis of the result errors with respect to the number of failures. (5) An analysis of the ABFT and recovery overheads. (6) An in-depth comparison of the fault-tolerant SGCT based ABFT with traditional checkpointing on a non-fault-tolerant SGCT based application. (7) A detailed evaluation of the infrastructural support in terms of load balancing, pure- and hybrid-MPI, process layouts, processor affinity, and so on.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
- 1.1 Problem Statement 1
- 1.2 Scope and Contributions 3
- 1.3 Thesis Outline 4

2 Background and Related Work **5**
- 2.1 Overview of Fault Tolerance 5
- 2.2 Failure Recovery Techniques 8
- 2.3 MPI-Level Fault Tolerance 10
- 2.4 The Sparse Grid Combination Technique 13
  - 2.4.1 Classical Sparse Grid Combination Technique 14
  - 2.4.2 Fault-Tolerant Sparse Grid Combination Technique 16
- 2.5 Related Work 17
- 2.6 Summary 19

3 Implementation Overview and Experimental Platform **21**
- 3.1 Parallel SGCT Algorithm Implementation 21
- 3.2 Hardware and Software Platform 23
- 3.3 Fault Injection 23
- 3.4 Performance Measurement 24

4 Application Level Fault Recovery by ULFM MPI **27**
- 4.1 Introduction 28
- 4.2 Fault Detection and Identification 28
  - 4.2.1 Process Failure Detection and Identification 28
  - 4.2.2 Node Failure Detection and Identification 29
- 4.3 Fault Recovery 30
  - 4.3.1 Faulty Communicator Reconstruction 30
    - 4.3.1.1 Spawning Based Recovery 30
    - 4.3.1.2 Shrinking Based Recovery 32
  - 4.3.2 Lost Data Recovery 33
- 4.4 Experimental Results 35
  - 4.4.1 Experimental Setup 35

4.4.2 Failure Identification and Communicator Reconstruction Overheads . . . 36
4.4.3 Failed Grid Data Recovery Overheads . . . 37
4.4.4 Approximation Errors . . . 39
4.4.5 Scalability . . . 39
4.5 Summary . . . 40

5 Fault-Tolerant SGCT with Applications **43**
5.1 General Methodology for the SGCT Integration . . . 44
5.2 Fault-Tolerant SGCT with the Gyrokinetic Plasma Application . . . 45
5.2.1 Application Overview . . . 45
5.2.2 Implementation of the SGCT Algorithm for Higher-Dimensional Grids . . . 46
5.2.3 Modifications to GENE for the SGCT . . . 48
5.2.4 Experimental Results . . . 49
5.2.4.1 Experimental Setup . . . 50
5.2.4.2 Execution Time and Memory Usage . . . 50
5.2.4.3 Approximation Errors . . . 53
5.3 Fault-Tolerant SGCT with the Lattice Boltzmann Method Application . 53
5.3.1 Application Overview . . . 53
5.3.2 Modifications to Taxila LBM for the SGCT . . . 54
5.3.3 Experimental Results . . . 56
5.3.3.1 Experimental Setup . . . 56
5.3.3.2 Execution Time and Memory Usage . . . 56
5.3.3.3 Approximation Errors . . . 58
5.4 Fault-Tolerant SGCT with the Solid Fuel Ignition Application . . . 58
5.4.1 Application Overview . . . 58
5.4.2 Modifications to SFI for the SGCT . . . 59
5.4.3 Experimental Results . . . 60
5.4.3.1 Experimental Setup . . . 62
5.4.3.2 Execution Time and Memory Usage . . . 62
5.4.3.3 Approximation Errors . . . 62
5.5 Failure Recovery Overheads . . . 63
5.5.1 Recovery Overheads for Shorter Computations . . . 63
5.5.2 Recovery Time Analysis for Longer Computations . . . 65
5.5.3 Overhead due to Computing Extra Grid Points . . . 67
5.5.4 Repeated Failure Recovery Overheads . . . 69
5.6 Summary . . . 70

6 Evaluation of the SGCT and Applications **71**
6.1 Introduction . . . 71
6.2 SGCT Performance Analysis . . . 73
6.3 Load Balancing and Communication Profiles . . . 75
6.4 Pure-/Hybrid-MPI Performance Analysis . . . 78

6.5 Effect of Process Layouts on Performance 79
6.6 Effect of Processor Affinity on Performance 80
6.7 Summary 81

7 Conclusion **83**
7.1 Future Work 85

A Process Recovery by ULFM MPI **87**

B Application Input Parameters **91**

List of Figures

2.1 Changes required in exascale systems. 6
2.2 Variation of MTTI with the number of CPUs. 7
2.3 The Sparse Grid Combination Technique. 14
2.4 A depiction of the 2D SGCT. 16

3.1 Message paths for the gather stage for the 2D direct combination method (not truncated) on a level $l = 5$ combined grid. 22

4.1 Techniques of recovering failed processes. 31
4.2 Process grid configurations of different sub-grids of the 2D FT-SGCT based applications with level $l = 4$ when the faulty communicator is shrunk as a recovery action. 32
4.3 Grid arrangements of the 2D SGCT with level $l = 4$ solving the general advection problem to demonstrate different data recovery techniques. . 34
4.4 Times for generating the failure information and repairing the faulty communicator for the 2D SGCT solving the general advection problem. 36
4.5 Failed grid data recovery overheads of the 2D SGCT solving the general advection problem. 38
4.6 Approximation errors of the 2D FT-SGCT for the general advection solver. 39
4.7 Overall parallel performance of the 2D SGCT solving the general advection problem with a single combination. 40

5.1 A demonstration of parallelization $p = 2$ on the non-SGCT dimension N_z of GENE for the 2D SGCT. 47
5.2 Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the GENE application. 49
5.3 Approximation errors of the FT-SGCT based GENE application. 51
5.4 A comparison of the 2D full grid and level $l = 5$ combined grid solutions for the GENE application. 52
5.5 Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the Taxila LBM application. 55

5.6 A comparison of the 2D full grid and level $l = 5$ combined grid solutions for the Taxila LBM application. 57
5.7 Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the SFI application. 59
5.8 A comparison of the 2D full grid and level $l = 5$ combined grid solutions for the SFI application. 61
5.9 Recovery overhead of a single occurrence of failures for shorter computation for GENE. 63
5.10 Expected relative recovery overhead for longer computation for GENE. 66
5.11 Relative overhead required in the SGCT to achieve an ABFT. 68
5.12 Repeated ULFM MPI failure recovery overheads of the 2D FT-SGCT with level $l = 5$ applied to the GENE application over 64 cores. 69

6.1 Execution time of the average of ten combinations of the direct SGCT in isolation. 72
6.2 Overall execution time of the general advection solver with the direct SGCT running over 1024 time-steps (MPI warm-up time excluded). . . 73
6.3 An analysis of the TAU-generated load balancing of the 2D direct SGCT computing the GENE application with *2d_big_6* input and a single combination (MPI warm-up time included). 74
6.4 An analysis of the IPM generated load balancing of the 2D direct SGCT solving the general advection problem on level $l = 11$ with a single combination (MPI warm-up time excluded). 76
6.5 A comparison of the pure- and hybrid-MPI performance for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). 77
6.6 A comparison of the performance due to different process layouts for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). 78
6.7 2D linear and block mapping of 32x4 process grid onto the cores of Raijin nodes. 79
6.8 A comparison of the performance due to the linear and block mappings for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). 80

List of Tables

4.1 ULFM MPI performance to recover multiple process failures. 37

5.1 Execution time breakdown of the 2D FT-SGCT with parallelization $p = 1$ and level $l = 5$ for the GENE application. 50
5.2 Execution time breakdown of the 2D FT-SGCT with level $l = 5$ for the Taxila LBM application. 56
5.3 Execution time breakdown of the 2D FT-SGCT with level $l = 5$ for the SFI application. 60

6.1 Parameters used in advection, GENE, Taxila LBM, and SFI experiments. 77

B.1 Parameters in `testsuite/big/parameters_6` file used in GENE experiments. 91
B.2 Parameters of Taxila LBM experiments from `tests/bubble_2D/input_data` and `tests/bubble_3D/input_data` files. 92

Chapter 1

Introduction

This thesis addresses the challenges and opportunities of achieving high performance fault tolerance of applications running on the upcoming exascale systems.

1.1 Problem Statement

Numerical solution of *Partial Differential Equations* (PDEs) is an important problem in computational science as PDEs are the basis of simulating all physical theorems[1]. The challenges that are encountered in all scientific simulations are thus essentially the same as solving the PDEs.

Today's largest *High Performance Computing* (HPC) systems consist of thousands of nodes which are capable of concurrently executing up to millions of threads to simulate the PDE based complex scientific problems within a feasible amount of time. The nodes within these systems are connected with high-speed network infrastructures to minimize communication costs [Ajima et al., 2009]. Significant effort is required to exploit the full performance of these systems. Extracting this performance is essential in different research areas such as climate, the environment, physics and energy which all are characterized by the complex scientific models they utilize.

In the near future, besides exploiting the full performance of such large systems, dealing with component failures will become a critical issue. Since the failure rate of a system is roughly proportional to the number of nodes of the system [Schroeder and Gibson, 2006], current HPC systems consisting of thousands of nodes experience significant number of component failures. For instance, a 382-days study on the 557 Teraflops Blue Gene/P system with 163,840 computing cores at Argonne National Laboratory showed that it experienced a failure (hardware) every 7.5 days [Snir et al., 2014]. Since the typical size of HPC systems is becoming larger as we approach exascale computing, the rate at which they experience failures is also increasing [Gibson

[1] *"... partial differential equations are the basis of all physical theorems. In the theory of sound in gases, liquid and solids, in the investigations of elasticity, in optics, everywhere partial differential equations formulate basic laws of nature which can be checked against experiments."*
– Bernhard Riemann (1826-1866)

et al., 2007]. A study in [Snir et al., 2014] assumed the *Mean Time To Failure* (MTTF) of an exascale system as 30 minutes.

Besides exascale computing, fault tolerance is also important in other areas, such as cloud computing, and scenarios, such as low power or adverse operating condition of the system. (a) In the large-scale and complex dynamic environments of cloud computing, there are several reasons such as expansion and shrinkage of the system size, update and upgrade of the system, online repairs, intensive workload on servers, and so on, that can induce failures and faults. The shrinkage of system components is required to exclude the faulty or high costly components of the system; whereas the expansion of system components is required to balance the server loads of the system. (b) In order to reduce the overall cost, sometimes system components (i.e., processors) are designed as very cheap to operate with low power consumption, which causes failures even with the moderate number of system components. Moreover, adverse operating condition scenarios also cause failures. The common type of failures due to these scenarios is 'bit-flips' in memory or logic circuitry, which is termed as *soft* faults.

The most commonly used *Checkpoint/Restart* [Hursey et al., 2007] technique, which restarts the application from the recently checkpointed state in the event of failures, has several limitations to achieve fault tolerance in exascale systems. One of the key limitations is that a large amount of time required to write a checkpoint could be close to the MTTF. Although a parallelization of the checkpoint write and computation/communication reduces the overall time, the key limitation is still in effect, together with a large amount of time required to read the checkpoint at restarts. Furthermore, the data volume of the future ultra-large-scale scientific simulations is expected to be increased, which in turn will increase the checkpoint write and read times.

Thus, there is an urgent need to develop large-scale fault-tolerant applications using application/user level or other non-Checkpoint/Restart technique based resiliency. Traditionally, large-scale applications use the *Message Passing Interface* (MPI) [Message Passing Interface Forum, 1993], which is a widely used standard for parallel and distributed programming of HPC systems. However, the standard does not include methods to deal with one or more component failures at run-time. In order to address this problem FT-MPI [Fagg and Dongarra, 2000] was introduced to enable MPI based software to recover from process failure (see [Ali and Strazdins, 2013] for details). However, development of FT-MPI was discontinued due to the lack of standardization [Bland, 2013b]. Recently, the MPI Forum's Fault Tolerance Working Group began work on designing and implementing a standard for *User Level Failure Mitigation* (ULFM) [Bland, 2013a] which introduces a new set of tools to facilitate the creation of fault-tolerant applications and libraries. These tools provide MPI users the ability to detect, identify, and recover from process failures. It is a great opportunity for application developers to use these tools to make their applications fault-tolerant.

Currently, there is a lack of practical examples which demonstrate the range of issues encountered during the development of fault-tolerant applications. Moreover, the amount of literature detailing the implementation and performance of the pro-

posed standard is very limited. Some of the work that is available assumes a fail-stop process failure model, i.e., a failed process is permanently stopped without recovering and the application continues working with the remaining processes [Hursey and Graham, 2011]. However, continuation with only the alive processes is not sufficient for all applications. As for example, some applications do not tolerate a reduction of the MPI communicator size due to maintaining a strict load balancing and, thus, require a recovery of the failed processes in order to finish the remaining computation successfully with balanced loads. Even the applications which are careless about load balancing require a major re-factoring effort in implementation if the communicator size is changed.

There appears to be an even greater lack of research on how to make existing, complex and widely used parallel applications fault-tolerant. In this thesis, we demonstrate how a general advection solver, and three existing real-world applications (the GENE gyrokinetic plasma simulation, Taxila Lattice Boltzmann Method application, and Solid Fuel Ignition application codes) can be made fault-tolerant using ULFM MPI and a form of algorithm-based (application/user level) fault tolerance obtained via modification of the Sparse Grid Combination Technique (SGCT). Our implementation and analysis include the restoration of failed processes and MPI communicators on either the existing or new (spare) nodes.

1.2 Scope and Contributions

There are different kinds of faults that may occur in a system. Based on the symptoms and consequences of each category, different types of strategies to follow to handle them. The level of effort needed to identify and handle them may vary from one category to the other. Out of many categories, some common types of faults are *transient* (faults that occur once and then disappear), *intermittent* (faults that occur, then vanish again, then re-occur, then vanish), *permanent* (faults that continue to exist until the faulty component is repaired or replaced), *fail-stop* (faults that define a situation where the faulty component either produces no output or produces output that clearly indicates that the component has failed), and *Byzantine* (faults that define a situation where the faulty component continues to run but produces incorrect results). Although it is desirable that a fault-tolerant technique will be able to handle all types of faults, but in practice, it is too hard to design and implement such a technique.

In this thesis, we are not handling all the above mentioned types of faults. The scope is narrowed down to handle only the permanent or fail-stop type of faults. More specifically, we are looking at the problem of recovering from the application process failures, caused by the hardware or software faults, from within the application.

The aim of this thesis under the stated scope is to:

- Demonstrate how ULFM as an MPI standard may be used to create a fault-tolerant application, and evaluate its current effectiveness. Our approach fea-

tures the preservation of communicator size and rank distribution after faults, the preservation of load balance, and either an exact or approximate data recovery for the failed processes using the SGCT based general advection solver.

- Detail how a scalable SGCT algorithm can be integrated not only into a general advection solver, but also into three existing and complex applications to make them highly fault-tolerant, and evaluate their effectiveness.
- Evaluate the capabilities of ULFM MPI to recover from a single or multiple real process/node failures for a range of complex applications.
- Perform a detailed experimental evaluation of the integrated applications including time and memory requirements, and parallelization on the non-SGCT dimensions.
- Perform an analysis of the result errors with respect to the number of failures, overhead due to computing some extra unknowns to achieve the fault tolerance, and an analysis of the recovery overheads. The latter includes a comparison with traditional checkpointing on a non-fault-tolerant SGCT based application.
- Perform a detailed analysis of the SGCT algorithm and the applications in terms of load balancing, pure- and hybrid-MPI, process layouts, processor affinity, and so on.

1.3 Thesis Outline

The body of this thesis is structured around the key contributions outlined above. Chapter 2 provides an overview of fault tolerance and surveys relevant fault-tolerant literature. It provides more detailed background on previous fault tolerance techniques. Chapter 3 gives an overview of our implementation, and experimental platform.

Chapters 4, 5, and 6 comprise the main body of the thesis, covering the key contributions. Chapter 4 evaluates the effectiveness of ULFM MPI for the implementation of application level resiliency in the application. This includes a detailed implementation guidelines for the detection, identification, and recovery of process and node failures with the ULFM MPI semantics. Chapter 5 evaluates the effectiveness of applying the fault-tolerant SGCT on three different types of existing complex parallel applications. At the same time, this chapter also evaluates the application level recovery overheads implemented by ULFM MPI on these applications, and compares these with the built-in checkpointing technique. Chapter 6 provides a detailed analysis of the infrastructural support and the evaluation of applications on this infrastructure with respect to combination algorithm's scalability, load balancing, pure- and hybrid-MPI, process layouts, processor affinity, and so on.

Finally, Chapter 7 concludes the thesis, describing how the contributions have identified, quantified, and addressed the challenges of achieving high performance fault tolerance of varieties of PDE based complex applications on the targeted exascale systems. It further identifies the key future directions for research.

Chapter 2

Background and Related Work

This chapter provides background information on fault tolerance basics, failure recovery techniques, MPI-level fault tolerance, the classical and fault-tolerant versions of the SGCT (computational model for the high-dimensional data processing), and related work to place the research contributions in context.

This chapter starts with a brief introduction to the field of fault tolerance in Section 2.1. Section 2.2 describes failure recovery techniques and Section 2.3 describes some MPI library based fault tolerance. A computational model (the SGCT) used for the high-dimensional data processing and its robust version are described in Section 2.4. An overview of the SGCT algorithm is presented in Section 3.1. Section 2.5 describes the research closely related to this thesis.

Sections 2.4 and 2.5 of this chapter are from the work published jointly with others as a part of the paper titled "Complex Scientific Applications Made Fault-Tolerant with the Sparse Grid Combination Technique" [Ali et al., 2016]. Section 2.4, of this chapter is also published jointly with others as a part of the paper titled "A Fault-Tolerant Gyrokinetic Plasma Application using the Sparse Grid Combination Technique" [Ali et al., 2015].

2.1 Overview of Fault Tolerance

The *European Exascale Software Initiative* (EESI) began their journey in the middle of 2010 with the hope of creating a common platform to tackle the issues that may arise in today's and upcoming HPC systems. This initiative seems as a driving force of participating in a competition among different nations for building the next generation supercomputers. For instance, in June 2011, Japanese K computer achieved the number one placing on the TOP500[1] list of the world's fastest supercomputers, with a performance in excess of 10 petaflops (10^{16} floating point operations per second). But today, China's Tianhe-2 replacing that positing, with a sustained performance of 33.86 petaflops, which is more than three times as powerful than K. This rate of increase puts HPC well on track to reach the next major milestone - exascale computing (10^{18} flops) - by the end of the decade.

[1]http://www.top500.org/

Achieving this milestone will certainly require major changes to hardware and software technologies as shown in Figure 2.1. As it will be so hard to increase the clock frequency in the future, an exascale system is likely to have approximately one billion processing elements (cores) [Ashby et al., 2010]. Delivering these large number of elements will require more power. To keep it in an acceptable window, i.e., around 20 megawatts (MW) compared to 12.66 MW and 17.81 MW for K and Tianhe-2, respectively, it will require the development of novel architectures, most likely with increased heterogeneity. Similar to clock frequency, it is hard to increase the performance of *Input/Output* (I/O) and memory systems compared to that of the processing elements. This will cause data movements on and off chip to dominate other operations, and create I/O bottlenecks on disk operations. This will require some software side solutions for maximizing local chip workload and placing part of the file system on the heterogeneous node.

Since the rate of component failures of a system is roughly proportional to the number of cores of the system [Schroeder and Gibson, 2006], an exascale system consisting of 1,000 times more cores than either the K computer (705,024 cores) or Tianhe-2 (3,120,000) will certainly suffer more frequent component failures. The *Mean Time To Interrupt* (MTTI), which is generally measured in days for today's leading supercomputers, will fall within the range of an hour in the exascale system due to this higher failure rate. A failure event analysis research at Los Alamos National Laboratory (LANL) with 140,000 interrupt events on 21 platforms shows remarkably similar trends of decreasing MTTI with the increase of number of cores in the system.

[2]DOE Exascale Initiative Roadmap, Architecture and Technology Workshop, San Diego, December, 2009.

	2010	2018	Factor Change
System peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Gf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 cpus	1,000 cpus	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System Size (nodes)	20 K nodes	1 M nodes	50
Total Concurrency	225 K	1 B	4,444
Storage	15 PB	300 PB	20
Input/Output bandwidth	0.2 TB/s	20 TB/s	100

Figure 2.1: Changes required in exascale systems[2].

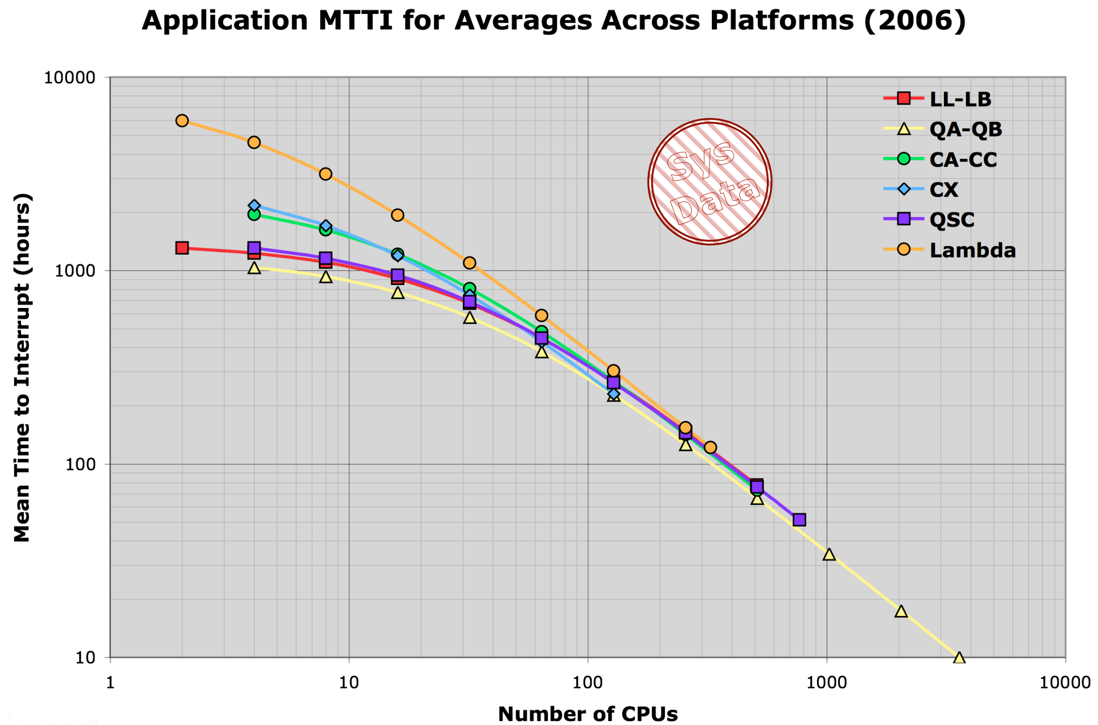


Figure 2.2: Variation of MTTI with the number of CPUs[3].

The results of this work presented in HPC-4 SDI/LCS seminar (October 10, 2007) are shown in Figure 2.2. Moreover, a study at the Oak Ridge National Laboratory showed that a 100,000-processor supercomputer with all its associated support systems could experience a failure every few minutes [Geist and Engelmann, 2002]. Current methods for dealing with failures – often just re-run the application as failures are very unlikely to occur in two successive runs – will be unable to cope with this increase in the frequency of failures. An appropriate and efficient approach capable of handling such frequent failures on the large systems will be needed to successfully run the software on these systems.

The sources of such frequent failures include memory soft and hard errors; disks, file system or I/O node errors (disk reconstruction time); network connection faults (fibers, connectors, laser, etc.); resource exhaustion (memory, disc quota, etc.); Operating System/run-time/library bugs; hardware errors (power supply, fans, water valves, water connectors, water pipes, etc.); operators, system administrators, user errors; inconsistent maintenance, i.e., libraries update errors; and so on. An experiment was carried out in LANL HPC systems over a period of a few years in [Schroeder and Gibson, 2010] to find out the root causes for system failures (both soft and hard). It is observed for these systems that hardware is the single largest source of faults, with 64% of all failures assigned to this category. Software is the second largest contributor, with 18% of all failures. It is also important to consider that the number of

[3]HPC-4 SDI/LCS seminar, October 10, 2007.

failures obtained from the undetermined cause is significant. Total failures assigned to this category is 14%. A detailed root cause information reveals that CPU (42.8%) and memory DIMMS (21.4%) are the largest fraction of all hardware-related failures. For the software category, they are "other software" (30.0%), OS (26.0%), and parallel file system (11.8%). The largest fraction of all environment-related failures are power outage (48.4%), UPS (21.2%), power spike (15.1%), and chillers (9.8%).

The set of possible solutions to deal with these failures, as reported in [Snir et al., 2014], is divided into three categories: the *hardware* approach, the *system* approach, and the *application* approach.

The hardware approach will add additional hardware in an exascale system to deal with failures on the hardware level. Although this will require the least effort in porting current applications, it will incur additional power consumption in the system. Moreover, as the hardware in exascale system becomes more complex, the software will become more complex and hence error-prone. In this scenario, new complexities arise in the system due to the introduction of additional hardware.

In the system approach, fault tolerance is achieved by applying both the hardware and system software in such a way that the application code remain unchanged. Although it may be convincing that changing the system software is less costly than that of the hardware, this approach may add additional complexities in the system and, hence, may increase its energy consumption.

In the application approach, application code is extended to handle resiliency. No changes in hardware and system software are required. Since there are no additional costs and complexities due to extra hardware and system software, this approach may be suitable for exascale platforms. Moreover, application developers have more options to select the most appropriate resiliency strategy for their applications.

2.2 Failure Recovery Techniques

Some failure recovery techniques are as follows.

Checkpoint/Restart

The classic system-level/automatic fault tolerance technique is the *Checkpoint/Restart* [Hursey et al., 2007]. It generally means the process of periodically storing the whole state of a computation in disk space such that its execution could be restarted from its recent saved state in the event of a failure. This storing is done in local disk storage or in remote disk storage or in both. In case of storing the state of large scale application, each of the tens of thousands of processes writes several gigabytes of data. This increases the overall checkpoint volume in the order of several tens of terabytes. This type of checkpointing causes an I/O bottleneck as the I/O bandwidth could not win the race of speed increment against the computational capabilities. This behaviour causes up to 25% of overhead in current petascale systems for a particular case as shown in [Schroeder and Gibson, 2007]. Furthermore, an application will no longer progressing if the MTTI is shorter than or equal to the application restart time. In such a scenario, without performing any actual computation, application starts next

checkpointing just after restarting from the recent checkpoint.

Diskless checkpointing
The *diskless checkpointing* approach [Plank et al., 1998] is proposed to reduce the overhead of the Checkpoint/Restart approach. It stores a reduced volume of checkpoint state data onto compute nodes' own memory without going to disk. It also needs some extra nodes to save a checksum of the memory checkpoint states so that it could be used to recover the memory checkpoints of the failed nodes. Although the performance of this technique is better than that of the disk-based Checkpoint/Restart method, there is a potentially significant memory I/O overhead to this method, especially in memory intensive applications. Moreover, the number of additional nodes to store the checksum will grow in proportion to the number of nodes running the application.

Replication
The *replication* technique [Ferreira et al., 2011] is proposed to solve the problem of large overheads of the Checkpoint/Restart technique and to exempt the requirements of storing checkpoints on memory of the diskless checkpointing approach. The idea of replication is that most applications leave some "wasted" spaces on the cluster machines on which they are executed. In order to efficiently use those spaces, multiple copies of the application are executed simultaneously. If there is any failure occurs, one of the replicated processes taking the charge of the original version of the application and the computation can continue onwards. This technique is applicable for some types of machines, especially those where the system utilization is not greater than 50%.

Message logging
The *message logging* technique [Bouteiller et al., 2003] is proposed to reduce the rollback overhead of the Checkpoint/Restart technique. It involves all processes to checkpoint their states without coordination and logging all communication operations in a stable media. Thus, in case of any failures, this log is analyzed to restart the execution of only the crashed processes, rather than every processes, from the recent local snapshot, and establishing the same communication with the help of the saved communication log. However, the overhead of this technique is proportional to the communication volume of the application. A significant amount of penalty is added by this technique for all messages transferred even if there is no fault throughout the whole computation.

Task pools with reassignment
The drawbacks of the replication technique can be solved by the *task pools with reassignment* technique. Instead of running multiple copies of the application simultaneously, it splits the work into some discrete tasks by a manager at run-time. These tasks then can be executed by any worker node. In the event of a failure, there is a provision of reassigning the affected tasks to other nodes. It represents a very effective method for implementing fault tolerance and is already used in, e.g., standard

MapReduce algorithms [Dean and Ghemawat, 2008]. However, this method can be vulnerable to the failure of the manager node responsible for scheduling.

Proactive migration
The *proactive migration* technique [Chakravorty et al., 2006] is proposed to solve the problem of recomputing the affected tasks from the beginning of the task pools with reassignment technique. In order to avoid the recomputation, it predicts the failures in nodes before they really happen and moves the running applications away from them before the fault occurs. Theoretically, this would allow applications to run on fault-prone systems without any modifications. But practically, the performance monitoring of the nodes must occur sufficiently quickly that the application can be migrated before the failure does occur. Otherwise, it will not be applicable.

The success of the proactive fault tolerance solutions depends solely on the accurate prediction of the failures and the ranges of failures that it could cover. Prediction techniques used to achieve fault tolerance should incur less overhead, as well as, the work lost due to wrong prediction should be small. The state-of-the-art researches of this category is based on the data mining approaches [Gainaru et al., 2013].

Algorithm-Based or User Level Fault Tolerance
In the *Algorithm-Based Fault Tolerance* (ABFT) technique [Huang and Abraham, 1984], numerical algorithms are modified to include methods for detecting and correcting errors. An extension of this is to develop new algorithms that are naturally resilient to faults. The major advantage of dealing with faults at the algorithm level is that the time-to-solution is roughly unchanged in the presence of faults. There may be an impact in terms of some loss of accuracy, but in many cases these are an acceptable compromise in order to guarantee a solution within a given time window.

Transactional Fault Tolerance
Transactional fault tolerance concept is closely related to the distributed database systems. This is used as a way of ensuring data consistency within distributed databases [Bernstein and Goodman, 1981]. Consistency will be achieved either by completing the submitted operation to the database successfully, or rolled back the database operation to a state prior to the operation was attempted. By performing updates in this atomic fashion, the database is protected from corruption in the case where the operation failed. With the popularity of concurrency in computing, transactional memory is introduced in [Herlihy et al., 1993] to assure the programmer that multiple concurrently running processes are not permitted writing on the same chunk of memory simultaneously. In order to achieve this goal, the ideas of transactions are currently considered into HPC, including a preliminary discussion of transactional fault tolerance in the MPI Standard.

2.3 MPI-Level Fault Tolerance

There are several MPI-level fault tolerance techniques available. These are as follows.

CoCheck
The *CoCheck* MPI [Stellner, 1996] is a combination of the Checkpoint/Restart and migration techniques. It uses a *single process checkpointer* which plays an important role of migrating the processes by saving the in-flight messages in a safe buffer and clearing the channel. This is achieved by exchanging a special message between the processes to indicate the clearance of the channel. If a process receives the special message, it assumes that there is no in-flight message left in the channel. Otherwise, it stores the special message in a special buffer as this is the in-flight message. When finally a process collects either the special or in-flight messages from all the processes destined to this process, it assumes that there is no ongoing message left in the channel. Hence, processes can now safely migrate with the checkpointer.

Starfish
The *Starfish* MPI [Agbaria and Friedman, 1999] combines group communication technology and the Checkpoint/Restart technique. This group communication technology allows the application to run without any disruption in the event that some of the nodes fail. Failure recovery is achieved by recomputing the part of the application which are disrupted due to failures. Recomputation is done either from the beginning, or from the recent checkpointed state by the Checkpoint/Restart technique. This is usually performed on the extra nodes added to the application by the Starfish on-the-fly. This run-time node adding feature allows Starfish to migrate the application processes from one node to another node with the assistance of the Checkpoint/Restart technique. Moreover, Starfish has the capability of performing both the application independent and application dependent fault tolerance.

MPI-FT
MPI-FT [Louca et al., 2000] is a fault-tolerant version of MPI. It performs failure recovery by means of reassigning tasks to the replacements for the dead processes. A detection technique is used for the detection of process failures. A centralized monitoring process (called the *Observer*), responsible for notifying the failure event to the rest of the alive processes, performs the recovery action. There are two modes of the recovery action. The first one performs distributed buffering of message traffics on each process. When the Observer detects process failures, it performs recovery by resending buffered messages from all the processes to the replacement processes, those are originally destined for the dead ones. The second one is based on the idea of centrally storing every message traffics by the Observer, and resend these to the replacements for the dead processes.

One of the problems encountered with MPI-FT for performing recovery action is the recovery of the dead communicator(s). This problem is solved by either preparing the spawning communicators in advance, covering every cases of the failure event by creating and using a communicator matrix, or spawning the replacement processes when the program starts executing. The disadvantages of this technique are that it pauses the synchronization due to the collective operations responsible for spawning the communicators, and the alive processes on the new communicator may have some pending messages destined for the old communicator which need to

be synchronized.

MPICH-V

MPICH-V [Bosilca et al., 2002] is a fault-tolerant version of MPI combining features from the uncoordinated Checkpoint/Restart and message logging techniques. With the MPICH-V run-time support, any application written in standard MPI could be made fault-tolerant. The key idea is that a *Dispatcher* coordinates the whole application execution by periodically collecting ''alive" messages from all the nodes, and at the same time keeps records of all the communications in stable *Channel Memories* (CM). In addition to this, every node checkpoints their task images to a *Checkpoint Server* (CS). If any ''alive" message is not received for a certain period of time, the Dispatcher assumes that the particular node is dead, and restarts the execution from the point of failure with the support of the CS. By this time, if that faulty node rejoins the system, the duplicate instance removal is managed by the CM. Moreover, network connection management for the alive and dead nodes is achieved by the CM.

The service provided by MPICH-V seems to be automatic and transparent to an application developer. However, periodic monitoring of all the nodes and periodic checkpointing of all the nodes task images to stable storages incur a large overheads to the application.

FT-MPI

FT-MPI [Fagg and Dongarra, 2000] offers a number of options for automatic process-level fault tolerance within the MPI library itself. This is achieved by simply calling a new communicator creation function, such as `MPI_COMM_DUP` or `MPI_COMM_CREATE`, in the application, with almost no impact on the user code.

FT-MPI provides the following three types of recovery modes to chose from by an application developer.

- The first recovery mode is SHRINK, which builds a new communicator excluding the failed processes and shrink the communicator size. Although the alive processes are ordered in the communicator, but for some applications where computation depends on the consistent values of the local ranks, this shrinkage could cause problems.
- The second recovery mode is BLANK. This is similar to SHRINK in the sense that all the failed processes are removed from the reconstructed communicators. However, without shrinking the communicator size, it replaces them with invalid process ranks. Although communication with the invalid ranks causes error, but those are left for future development to replace with new processes so that there is no disruption in inter-process communication.
- The third and most well-supported recovery mode is REBUILD. It automatically replaces failed processes by the newly created processes. The original communicator size and the process rank orders are left unchanged. With default communicator (`MPI_COMM_WORLD`), newly created processes are automatically restarted with the same command-line parameters as the original

processes. However, for the other communicators they must be reconstructed manually.

Initially, FT-MPI was built on the top of *Parallel Virtual Machine* (PVM) due to the unavailability of proper MPI run-time. Later, the HARNESS run-time [Fagg et al., 2001], originally implemented in Java, was rewritten in C to be used for FT-MPI. This run-time provides important features such as the ability to create new processes, examining their health, and monitor the status of all processes executing the application.

Although FT-MPI had lots of functionalities to provide the fault tolerance support, it was never adopted into the MPI standard due to the lack of standardization, and its development was discontinued.

ULFM MPI

User Level Failure Mitigation (ULFM) [Bland, 2013a] MPI can be considered as an attempt of resolving the non-standardization issue of FT-MPI. The MPI Forum's Fault Tolerance Working Group began the implementation of standard fault-tolerant MPI by introducing a new set of semantics on the top of the existing standard MPI library [Bland, 2013b]. Semantics of the draft standard include the detection and identification of process failures, propagating the failure information within the alive processes in the faulty communicator, and so on. Usually process failures are detected by checking the return code of the collective communication routines. With the *run-through stabilization* mode [Fault Tolerance Working Group] of ULFM MPI, surviving processes can continue their operations while others fail. The alive processes can form a fully operational new communicator without getting any disruption from the dead processes. It is also possible to create the replacement processes for the failed ones to recover the original communicator. Based on the requirements of the application, either the local or global recovery is also possible.

ULFM MPI supports the coordinated Checkpoint/Restart without modification. It is also possible to create the uncoordinated Checkpoint/Restart without requiring the application to restart entirely. The implementation of message-logging techniques and transactional fault tolerance on the top of ULFM MPI also achieves some benefits. Furthermore, ABFT techniques could be easily integrated with ULFM MPI. For details, see Chapter 4.

2.4 The Sparse Grid Combination Technique

PDEs are typically solved numerically by first discretizing the domain as points on a full regular grid. This suffers from the *curse of dimensionality*, that is, with uniform discretization across all dimensions there is an exponential increase of the number of grid points as the dimensionality increases. In order to solve this problem, high-dimensional PDEs may be solved on a *sparse grid* [Bungartz and Griebel, 2004] consisting of relatively fewer grid points than the regular isotropic grid.

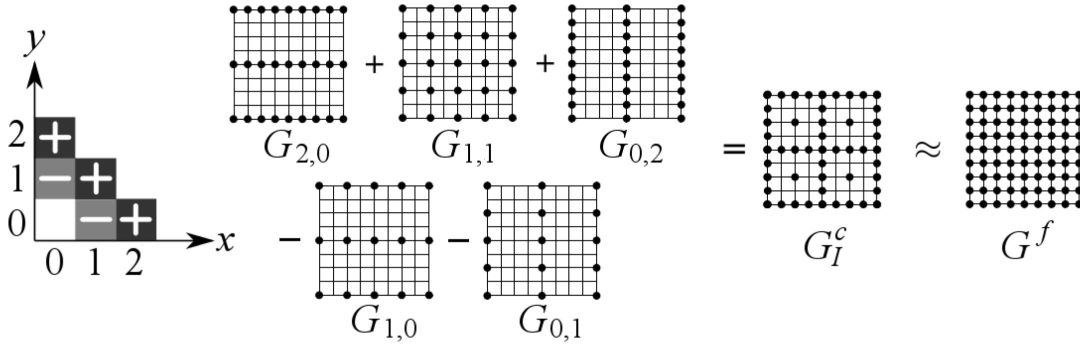


Figure 2.3: The Sparse Grid Combination Technique. $G_{i,j}$, G_I^c, and G^f represent the sub-grid, sparse or combined grid, and full grid equivalent to the sparse grid, respectively, for the 2D case. A distinct set of processes computes each $G_{i,j}$ in parallel via domain decomposition. Solutions on $G_{i,j}$ are linearly combined to approximate the solution of G^f on G_I^c. Multiple processes are also running on G_I^c.

2.4.1 Classical Sparse Grid Combination Technique

The *Sparse Grid Combination Technique* (SGCT) [Griebel, 1992; Griebel et al., 1992b] is a method of approximating the sparse grid solution which in turn approximates the full grid solution. Instead of solving the PDE on a full isotropic grid, it is solved on a set of small *anisotropic* regular grids referred to as *sub-grids* or *component grids*. Finally, solutions on these sub-grids are linearly combined to approximate the solution on the sparse grid (or, in this context, called *combined grid*). For the 2D problem, this technique is illustrated in Figure 2.3. This technique can be applied in principle to any PDE, but sufficient smoothness of the solution is required for high accuracy.

Suppose that each sub-grid $G_{i,j}$ in 2D has $(2^{i+1}+1) \times (2^{j+1}+1)$ grid points with a grid spacing of $h_x = 2^{-i-1}$ and $h_y = 2^{-j-1}$ in the x and y directions, respectively, where $i, j \geq 0$. With a 2D domain, the grid points of $G_{i,j}$ are $\{(\frac{x}{2^{i+1}}, \frac{y}{2^{j+1}})|x = 0, 1, \cdots, 2^{i+1}, y = 0, 1, \cdots, 2^{j+1}\}$. In the more general case, the index space for the grids will be some finite $I \subset \mathbb{N}^d$, where d is the grid dimension, and the set of grids of interest can be denoted by $\{G_{\underline{i}}, \underline{i} \in I\}$. If $u_{\underline{i}}$ denotes the approximate solution of a PDE on $G_{\underline{i}}$, the combination solution u_I^c on grid G_I^c generally takes the form

$$u_I^c = \sum_{\underline{i} \in I} c_{\underline{i}} u_{\underline{i}}, \tag{2.1}$$

where the $c_{\underline{i}} \in \mathbb{R}$ are the combination coefficients. Clearly, the accuracy of the combination technique approximation depends on the choice of the index space I of the sub-grids and their respective coefficients. In 2D, good choices of the coefficients are ± 1 [Larson et al., 2013a]. For instance, in the classical case, we have for level l the set $I = \{(i,j)|i,j \geq 0, l-2 \leq i+j \leq l-1\}$ and the combination coefficients are $c_{i,j} = 1$

if $i + j = l - 1$ and $c_{i,j} = -1$ if $i + j = l - 2$. This provides the combination formula

$$u_I^c = \sum_{i+j=l-1} u_{i,j} - \sum_{i+j=l-2} u_{i,j}. \tag{2.2}$$

Note that level $l = 3$ for the classical SGCT shown in Figure 2.3.

The computation on sub-grids $G_{\underline{i}}$ and their combinations are performed in the following way. At a time-step t_i, PDE instances are computed (solved) concurrently on each $G_{\underline{i}}$ with the corresponding grid points and spacing. This is continued for T successive time-steps with step-size Δt. After that, all the solutions $u_{\underline{i}}$ on $G_{\underline{i}}$ are assembled to get the combined solution u_I^c on grid G_I^c for time-step t_{i+T}. Then, u_I^c is projected for all the sub-grids $G_{\underline{i}}$ with the properly adjusted weights, and the whole process is repeated with $t_i = t_{i+T}$.

In this thesis, we use the notion of *single* and *multiple* combinations. If the above mentioned process is not repeated, we call it a single combination. In this case, T becomes the same as the total number of time-steps applied to compute the solution on the equivalent full grid G^f, say T'. On the other hand, with multiple combinations, the process is repeated for multiple times, but with $T < T'$. If we want to perform n repeated combinations, then T becomes T'/n.

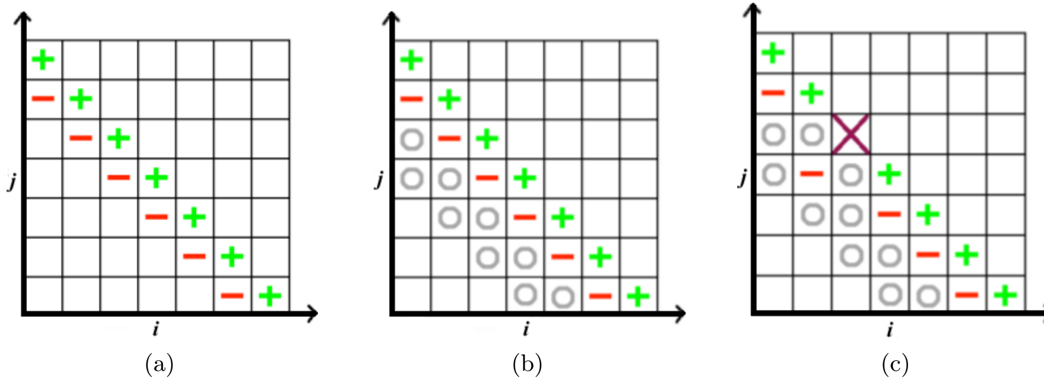
In this thesis, we also use so-called 'truncated' combinations [Benk and Pfluger, 2012], where, for the 2D case, each sub-grid has $(2^{i+1} + 1) \times (2^{j+1} + 1)$ points, for some $(i, j) \geq (i' + 1 - l, j' + 1 - l)$. This avoids the problem of minimum dimension size imposed by some applications. Furthermore, it allows us to avoid the use of highly anisotropic grids (e.g. $G_{0,l-1}$), which have been known to contribute least towards the accuracy of the sparse grid solution or cause convergence problems [Benk and Pfluger, 2012], and enabling us to concentrate process resources on more accurate sub-grids. In this context, we use a different notion of level to that described previously, which describes how much smaller the sub-grids are relative to some full grid $G_{i',j'}$. In particular, a level $l \leq \min\{i' + 1, j' + 1\}$ in this context consists of sub-grids from the index set

$$I = \left\{ (i,j) : \begin{array}{c} (i' + 1 - l, j' + 1 - l) \leq (i,j) \\ i' + j' - l \leq i + j \leq i' + j' + 1 - l \end{array} \right\}. \tag{2.3}$$

Similarly, for the 3D SGCT with a reference full grid $G_{i',j',k'}$, a level $l \leq \min\{i' + 1, j' + 1, k' + 1\}$ consists of sub-grids from the index set

$$I = \left\{ (i,j,k) : \begin{array}{c} (i' + 1 - l, j' + 1 - l, k' + 1 - l) \leq (i,j,k) \\ i + j + k \leq i' + j' + k' + 2 - 2l \\ i + j + k \geq i' + j' + k' - 2l \end{array} \right\}. \tag{2.4}$$

Two levels of parallelism are achieved in the SGCT computation. Firstly, different sub-grids are computed in parallel. Secondly, each sub-grid, $G_{\underline{i}}$, is assigned to a different process group and is computed in parallel via domain decomposition.



(a) (b) (c)

Figure 2.4: A depiction of the 2D SGCT. '+', '–', and 'o' on a sub-grid denotes the computed solution on that sub-grid is added, subtracted, and ignored, respectively, on the combined solution. 'x' on a sub-grid denotes the solution on that sub-grid is lost and ignored in the combination. (a) Classical SGCT, (b) fault-tolerant SGCT with extra smaller sub-grids on two lower layers (marked with 'o') and without any loss of sub-grid solution, and (c) fault-tolerant SGCT in the event of a lost solution on sub-grid $G_{2,4}$.

In contrast to the full grid approach which consists of $\mathcal{O}(h_l^{-d})$ grid points, the SGCT consists of only $\mathcal{O}(h_l^{-1}(\log_2 h_l^{-1})^{d-1})$ grid points, where $h_l = 2^{-l}$ denotes the employed grid spacing with level l, and d is the dimension. The accuracy of the solution obtained from the SGCT deteriorates only slightly from $\mathcal{O}(h_l^r)$ to $\mathcal{O}(h_l^r(\log_2 h_l^{-1})^{d-1})$ for a sufficiently smooth solution of order r methods [Garcke and Griebel, 2000].

2.4.2 Fault-Tolerant Sparse Grid Combination Technique

A fault-tolerant adaptation of the SGCT has been studied in [B. Harding and M. Hegland, 2013]. In this thesis, we refer to this adaptation as *Fault-Tolerant SGCT* (FT-SGCT). It is observed that the solution on even smaller sub-grids can be computed at little extra cost and that this added redundancy allows combinations with alternative coefficients to be computed. When a process failure affects one or more processes involved in the computation of one of the larger sub-grids, the entire sub-grid is discarded. In the event that some sub-grids have been discarded one must modify the combination coefficients such that a reasonable approximation is obtained using solutions computed on the remaining sub-grids. In 2D, this involves finding $c_{i,j}$ for formula (2.1) for which $c_{i,j} = 0$ for each $u_{i,j}$ which was not successfully computed. For a small number of failures this is typically done by starting with formula (2.2) and subtracting hierarchical surplus approximators of the form $u_{i',j'} - u_{i'-1,j'} - u_{i',j'-1} + u_{i'-1,j'-1}$ such that the undesired $u_{i,j}$ drop out of the formula whilst introducing some of the smaller sub-grids which were also computed. After a combination, all sub-grids may be restarted from the combined solution, including those which had previously failed. An approach for the general computation of combination

coefficients is described in [Harding et al., 2015].

For the 2D fault-tolerant SGCT computations in this thesis, two extra layers (or diagonals) of sub-grid solutions $u_{i,j}$ are computed satisfying $i + j = l - 3$ and $i + j = l - 4$. These two extra layers of sub-grids have levels $l - 3$ and $l - 4$, respectively. During fault-free operation these extra sub-grid solutions are not used in the combination formula (2.2). Rather, all sub-grid solutions $u_{i,j}$ with $i + j = l - 1$ and $i + j = l - 2$ are used. However, if any of the sub-grid solutions $u_{i,j}$ with $i + j = l - 1$ or $i + j = l - 2$ do not complete due to a fault, some of the extra and remaining unaffected sub-grid solutions $u_{i,j}$ are used in an alternate combination of the form (2.1) so that the combination gives the best result. An example of the default combination, an alternative leaving extra sub-grids unused, and an alternative using one of the extra sub-grids is depicted in Figure 2.4. For the 3D fault-tolerant SGCT computations in this thesis, one additional layer (or diagonal) of sub-grids with level $l - 4$ was computed (with the 3 layers $l - 1$, $l - 2$, and $l - 3$ necessarily computed for the default combination).

In this thesis, we also use 'truncated' combinations [Benk and Pfluger, 2012] for the FT-SGCT. The lower limit of formula (2.3) is changed to achieve this when extra layers are added, e.g. $i' + j' - l - 2 \leq i + j$ for two extra layers. Similarly, lower limit of formula (2.4) could be updated to achieve the 3D FT-SGCT.

2.5 Related Work

This thesis work lies at the intersection of four active research and development areas – parallelization of the SGCT, recovery of process and node failures with ULFM MPI, *Algorithm-Based Fault Tolerance* (ABFT) technique, and evaluation of the effectiveness of applying the SGCT to the GENE plasma micro-turbulence, Taxila *Lattice Boltzmann Method* (Taxila LBM), and *Solid Fuel Ignition* (SFI) application codes. Below we summarize and contrast work most closely related to ours.

A technique for replacing only a single failed process in the communicator and matrix data repair for a QR-Factorization problem was proposed in [Bland, 2013b]. Process failure was handled by ULFM MPI, and data repair was accomplished by using a reduction operation on a checksum and remaining data values. The author analyzed the execution time and overhead on a fixed number of processes in the presence of a single process failure. A detailed performance analysis of the recovery mechanism for multiple process failures, however, was not presented. Nor was the technique applied to a varying number of processes in other realistic parallel applications. A detailed performance evaluation of different ULFM MPI routines used to tolerate process failures was found in [Bland et al., 2012].

A fault-tolerant implementation of a multi-level Monte Carlo simulation that avoids checkpointing or recomputation of samples was proposed in [Pauli et al., 2013]. It used ULFM MPI to recover the communicator after failures by sacrificing its original size. A periodic reduction strategy was incorporated to all the samples unaffected by failures in the computation of the final result, and simply excluded the

samples affected by failures. The periodic communication/reduction is likely to be costly across multiple nodes, and the experimental results relating to multiple nodes were not provided. Reconstruction of the faulty communicator was not considered, nor was data recovery implemented.

Local Failure Local Recovery (LFLR) was proposed in [Teranishi and Heroux, 2014]. It inherited the idea from diskless checkpointing [Plank et al., 1998] in which some spare processes accommodated a space for data redundancy and local checkpointing. This allows an application developer to perform a local recovery without disrupting the execution of whole application when a process fails. The idea is to split the original communicator into several group communicators and dedicate a spare process for each group to store the parity checksum of the corresponding group. In the event of a single process failure, the corresponding spare process replaces the failed one with ULFM MPI, and recover the lost data locally from the local memory checksum. It requires local checksum to be updated periodically, which seems to be costly. Spare processes are used in the LFLR approach to handle only a single process failure. In this thesis, extra processes are also used for a small amount of redundant computations, but we are able to tolerate multiple process/node failures.

A customized run-time based simplified programming model called *Fault-Tolerant Messaging Interface* (FMI) was designed and implemented in [Sato et al., 2014] to improve the resilience overheads of the existing multi-level checkpointing method and MPI implementations. The semantics needed to write applications were similar to MPI, but the resiliency of applications was ensured by the FMI interface. Scalable failure detection with the help of a log-ring overlay network, fast in-memory checkpointing on spare nodes, and dynamically allocating spare compute resources in the event of failures were proposed. Although the main objective of providing resiliency to applications is the same, we are applying an ABFT for the approximate recovery of multiple failures, rather than the exact recovery through diskless checkpointing. Our failure detection and process recovery techniques are also different.

Early work in the parallelization of the SGCT for Laplace's equation and the 3D Navier-Stokes system were reported in [Griebel, 1992] and [Griebel et al., 1996], respectively. However, fault-tolerant issues were not considered.

ABFT techniques for creating robust PDE solvers based on the FT-SGCT were proposed in [Larson et al., 2013a; B. Harding and M. Hegland, 2013]. The proposed solver can accommodate the loss of a single or multiple sub-grids. Grid losses were tolerated by either deriving new combination coefficients to excise a faulty sub-grid solution or approximating a faulty sub-grid solution by projecting the solution from a finer sub-grid. This work, however, was implemented using simulated, rather than genuine, process failures. Furthermore, this work assumed that an application process failure is followed by a recovery action such as communicator repair, but did not actually implement such a mechanism. Finally, the results used a simple advection solver, whereas the work in this thesis uses real-world and pre-existing applications.

The first application of the SGCT to GENE was reported in [Kowitz et al., 2012]. Under this scheme, sub-grid instances of GENE were run, with their respective out-

puts written to data files that were subsequently combined to compute the SGCT solution. Complimentary work to ours on load balancing of GENE sub-grid instances and an alternative hierarchization based implementation of the SGCT has been reported in [Heene et al., 2013] and [Hupp et al., 2013], respectively. None of these aforementioned efforts has investigated the fault-tolerant possibilities of the SGCT for this application, nor have they implemented any alternative fault-tolerant techniques.

The effectiveness of ABFT by applying the FT-SGCT to GENE was analyzed in [Parra Hinojosa et al., 2015; Pflüger et al., 2014]. An analysis of solution accuracies in the event of several sub-grids lost, and the overhead of computing redundant smaller sub-grids were presented there. The load balancing implemented there was from the developed load model from a *master-slave* parallelism model. In this thesis, we contribute the tolerance of real process and node failures with ULFM MPI, which was absent there. Moreover, we provide a load balancing scheme on a global *Single Program Multiple Data* (SPMD) parallelism model and show how several SGCTs could be applied to the non-SGCT dimensions concurrently[4].

2.6 Summary

This chapter introduces key background material. We discuss the importance of fault tolerance, the reasons behind the failure of supercomputer nodes, an overview of some failure recovery techniques, including fault tolerance techniques implemented on the top of the MPI library. We discuss the SGCT and a fault-tolerant version of the SGCT, which provide necessary background for the key contributing thesis chapters. We further discuss some research work closely related to the contributions of this thesis. Before we move to the primary contributions of the thesis, we next give an overview of our implementation, and experimental platform.

[4]The concept of non-SGCT dimensions arises from the scenario where the total number of dimensions of the SGCT is smaller than that of the application solution field. Non-SGCT dimensions could be any of the dimensions among the lower dimensions which are usually forming blocks to fit into the SGCT dimensions. As for example, if we have two SGCT dimensions (say, x and y), and three field dimensions (say, x, y, and z; where z is the lower dimension), then z is the non-SGCT dimension. In this scenario, each element in the SGCT dimensions will be a block of elements with block size equals to the size of dimension z. For details, see Section 5.2.2.

Chapter 3

Implementation Overview and Experimental Platform

This chapter presents an overview of our implementation, hardware and software platform, fault injection technique, and measurement methodologies that we use throughout the evaluations presented in this thesis.

3.1 Parallel SGCT Algorithm Implementation

An implementation of the *direct* SGCT algorithm is used in this thesis. The key idea of this algorithm is to perform a scaled addition of part of each sub-grid's solution $u_{\underline{i}}$ in $P_{\underline{i}}$ to P^c to get the combined (or sparse) grid solution u_I^c.

With the direct SGCT algorithm, each PDE instance whose solution is $u_{\underline{i}}$ is run on a distinct set of processes denoted by $P_{\underline{i}}$ and is arranged in a logical d-dimensional grid. The algorithm consists of first a *gather* stage, where each process in $P_{\underline{i}}$ sends its portion of $u_{\underline{i}}$ to each of the corresponding (in terms of physical space) processes in a logical d-dimensional grid P^c to be scaled added into u_I^c. This is illustrated in Figure 3.1. The portion of $u_{\underline{i}}$ is selected based on the local to global mapping of processes in each d dimension from $P_{\underline{i}}$ to P^c. Suppose, a 2D process grid $P_{\underline{i}}$ is represented by $\{P_{\underline{i}}^x, P_{\underline{i}}^y\}$, and P^c by $\{P_x^c, P_y^c\}$. If $P_{\underline{i}}^x = P_x^c$ and $P_{\underline{i}}^y = P_y^c$, then the whole solution $u_{\underline{i}}$ is scaled added into u_I^c (initially u_I^c is empty) with an exact mapping of processes. Otherwise, solution $u_{\underline{i}}$ is split into $\frac{P_x^c}{P_{\underline{i}}^x}$ and $\frac{P_y^c}{P_{\underline{i}}^y}$ parts in x and y dimensions, respectively, and then scaled added each chunk of $u_{\underline{i}}$ into u_I^c. In this case, each process in $P_{\underline{i}}^x$ and $P_{\underline{i}}^y$ is mapped into $\frac{P_x^c}{P_{\underline{i}}^x}$ and $\frac{P_y^c}{P_{\underline{i}}^y}$ processes of P_x^c and P_y^c, respectively. Finally, each process in P^c then gathers the $|I|$ versions of each point of the full grid (using interpolation where necessary), and performs the summation according to formula (2.1) to get the combined (or sparse) grid solution u_I^c, which can be used as an approximation to the full grid solution. The use of interpolation in turn requires that a 'halo' of neighbouring points (in the positive direction, for our implementation) have been filled by a halo exchange operation by each process in each $P_{\underline{i}}$ and is also sent in the gather stage. For reasons of efficient resource utilization, P^c is made up of a (normally near-maximal) subset of all processes in $\cup_{\underline{i} \in I} P_{\underline{i}}$.

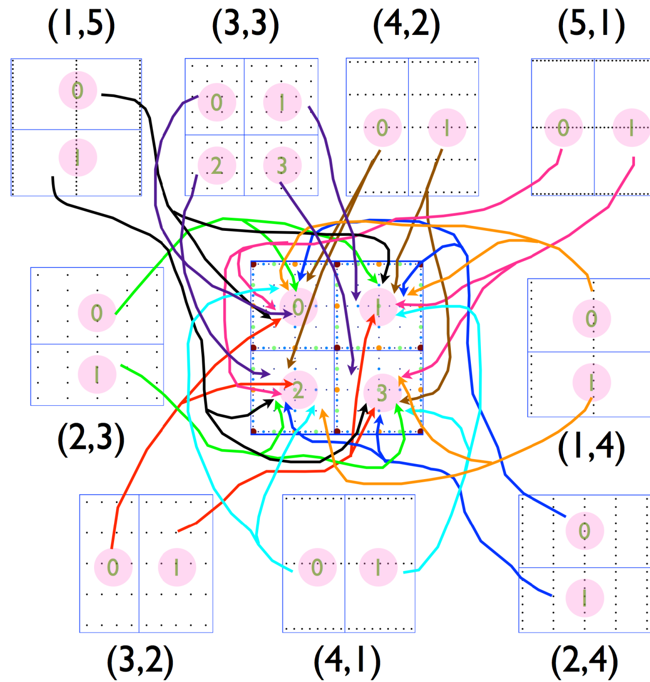


Figure 3.1: Figure 4 from [Larson et al., 2013b]. Message paths for the gather stage for the 2D direct combination method (not truncated) on a level $l = 5$ combined grid. The combined grid and component grid (3,3) have 2×2 process grids, all others have 2×1 or 1×2 process grids.

Similarly, in the scatter stage, a reverse mapping of processes from P^c to $P_{\underline{i}}$ is done to scatter a down sample of u_I^c to $P_{\underline{i}}$, iteratively, for each $\underline{i} \in I$.

Further details on the algorithm using a full grid representation of the combined grid G_I^c are available in [Strazdins et al., 2015]. An improved version of this algorithm, where a *partial* sparse grid, rather than a full grid, representation of G_I^c is used to perform an efficient interpolation on G_I^c, is available in [Strazdins et al., 2016b]. We used this improved version of the algorithm in this thesis, except where otherwise indicated.

In terms of load balancing, we used a simple strategy to balance the loads among the processes. The same number ($p' \in \mathbb{N}$) of processes is allocated on each of the distinct set of processes $P_{\underline{i}}$ for each sub-grid on the uppermost diagonal (i.e., for the 2D case, each $G_{i,j}$ with $i + j = l - 1$) in the grid index space. Each sub-grid on the next lower diagonals (i.e., for the 2D case, each $G_{i,j}$ with $i + j = l - 2$, $i + j = l - 3$, and $i + j = l - 4$) is allocated $\lceil p'/2 \rceil$, $\lceil p'/4 \rceil$, and $\lceil p'/8 \rceil$ processes, respectively. Details are discussed and analyzed in Section 6.3 of Chapter 6.

A failure of computing nodes or application processes causes the loss of some processes on some grids $G_{\underline{i}}$. This is handled as follows. Before the SGCT algorithm is applied, the loss of any processes in $P_{\underline{i}}$ is detected using ULFM MPI (see Chapter 4 for details). Replacement processes are then created (with the same process grid size as $P_{\underline{i}}$) on the same node when that node is still available (i.e., the failure is not

permanent). Otherwise, replacements are created on a spare node. Following the reconstruction of communicators, an alternate combination formula (see Section 2.4) is derived which sets a combination coefficient of $c_{\underline{i}} = 0$ for the lost sub-grid solutions $u_{\underline{i}}$. Note that this formula can be computed on all current processes. In this case, the gather of $u_{\underline{i}}$ on the replaced $P_{\underline{i}}$ and P^c is not performed. Note that the replaced $P_{\underline{i}}$ and P^c do participate in the scatter operation so that they are populated with the combined data to replace the lost data.

3.2 Hardware and Software Platform

All experiments were conducted on the Raijin cluster managed by the *National Computational Infrastructure* (NCI) with the support of the Australian Government. Raijin has a total of 57,472 cores distributed across 3,592 compute nodes each consisting of dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors (i.e., 16 cores) with InfiniBand FDR interconnect, a total of 160 terabytes (approximately) of main memory, and 10 petabytes (approximately) of usable fast filesystem operated by the `x86_64 GNU/Linux` OS [Cit].

We used git revision `icldistcomp-ulfm-46b781a8f170` of ULFM MPI (as of 13 December 2014) under the development branch 1.7ft of Open MPI for our experiments. The parameters for the collective communications for `mpirun` were set to `coll tuned,ftbasic,basic,self`. The value of the MCA parameter `coll_ftbasic_method` was set to 1 to choose the 'Two-Phase Commit' as an agreement algorithm for the failure recovery. The 'Log Two-Phase Commit' option was more scalable than the used one, but could not be used in our experiments due to its instability. All the source code (including ULFM MPI) were compiled with `GNU-4.6.4` compilers using the optimization flag `-O3`. The versions of PETSc [Balay et al., 2014] and MPI were `petsc-3.5.2` and `openmpi-1.4.3` (used for the simulations with non-real process failures), respectively.

Although InfiniBand interconnect was used in the Raijin cluster, the BTL component TCP was used while executing applications. Due to an issue in `icldistcomp-ulfm-46b781a8f170` distribution of ULFM MPI, the execution of applications reported an warning that there was an error initializing the OpenFabrics device. This issue has been fixed in the recently released ULFM MPI version 1.0 and subsequent commits, but considering the wastage of lots of CPU hours, we did not repeat the experiments.

3.3 Fault Injection

Faults were injected into the application by aborting a single or multiple MPI application processes at a time (with the exception of process 0 as it held critical data) by the run-time system call `kill(getpid(), SIGKILL)` at some point before performing the combination of the sub-grid solutions. The same effect was observed when the processes were killed by the `kill -9 <PID>` command from the command-line,

where <PID> was the application process identification number (either a single or multiple) extracted by the `ps -A | grep <executable_application_name>` command from the command-line when the application was in execution. The processes were also killed repeatedly (not at a single time) to analyze the repeated failure recovery performance of the application.

3.4 Performance Measurement

The IPM profiling tool [Wright et al., 2009; IPM] was used to report the total memory usage of the applications. The deallocation of memories in the application code was disabled to generate the memory usage reports. The reason behind this approach was that a mixture of different programming languages were used to create the SGCT based applications. As for example, the SGCT was implemented in C++, but the applications integrated with it was implemented either in FORTRAN or in PETSc [Balay et al., 2014] (details in Chapter 5). With these interoperable programming languages, the best way of generating memory reports by the IPM tool was unclear. Thus, the `MPI_Pcontrol()` function was called in the main C++ program to create a code region consisting of computing the sub-grids and combining the sub-grids' solutions to generate the memory usage reports. Moreover, to save CPU hours, the number of time-steps was set to 1 or as minimum as possible in this context.

Both the TAU [Shende and Malony, 2006] and IPM [Wright et al., 2009; IPM] profiling tools were used to analyze the load balancing and communication profiles. The ways of generating profiles for computing the sub-grids and performing the combination as a whole, and in isolation were a little bit different. For the former case, the sequence of instructions were: (1) setting a barrier (by the `MPI_Barrier()` function call) before the computation of sub-grids, (2) start a code region by the `MPI_Pcontrol()` function call function, (3) compute the sub-grids, (4) perform the combination, and (5) end the code region by the `MPI_Pcontrol()` function call. For the latter case, the sequence of instructions were: (1) setting a barrier before the computation of sub-grids, (2) start a code region by the `MPI_Pcontrol()` function call, (3) compute the sub-grids, (4) end the code region by the `MPI_Pcontrol()` function call, (5) setting a barrier before performing the combination, (6) start a code region by the `MPI_Pcontrol()` function call, (7) perform the combination, and (8) end the code region by the `MPI_Pcontrol()` function call.

`MPI_Wtime()` function was used to measure the execution performance of the applications. In order to measure the whole application running time in isolation, barriers were placed in the same way as they were used for the analysis of the load balancing and communication profiles. Throughout this thesis, 'sec' and 'msec' represent seconds and milliseconds, respectively.

Approximation error was represented by the relative l_1 error of the combined field. It was computed by $\frac{\|u'-u\|_1}{\|u\|_1}$, where u was the field of the full grid solution and u' was that of the combined grid produced by the SGCT or its variants. This time, a full grid, rather than a partial sparse grid, representation of the combined grid G_I^c

was used.

Chapter 4

Application Level Fault Recovery by ULFM MPI

Studies show that the resiliency from the hardware and OS levels will incur more complexities and more power consumption. The design consideration of the next generation peta and exascale systems should keep the power consumption of each component at a minimum level to operate a large number of components together. Considering this constraint, the hardware design of large systems is targeted to remain as simple as possible, resulting no fault tolerance consideration from the system level. Achieving resiliency from the OS level requires a combination of major-changed system software and hardware to work together. This adds additional complexities in the system and increases the power consumption. As a result, a technique which does not increase the complexities and power consumption of the systems may be the suitable approach.

Handling resiliency from the application level does not add extra complexities to the system, and hence no additional power consumption. Although developers need to handle resiliency as a part of their application code, it provides more flexibility to the developers.

This chapter describes detailed implementation studies using the recent draft of the ULFM MPI semantics for the detection, identification, and recovery of process and node failures.

The organization of this chapter is as follows. Section 4.2 describes how the ULFM MPI semantics are used to detect and identify which application processes and nodes fail. Section 4.3 describes the techniques of fixing the broken communicator without and with losing its original size by ULFM MPI, and recovering the data of the failed processes/nodes. Experimental evaluation of the application level failure recovery is presented in Section 4.4.

This chapter describes work published jointly with others as "Application Fault Tolerance for Shrinking Resources via the Sparse Grid Combination Technique" [Strazdins et al., 2016a], and "Application Level Fault Recovery: Using Fault-Tolerant Open MPI in a PDE Solver" [Ali et al., 2014].

4.1 Introduction

User Level Failure Mitigation (ULFM) [Bland, 2013a] is a draft standard for the fault-tolerant model of MPI. Implementation of this standard is initiated by the MPI Forum's Fault Tolerance Working Group by introducing a new set of tools into the existing MPI to create fault-tolerant applications and libraries. This draft standard allows application programmers to design their recovery methods and control them from the user level, rather than specifying an automatic form of fault tolerance managed by the OS or communication library [Bland et al., 2012; Bland, 2013b].

ULFM works on the *run-through stabilization* mode [Fault Tolerance Working Group], where surviving processes can continue their operations while others fail. Any MPI operation that involves a failed process will raise an MPI exception rather than waiting for an indefinite period of time to succeed the operation. If a point-to-point communication operation is unsuccessful due to a process failure, surviving process reports the failure of the partner process. With collective communication where some of the participating processes fail, some processes perform successful operations while others report process failures, which leaves the state as non-uniform. In such a scenario, the knowledge about failures is explicitly propagated and prohibit any further communication on the given communicator by setting the communicator in the revoked state.

Currently, automatic replacement of a failed process like FT-MPI [Fagg and Dongarra, 2000] is not possible in ULFM MPI. An application developer needs to use the ULFM MPI semantics to recover from process/node failures. Process failures can be detected using the return code of the ULFM MPI communication routines. By examining the return codes, one may identify which processes failed (if any) in a communicator. In the event of a failure, the communicator is revoked. A new communicator containing all the surviving processes can then be created by shrinking the revoked communicator. The size of this new communicator will obviously be smaller than that of the original communicator, however, the original size can be preserved by spawning replacement processes to merge with the new communicator.

4.2 Fault Detection and Identification

The first and most important step of fault-tolerant implementation of any application is to detect if any component failure occurs, and to list them if they occur. By component failure, we want to stick with process failure caused by any type of faults. Since node failure causes all the processes on the node to die immediately, it can also be detected in terms of detecting chunk of process failures.

4.2.1 Process Failure Detection and Identification

Process failure detection requires to create and attach a customized error handler to each MPI communicator. Since `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN` do not provide any guarantee that any further communication can occur after the

failure, ULFM MPI provides its own semantics to create a customized error handler to be attached to each communicator. The customized error handler includes the `OMPI_Comm_failure_ack()` and `OMPI_Comm_failure_get_acked()` functions to acknowledge and manage the local failures.

An application checks the return code of the collective function `MPI_Barrier()` to test if any process failure occurs in the communicator. With failure it returns code other than `MPI_SUCCESS`, and propagates the knowledge about the failure through the customized error handler. At the same time, it revokes the communicator by the `MPI_Comm_revoke()` function call so that no communication can occur in the communicator until it is fixed. Then it shrinks the communicator containing only the surviving processes by the `MPI_Comm_shrink()` function call. Then an old and new groups are created from the broken and shrunken communicators, respectively, by the `MPI_Comm_group()` function call. Finally, these two groups are compared by the `MPI_Group_compare()` and `MPI_Group_difference()` function calls to create a group containing only the lost processes, and then this group is translated to create a globally consistent list of failed processes by the `MPI_Group_translate_ranks()` function call.

For details, refer to Algorithm 5 of Appendix A.

4.2.2 Node Failure Detection and Identification

The failed process list could be analyzed to determine whether they are resulted in due to process failures or node failures. This requires a rankmap file (called hostfile) containing a list of compute nodes which are used by an application using the `--hostfile` option of `mpirun`. Each line entry of the hostfile contains distinct node name followed by `SLOTS` number. `SLOTS` is set to the total core count of a node, and each MPI process is mapped onto a distinct core. This configuration is used to identify node failures in terms of process failures as follows.

We used a simple idea to detect node failures. If the total failed process count is a multiple of `SLOTS` (say, $n \times \texttt{SLOTS}$, where $n \in \mathbb{N}$ and $n > 0$) for homogeneous nodes like Raijin, we could consider it as node failure. However, this may also happen due to non-node failures, where some nodes experience process failures, rather than node failures, and the sum of them is $n \times \texttt{SLOTS}$. As a result, checking only this condition is not enough for detecting node failures. An algorithm able to detect node failures first sorts all the failed process ranks into an ascending order, and then creates groups of failed processes with successive ranks, each with size `SLOTS`. If each group's total rank count is `SLOTS`, and starting rank is $SRANK = s \times \texttt{SLOTS}$, where $s \in \mathbb{N}$, then it may be node failures. If some failed processes are still left after this grouping, these out-of-group failed processes are due to process (i.e., not node) failures.

The identification of nodes which are believed to be failed is achieved by getting the hostfile line indices of the failed nodes (started from 0), simply calculated by $\lfloor SRANK/\texttt{SLOTS} \rfloor$, and extract the node names from these indices.

4.3 Fault Recovery

The second step of fault-tolerant implementation after detecting and identifying component failures is to reconstruct the faulty communicator. There are two ways of accomplishing this. One is to reconstruct the faulty communicator by preserving the original communicator size and rank distribution by creating and restoring the replacement processes for the dead ones, and the other one is with sacrificing the original communicator size by excluding the dead processes.

4.3.1 Faulty Communicator Reconstruction

In this section, we discuss both the spawning and shrinking based techniques.

4.3.1.1 Spawning Based Recovery

In this technique, faulty communicator reconstruction is done by spawning the replacement MPI processes by the `MPI_Comm_spawn_multiple()` function call. With process failures, spawning is done on the nodes which experience process failures. The node identity of a failed process with rank r is determined by extracting the node name from the hostfile with line index $\lfloor r/\texttt{SLOTS} \rfloor$. In case of node failures, replacement processes are created on the spare nodes listed in the hostfile. With repeated node failures, application tracks which spare nodes are currently available and which are already used. This is done by maintaining hostfile's spare line indices as *unused* or *used*, and extract the node name of an unused line index.

The extracted node names where to launch the replacement processes are used to set the `MPI_Info` object by the `MPI_Info_set()` function call, and pass the object to the `MPI_Comm_spawn_multiple()` function, along with other parameters, to spawn the processes on these nodes. The spawned processes are referred to as *child* processes and the rest are referred to as *parents*. The child and parent processes have their own intercommunicators through which they communicate with their own processes. Attaching the child to the parent is accomplished by merging their intercommunicators by the `MPI_Intercomm_merge()` function call. The ranks of the child processes on the merged (reconstructed) communicator should be the same as they were in the original communicator (before failure) so that there is no disruption in the application's original communication pattern. This is achieved by ordering the ranks by the `MPI_Comm_split()` function call with properly selected keys as input to it. Finally, the identity of the child communicator is converted into parent communicator by assigning `MPI_COMM_NULL` to it. In this way the reconstructed communicator becomes ready to use within the application.

Figure 4.1 shows the overall technique of reconstructing a faulty communicator including the ordering of child processes in the communicator. Some high-level/mid-level algorithms, and an implementation of these algorithms to be used as a toolkit of fault-tolerant application implementation are presented in Appendix A.

In order to advance the fault-tolerant application with the reconstructed communicator, some non-trivial modifications are needed to handle the initialization of

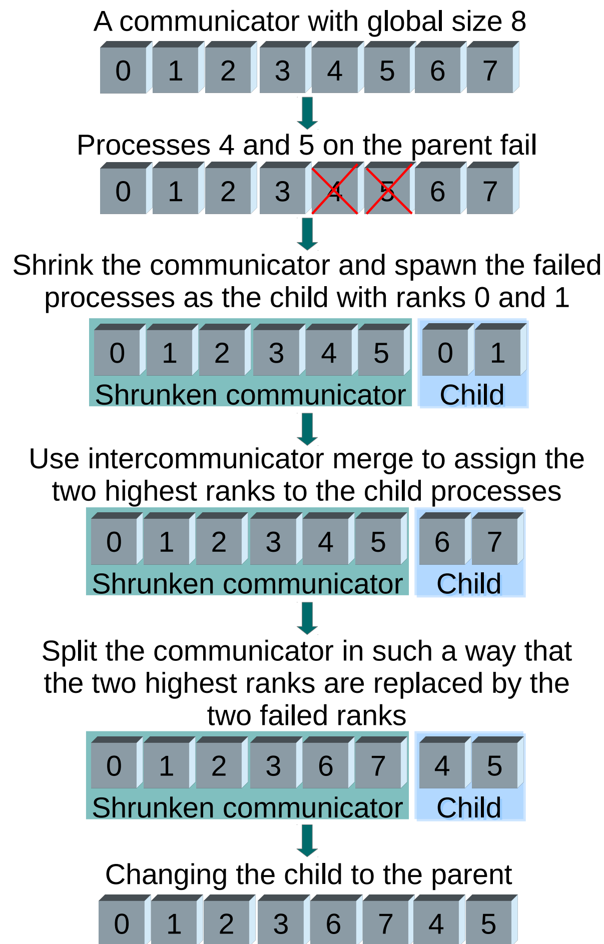
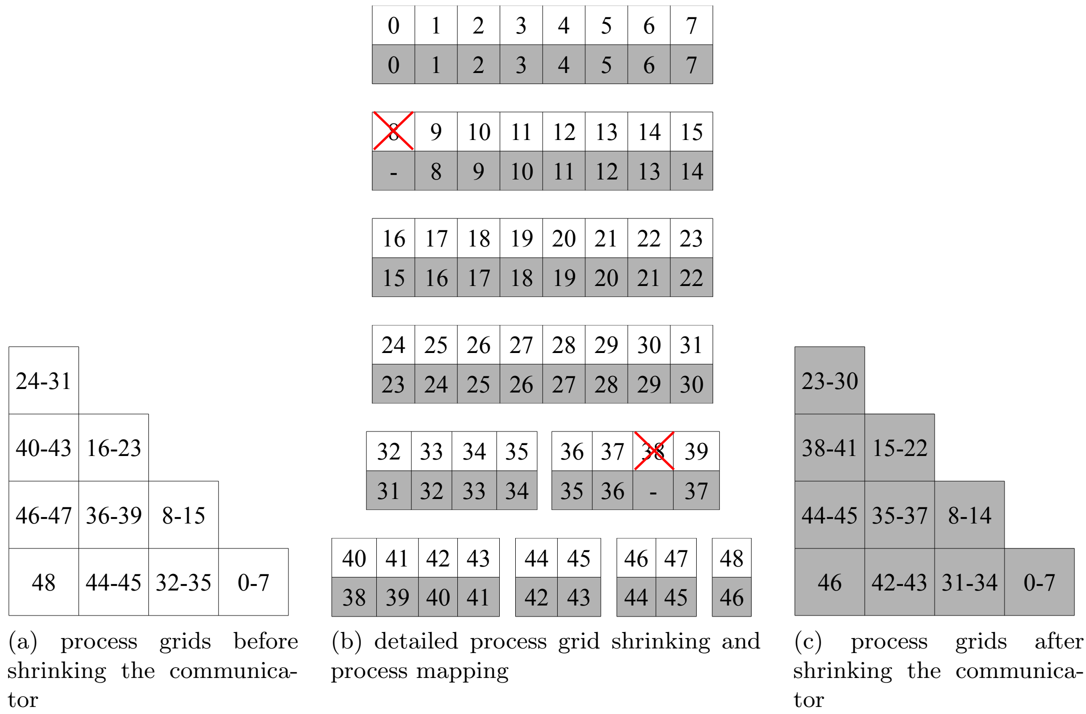


Figure 4.1: Techniques of recovering failed processes and assigning the same ranks as they were before the failure in the original communicator.

the spawned processes. The customized error handler is set up for the spawned processes. Run-time variables those are passed in the application as a command-line arguments are also set for the newly created processes. Consistency of data of some variables between the existing old and newly created processes is required. Some of these variables include the number of failed processes, list of failed processes, simulation step counter, and the others. This could be achieved by copying the data of process 0. If different groups of communicators split from the original global communicator are used in the application, these groups are created from the reconstructed global communicator for the spawned processes. Furthermore, the declaration, memory allocation, and initialization of some application variables are required.



(a) process grids before shrinking the communicator

(b) detailed process grid shrinking and process mapping

(c) process grids after shrinking the communicator

Figure 4.2: Process grid configurations of different sub-grids of the 2D FT-SGCT based applications with level $l = 4$ when the faulty communicator is shrunk as a recovery action. Numbers with white and gray background cell represents the MPI processes before and after shrinking the communicator, respectively. Mark 'X' represents the failure of an MPI process.

4.3.1.2 Shrinking Based Recovery

In this technique, fault tolerance is achieved without spawning the replacement processes. After detecting the process or node failures, the faulty communicator is shrunk, containing only the alive processes to complete the rest of the computation. With an SGCT-based application, this is achieved by shrinking the process grid and updating the data structures of the sub-grids that are experiencing failures. The sub-grids whose processes are not lost continue their operations without having disruption. However, after shrinking the communicator, a mapping of processes is required to access the appropriate data owned by each sub-grid.

An example of the overall approach is shown in Figure 4.2. The process grid for each sub-grid without any failures is shown in Figure 4.2a. Suppose, processes 8 and 38 fail. After this failure, the communicator is shrunk excluding processes 8 and 38. The process grids of sub-grids containing processes 8 and 38 are shrunk as shown in Figure 4.2b. The data structures of each process of these process grids are updated to adjust the whole range of grid points of the corresponding sub-grids. The remaining processes of the other sub-grids do not need to update their data structures. But a mapping of processes is required to point to the appropriate sub-grid data. As for

example in Figure 4.2b, processes 38 to 41 of the shrunken communicator play the role of processes 40 to 43 of the un-shrunken communicator. Otherwise, a disruption in communication and/or unexpected data transfer will happen. The process grid for each sub-grid after shrinking the faulty communicator is shown in Figure 4.2c.

Comparing Figures 4.2a and 2.4 indicates the ordering we use to delineate the process grid index space I.

4.3.2 Lost Data Recovery

The third and final step of fault-tolerant implementation after the reconstruction of faulty communicator is to recover the data of the failed processes. For the SGCT based 2D general advection solver (details in Section 4.4.1), this involves recovering the data of all the processes executing in parallel on a sub-grid, although only some of the processes of the corresponding sub-grid experience failures. Since any communication attempt with the failed processes raises an exception while trying to update the grid data, some of the surviving processes on an affected sub-grid contain invalid data. So, data recovery only for the failed processes on a sub-grid is insufficient for this application.

We have implemented and analyzed three techniques for recovering the lost data of the grids experiencing failures. These techniques are called the *Checkpoint/Restart* based SGCT (CR-SGCT), *Resampling and Copying* based SGCT (RC-SGCT), and alternate combination formula based Fault-Tolerant SGCT (FT-SGCT).

- The CR-SGCT supports exact data recovery. It periodically takes checkpoints onto disks while the computation of each sub-grid is in progress. In the event of process or node failures, it restarts with the recent checkpointed data and performs a recomputation for a number of time-steps the application advances from the last checkpoint event to the failure detection event. Figure 4.3a shows the grid arrangements of this. Total number of checkpoints, C, of the application is calculated by

$$C = \lfloor T_{CR}/T_{OC} \rfloor, \quad (4.1)$$

  where

  - $T_{CR} := T_{CR}(N)$ is the total run time of the CR-SGCT based application on N nodes.
  - $T_{OC} = \sqrt{2T_{WR} \cdot T_{fn}/N}$ is the optimal time between checkpoints proposed by Young [Young, 1974], where
    - $T_{WR} := T_{WR}(N)$ is the time required to write a checkpoint on N nodes.
    - T_{fn} is the *Mean Time Between Failures* (MTBF) on each node.
    - T_{fn}/N is the MTBF across N nodes.

- The RC-SGCT supports a combination of exact and approximate data recovery techniques. It creates a redundancy of the upper diagonal sub-grids so that the lost data of the upper diagonal sub-grids can be exactly recovered by copying

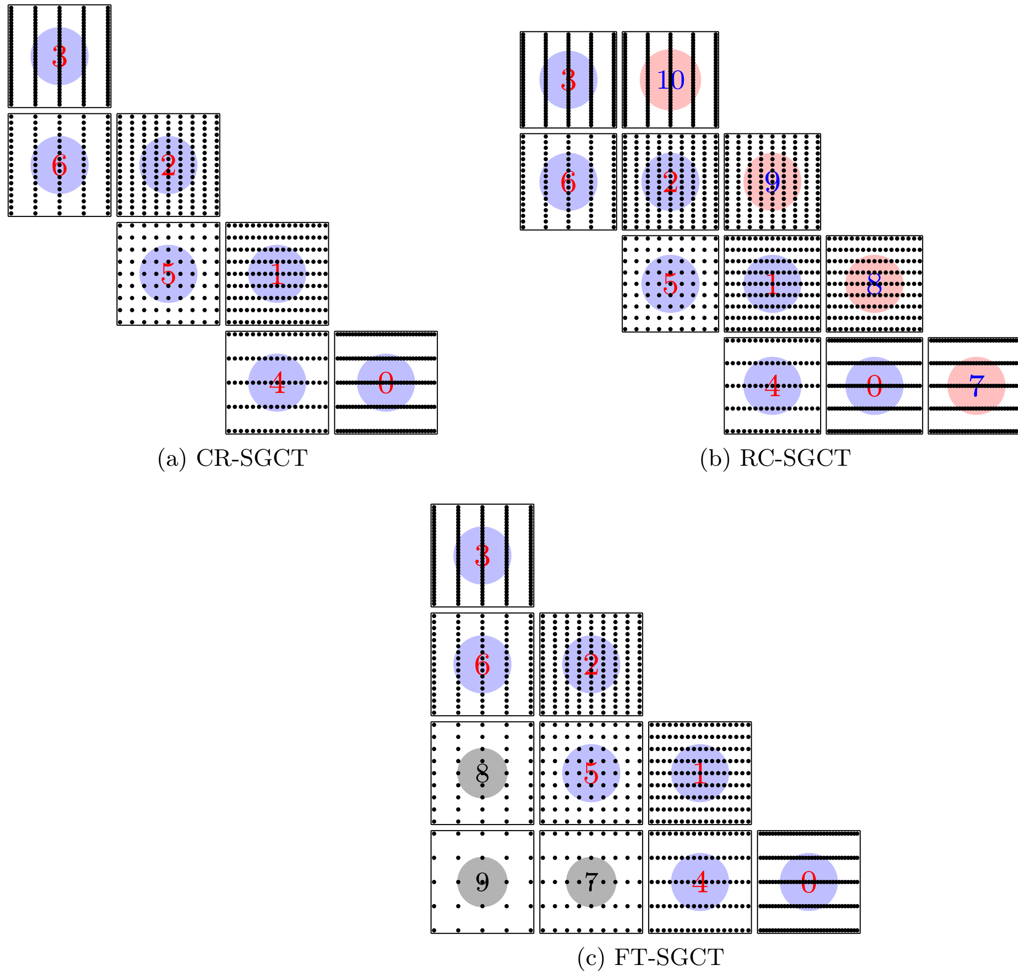


Figure 4.3: Grid arrangements of the 2D SGCT with level $l = 4$ solving the general advection problem to demonstrate different data recovery techniques. A distinct set of processes computes each sub-grid in parallel via domain decomposition. (a) Each sub-grid data is periodically checkpointed to the disk. Data recovery is achieved by reading the corresponding recent checkpoint file. (b) Grids with IDs 7, 8, 9, and 10 are the duplicate sub-grids of the upper diagonal sub-grids with IDs 0, 1, 2, and 3, respectively, and vice versa. The lost data of a sub-grid on either the duplicate or upper diagonal layer is recovered by copying the data from the corresponding duplicate sub-grid. The lost data of lower diagonal sub-grids with IDs 4, 5, and 6 are recovered by resampling the corresponding upper diagonal sub-grid's data. (c) Grids with IDs 7, 8, and 9 are the sub-grids on the extra two layers. Some of these are used to form FT-SGCT when some of the remaining regular sub-grids are lost. In this way it generates almost the same solution of classical SGCT without faults.

from its redundancy (and vice versa). As each lower diagonal sub-grid is a sub-set of the upper diagonal sub-grid (finer) above it, a resampling of the upper diagonal grid is used to recover the lost data of the lower diagonal sub-grid below it. According to the grid arrangements of Figure 4.3b, exact data

recovery of sub-grid 0 is done from sub-grid 7, 7 from 0, 1 from 8, 8 from 1, 2 from 9, 9 from 2, 3 from 10, and 10 from 3. Alternatively, approximate data recovery of sub-grid 4 is done by resampling data from sub-grid 1, 5 from 2, and 6 from 3.

- The FT-SGCT performs an approximate data recovery of the lost sub-grids. It employs some extra layers of sub-grids below the lower diagonal sub-grids. Figure 4.3c shows the grid arrangements of this, where the number of extra layers used in the implementation is two. In the event of failures, all the surviving sub-grids, including those on the extra layers, are assigned new coefficients for the combination, and then a sample of the combined solution is used as recovered data of the sub-grid whose data was lost [B. Harding and M. Hegland, 2013; Harding and Hegland, 2013; Harding et al., 2015]. Note that, unlike the CR-SGCT and RC-SGCT techniques, data recovery with this technique is only possible when the combination of sub-grid solutions is complete.

4.4 Experimental Results

In this section, the experimental setup, an analysis of the failure identification and communicator reconstruction performance, failed grid data recovery overheads, the approximation error of the solutions computed with the FT-SGCT, and the scalability achieved for the 2D general advection solver are presented.

4.4.1 Experimental Setup

A scalable PDE solver was chosen to make it fault-tolerant, and all the experiments of this chapter were conducted with this solver. It solved the scalar advection equation in two and three spatial dimensions using a MacCormack [MacCormack, 2003] and Lax–Wendroff [Lax and Wendroff, 1960] schemes, respectively. Throughout this thesis, we refer to this solver as the general *advection* solver.

In our experiments, 2D SGCT applied to the general advection solver was run for 2^{13} time-steps. Full grid size was $(2^{13}+1) \times (2^{13}+1)$, level was $l = 4$, and the number of combinations was one. At some stage failure detection was tested, and if needed, the recovery process was initiated for the RC-SGCT, and FT-SGCT techniques. However, for the CR-SGCT technique, failure detection was tested prior to initiating the checkpoint write onto disk, and if required, it restarted from the recent checkpoint data rather than writing onto disk. When the solver completed its execution for all time-steps, the sub-grids were combined and the overall solution was tested for accuracy.

There are some constraints on the ranks of processes which can fail, and the indices of sub-grids whose data can be lost. Failures in the RC-SGCT technique should not occur at the same time on any of the sets of two sub-grids which are in communication for recovering the data. For example, failures should not occur simultaneously on sub-grids 3 and 6, or 2 and 5, or 1 and 4, or 0 and 7, or 1 and 8,

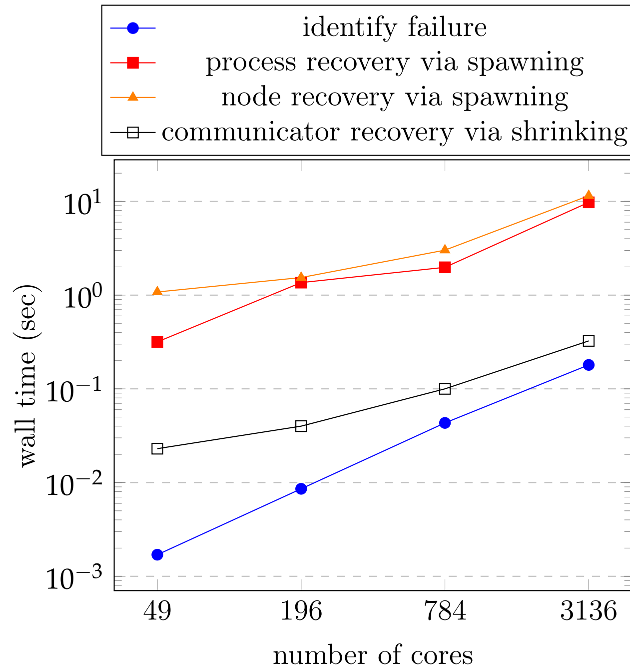


Figure 4.4: Times for generating the failure information and repairing the faulty communicator for the 2D SGCT solving the general advection problem when multiple failures occur. 'identify failure' represents the detection and identification of multiple processes failure. Each of the process, node, and communicator recoveries includes the time of 'identify failure' plus the time of the corresponding recovery via spawning the replacement processes or shrinking the communicator. The results shown are an average of five experiments.

or 2 and 9, or 3 and 10, according to the grid arrangements shown in Figure 4.3b. However, there are no such constraints in the CR-SGCT and FT-SGCT techniques. But for all of these techniques, there is a common constraint that process 0 must be alive in the application as it holds critical data.

Experimental results of Figures 4.4 and 4.7 consist of real (process or node) failures, but for Figures 4.5 and 4.6, they are non-real (simulated), i.e., only assuming that failures are occurring. MTBF on each node T_{fn} is set to 22.66 hours and disk write latency T_{WR} is measured as 0.03 sec.

4.4.2 Failure Identification and Communicator Reconstruction Overheads

Figure 4.4 shows the ULFM MPI performance for detecting and identifying (process or node) failures, recovering the failed processes or nodes via spawning the replacement processes, and recovering the faulty communicator via shrinking. It is observed from the experimental results that the wall times for generating the failure information and reconstructing the faulty communicator increase with core count. It is also observed that the maximum times required to reconstruct the faulty communicator due to multiple process and node failures are around 10 sec and 12 sec, respectively,

on 3,136 cores. This data seems reasonable as we conducted experiments with the beta version of ULFM MPI. However, which MPI functions are contributing most to this reconstruction process is worth investigating. It is observed from Table 4.1 that the major reconstruction times are contributed by `MPI_Comm_spawn_multiple()` and `MPI_Intercomm_merge()` functions.

It is also observed from Figure 4.4 that the shrinking based communicator recovery is less expensive than the spawning based recovery; with 3,136 cores, a 30× improvement is observed.

4.4.3 Failed Grid Data Recovery Overheads

Figure 4.5 shows the data recovery overheads of the failed grids for the CR-SGCT, FT-SGCT, and RC-SGCT techniques, when multiple or a single grids' data are lost. Recovery overheads of the CR-SGCT technique include times for creating all the checkpoints onto disk, reading the recent checkpoint, and the recomputations that are needed. With the FT-SGCT technique, only the time needed for creating the combination coefficients, rather than recovering the grid data which is happened as a compulsory stage later, is used as recovery overhead. Recovery overheads of the RC-SGCT technique include times for copying and/or resampling data from the finer sub-grids.

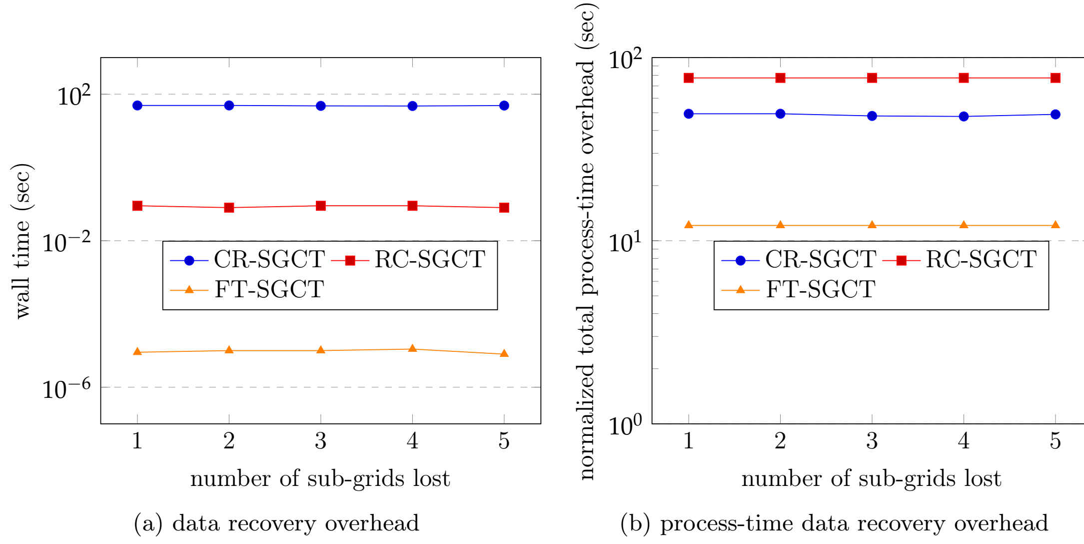
For these experiments, we have simulated failures on up to 5 sub-grids in order to determine the effect of the number of lost sub-grids on the data recovery time. Thus, the results do not include faulty communicator reconstruction time. We observe that in all cases data recovery times are almost independent of the number of lost sub-grids.

It is observed from Figure 4.5a that the CR-SGCT technique shows the highest overhead, FT-SGCT shows the lowest overhead, and RC-SGCT is in between these two.

A comparison of the data recovery overheads presented in Figure 4.5a may not be fair due to executing different number of application instances for the three techniques. To make a fair comparison, we can consider process-time data recovery

# cores	average wall time (sec)			
	`OMPI_Comm_shrink()`	`OMPI_Comm_agree()`	`MPI_Comm_spawn_multiple()`	`MPI_Intercomm_merge()`
49	0.002	0.01	0.23	0.03
196	0.008	0.01	0.28	1.02
784	0.043	0.01	0.81	1.04
3136	0.173	0.01	7.91	1.21

Table 4.1: ULFM MPI performance to recover multiple process failures.



(a) data recovery overhead

(b) process-time data recovery overhead

Figure 4.5: Failed grid data recovery overheads of the 2D SGCT solving the general advection problem. The number of processes on each diagonal (including duplicate), lower diagonal, upper extra layer, and lower extra layer sub-grid is 8, 4, 2, and 1, respectively. Results shown are an average of five experiments.

overheads. These process-time data recovery overheads are calculated by

$$T'_{REC,CR} = C \cdot T_{WR} + T_{REC,CR}\,,$$
$$T'_{REC,RC} = (T_{REC,RC} \cdot P_{RC} + T_{APP,RC}(P_{RC} - P_{CR}))/P_{CR}\,, and$$
$$T'_{REC,FT} = (T_{REC,FT} \cdot P_{FT} + T_{APP,FT}(P_{FT} - P_{CR}))/P_{CR}\,,$$

where

- $T'_{REC,CR}$, $T'_{REC,RC}$, and $T'_{REC,FT}$ are the normalized total process-time overheads of the CR-SGCT, RC-SGCT, and FT-SGCT techniques, respectively (these are normalized with respect to the number of processes used in the CR-SGCT).
- C is the optimal number of checkpoints of the application (see 4.1).
- T_{WR} is the single checkpoint write time onto disk by a process.
- $T_{REC,CR}$ is the recovery time of the CR-SGCT technique (time for reading checkpoint file and performing recomputation).
- $T_{REC,RC}$ is the recovery time of the RC-SGCT technique (time for copying and/or resampling sub-grid data).
- $T_{REC,FT}$ is the recovery time of the FT-SGCT technique (time for calculating combination coefficients).
- $T_{APP,RC}$ and $T_{APP,FT}$ are the total application times (excluding communicator reconstruction time) of the RC-SGCT, and FT-SGCT techniques, respectively.
- P_{CR}, P_{RC}, and P_{FT} are the total number of processes used for the CR-SGCT, RC-SGCT, and FT-SGCT techniques, respectively.

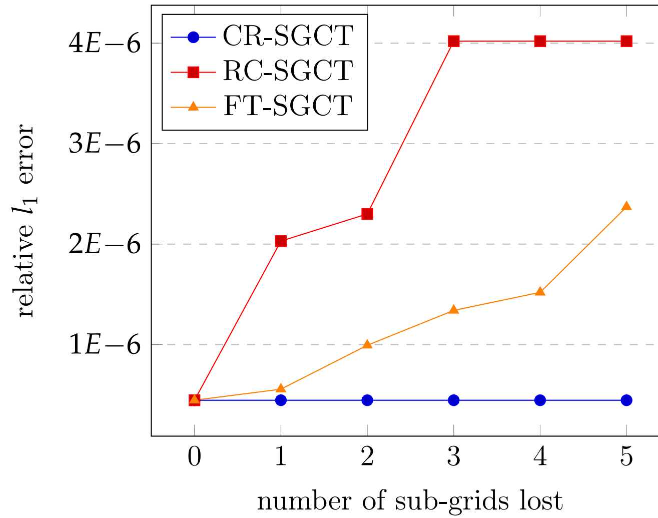


Figure 4.6: Approximation errors of the 2D FT-SGCT for the general advection solver. The number of processes on each diagonal (including duplicate), lower diagonal, upper extra layer, and lower extra layer sub-grid is 8, 4, 2, and 1, respectively. The results shown are an average of 20 experiments.

It is observed from the measured process-time data recovery overheads presented in Figure 4.5b that the RC-SGCT shows more process-time overheads, the FT-SGCT shows less process-time overheads, and the CR-SGCT is in between these two.

4.4.4 Approximation Errors

The accuracy of the combined solution with or without failures is shown in Figure 4.6. The error here is the average of the l_1-norm of the difference between the combined grid solution and exact analytical solution (which can be calculated for advection from the initial conditions).

It is observed that the CR-SGCT shows an error independent of the number of sub-grids lost, as it has exact data recovery; the error simply reflects that of an advection solver using the SGCT at the given grid resolutions. However, the average approximation errors of the other two techniques grow as the number of lost sub-grids increases; error of the RC-SGCT is more than the FT-SGCT. This observation indicates that resampling a lower resolution lost sub-grid from a high resolution sub-grid is less accurate than the FT-SGCT, which utilizes data from a lower resolution sub-grid for each sub-grid that is lost.

4.4.5 Scalability

The overall parallel performance of the application, when it experiences the failure of multiple real MPI processes, is shown in Figure 4.7. There are two trends observed from the figure. The first one is that execution of both the RC-SGCT and CR-SGCT takes longer time compared to the FT-SGCT. The second one is that there is a less

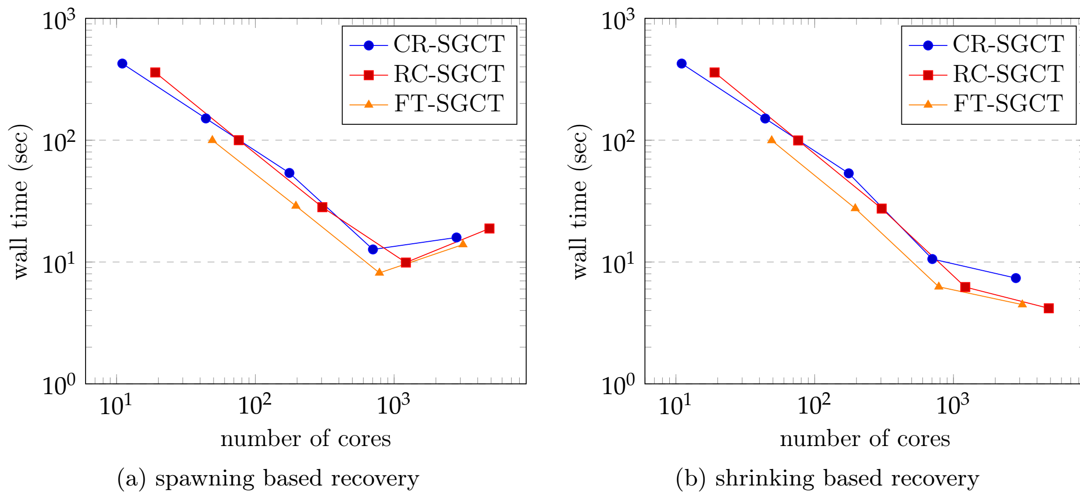


Figure 4.7: Overall parallel performance of the 2D SGCT solving the general advection problem with a single combination. Execution time in (a) includes faulty communicator reconstruction time by spawning the replacement MPI processes, data recovery time, and application running time. Execution time in (b) includes the same timing components as (a), but the faulty communicator reconstruction is done by shrinking the communicator. The results shown are an average of five experiments.

degradation of scalability for the shrinking based recovery than the spawning based recovery after 1,000 cores.

The first trend is explained by noting that the FT-SGCT involves only a relatively small amount of extra computations, whereas the RC-SGCT has a significant degree of replication of computations and the CR-SGCT performs many disk I/O operations (including some recomputations). The second trend is justified by stating that, unlike the spawning based recovery, the shrinking based recovery does not need to spawn the replacement processes and merge the intercommunicators. It is observed from Table 4.1 that the spawning and merging beyond 1,000 cores are expensive.

The C/C++ source codes for our implementation are available to conduct further research on these at https://github.com/mdmohsinali/SGCT-Based-Fault-Tolerant-Advection-Solver.

4.5 Summary

This chapter evaluates the effectiveness of ULFM MPI for the implementation of application level resiliency applied to the SGCT based 2D general advection solver. This includes detailed implementation guidelines for the detection and identification of process and node failures, and recovery of the faulty communicator by means of preserving and sacrificing its original size with the ULFM MPI semantics. Analysis reveals that the shrinking based recovery is less expensive than the spawning based

recovery, which sets a hope for making an application fault-tolerant from the application level. Comparing three data recovery techniques it is observed that the FT-SGCT performs cost-effective data recovery with only a small loss of accuracy.

The analysis of this chapter was limited to a simple benchmark written for the SGCT based 2D general advection solver. The next chapter will focus on how well ULFM MPI performs on some existing complex real-world applications. At the same time, it will focus on the evaluation of the effectiveness of applying the FT-SGCT on these applications.

Chapter 5

Fault-Tolerant SGCT with Applications

The previous chapter describes how the ULFM MPI semantics can be used to detect and identify process and node failures, and fix the faulty communicator as a support of creating the SGCT based robust 2D general advection solver. It also compares three data recovery techniques and observes that the alternate combination formula based application level data recovery is the most efficient technique compared to the other two for this application. However, there is a lack of practical examples demonstrating the range of issues encountered during the implementation of the application level resiliency on different types of complex and widely-used existing parallel applications. At the same time, the evaluation of how effective the SGCT is on these parallel applications is not adequate.

In order to fill some of these gaps, this chapter describes the integration of three existing complex parallel applications into the SGCT (or FT-SGCT), and evaluates their effectiveness. It also describes the process and node failure recovery overheads through ULFM MPI on these applications, and makes a comparison with traditional checkpointing technique.

The organization of this chapter is as follows. Section 5.1 describes a general methodology for the SGCT integration. Sections 5.2, 5.3, and 5.4 describe the existing GENE gyrokinetic plasma application, the Taxila Lattice Boltzmann Method (Taxila LBM) application, and the Solid Fuel Ignition (SFI) application, including how they are adapted to become fault-tolerant using a highly scalable SGCT algorithm, and their experimental evaluations, respectively. Failure recovery overheads for shorter and longer computations, overheads due to achieve robustness in the SGCT, and repeated failure recovery overheads are described in detail in Section 5.5.

This chapter describes work published jointly with others as "Complex Scientific Applications Made Fault-Tolerant with the Sparse Grid Combination Technique" [Ali et al., 2016]. Contents of Section 5.2 of this chapter are also published jointly with others as "A Fault-Tolerant Gyrokinetic Plasma Application using the Sparse Grid Combination Technique" [Ali et al., 2015].

```
1  W: global communicator;
2  G = {G_i}: set of sub-grids;
3  C = {C_i}: set of sub-grid communicators created from W;
4  T': number of time-steps;
5  n: number of combinations;
6  g = {g_i}: set of fields returned from the application computed on G;
7  u = {u_i}: corresponding set of sub-grid solutions used in equation (2.1);
8  u_I^c: combined solution of the SGCT;
9  for each C_i ∈ C do in parallel
10     u_i ← null; //makes runApplication() initialize g_i
11 for each of the n combinations do
12     for each C_i ∈ C do in parallel
13         g_i ← runApplication(u_i, G_i, C_i, T'/n);
14         u_i ← g_i; //on their common points
15         updateBoundary(u_i, C_i);
16     reconstructFaultyCommunicator(W); //using ULFM MPI (details in
       Chapters 4 and A)
17     u_I^c ← gather(u, W); //reconstructed grids do not participate
18     u ← scatter(u_I^c, W);
```

Algorithm 1: Main function of the modified application. Operations are assumed to be applied in parallel over all processes in the relevant communicator.

5.1 General Methodology for the SGCT Integration

Our underlying SGCT algorithm is implemented in C++ [Strazdins et al., 2015]. Applications implemented either in C++ or other languages require an integration with the SGCT to exchange the $\{u_{\underline{i}}\}$ of equation (2.1) between them. The implementation of this integration will depend on the language of the application and its complexity. It typically requires a minor modification of the application.

Algorithm 1 describes the main program, written in C++, calling into the application. A global communicator W is used to create a set of sub-grid communicators $\{C_{\underline{i}}\}$ to simultaneously run several (one for each $\underline{i} \in I$) application instances (line 12), with the application itself being called for the specified number of time-steps (i.e. T'/n) on sub-grid $G_{\underline{i}}$ on line 13. The first time this is called, $u_{\underline{i}}$ is `null`, and `runApplication()` uses the initial condition data to initialize $g_{\underline{i}}$; afterwards, it uses $u_{\underline{i}}$ instead. This computes a set of application fields $\{g_{\underline{i}}\}$. In the current implementation, the number of unknowns in dimensions over which the SGCT is applied must be a power of 2. This restriction could be removed in future implementation.

The set of sub-grid solutions $\{u_{\underline{i}}\}$, used in formula (2.1), will have an extra element padded out in the SGCT dimensions. On each process, the storage for $\{u_{\underline{i}}\}$ will also have room for halo elements. Common elements are copied over from $g_{\underline{i}}$ to $u_{\underline{i}}$ (line 14). Boundary updates and halo exchanges are also performed (line 15). The former are needed to 'pad out' $g_{\underline{i}}$, whose sizes are normally a power of 2, to $u_{\underline{i}}$, whose sizes must be a power of two plus one for the SGCT. The latter are needed because our SGCT algorithm uses interpolation [Strazdins et al., 2015].

Process or node failures may happen after line 9 in Algorithm 1 once the program starts executing. In order to tolerate these failures, the faulty communicator is reconstructed (line 16, details are in Chapter 4). Then the FT-SGCT is applied using the communicator W, with the combined solution u_I^c being used to re-initialize the sub-grid solutions $\{u_{\underline{i}}\}$ to facilitate further computations (lines 17–18).

Multiple combinations over the simulation may also improve the accuracy of the combined solution. Additionally, it may cause less data loss in the presence of failures. Since it combines component solutions several times, the effect of failures may be restricted to the subsequent combination.

It is necessary to exchange $\{u_{\underline{i}}\}$ between the application and the SGCT implementation through an interface in lines 13 and 18.

The main program uses the SGCT implementation to determine the parallel domain decomposition to be used by the application. $C_{\underline{i}}$ in line 13 is used to create $P_{\underline{i}}$, arranged on a logical d-dimensional process grid, to achieve this.

The main program creates a different directory for each application instance with a customized input parameter file containing the necessary values required for this instance.

MPI error handlers are attached to all MPI communicators and sub-communicators in the code. With a communicator or sub-communicator `comm`, the error handler function utilizes the `OMPI_Comm_failure_ack(comm, ...)` and `OMPI_Comm_failure_get-_acked(comm, ...)` functions provided by ULFM MPI for sending and receiving acknowledgments upon detecting process failures. These acknowledgments continue to be used throughout the process of respawning lost processes and reconstructing communicators.

5.2 Fault-Tolerant SGCT with the Gyrokinetic Plasma Application

In this section, we discuss in detail the technique of applying the FT-SGCT to the gyrokinetic plasma application (GENE), including experimental evaluations. An overview of the application is presented in Section 5.2.1. Section 5.2.2 describes how the SGCT algorithm is integrated into the higher-dimensional grids, and includes a detailed discussion of parallelization over non-SGCT dimensions. The modifications required for the application of GENE are discussed in Section 5.2.3. Finally, an experimental evaluation of the FT-SGCT for GENE is presented in Section 5.2.4.

5.2.1 Application Overview

The *Gyrokinetic Electromagnetic Numerical Experiment* (GENE) [Jenko and the GENE development team, 2014; Görler et al., 2011] is a plasma micro-turbulence application code. It contains a multi-dimensional solver of gyrokinetic equations for a field comprised of ion and electron densities defined on a fixed grid in a five-dimensional phase space (x, y, z, v, u). The physical space is toroidal (e.g. a tokamak) with a mag-

netic field whose lines move around the torus. x, y, and z represent the spatial coordinates in the direction radial, perpendicular, and parallel to the magnetic field. v and u are the velocities in the z and y dimensions, respectively. GENE imposes a minimal resolution of 16 grid points in the velocity dimensions.

This field is one of the main outputs of GENE; from it, GENE computes gyroradius-scale fluctuations and transport coefficients.

The code base [Jenko and the GENE development team, 2014] is written in FORTRAN 90 and utilizes hybrid MPI/OpenMP parallelization. High scalability to 10,000 cores has been reported [Görler et al., 2011]. Due to the relative smoothness of the solution, the SGCT has yielded good results in producing a relatively accurate solution on important problem sets [Kowitz et al., 2012].

Internally, GENE uses a complex precision array (`g_1`) representing the density of each particle (species) of interest in phase space. The number of species s is typically in the range $1 \leq s \leq 4$. As well as the dimensions above, s adds a sixth dimension to the array. GENE is capable of generating this field from initial conditions data or from reading `g_1` from a previously stored checkpoint (made by an MPI parallel I/O call).

All processes in a running GENE instance read the same parameter file `parameters`, which include things such as the sizes of each grid dimension, number of time-steps, maximum Δt for each time-step etc.

For performing the 'initial value' computation [Eriksson et al., 1996] of GENE, the main subroutine `rungene()` initializes the communicator for the simulation, reads parameters and checkpoint data (when applicable), sets the maximum time-step and calls the `initial_value()` function, which contains the time evolution loop. In each time-step, the electromagnetic fields are computed from `g_1`, and the gyrokinetic equations are applied to produce an update for `g_1` for that time-step.

5.2.2 Implementation of the SGCT Algorithm for Higher-Dimensional Grids

The field of GENE, regarded as a field of real numbers, can be thought of an array with a dimensionality of

$$D = (2, N_x, N_y, N_z, N_v, N_u, s) \,.$$

The first element in D arises from the field being complex (note that the SGCT uses only additions and multiplications with real coefficients).

Our implementation performs the SGCT across such a field as follows. The dimensions for the SGCT must be a contiguous sub-vector of D. Any remaining lower dimensions of D can be dealt with by an extension to the algorithm to operate on blocks of $b \geq 1$ real elements. Any remaining higher dimensions can be dealt with by applying the SGCT iteratively over these dimensions. If the above proves restrictive, the grid can be transposed to get the desired ordering in D.

For example, to perform a 2D combination on the N_v and N_u dimensions, we

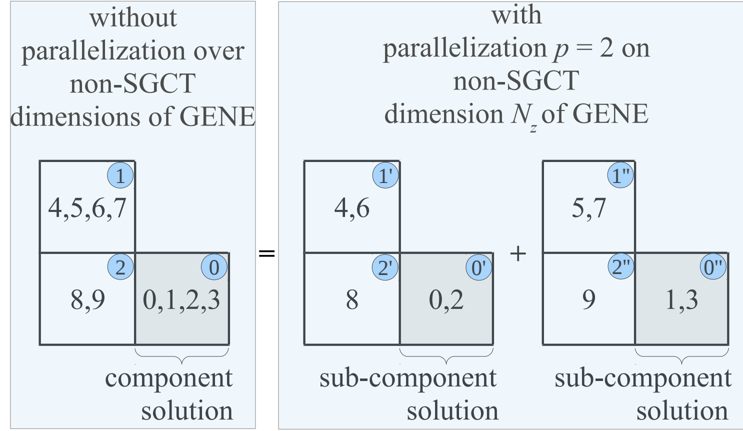


Figure 5.1: A demonstration of parallelization $p = 2$ on the non-SGCT dimension N_z of GENE for the 2D SGCT. The number inside the circle (without $'$ and $''$) denotes grid id for a sub-grid where component (or sub-grid) solution is available after the execution of GENE. For a grid x; x' and x'' are the grid ids of sub-grids holding partitioned sub-component solutions. The number outside the circle represents processes working on a particular sub-grid. In this case, instead of applying a single SGCT with sub-grids 0, 1, and 2, two SGCTs are applying in parallel: one with $0'$, $1'$, and $2'$ sub-grids, and the other with $0''$, $1''$, and $2''$ sub-grids.

use a block factor of $b = 2N_xN_yN_z$ and, if $s = 2$, iterate the SGCT over each of two array slices in the s dimension. GENE is generating the density for each species $\in s$ separately. They are stored into a single multi-dimensional array, where the higher dimension is species s. We need to apply the SGCT to each of the generated density, extracted as a slice from the multi-dimensional array, separately.

Our SGCT algorithm supports parallelization over arbitrary process grids in the SGCT dimensions. To support a parallelization to a total factor of $p \in \mathbb{N}$ in the non-SGCT dimensions, p independent SGCT computations can be performed in parallel. Non-SGCT dimensions could be any of the dimensions among the lower dimensions which are usually forming blocks. As for example for the 2D combination on the N_v and N_u dimensions, this could be any combination of N_x, N_y, N_z. In this case, the process grid $\tilde{P}_{\underline{i}}$ used to advance the simulation on sub-grid $G_{\underline{i}}$ will be split into p process sub-grids ($P_{\underline{i}}$), each of which is then passed to the appropriate instance of the SGCT computation. The combined grid's process sub-grid (P^c) used in each instance will be disjoint from the combined grid process sub-grids in other instances.

The implementation of this requires the careful construction of MPI communicators for each process sub-grid; the details of this are as follows.

Parallelization on the non-SGCT dimensions changes the original block definition and their placements. With a parallelization $p > 1$, an original block b is divided into p sub-blocks $b_0, b_1, \cdots, b_{p-1}$, each of size b/p, which are distributed across consecutive p processes $\tilde{P}'_0, \tilde{P}'_1, \cdots, \tilde{P}'_{p-1}$, respectively, of $\tilde{P}'(= \tilde{P}_{\underline{i}})$. In order to construct the MPI communicator for each process sub-grid to perform p independent SGCT

computations in parallel, $\tilde{P}'$ on sub-grid $G_{\underline{i}}$ is split into p process sub-grids $P'(= P_{\underline{i}})$ in such a way that each $P'_k|k = 0, 1, \cdots, p-1$ contains $\tilde{P}'_{j\cdot p+k}|j = 0, 1, \cdots, \frac{\tilde{P}'}{p} - 1$ processes. An example of this technique is shown in Figure 5.1.

5.2.3 Modifications to GENE for the SGCT

Some modifications to the GENE source were required to exchange $\{u_{\underline{i}}\}$ of equation (2.1) between the SGCT and GENE. On each process, $\{u_{\underline{i}}\}$ corresponds to the `g_1` array (a 5D complex density field of GENE).

Referring to the main program of Algorithm 1, after copying common elements from $g_{\underline{i}}$ to $u_{\underline{i}}$ at line 14, the padded elements in $u_{\underline{i}}$ are initialized to 0 for the velocity dimensions. For the spatial dimensions, boundary conditions are applied, including a halo exchange (line 13). For 3D configurations using the N_z dimension, shifts need also to be applied [Görler, 2009], as flux lines traverse the tokamak in a helical fashion. In this case we copy $g_{\underline{i}}$ into an existing GENE array `f_` (dimensioned to include all GENE boundaries) and call into the GENE code itself as follows:

```
call exchange_5df(f_)
call exchange_v(f_)
call exchange_mu(f_)
```

and then copy into $u_{\underline{i}}$ the corresponding elements in `f_`.

The main program uses the SGCT implementation to determine the parallel domain decomposition to be used by GENE. Furthermore, each GENE instance will have different global sizes for the dimensions used in the combination (e.g. for the 3D SGCT, these might be N_z, N_v and N_u). The main program also creates a different directory for each GENE instance with a customized `parameters` file containing the above values for this instance.

A standard tool is utilized for the interoperability between C and FORTRAN as an interface. We used the intrinsic module `ISO_C_BINDING` and the *language-binding-spec* attribute `BIND` for this interoperability. A C wrapper of the C++ code is used to hide the C++ code from the interoperation. A list of modifications that were made on GENE are as follows.

- The `runApplication()` function at line 13 of Algorithm 1 replaces the `main` subroutine of GENE. It calls into the top-level GENE (FORTRAN) subroutines as follows:

```
call check_for_scan(...)
call initialize_comm_scan(...)
call check_for_diagdir(...)
call rungene(...)
call erase_stop_file
call create_finished_file
call finalize_comm_scan(...).
```

One extra parameter is added to the `rungene()` subroutine to store the sub-grid communicator $C_{\underline{i}}$ to determine the process rank and communicator size with that parameter. The other subroutines are not modified.

- In the `initial_value()` subroutine, called by the `rungene()` subroutine, a C++ function call `c_get_g1()` is made with FORTRAN's `INTERFACE` block at the end of the time evolution loop. This passes `g_1` (and other associated data) from GENE to the SGCT to initialize each sub-grid solution $u_{\underline{i}}$.

- $u_{\underline{i}}$, passed to the `initial_value()` subroutine via calling the `rungene()` subroutine, is used to initialize `g_1` field before entering the time-loop. This is required for repeated combinations over time. For a single combination, this is not used (passed as `null`). GENE initializes `g_1` by itself.

5.2.4 Experimental Results

In this section, the experimental setup, an analysis of the execution performance, and the memory consumption of both the FT-SGCT and equivalent full grid computations for GENE are presented. Following this, we discuss the approximation errors of the solutions computed with FT-SGCT for GENE.

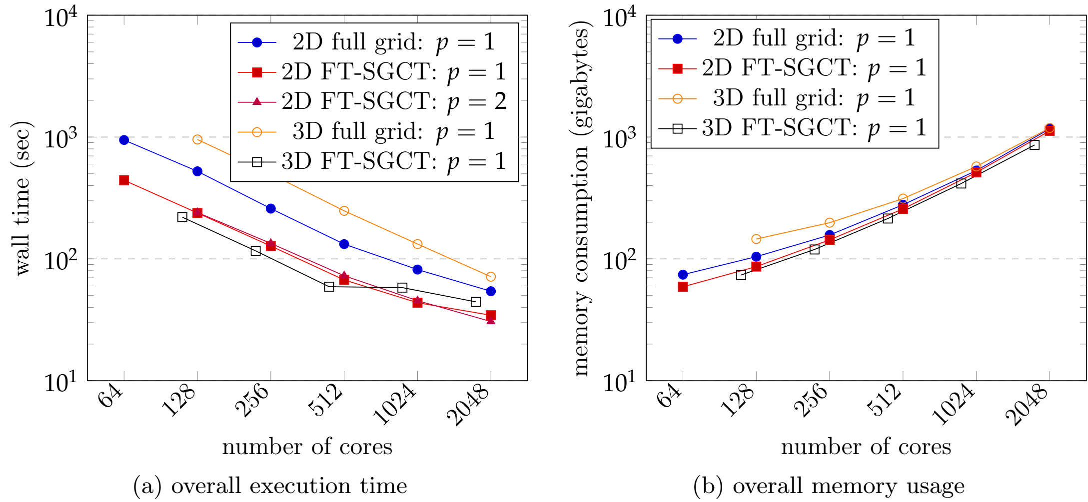


Figure 5.2: Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the GENE application. *2d_big_6* and *3d_big_6* inputs are used for the 2D and 3D FT-SGCT computations, respectively, which are different. The results shown are an average of five experiments. p represents parallelization on the non-SGCT dimension (N_z for 2D, N_y for 3D).

# cores	GENE simulation (sec)	combination algorithm (sec)
64	441.01	0.373
128	236.98	0.252
256	126.46	0.151
512	66.30	0.115
1024	43.10	0.092
2048	33.93	0.076

Table 5.1: Execution time breakdown of the 2D FT-SGCT with parallelization $p = 1$ and level $l = 5$ for the GENE application presented in Figure 5.2a.

5.2.4.1 Experimental Setup

Experiments were conducted based on a problem from the GENE testsuite as shown in Table B.1. One is called *2d_big_6*, with a full grid size $(N_v, N_u) = (2^8, 2^8)$ for the 2D FT-SGCT, $N_x = 64, N_y = 4, N_z = 16$, and the level of the SGCT is $l = 5$. For a 2D FT-SGCT for $l = 5$ with power of two sizes for the component's process grids, it turns out that the total number of processes is also a power of two, permitting a head-to-head comparison with the full grid results. The other is called *3d_big_6*, with a full grid size $(N_z, N_v, N_u) = (2^6, 2^8, 2^8)$ for the 3D FT-SGCT, $N_x = 32, N_y = 4$ and the level of the FT-SGCT is $l = 4$. For both cases we set the number of species s to be 1, the time-steps to 100, the maximum $\Delta t = 10^{-3}$, and the grid type for the N_u dimension to be 'equidist' (by setting "mu_grid_type = 'equidist"' as shown in Table B.1). Number of processes in various directions were appropriately set from the main program of Algorithm 1.

The selection of levels $l = 5$ and $l = 4$ (for the 2D and 3D versions respectively) is one of the many possible examples. The execution time, concurrency, and accuracy of the combined solution will be varied with the change of level.

5.2.4.2 Execution Time and Memory Usage

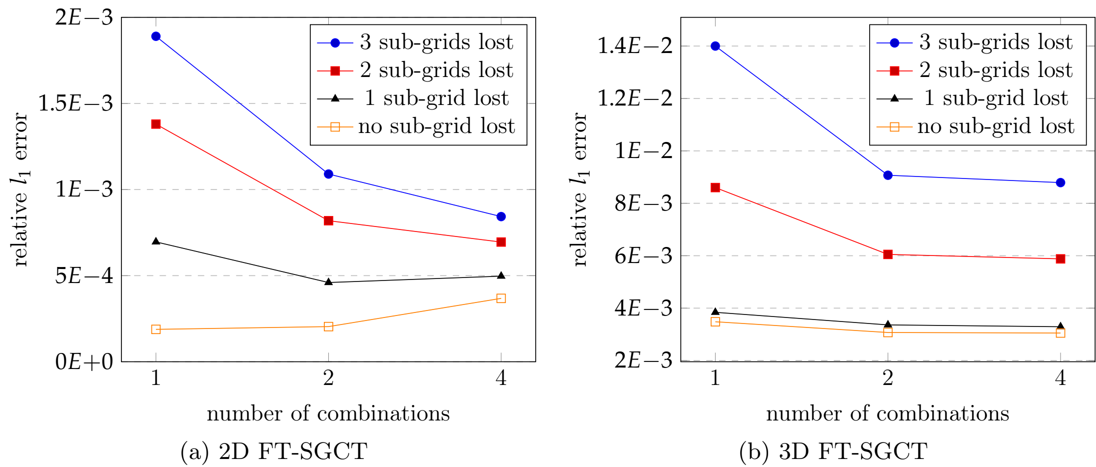
Figure 5.2a gives execution performance of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computations for GENE. It is observed that the 2D FT-SGCT is approximately 2× faster than the equivalent full grid computation. The difference in this speed reflects the reduced amount of work enabled by the FT-SGCT compared to the full grid, which for level $l = 5$ on the 2D case is approximately one half. Execution time of the 3D FT-SGCT shows a clear advantage in terms of reduced amount of work, which for $l = 4$ in this case is approximately one quarter. This causes the 3D FT-SGCT to be approximately 4× faster than the equivalent full grid computation.

Figure 5.2a also provides evidence that our FT-SGCT implementation supports

parallelization over non-SGCT dimensions for GENE. For the 2D case, we choose a parallelization $p = 2$ on the non-SGCT dimension N_z, where N_v and N_u hold process sub-grid for the FT-SGCT. Each process sub-grid experiences a reduction of process count by a factor of p on one of the FT-SGCT dimensions. This occurs on either the N_v or N_u dimension, or both. For $p = 2$, the larger dimension is reduced (or, N_u, when they are the same). It is observed that this process grid adjustment still shows a similar execution time.

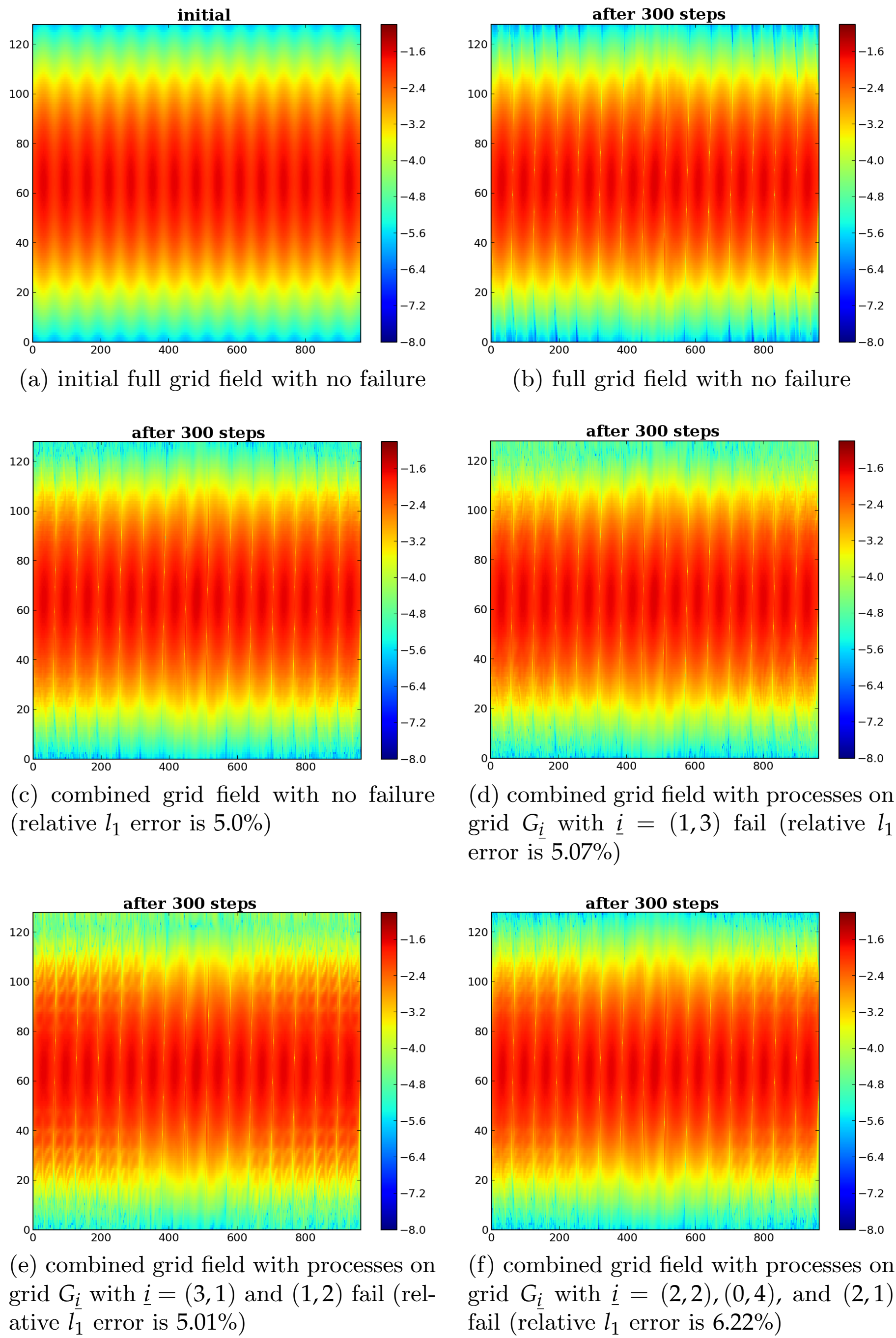
Table 5.1 shows the overall performance of the 2D FT-SGCT for the GENE application in isolation. It is observed that performing the combination is not costly. It should be noted however these times are for when the SGCT algorithm has been called before the application starts to exclude the Open MPI warm-up time. As explained in [Strazdins et al., 2016b], the cost of the first call to the SGCT can be much greater, due to the fact that Open MPI is required to setup new connections between processes operating on separate sub-grids. For a detailed analysis and evaluation of the combination algorithm, see [Strazdins et al., 2016b, 2015].

Figure 5.2b gives a comparison of memory usage of the 2D and 3D FT-SGCT with the equivalent full grid computation for GENE. It is observed that the memory requirement of the FT-SGCT is roughly the same as that of the equivalent full grid for the relatively small SGCT level used. A bigger improvement in memory usage



(a) 2D FT-SGCT (b) 3D FT-SGCT

Figure 5.3: Approximation errors of the FT-SGCT based GENE application. *2d_big_6* and *3d_big_6* inputs are used for the 2D and 3D FT-SGCT computations, respectively, which are different. The results shown are an average of a number of experiments for the loss of one sub-grid, two sub-grids, and three sub-grids such that the occurrence of faults maintains an uniform distribution over all computing nodes. In order to do so, we injected 62% of failures on the first diagonal sub-grids (on larger sub-grids, except sub-grid 0 or $G_{4,0}$ on level $l = 5$ containing process 0), 25% on the second diagonal sub-grids, 10% on the third diagonal sub-grids, and 3% on the fourth diagonal sub-grids (on smaller sub-grids) for the 2D FT-SGCT. For the 3D FT-SGCT, these are 72%, 22%, 5%, and 1%, respectively. However, for the non-failure case, only one experiment is done.



(a) initial full grid field with no failure

(b) full grid field with no failure

(c) combined grid field with no failure (relative l_1 error is 5.0%)

(d) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (1,3)$ fail (relative l_1 error is 5.07%)

(e) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (3,1)$ and $(1,2)$ fail (relative l_1 error is 5.01%)

(f) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (2,2), (0,4)$, and $(2,1)$ fail (relative l_1 error is 6.22%)

Figure 5.4: A comparison of the 2D full grid and level $l = 5$ combined grid solutions with $(N_v, N_u) = (2^7, 2^7)$, $N_x = 16$, $N_y = 1$, $N_z = 64$, and species $s = 1$ of standard test 1 (with `standard/parameters_1` parameter file), and a single combination for the GENE application. y-axis and x-axis of each plot represents 'velocity parallel to field' and '(radial dimension) $\times$ (parallel to field dimension)', respectively.

would be expected for larger levels.

5.2.4.3 Approximation Errors

Figure 5.3 shows the approximation errors of the 2D and 3D FT-SGCT with varying number of combinations and varying number of lost sub-grids for GENE. It is observed that the error is acceptably low for both the 2D and 3D cases. Moreover, multiple combinations clearly show the advantages with respect to accuracy, but with an exception of the 2D FT-SGCT on the non-failure case. Previous studies on theoretical error bounds and numerical results for advection show an advantage to a certain number of combinations [Lastdrager et al., 2001], which one might expect could also be realized for GENE. However, why the accuracy of the 2D FT-SGCT for GENE on the non-failure case is not improving for multiple combinations is under investigation.

Furthermore, Figure 5.4 shows some plots of the combined and equivalent full grid solution densities in the absence or presence of faults for GENE. It demonstrates how much a field deteriorates due to approximation error. It is observed that even with approximately 6% relative l_1 error of the combined field there is no significant difference between the combined and equivalent full grid fields.

The C/C++ and FORTRAN source codes for our implementation are available at https://github.com/mdmohsinali/SGCT-Based-Fault-Tolerant-GENE.

5.3 Fault-Tolerant SGCT with the Lattice Boltzmann Method Application

In this section, we discuss the Taxila Lattice Boltzmann Method (Taxila LBM) application, the modifications needed on Taxila LBM to apply the SGCT, and the experimental evaluation of the FT-SGCT for Taxila LBM.

5.3.1 Application Overview

The *Lattice Boltzmann Method* (LBM) is used to solve the discrete Boltzmann equation. The equation is used to evolve the general form of a distribution function for a lattice

$$f^k(\mathbf{x} + \mathbf{e}_i\Delta t, \mathbf{e}_i, t + \Delta t) - f^k(\mathbf{x}, \mathbf{e}_i, t) = \Omega^k_{\text{coll}} + \Omega^k_{\text{forces}},$$

where $\mathbf{x}$ represents the sites of a discrete lattice on which particles at time t moving with fixed velocities $\mathbf{e}_i$ from site to site, i is the index of n_i available velocity directions including the stationary ($i = 0$), k represents the index of n_k components, and Ω_{coll} and Ω_{forces} are the momentum changes in the particle distribution caused by collisions and other forces, respectively.

The popularity of LBM for the pore-scale simulation is increasing due to its capability of including complex geometries without putting more effort. Other benefits

include multiple relaxation times, increased isotropy, improved accuracy, and physical fidelity of the method. The explicit type of algorithm and comparatively huge local task provide a benefit of achieving high computational efficiency [Coon et al., 2014].

Taxila LBM [Coon et al., 2014; Porter et al., 2012] is an open-source software framework of the LBM for simulation of flow in porous and geometrically complex media. It is based upon the Shan-Chen model [Shan and Chen, 1993]. This framework provides some excellent features: (1) It is shown that the implementation is scalable to tens of thousands of cores on Jaguar/Titan. (2) Its FORTRAN 90 based PETSc [Balay et al., 2014] modular implementation is easily extendable. (3) It provides the flexibility of solving D2Q9 (2-dimensional, 9 lattice velocities), D3Q19 (3-dimensional, 19 lattice velocities), and other mesh dependencies, on the 2D or 3D grids. (4) It supports arbitrary, heterogeneous boundary conditions, and multiple mineral/wall materials. (5) It can operate on multi-phase systems with different phase viscosities and/or molecular masses. (6) It provides the flexibility to include higher order derivatives or multiple relaxation times to improve the stability at large viscosity ratios.

In this thesis, we chose an example application available under `tests/bubble_3D` of [Tax, 2015]. This example is called a *bubble test*, in which two partially miscible fluids are initialized in contact with each other. Due to surface tension, the fluids equilibrate: one fluid forms a spherical bubble inside the other with a nonzero thickness interface between the two fluids. The pressure difference between the fluids measures surface tension, while the thickness of the interface measures miscibility. In typical applications, these tests are a calibration step to ensure model parameters result in physical properties consistent with the fluids to be modeled, such as supercritical CO_2 and water, or air and water.

By default, the bubble test provides a distribution function field as output. It is also possible to extract either the density, total velocity, total density, or pressure fields. In our experiment, we choose the density field as output.

5.3.2 Modifications to Taxila LBM for the SGCT

Some modifications to the Taxila LBM source were required to exchange $\{u_{\underline{i}}\}$ of equation (2.1) between the SGCT and Taxila LBM. On each process, $\{u_{\underline{i}}\}$ corresponds to the `rho` array (either 2D or 3D) of Taxila LBM.

An interface is created to communicate between C++ and PETSc (FORTRAN 90 code). A standard tool is used for the interoperability between C and FORTRAN. A C wrapper of the C++ code is used to hide the C++ code from the implementation. An intrinsic module `ISO_C_BINDING` and the *language-binding-spec* `BIND` is used for the interoperability.

In the original Taxila LBM implementation, the bubble format is fixed. We made it adaptive based on the number of grid points in each dimension. For each grid, we define the suspending bubble as a square/rectangle of length a one-half of the number of grid points in each dimension, and placed it at the center of the corresponding

grid.

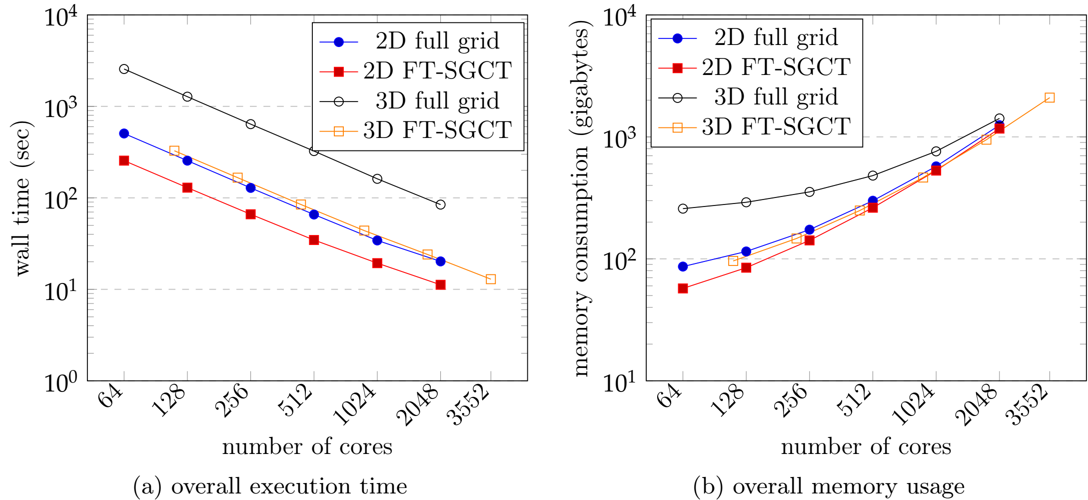
The default global communicators in the `LBMCreate()` and subsequent subroutines are replaced by the sub-grid communicator $C_{\underline{i}}$ passed from the main program of Algorithm 1. Similarly, process grid configuration, generated from $C_{\underline{i}}$, and sub-grid $G_{\underline{i}}$ configuration are also passed as parameters onto these subroutines to make these configurations consistent on both the SGCT and Taxila LBM sides.

The original Taxila LBM requires a single parameter file `input_data` to initialize the application, but we need $|I|$ versions of the customized `input_data` files with one for each sub-grid $G_{\underline{i}}$ to initialize the application on each sub-grid $G_{\underline{i}}$. These are created and placed in separate directories on-the-fly.

Since density is the field of interest to which the SGCT is applied, we extract the local `rho` field after finishing the execution of the `LBMRun()` subroutine. A local pointer `lbm%flow%distribution%rho_a` is used to achieve this. Then a C++ function call `c_get_rho()` is made with FORTRAN's `INTERFACE` block to pass this pointer (and the other associated data) from Taxila LBM to the SGCT to initialize each sub-grid solution $u_{\underline{i}}$. After the initialization achieved by copying the common elements from $g_{\underline{i}}$ to $u_{\underline{i}}$ (line 12 of Algorithm 1), periodic boundary conditions are applied for all dimensions (line 13) (including a halo exchange operation). A block factor of $b = 1$ is used here.

Process or node failures are handled by the similar way as it is handled for the GENE application.

In order to facilitate multiple combinations, $u_{\underline{i}}$ is passed into the `LBMCreate()`



(a) overall execution time

(b) overall memory usage

Figure 5.5: Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the Taxila LBM application. 2D FT-SGCT computers $2^{13} \times 2^{13}$ grid points with level $l = 5$. 3D FT-SGCT computes $2^9 \times 2^9 \times 2^9$ grid points with level $l = 4$. The results shown are an average of five experiments (200 time-steps each).

and subsequent subroutines to initialize `rho`, before the execution of the `LBMRun()` subroutine. For a single combination, this is not used; Taxila LBM initializes `rho` by itself.

5.3.3 Experimental Results

In this section, the experimental setup, an analysis of the execution performance, and memory consumption of both the FT-SGCT and equivalent full grid computations for Taxila LBM are presented. Following this, we discuss the approximation errors of the solutions computed with the FT-SGCT for Taxila LBM.

5.3.3.1 Experimental Setup

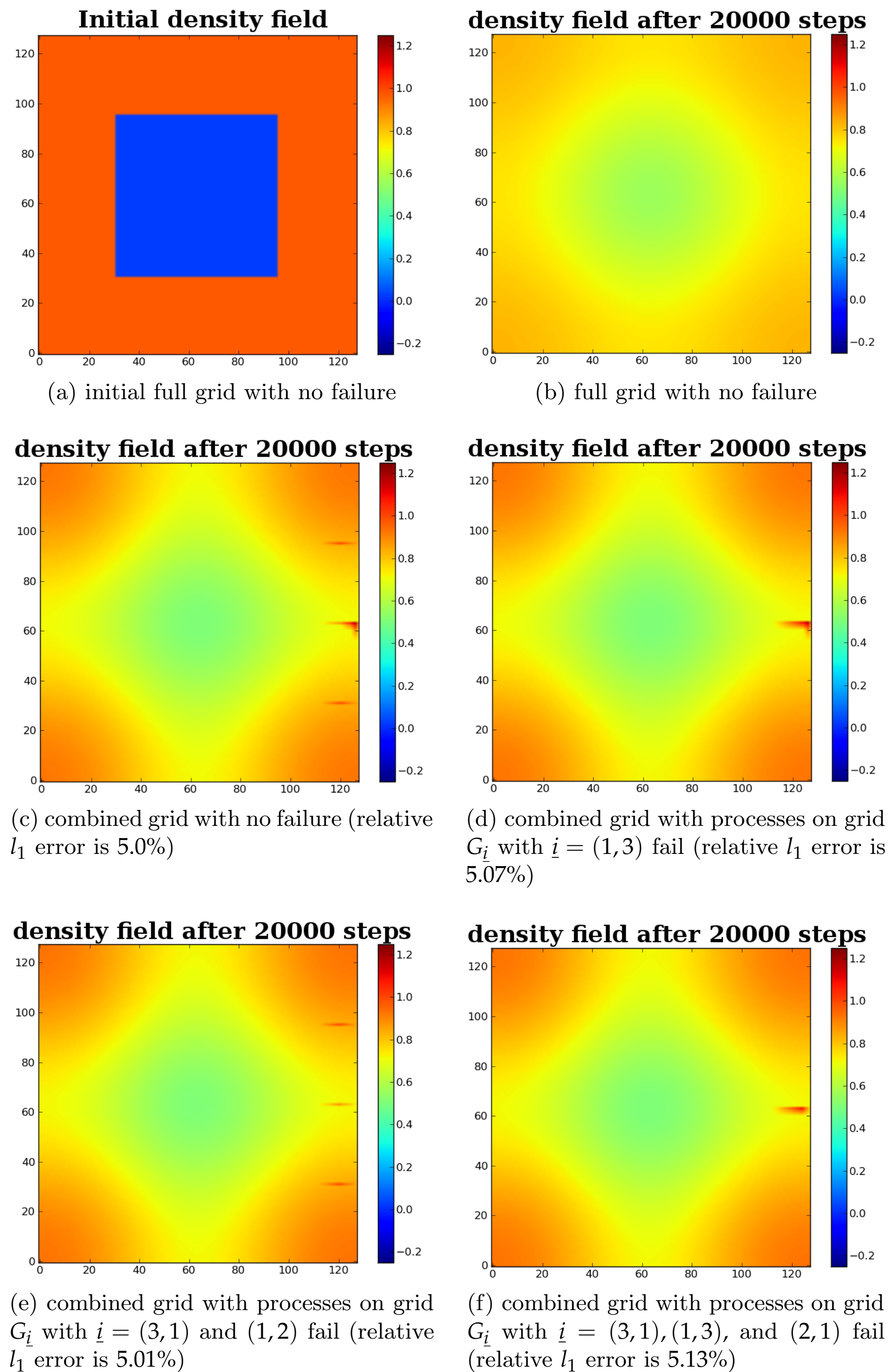
Experiments were conducted with parameters in `input_data` file as shown in Table B.2. The time-steps of the experiments were set by `-npasses <VALUE>`. Full grid dimensions for the 2D and 3D cases were `(-NX, -NY) =` $(2^{13}, 2^{13})$ with level $l = 5$ and `(-NX, -NY, -NZ) =` $(2^9, 2^9, 2^9)$ with level $l = 4$, respectively. The number of processes in the x, y, and z directions (`-da_processors_x, -da_processors_y, -da_processors_z`) for the 3D case (and for the 2D case with no z-dimensional value) were appropriately set from Algorithm 1.

5.3.3.2 Execution Time and Memory Usage

Figure 5.5a shows the execution performance of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computations for Taxila LBM. It is observed that the 2D FT-SGCT is approximately 2× faster than the equivalent full grid computation due to comparatively reduced amount of work in the FT-SGCT. It is also observed that the 3D FT-SGCT is approximately 4× faster than the equivalent full grid computation due to computing less work than the 2D case.

# cores	Taxila LBM simulation (sec)	combination algorithm (sec)
64	254.68	$6.3E^{-2}$
128	128.28	$3.8E^{-2}$
256	64.90	$2.4E^{-2}$
512	32.99	$1.0E^{-2}$
1024	17.66	$5.5E^{-3}$
2048	9.48	$3.1E^{-3}$

Table 5.2: Execution time breakdown of the 2D FT-SGCT with level $l = 5$ for the Taxila LBM application presented in Figure 5.5a.



(a) initial full grid with no failure

(b) full grid with no failure

(c) combined grid with no failure (relative l_1 error is 5.0%)

(d) combined grid with processes on grid $G_{\underline{i}}$ with $\underline{i} = (1,3)$ fail (relative l_1 error is 5.07%)

(e) combined grid with processes on grid $G_{\underline{i}}$ with $\underline{i} = (3,1)$ and $(1,2)$ fail (relative l_1 error is 5.01%)

(f) combined grid with processes on grid $G_{\underline{i}}$ with $\underline{i} = (3,1),(1,3)$, and $(2,1)$ fail (relative l_1 error is 5.13%)

Figure 5.6: A comparison of the 2D full grid and level $l = 5$ combined grid solutions with $2^7 \times 2^7$ grid points and a single combination for the Taxila LBM application. Since simulation on larger grid for 20000 time-steps is very expensive, we choose this smaller grid setting for the results of this figure only. The fields shown are for the first component out of two components.

Table 5.2 shows the overall performance of the 2D FT-SGCT for the Taxila LBM application in isolation. It is observed that performing the combination is again of little relative cost. It should be noted however these times are again for when the SGCT algorithm has been called before the application starts to exclude the Open MPI warm-up time.

Figure 5.5b shows the amount of memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computations for Taxila LBM. It is observed that the memory requirement of the FT-SGCT is roughly the same as that of the equivalent full grid for the relatively small SGCT level used. A bigger improvement in memory usage would be expected for larger levels.

5.3.3.3 Approximation Errors

The computed relative l_1 approximation errors of the 2D and 3D FT-SGCT in the absence of faults for Taxila LBM with a single combination are $1.13E^{-2}$ and $3.98E^{-2}$, respectively, which seems acceptably low. For multiple combinations, it is observed that the accuracy of the combined solution is not improved.

Furthermore, Figure 5.6 shows some plots of the combined and equivalent full grid solution densities in the absence or presence of faults for Taxila LBM. It demonstrates how much a field deteriorates in the presence of faults. It is observed that even with approximately 5% approximation error there is no significant difference between the combined and equivalent full grid fields. It is also observed that there are some unexpected red spikes on the combined grid fields. This is due to an implementation issue of the FT-SGCT, and should be fixed.

The C/C++ and PETSc (FORTRAN 90) source codes for our implementation are available at https://github.com/mdmohsinali/SGCT-Based-Fault-Tolerant-Taxila-LBM.

5.4 Fault-Tolerant SGCT with the Solid Fuel Ignition Application

In this section, we discuss the Solid Fuel Ignition (SFI) application, the modifications needed on SFI to apply the SGCT, and the experimental evaluation of the FT-SGCT for SFI.

5.4.1 Application Overview

The Bratu problem in three-dimensional coordinates is defined by the equation

$$-\Delta u(x,y,z) - \lambda \exp^{u(x,y,z)} = 0, 0 < x,y,z < 1,$$

where Δ is the Laplace operator and λ (Bratu parameter) defines the magnitude of the nonlinearity. The boundary conditions are $u(x,y,z) = 0$ for $x = 0$, $x = 1$, $y = 0$,

$y = 1, z = 0, z = 1$. It is used in *Solid Fuel Ignition* (SFI) models [Bebernes and Eberly, 1989], heat transfer via radiation, nano-technology, cosmology, and so on.

In this thesis, we chose an application solving the Bratu problem for the modeling of SFI (or combustion) with $\lambda = 6.0$ $(0 \leq \lambda \leq 6.81)$. The application is an example code of PETSc [Balay et al., 2014] demonstrating the nonlinear *SNES* solver. It uses distributed arrays (`DMDAs`) to partition the parallel grid. A finite difference approximation with the usual 7-point stencil for 3D (5-point for 2D) is used to discretize the boundary value problem to obtain a nonlinear system of equations. The 3D and 2D versions of the code are available in [sfi, 2015b] and [sfi, 2015a], respectively.

This targeted application is not as complex compared to the previous applications. However, evaluation of this application will provide us a general idea about the evaluation of large complex applications modeling SFI.

5.4.2 Modifications to SFI for the SGCT

Some modifications to the SFI source were required to exchange $\{u_i\}$ of equation (2.1) between the SGCT and SFI. On each process, $\{u_i\}$ corresponds to vector `x` of the `SNESSolve()` function (either 2D or 3D) of SFI.

The PETSc code base of the 2D SFI is in FORTRAN 90. A standard tool is used for the interface between C and FORTRAN. A C wrapper of the C++ code is used to hide the C++ code from the implementation. An intrinsic module `ISO_C_BINDING` and the *language-binding-spec* `BIND` is used for the interoperability. For the 3D SFI, the PETSc code base is in C. Thus, no special interoperability is required.

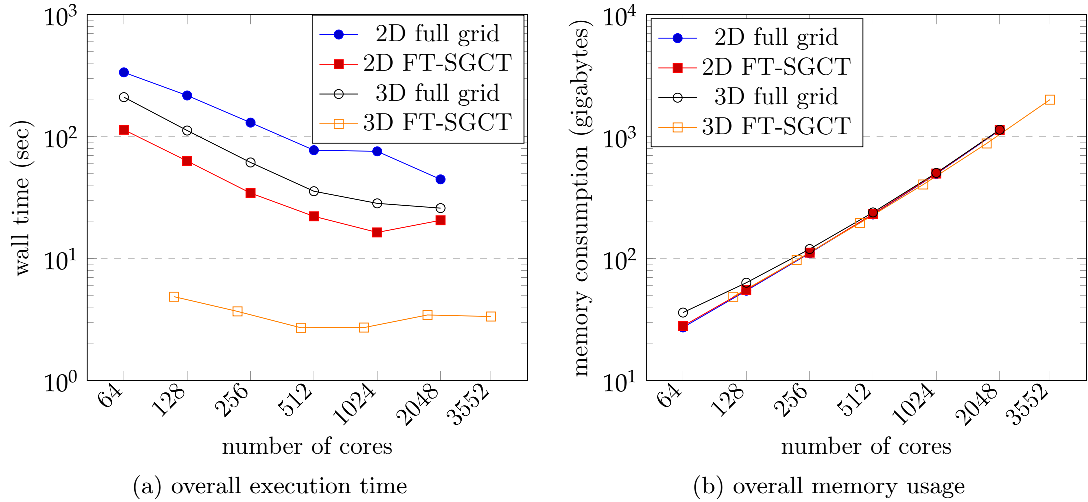


Figure 5.7: Overall execution time and memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computation with a single combination when there is no fault throughout the computation for the SFI application. 2D FT-SGCT computers $2^{11} \times 2^{11}$ grid points with level $l = 5$. 3D FT-SGCT computes $2^8 \times 2^8 \times 2^8$ grid points with level $l = 4$. The results shown are an average of five experiments.

The default global communicators in the `SNESCreate()` and `DMDACreate2d()` (or `DMDACreate3d()` for 3D) functions are replaced by the sub-grid communicator $C_{\underline{i}}$ passed from the main program of Algorithm 1. Similarly, process grid configuration, generated from $C_{\underline{i}}$, and sub-grid $G_{\underline{i}}$ configuration are also passed as parameters onto the `DMDACreate2d()` (or `DMDACreate3d()` for 3D) function to make these configurations consistent on both the SGCT and SFI sides.

The `DMDAVecGetArrayF90()` and `DMDAVecRestoreArrayF90()` (or the `DMDAVec-GetArray()` and `DMDAVecRestoreArray()`, respectively, for the C version) functions are used to access the solution vector `x` after the execution of the `SNESSolve()` function. Then the `c_get_sfi_field()` function is called to pass the solution vector `x` to the SGCT to initialize each sub-grid solution $u_{\underline{i}}$. After the initialization achieved by copying the common elements from $g_{\underline{i}}$ to $u_{\underline{i}}$ (line 12 of Algorithm 1), periodic boundary conditions are applied for all dimensions (line 13) (including a halo exchange operation). Similar to Taxila LBM, a block factor of $b = 1$ is used here.

Process or node failures are handled by the similar way as it is handled for the GENE application.

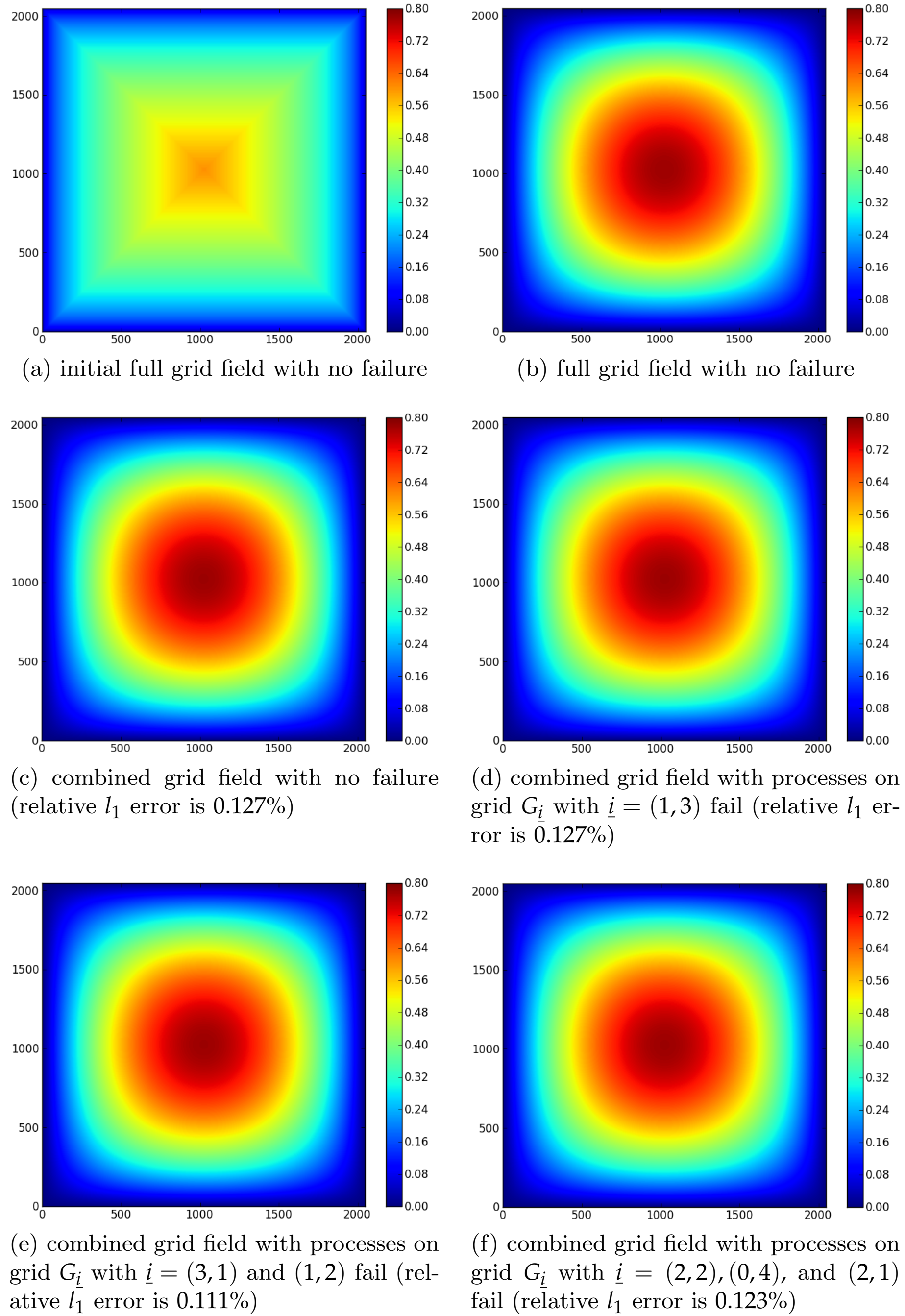
Multiple combinations are achieved by passing $u_{\underline{i}}$ into SFI to initialize `x` vector before the execution of the `SNESSolve()` function. For a single combination, this is not used; SFI initializes `x` by itself.

5.4.3 Experimental Results

In this section, the experimental setup, an analysis of the execution performance, and memory consumption of both the FT-SGCT and equivalent full grid computations for SFI are presented. Following this, we discuss the approximation errors of the solutions computed with the FT-SGCT for SFI.

# cores	SFI simulation (sec)	combination algorithm (sec)
64	113.40	$5.73E^{-3}$
128	62.11	$4.67E^{-3}$
256	33.28	$3.58E^{-3}$
512	20.66	$2.69E^{-3}$
1024	14.79	$1.84E^{-3}$
2048	18.87	$1.24E^{-3}$

Table 5.3: Execution time breakdown of the 2D FT-SGCT with level $l = 5$ for the SFI application presented in Figure 5.7a.



(a) initial full grid field with no failure

(b) full grid field with no failure

(c) combined grid field with no failure (relative l_1 error is 0.127%)

(d) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (1, 3)$ fail (relative l_1 error is 0.127%)

(e) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (3, 1)$ and $(1, 2)$ fail (relative l_1 error is 0.111%)

(f) combined grid field with processes on grid $G_{\underline{i}}$ with $\underline{i} = (2, 2), (0, 4)$, and $(2, 1)$ fail (relative l_1 error is 0.123%)

Figure 5.8: A comparison of the 2D full grid and level $l = 5$ combined grid solutions with $2^{11} \times 2^{11}$ grid points and a single combination for the SFI application.

5.4.3.1 Experimental Setup

Experiments were conducted with Bratu parameter $\lambda = 6.0$ (`-par 6.0`) and Jacobian finite difference approximation (`-snes_fd`). Full grid dimensions for the 2D and 3D cases were $2^{11} \times 2^{11}$ with level $l = 5$ and $2^8 \times 2^8 \times 2^8$ with level $l = 4$, respectively. The number of processes in the x, y, and z directions (`Nx, Ny, Nz`) for the 3D case (and for the 2D case with no z-dimensional value) were appropriately set from Algorithm 1.

5.4.3.2 Execution Time and Memory Usage

Figure 5.7a shows the execution performance of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computations for SFI. It is observed that the 2D FT-SGCT is approximately $3\times$ faster than the equivalent full grid computation due to a reduced amount of work required for the FT-SGCT. It is also observed that the 3D FT-SGCT is approximately $9\times$ faster than the equivalent full grid computation due to reduced computational work compared to the 2D case.

Table 5.3 shows the overall performance of the 2D FT-SGCT for the SFI application in isolation. It is observed that the overhead of performing the combination is very low. It should be noted however these times are for when the SGCT algorithm has been called before the application starts to exclude the Open MPI warm-up time.

Figure 5.7b shows the amount of memory usage of the 2D FT-SGCT, 3D FT-SGCT, and the equivalent full grid computations for SFI. It is observed that the memory requirement of the FT-SGCT is roughly the same as that of the equivalent full grid for the relatively small SGCT level used. A bigger improvement in memory usage would be expected for larger levels.

5.4.3.3 Approximation Errors

The computed relative l_1 approximation errors of the 2D and 3D FT-SGCT in the absence of faults for SFI with a single combination are $1.27E^{-3}$ and $1.28E^{-3}$, respectively, which seems acceptably low. Indeed, these low error rates make SFI a highly suitable application for the SGCT. For multiple combinations, it is observed that the accuracy of the combined solution is not improved.

Furthermore, Figure 5.8 shows some plots of the combined and equivalent full grid solution densities in the absence or presence of faults for SFI. It demonstrates how much a field deteriorates in the presence of faults. It is observed that with approximately 0.125% approximation error there is no significant difference between the combined and equivalent full grid fields.

The C/C++ and PETSc (FORTRAN 90) source codes for our 2D implementation are available at https://github.com/mdmohsinali/SGCT-Based-Fault-Tolerant-2D-SFI.

The C/C++ and PETSc (C) source codes for our 3D implementation are available at https://github.com/mdmohsinali/SGCT-Based-Fault-Tolerant-3D-SFI.

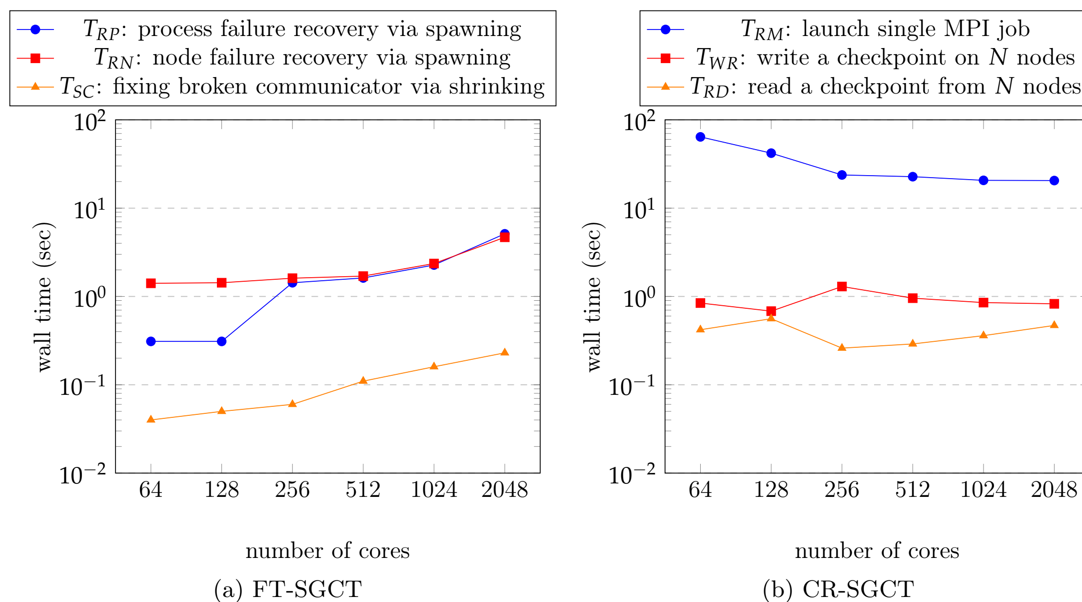


Figure 5.9: Recovery overhead of a single occurrence of failures for shorter computation for GENE. The results shown are an average of five experiments for *2d_big_6* input applied to the 2D SGCT for GENE.

5.5 Failure Recovery Overheads

In this section, we analyze the recovery overheads of the FT-SGCT (alternate combination formula based fault-tolerant SGCT) and CR-SGCT (applying Checkpoint/Restart technique to the non-fault-tolerant SGCT) for GENE. Then we analyze the recovery overheads of the FT-SGCT in terms of computing extra unknowns. Finally, we analyze repeated failure recovery overheads for GENE.

The failure recovery overheads for the other two applications (Taxila LBM and SFI) are almost the same as GENE. So, they are not presented separately.

5.5.1 Recovery Overheads for Shorter Computations

Figure 5.9 shows the component timings that are used to estimate the recovery overheads of the two approaches. The first component timing is related to the implementation of the FT-SGCT, which uses ULFM MPI and the algorithm-based recovery (in terms of applying alternate combination formula) on the SGCT to recover from failures. The second component timing is related to the implementation of the CR-SGCT, which uses a Checkpoint/Restart based recovery on the SGCT to recover from failures. The components are generated using *2d_big_6* input of GENE, and will be used for measuring the overheads of both the shorter and longer computations.

The notations in this figure are as follows:

- $T_{RP} := T_{RP}(N)$ is the time taken to recover from process failures and recon-

struct the broken communicators on N nodes using ULFM MPI (including acknowledgments performed in the alive processes, and spawning the replacement processes on the same node) for a single occurrence of faults.

- $T_{RN} := T_{RN}(N)$ is the time taken to recover the failed nodes from failures and reconstruct the broken communicators on N nodes using ULFM MPI (including acknowledgments performed in the alive processes, and spawning the replacement processes on a spare node) for a single occurrence of faults.
- $T_{SC} := T_{SC}(N)$ is the time taken to recover the broken communicators on N nodes via shrinking the communicators.
- $T_{WR} := T_{WR}(N)$ is the time required to write a global checkpoint on N nodes.
- $T_{RD} := T_{RD}(N)$ is the time required to read a checkpoint from N nodes.
- T_{RM} is a single MPI launch time (when restarts from a checkpoint after failure), which can be calculated by

$$T_{RM} = (t_1 + t_2) - (t_3 + T_{RD}), \tag{5.1}$$

  where

  - t_1 is the system time of running the CR-SGCT for 50 time-steps with global checkpoint write at the end (no checkpoint read),
  - t_2 is the system time of running the CR-SGCT for 50 time-steps after initializing from checkpoint (written on previous step) and no global checkpoint write,
  - t_3 is the system time of running the CR-SGCT for 100 time-steps with global checkpoint write at the end (no checkpoint read).

Note that checkpoint write and read in GENE could be enabled by "write_checkpoint = T" and "read_checkpoint = T", respectively, as parameters along with others as shown in Table B.1. Similarly, these could be disabled by replacing 'T' with 'F'.

Based on the process times, it is possible to estimate the recovery overheads for the shorter computations (since shorter computation saves CPU hours). The overheads of the FT-SGCT of a one-off process and node failures are T_{RP} and T_{RN}, respectively. With the CR-SGCT, the overhead is the sum of T_{WR}, T_{RM}, and T_{RD} for a single occurrence of failures. This excludes the overhead of backtrack time. It is observed that the one-off failure recovery overhead of the CR-SGCT is approximately $4\times$ larger than the spawning based recovery overhead of the FT-SGCT (both for process and node failures). With the shrinking based recovery, overhead of the FT-SGCT is approximately $90\times$ less than that of the CR-SGCT. We expect this gap to increase in future mature ULFM MPI releases.

It should be noted that we would expect that $T_{RP}, T_{RN} \ll T_{RM}$ for a mature MPI implementation, as recovering a failed process involves inherently less work than relaunching the job from scratch, and that we expect the gap to increase in future ULFM MPI releases.

5.5.2 Recovery Time Analysis for Longer Computations

Using results gathered so far, we will estimate the overhead of our implementation for longer computations and different frequencies of faults. We will compare this estimate with the time of the CR-SGCT taking into account the overhead generated by having to backtrack to the previous checkpoint when failures occur. It is assumed that the occurrence of faults is independent and identically distributed on each compute node. Further, it is assumed that faults are exponentially distributed and therefore the failure rate is constant. The other variables we use are as follows:

- T_{fn} is the *Mean Time Between Failures* (MTBF) on each compute node.
- N is the number of nodes used in the computation.
- T_{fn}/N is mean time between failures for a computation across N nodes.
- $T_{FT} := T_{FT}(N)$ is the total run time of the FT-SGCT implementation on N nodes.
- C is the number of combinations throughout the computation.

Experimental results presented in Figures 5.2a and 5.9 allow us to estimate T_{FT} and T_{RP}, respectively, for different N. Note that in order to have a reasonable approximation error, it is sensible to choose C such that at most 1 fault occurs on average between combinations, that is, $C \geq T_{FT}/(T_{fn}/N) = N \cdot T_{FT}/T_{fn}$. The only overhead of the spawning based recovery is from the recovery of processes and reconstruction of communicators using ULFM MPI when a failure occurs. As the expected number of failures is equal to the number of combinations, one has the additional overhead $C \cdot T_{RP}$. Note that recovery in the SGCT algorithm only occurs prior to each combination, application instances not affected by the failures continue to run independently up to the combination at which time the status of processes within the global communicator is checked. Thus, $C \cdot T_{RP}$ is actually an upper bound on the overhead for process recovery throughout the computation. Thus, the expected overhead is bounded above by

$$C \cdot T_{RP} = \frac{N \cdot T_{FT}}{T_{fn}} T_{RP} \, .$$

On the other hand, since the only overhead of the shrinking based recovery is from the recovery of the broken communicators via shrinking the communicators, the expected overhead of this recovery is bounded by

$$C \cdot T_{SC} = \frac{N \cdot T_{FT}}{T_{fn}} T_{SC} \, .$$

One sees that this is inversely proportional to the MTBF per node. Note that the occurrence of faults obviously affects the error of the SGCT. The estimation of this error, however, is more involved.

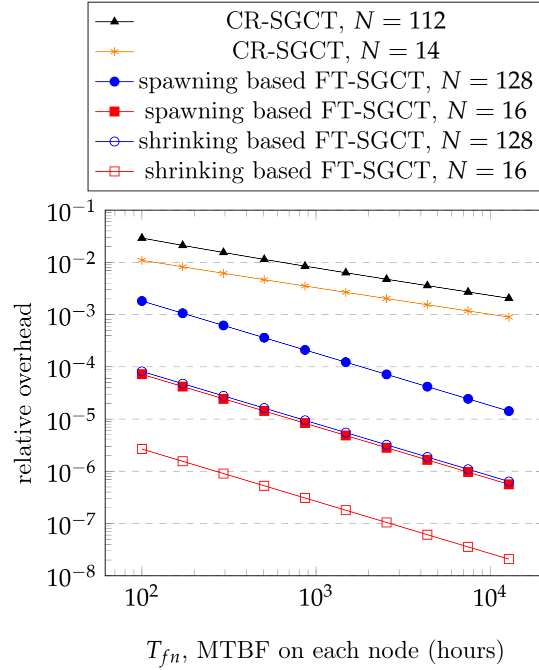


Figure 5.10: Expected relative recovery overhead for longer computation for GENE. The results shown are an average of five experiments for *2d_big_6* input applied to the 2D SGCT for GENE.

We will compare the algorithm-based recovery overheads of the FT-SGCT with the typical overhead of the Checkpoint/Restart applied to the SGCT computation. Here we define some additional values of interest.

- $T_{CR} := T_{CR}(N)$ is the total run time of the SGCT computation on N nodes using Checkpoint/Restart for recovery from faults.
- $T_{CR}/(T_{fn}/N) = N \cdot T_{CR}/T_{fn}$ is the expected number of faults throughout the computation.
- $T_{OC} = \sqrt{2T_{WR} \cdot T_{fn}/N}$ is the optimal time between checkpoints proposed by Young [Young, 1974].
- T_{CR}/T_{OC} is the total number of checkpoints throughout the computation.
- $T_R := T_R(N)$ is the total recovery time after a fault including restarting MPI and reading a checkpoint on N nodes. This is equivalent to $T_{RM} + T_{RD}$.
- $T_B = T_{OC}/2$ is the average backtrack time when a fault occurs, that is, the typical time between the last checkpoint and a failure for which recomputations must be done.

Experimental results summarized in Figures 5.2a and 5.9 allow us to estimate T_{CR}, T_{WR}, and T_R for some different values of N for GENE. Similarly, results from Figures 5.5a and 5.7a can be used with the similar type of results of Figure 5.9 for the

Taxila LBM and SFI applications to estimate these parameters for these two applications, respectively.

The total overhead for Checkpoint/Restart consists of two components. The first is the writing of checkpoints, which throughout the computation is

$$\frac{T_{CR}}{T_{OC}} T_{WR} = T_{CR} \frac{\sqrt{N \cdot T_{WR}}}{\sqrt{2T_{fn}}} .$$

Additionally, for each failure, MPI must be restarted, a checkpoint read, and recomputation done up to the point at which the failure occurred. This overhead is the restart time plus the typical recomputation time multiplied by the expected number of faults, that is,

$$N \frac{T_{CR}}{T_{fn}} (T_R + T_B) = T_{CR} \left(\frac{N T_R}{T_{fn}} + \frac{\sqrt{N \cdot T_{WR}}}{\sqrt{2T_{fn}}} \right) .$$

Adding the two together the total Checkpoint/Restart overhead is

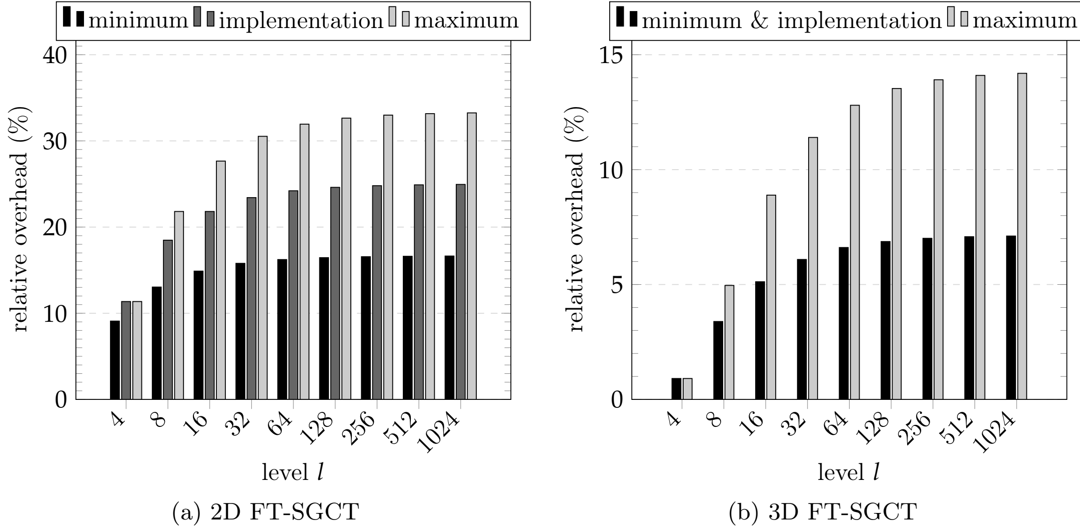
$$\frac{N \cdot T_{CR}}{T_{fn}} T_R + T_{CR} \frac{\sqrt{2N \cdot T_{WR}}}{\sqrt{T_{fn}}} .$$

Note that this overhead obviously extends the execution time of the application thus exposing it to more faults and that the same applies for the overhead with algorithm-based recovery. One may, however, divide the application execution time out of both overheads, and instead compare the overheads relative to the application execution times. We are particularly interested in how the two compare as the time between failures varies. The change in relative overheads with respect to T_{fn} is plotted in Figure 5.10 using representative values for the remaining variables obtained from the previous figures. It is observed that the overhead of the algorithm-based approach is significantly less than the equivalent computation done using Checkpoint/Restart.

If we compare the CR-SGCT $N = 112$ and $N = 14$ with the spawning based FT-SGCT $N = 128$ and $N = 16$, respectively, we observe approximately $15 - 100\times$ less relative overhead in the FT-SGCT than the CR-SGCT for the given lowest value of T_{fn}. Similarly, if we compare the shrinking based FT-SGCT $N = 128$ and $N = 16$ with the CR-SGCT $N = 112$ and $N = 14$, respectively, we observe approximately $350 - 3,500\times$ less relative overhead in the FT-SGCT than the CR-SGCT for the given lowest value of T_{fn}.

5.5.3 Overhead due to Computing Extra Grid Points

Some extra sub-grid computations are needed in the SGCT to achieve an algorithm-based fault resiliency called FT-SGCT. The amount of redundancy determines the accuracy of the combined solution when multiple sub-grids are lost at a time. This could be selected based on the reliability of the system on which the application is running. Applications running on the less reliable system should increase the



(a) 2D FT-SGCT

(b) 3D FT-SGCT

Figure 5.11: Relative overhead required in the SGCT to achieve an ABFT. It is calculated by dividing the total grid points in the SGCT out of extra grid points computed in the FT-SGCT. For the 2D case, minimum, implementation, and maximum are calculated with $e = 1, 2$, and $l - 2$, respectively, of equation (5.2). Similarly, for the 3D case, minimum & implementation and maximum are calculated with $e = 1$ and $l - 3$, respectively, of equation (5.3).

amount of redundancy to keep the approximation error in an acceptable level.

The number of grid points on each sub-grid of a layer/plane is half that of its upper (higher) layer/plane. With the current load balancing strategy, this principle also applies to the number of cores, if single-threaded MPI processes are used. For the 2D FT-SGCT, the overhead of computing extra grid points relative to computing the total grid points in the SGCT, is defined as

$$\begin{aligned}\text{relative overhead} &= \frac{\text{total unknowns in the FT-SGCT} - \text{total unknowns in the SGCT}}{\text{total unknowns in the SGCT}} \\ &= \frac{\text{extra unknowns}}{\text{regular unknowns}} \\ &= \frac{\sum_{i=1}^{e} \frac{(l-i-1)}{2^{i+1}}}{\sum_{i=1}^{2} \frac{(l-i+1)}{2^{i-1}}},\end{aligned} \tag{5.2}$$

where $l > 2$ is the level of the 2D SGCT, and $1 \leq e \leq l - 2$ is the number of extra layers. Similarly, for the 3D case, it is defined as

$$\text{relative overhead} = \frac{\sum_{i=1}^{e} \frac{(l-i-1)(l-i-2)}{2^{i+3}}}{\sum_{i=1}^{3} \frac{(l-i+2)(l-i+1)}{2^{i}}}, \tag{5.3}$$

where $l > 3$ is the level of the 3D SGCT, and $1 \leq e \leq l - 3$ is the number of extra

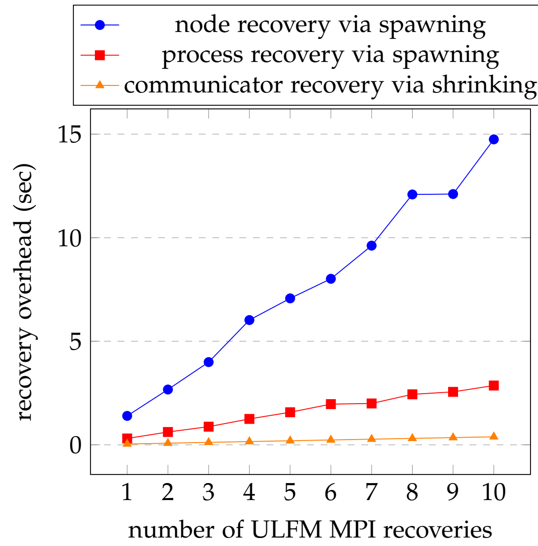


Figure 5.12: Repeated ULFM MPI failure recovery overheads of the 2D FT-SGCT with level $l = 5$ applied to the GENE application over 64 cores. The application is running with multiple combinations, and before performing each combination a real process fails. This failure is assumed to be due to a fault in a process or a node. The overhead includes recovery of all failures. The results shown here are an average of five experiments.

planes.

The relative overhead of our 2D FT-SGCT implementation (with level $l = 5$ and two extra layers) is 14.29%, whereas for the 3D case (with level $l = 4$ and one extra plane), it is 0.91%.

The minimum, maximum, and the implementation-specific relative overheads of the 2D and 3D FT-SGCT for various levels are shown in Figure 5.11. It is observed that the maximum relative overhead of the 3D SGCT is more than 2× lower compared to the 2D case. This indicates that the relative overhead will be significantly reduced if combination is performed on higher dimensions, rather than on lower dimensions.

5.5.4 Repeated Failure Recovery Overheads

T_{RP}, T_{RN}, and T_{SC} in Figure 5.9 are the spawning based process failure recovery overhead, spawning based node failure recovery overhead, and shrinking based communicator recovery overhead, respectively, for a single occurrence of faults. But in practice, process or node failures may happen repeatedly. So, it is an interesting task to find out how costly the repeated failure recoveries are. An experiment measuring the overhead of up to 10 repeated failure recoveries of GENE over 64 cores is shown in Figure 5.12. It shows that our implementation is robust to repeated failures and recoveries. The spawning based recovery overheads for each subsequent process and node failures are approximately 0.3 sec and 1.4 sec, respectively. With the shrinking based recovery, each subsequent communicator recovery takes approximately 0.04 sec.

5.6 Summary

This chapter evaluates the effectiveness of applying the FT-SGCT on three different types of existing complex parallel applications. Experimental evaluations reveal that applying the FT-SGCT on these applications show competitive execution times and acceptably low approximation errors, in comparison with the equivalent full grid computation. This chapter also evaluates the application level or algorithm-based recovery overheads implemented by ULFM MPI, and compares these with the checkpointing technique. Analysis shows that the application level recoveries are less expensive than the checkpointing, and relatively small amount of redundancy is required in the SGCT to achieve an algorithm-based fault resiliency.

SGCT algorithm is the central part of the application level fault tolerance. Detailed experimental evaluation of this algorithm was absent here. To fill this gap, the next chapter will focus on an in-depth analysis of the SGCT and applications.

Chapter 6

Evaluation of the SGCT and Applications

The previous chapter demonstrates that the combination algorithm is effective for a range of applications to be used as the basis of an ABFT. It also shows that this algorithm is computationally faster than the equivalent full grid simulation. However, there is a lack of in-depth performance analysis of this algorithm and the applications in various aspects, such as scalability, load balancing, shared memory parallelism, and so on, which could help optimizing the simulation to execute on the system.

In order to fill some of these gaps, this chapter provides a detailed analysis of the *direct* combination algorithm and the applications with respect to combination algorithm's scalability, load balancing, pure- and hybrid-MPI, process layouts, processor affinity, and so on.

The organization of this chapter is as follows. Detailed performance analysis of the combination algorithm is presented in Section 6.2. Section 6.3 analyzes the load balancing among the processes. A comparison of the pure- and hybrid-MPI performance on the combination algorithm and applications is presented in Section 6.4. An analysis of the effect of decomposed domain shape on application execution performance is discussed in Section 6.5. Section 6.6 explores if there is any effect of processor affinity on performance for the combination algorithm and applications.

The analysis of Sections 6.3 to 6.6 of this chapter is limited to the 2D version of the applications. The 3D version shows similar characteristics. Thus, the results of the 3D version are not presented and analyzed separately.

Contents of this chapter are accepted for publication jointly with others as a part of the paper titled "Design and Analysis of Two Highly Scalable Sparse Grid Combination Algorithms" [Strazdins et al., 2016b].

6.1 Introduction

Distributed parallel computation methodology of a problem consists of four stages: *partitioning*, *communication*, *agglomeration*, and *mapping* [Foster, 1995a]. Partitioning consists of decomposing the original problem into several small problems so that they can be computed concurrently. Communication coordinates the computation of

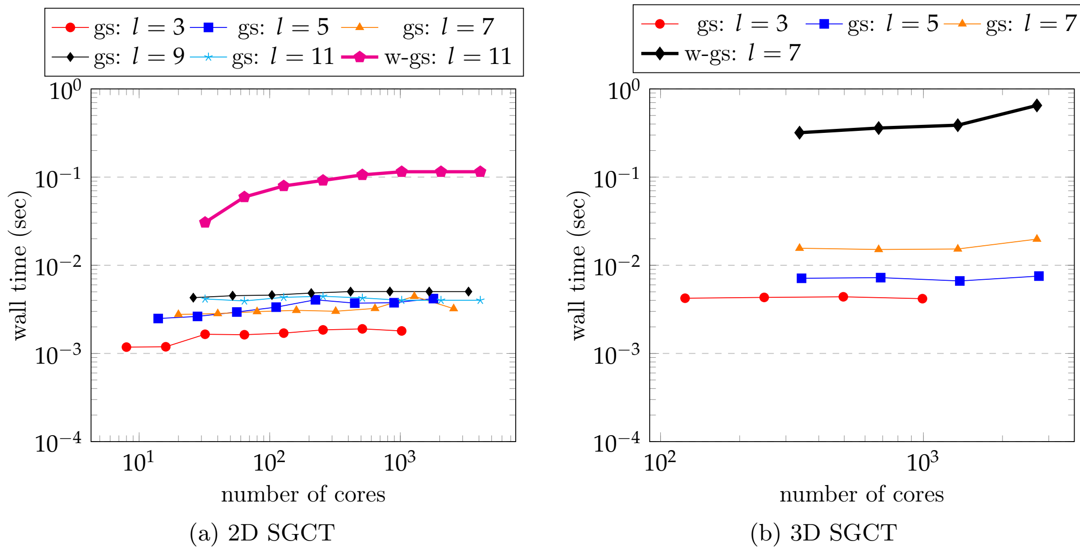


Figure 6.1: Execution time of the average of ten combinations of the direct SGCT in isolation. The workload of each core is 2^{14} grid points. 'l' represents the level of the SGCT. 'gs' and 'w-gs' denote the combination time, when MPI warm-up time is excluded and included, respectively. ISend/Recv-based implementations are used. The results shown are an average of five experiments.

these small problems. In order to achieve benefit from the parallel computation, the communication should be minimum, or should entirely overlap with the computation. Agglomeration combines the smaller problems into larger ones, if necessary, to minimize the overall communication. This should be done in such a way that the *surface-to-volume* ratio of each agglomerated problem is decreased, where surface and volume are proportional to communication and computation, respectively [Foster, 1995b]. Note that one of the difficulties of high-dimensional problems is the high surface-to-volume ratio (also known as concentration of measure in mathematics). Mapping corresponds to assigning each agglomerated problem to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. This is achieved by balancing loads among processors so that they finish their computations nearly at the same time. Communication costs could be further reduced by minimizing the intra-node communication by introducing shared-memory parallelism.

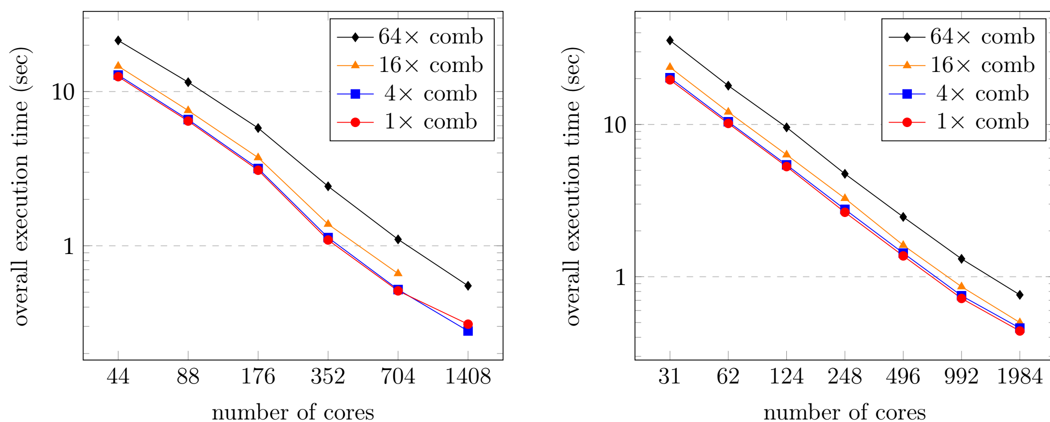
This chapter analyzes some of these factors in detail on the performance of the combination algorithm, a general advection solver, and three real-world applications. At the same time, detailed performance analysis of the combination algorithm in terms of scalability, combination overhead, and other key indicators is presented.

6.2 SGCT Performance Analysis

Referring to all the previous experiments we were confined to combination levels $l = 4$ or $l = 5$. However, according to [Harding and Hegland, 2013], SGCT with higher level provides some benefits. First, it may provide more accurate combined solution. Second, it increases the parallelism. An analysis is carried out to investigate how the SGCT performance varies with the change of level. Comparing 'gs' results in Figure 6.1, it is observed that the performance gaps for different levels are very small for both the 2D and 3D versions of the SGCT.

All the 'gs' results presented in Figure 6.1 are for 'warmed' timings where an SGCT was performed before the timing was taken (i.e., timings excluding warm-up time). Figure 6.1 also shows the timings for the direct SGCT without this 'warmed' timing (i.e., timings including warm-up time), which are labeled as 'w-gs'; we see a degradation in performance by a factor of approximately 20. This is caused by Open MPI setting up new (inter-grid) connections between processes in order to perform the SGCT. While not part of the SGCT itself, this overhead needs to be taken into account by any application using the SGCT.

It is reported in Section 5.2.4.3 of the previous chapter that multiple combinations of the SGCT reduce the approximation error of the combined solution. In this case, instead of performing only a single combination, multiple SGCT (*gather* and *scatter*) are applied. An experiment is conducted to measure the overhead of the repeated SGCT. Strong scaling results in Figure 6.2 indicate that the application scales even with the SGCT applied relatively frequently. The overhead due to repeated combination also seems to be small.



(a) 2D SGCT based application with level $l = 4$ and $(2^{13}+1) \times (2^{13}+1)$ grid points

(b) 3D SGCT based application with level $l = 3$ and $(2^9+1) \times (2^9+1) \times (2^8+1)$ grid points

Figure 6.2: Overall execution time of the general advection solver with the direct SGCT running over 1024 time-steps (MPI warm-up time excluded). ISend/Recv-based implementations are used. 'comb' represents combination. The results shown are an average of five experiments.

Although the combination algorithm seems to be computationally efficient and scalable, a thorough investigation is required to find out the operation which is relatively expensive compared to all the operations involved in the combination. This will provide an opportunity to improve performance in the future. To achieve this objective, an experiment similar to Figure 6.1a is carried out. But this time, a blocking BSend based implementation is used so that timing for one component is not overlapped with the others due to the asynchronous communication (ISend/Recv). Comparing the costs of all the operations related to the combination, it is observed that the interpolation operation seems to be relatively costly for the SGCT algorithm. For instance, with level $l = 9$ 2D SGCT, interpolation takes $20\% - 52\%$ of the total combination time (i.e., combination and interpolation takes $6.42E^{-3}$ sec and $3.37E^{-3}$ sec, respectively, on 26 cores; with 1664 cores, these timings are $1.77E^{-2}$ sec and $3.45E^{-3}$ sec, respectively). The 3D version also shows the similar characteristics to this. Thus, the interpolation routine should be targeted for the further improvement of the direct SGCT algorithm, even though it does not dominate the total combination time in these tests. Otherwise, we should consider the *hierarchical* SGCT algorithm which does not need to do interpolation [Strazdins et al., 2015].

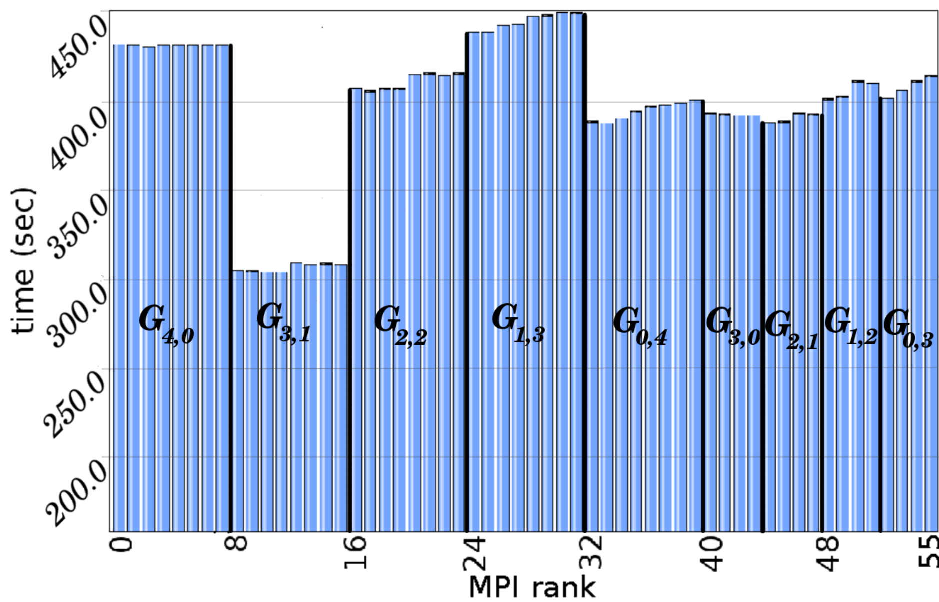


Figure 6.3: An analysis of the TAU-generated load balancing of the 2D direct SGCT computing the GENE application with *2d_big_6* input and a single combination (MPI warm-up time included). $p' = 8$ processes are allocated to each of the upper diagonal of sub-grids.

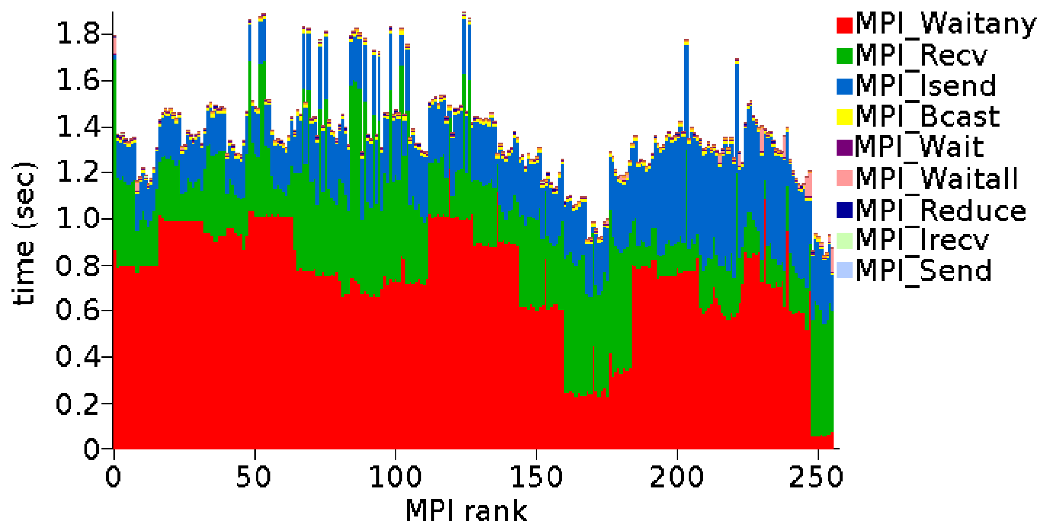
6.3 Load Balancing and Communication Profiles

As mentioned in Section 2.5 (Related Work), there are different ways of load balancing for the SGCT. A simple and static *master-slave* load balancing model for the SGCT is proposed in [Griebel et al., 1993; Garcke and Griebel, 2001]. With the assumption that a single core will compute multiple tasks (sub-grids), the master sorts the tasks in descending order based on the number of unknowns of the sub-grids. Then for a given P number of cores (sufficiently less than the number of tasks), the master distributes the first P tasks among the P cores. After that, the remaining tasks are distributed among the cores in such a way that the current largest task is assigned to one of the cores with the smallest workload. This policy is repeated until all the tasks are distributed among the cores. A similar strategy is followed in [Griebel et al., 1992a] to calculate the load using the number of unknowns, and both a static and dynamic load balancing strategies are proposed. With the static version, all the loads are distributed in advance by following the same policy of [Griebel et al., 1993; Garcke and Griebel, 2001] across all the multi-core groups (rather than a single-core group of [Griebel et al., 1993; Garcke and Griebel, 2001]) owned by the sub-grids. Later, this strategy is adopted in [Harding et al., 2014]. With dynamic load balancing, each group is assigned only a single task from the sorted list of tasks before initiating the computation. Whenever a core group finishes with its current task, the next task from the list is assigned to it. This techniques is iterated until the whole computation is finished. A similar static and dynamic load balancing technique is proposed in [Heene et al., 2013]. Only the difference is that it estimates the load of a sub-grid with the number of unknowns and the anisotropy of the discretization of the sub-grid.

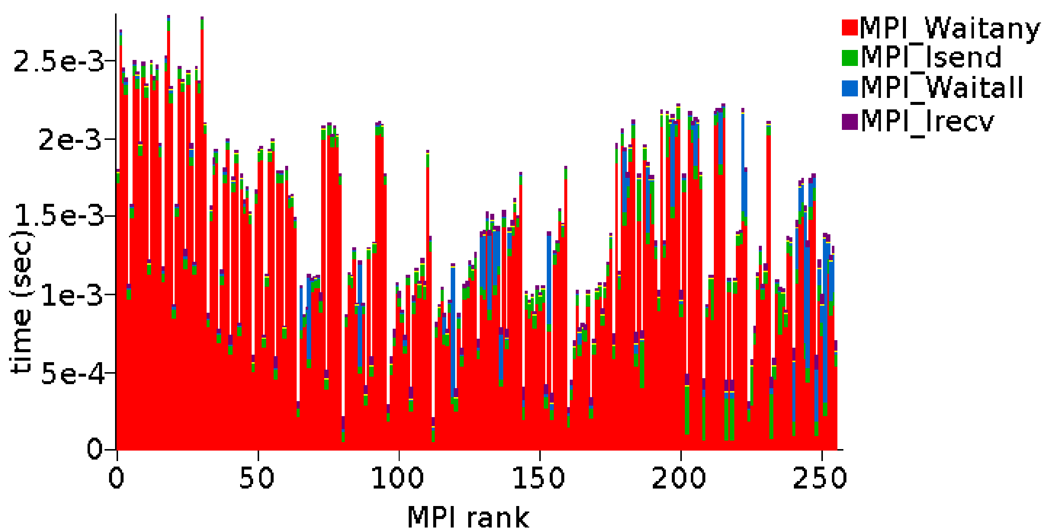
A simple strategy is used on a global SPMD parallelism model in this thesis to balance the load among the processes to execute the SGCT based applications. The same number ($p' \in \mathbb{N}$) of processes is allocated on each of the distinct set of processes $P_{\underline{i}}$ on the uppermost diagonal in the grid index space (see Figure 2.4; in the 3D case, the diagonal becomes a plane and three planes are required for the non-fault-tolerant case). The next lower diagonal is allocated $\lceil p'/2 \rceil$ processes. This strategy balances the amount of data points and hence work across each process, which approximates to a first order to the load for that process. To support the alternate combination technique (i.e., FT-SGCT), four diagonals/planes of sub-grids are used, and $\lceil p'/4 \rceil$ and $\lceil p'/8 \rceil$ processes are allocated for each sub-grid on the next two lower diagonals, respectively.

An analysis of load balancing is carried out based on this balancing strategy. The TAU profiling tool [Shende and Malony, 2006] is used to generate the reported timings. It is observed from Figure 6.3 that the first order approximation of loads to each process sufficiently balances loads among the MPI processes (ranks) for the 2D direct SGCT computing the GENE application. However, it can be seen that the second sub-grid is computed fastest, and the first and fourth sub-grids are slowest.

The GENE application is observed to be very sensitive to the number of processes allocated to each of its five dimensions (excluding species). In our current



(a) computing whole application: MPI task



(b) performing a single combination: MPI task

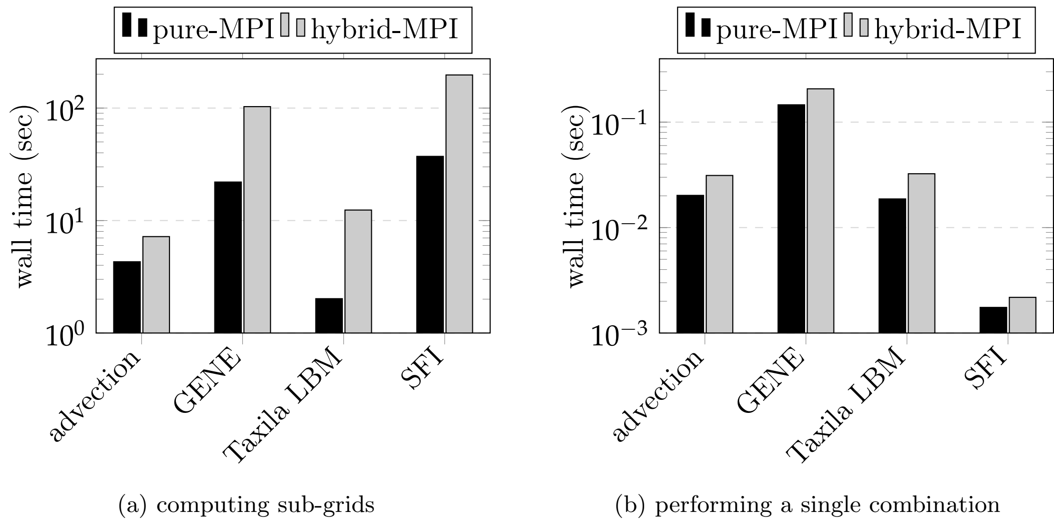
Figure 6.4: An analysis of the IPM generated load balancing of the 2D direct SGCT solving the general advection problem on level $l = 11$ with a single combination (MPI warm-up time excluded). $p' = 16$ processes are allocated to each of the upper diagonal of sub-grids. The workload per core is 2^{14} points and the number of time-steps is 2^{14}. Total execution time is 4.06 sec, and combination time is 0.0058 sec.

implementation, we set processes only for the two or three dimensions (for the 2D and 3D FT-SGCT respectively), and the remaining are set to 1. The process grid for the second sub-grid properly distributes the processes across all the dimensions, so it is computed fastest.

2D application	full grid size	time-steps	others
advection	$(2^{13}+1) \times (2^{13}+1)$	2^{13}	-
GENE	$N_v \times N_u = 2^8 \times 2^8$	10	Table B.1
Taxila LBM	$2^{13} \times 2^{13}$	20	Table B.2
SFI	$2^{11} \times 2^{11}$	default	-

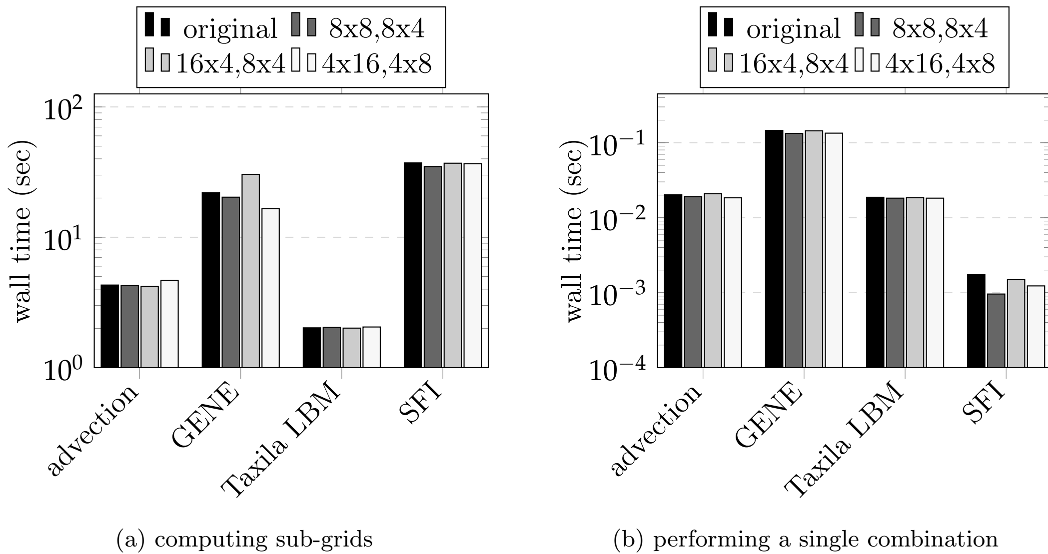
Table 6.1: Parameters used in advection, GENE, Taxila LBM, and SFI experiments.

In order to analyze the load balancing a little bit deeper, Figure 6.4 gives IPM profiles [Wright et al., 2009; IPM] for the 2D general advection solver with a level $l = 11$ direct SGCT (profiles for the GENE application are also similar to this application). We can clearly see the structure of the advection sub-problems. `MPI_Isend` and `MPI_Recv` are used only in the advection phase, being used for halo exchanges. The fluctuations in `MPI_Waitany` in part (a) indicates load imbalance across the process grid caused by differences in computational speed across the different sets of advection problems. It is also the dominant communication time for the SGCT operation (part (b)). Since we are using a simple load balancing strategy, this little amount of load imbalance is expected.



(a) computing sub-grids (b) performing a single combination

Figure 6.5: A comparison of the pure- and hybrid-MPI performance for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). With pure-MPI, $p' = 64$ processes, each process contains a single thread and each thread maps to a single core, are allocated to each of the upper diagonal of sub-grids. With hybrid-MPI, $p' = 8$ processes, each process contains 8 threads and each thread maps to a single core, are allocated to each of the upper diagonal of sub-grids. Other necessary parameters are from Table 6.1. The results shown are an average of five experiments.



(a) computing sub-grids (b) performing a single combination

Figure 6.6: A comparison of the performance due to different process layouts for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). $p' = 64$ processes are allocated to each of the upper diagonal of sub-grids. Other necessary parameters are from Table 6.1. The results shown are an average of five experiments.

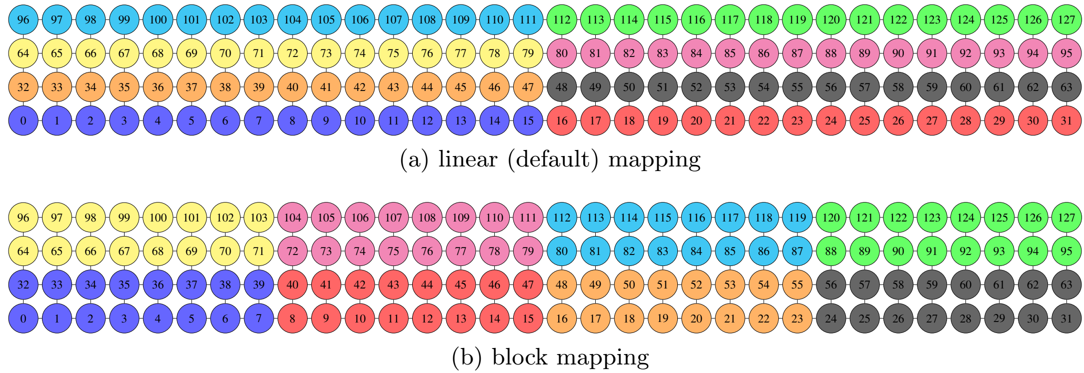
6.4 Pure-/Hybrid-MPI Performance Analysis

The combination algorithm (and the applications) seems to be communication-intensive. With a significantly larger number of cores, communication cost is expected to dominate over computation cost. A possible way of resolving this issue is to minimize the explicit intra-node communication and to make more efficient use of the shared memory of the node by hybrid-MPI.

We use the simplest execution model for hybrid-MPI. We introduce "#pragma omp parallel for" directives in front of individual loops in order to achieve the fine-grained loop-level work sharing.

Experiments are carried out on Raijin cluster to compare the performance of the pure- and hybrid-MPI. Note that a Raijin node consists of 2 processor sockets, each comprising 8 cores. With the pure version, each MPI process is mapped to each core, and a single OpenMP thread is working for each process. Alternatively, the hybrid version maps each MPI process to each socket, and each MPI process runs with 8 OpenMP threads. With this setup 1/8-th of MPI processes are used in hybrid-MPI compared to pure-MPI, but the same core counts are used in both cases for each experiment. As a result, total amount of inter-process communications in the hybrid version is 8 times less than that of the pure version. For both cases, the levels of thread provided by Open MPI are `MPI_THREAD_SINGLE`.

An experiment is carried out to evaluate the performance of the pure- and hybrid-MPI on the 2D direct SGCT applied to a general advection solver and three real-world



(a) linear (default) mapping

(b) block mapping

Figure 6.7: 2D linear and block mapping of `32x4` process grid onto the cores of Raijin nodes. Processes with the same color are mapped onto the cores of the same node. Different color represents they are mapped onto the different nodes.

applications. It is observed from the experimental results of Figure 6.5 that the hybrid version is computationally more expensive than the pure version. An analysis reveals that the overall communication is reduced in hybrid-MPI compared to pure-MPI, but the scheduling of threads to cores by an MPI process causes that process to wait for a longer period of time. As for example, from the IPM [Wright et al., 2009; IPM] profile of a 2D general advection solver it is observed that the communication is reduced from 16.87% to 1.43% in hybrid-MPI than pure-MPI. An analysis of pure- and hybrid-MPI performance for the GENE application in [Sáez and Soba, 2008] also shows that GENE execution in hybrid-MPI is 10% slower than pure-MPI. Detailed analysis shows that hybrid-MPI reduces the communication in GENE, but the idle time in the working scheduling of the threads in the hybrid version is the root cause of decreasing the overall performance compared to the pure version. Reducing this thread scheduling overhead at run-time could boost up the hybrid-MPI performance. This could possibly be investigated further in the future.

6.5 Effect of Process Layouts on Performance

Process layouts of each sub-grid determine the shape of the decomposed domain. Changing domain shape changes the communication pattern for both the computation of application on each sub-grid and performing the combination. As for example, 1D and 2D process grids may need to update the boundary values on two and four directions, respectively. Each application has its own communication requirements, and so process layouts may affect them differently. We analyze the effect on a general advection solver and three real-world applications with different process layouts.

Process layouts for 'original' (original or proposed) of the 2D $G_{\underline{i}}$ with $\underline{i} = (3,0)$, $(2,1)$, $(1,2)$, $(0,3)$, $(2,0)$, $(1,1)$, and $(0,2)$ are `16x4`, `8x8`, `8x8`, `4x16`, `8x4`, `8x4`, and `4x8`, respectively. Process layouts of the 2D full grid are `16x16`. In the 2D case, a

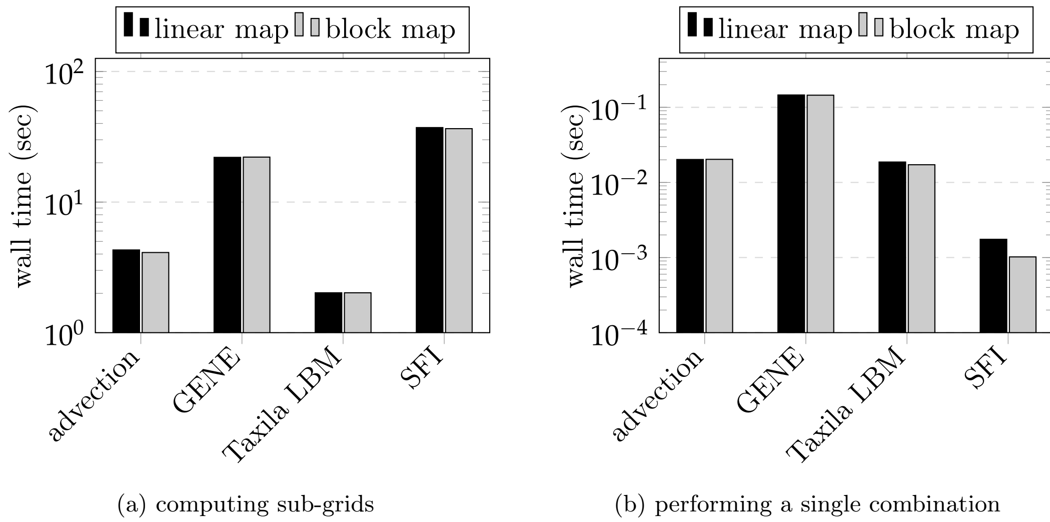


Figure 6.8: A comparison of the performance due to the linear and block mappings for the 2D direct SGCT with level $l = 4$ and a single combination (MPI warm-up time excluded). $p' = 64$ processes are allocated to each of the upper diagonal of sub-grids. Other necessary parameters are from Table 6.1. The results shown are an average of five experiments.

legend `AxB,CxD` indicates that the process layouts of each sub-grid on upper and lower layers are `AxB` and `CxD`, respectively.

An experiment is carried out to analyze the effect of process layouts on the 2D direct SGCT applied to a general advection solver and three real-world applications. In this case, each single-threaded MPI process is mapped onto a distinct CPU core. It is observed from the experimental results of Figure 6.6 that process layouts affect the performance differently on different applications. As for example, process layouts `8x8,8x4` and `4x16,4x8` reduce computation costs for the GENE and SFI applications (both sub-grids and combination) compared to the original layouts, but there is no significant difference in performance is observed for the other two applications. So, if the SGCT needs to change its own layouts to adapt to the application, it will not affect the performance significantly.

6.6 Effect of Processor Affinity on Performance

Each process is mapped onto physical processor core(s) to execute the task assigned to it. It should be done in such a way that the overall execution time (both sub-grids and combination) is minimized. There are two strategies a mapping algorithm follows to achieve this. First, tasks that are able to execute concurrently are mapped onto different processors to increase the concurrency. Second, tasks that need frequent communication are mapped onto the same processor so as to reduce the communication cost. Application instances executing on each sub-grid are mapped onto different processors to increase the concurrency/parallelism, but this increases the

communication cost when the sub-grid solutions are combined into a single combination grid. Alternately, an attempt can be taken to decrease the communication cost for combination, but it sacrifices the application concurrency/parallelism.

In order to analyze the effect of processor affinity on performance, each single-threaded MPI process is mapped onto a distinct CPU core. The process-to-core mapping of all the previous benchmarks were done by mapping the processes to cores linearly (default), similar to the approach shown in Figure 6.7a for the 2D process grid. It is an interesting part of research to investigate what happens to performance if processes are mapped block-wise, similar to the approach shown in Figure 6.7b for the 2D process grid.

An experiment is conducted on the 2D direct SGCT applied to a general advection solver and three real-world applications to compare the performance of the linear and block mappings. Experimental results shown in Figure 6.8 reveal that the block mapping increases the combination performance for the Taxila LBM and SFI applications compared to the linear mapping. Moreover, the block mapping seems to be a little bit faster compared to the linear mapping for computing advection on sub-grids. For the other cases, however, no significant performance difference is observed. Thus, run-time block process-to-core mapping can be applied to the SGCT based applications to enhance their performance.

6.7 Summary

This chapter presents an in-depth performance analysis of the combination algorithm and applications. It is observed that the combination algorithm is scaling with level, and the repeated application of the combination algorithm adds only a small overhead. It is also observed that it is possible to roughly balance the loads among the processes with a simple strategy. The communication of a highly communication-intensive application could be reduced by a factor with the shared-memory implementation by hybrid-MPI, but with an unacceptable degradation of performance in the solver. Process layouts responsible for determining the shape of the decomposed domain do not appear to significantly affect the performance of the combination (and applications as well), which facilitates the use of application-constrained layouts when integrating with the SGCT. An analysis of the block and linear mapping shows that the block mapping may improve the performance of computing the sub-grids and performing the combination for some of the applications.

Chapter 7

Conclusion

Considering the power consumption, clock frequencies, node size, performance, and other system requirements, frequent component failures seem to be inevitable in upcoming exascale systems. The MTBF is expected to be so small that even a highly tuned checkpointing technique could not be applied efficiently as fault recovery for the ultra-large-scale scientific simulations running on them. The increasing volume of simulation data also makes the checkpointing technique infeasible. A cost-effective model to process the increased amount of scientific simulation data and a feasible technique to recover from faults in an exascale system are thus two of the many urgent requirements for the next generation ultra-large-scale scientific simulations.

In this thesis, we have presented an overview of a general parallel Sparse Grid Combination Technique (SGCT) algorithm and its associated load balancing strategy, which is used to process ultra-large-scale scientific simulation data in a cost-effective way with only small loss of accuracy. The algorithm can be applied over several of the dimensions of a multi-dimensional field of a time-evolving PDE application. It also easily supports parallelization in the non-SGCT dimensions. Thus, it is capable of supporting extremely large-scale applications.

We have shown how, using a general methodology, it can be integrated into three existing, complex real-world applications – the GENE gyrokinetic plasma application, Taxila Lattice Boltzmann Method (Taxila LBM) application, and Solid Fuel Ignition (SFI) application – with minimal changes to the source code to evaluate its effectiveness. The software engineering effort required to integrate the SGCT into the application was acceptable. Even if the application has the complication of non-trivial boundary conditions, which needs to be applied for the SGCT, it is likely the application itself will provide the required code, as was the case for the GENE application. Not only can the application benefit from the efficiency-accuracy trade-offs of the SGCT, for a relatively small amount of extra effort, it can also be made fault-tolerant in the form of an Algorithm-Based Fault Tolerance (ABFT).

For relatively large fields of GENE, the large core count overhead of the combination was of the order of 1 to 100 msec (excluding MPI warm-up time), which is easily acceptable compared with the current and near future MTBFs. Successful integration of the Taxila LBM and SFI applications with the SGCT also shows the similar characteristics. Furthermore, we have shown that the 2D, and especially the 3D, SGCT have significant computational efficiencies compared with traditional full

grid simulation while having acceptable losses in accuracy. We did find however that the 'smoothness' of the dimensions of the field chosen for the SGCT were important in this respect. In the case of multiple failures, multiple combinations can be used to reduce the error.

Using a recent release of User Level Failure Mitigation (ULFM) MPI, we have shown how it can be applied to recover a HPC application from process/node failures. We designed and implemented a toolkit which could be directly integrated into any grid based applications as a part of making them fault-tolerant.

We have shown that the integration of ULFM MPI with the fault-tolerant version of the SGCT can be used to recover complex applications from both process and node failures. Compared with the built-in checkpoint infrastructure in the GENE application and job restart from the checkpoint, our spawning and shrinking based approaches have approximately 1/4 and 1/90 of the overhead, respectively, for a one-off failure excluding the overhead of backtrack time. An analysis for a long-running application taking the backtrack time into account shows that our spawning based technique has an overhead between one and two orders of magnitude less. A similar analysis for the shrinking based technique shows an overhead between two and three orders of magnitude less than the checkpointing technique. We expect this performance gap to increase as the relatively recent ULFM MPI matures. An experiment with repeated failures shows that the ULFM MPI implementation is robust to repeated failures and recoveries.

We have analyzed in our experiments that only 14.29% and 0.91% redundancies are required to make the direct SGCT fault-tolerant for the current implementation of the 2D and 3D cases, respectively. An increase of the number of sub-grids leads only to a very slow increase of the required redundancies.

We have shown that the combination algorithm achieves higher scalability suitable for exascale computing. We have also shown that the communication of the SGCT based application could be reduced by applying hybrid-MPI, but with an unacceptable degradation of performance in the application. It has been shown that changing the shape of the decomposed domains does not affect the performance of the SGCT based applications significantly, and thus application-constrained process layouts could be used in the integrated applications. We have also analyzed the block and linear mappings and shown that the block mapping may improve the performance of computing the sub-grids and performing the combination for some of the applications.

We expect that our SGCT based ABFT technique will show significant advantages on current and future platforms beyond the ones we had access to for this thesis. This includes both much larger systems where checkpointing overheads become prohibitive, and systems whose components are less reliable than supercomputer nodes, e.g. very cheap processors operated at minimal voltage in order to save power.

7.1 Future Work

The potential future directions for this line of research are as follows.

Since the integration methodology of the SGCT/FT-SGCT and an application is general, this could be applied to other complex PDE based applications suitable for the SGCT. Some example applications could be: (1) *Overture*[1], which uses finite volume and finite differences on regular meshes. (2) *OpenFOAM*[2], which is a relatively well-known computational fluid dynamics toolbox. It is mainly designed to use meshes but would accept structured rectangular meshes. (3) *OpenLB*[3], which is an open-source lattice Boltzmann code to address a vast range of problems in computational fluid dynamics. (4) *Flashcode*[4], the "Direct Solvers for Uniform Grid" part of this code.

In this thesis, the combination algorithm is applied on up to 3 dimensions. This can be extended to higher dimensions. The higher the dimensions the SGCT is applied to, the higher the computational efficiency expected to be achieved. Such extension may reduce the computational complexity of ultra-large-scale scientific simulations.

The work in this thesis handles the *permanent* or *fail-stop* type of faults, where failure detection seems to be straight-forward and relatively easy using the ULFM MPI semantics. This research could be extended to handle soft faults, where fault detection is one of the key challenges. Since the faulty components continue to execute the application without reporting any symptom of the soft fault, it is difficult to detect. The detection could be performed by scanning the memory footprints for any unusual activity happening there. If so, an explicit signal should be issued to the process accessing the affected part of the memory so that the remaining processes can detect it as a process failure by the ULFM MPI semantics. Some system-level research which could detect unusual memory operations are available in [Allan, 2014; Narsale and Huang, 2009; Ndai et al., 2005; Meaney et al., 2005]; and these could be integrated with the work in this thesis to tolerate soft faults.

[1] http://www.overtureframework.org/
[2] http://www.openfoam.com/
[3] http://optilb.com/openlb/
[4] http://flash.uchicago.edu/site/flashcode/

Appendix A

Process Recovery by ULFM MPI

Fault tolerance is a promising and trending research area in this decade. ULFM MPI is moving this research area a step forward. However, it is not completely clear to the researchers how effective ULFM MPI is in this area, as there are only a limited amount of practical work available. Additionally, the details of actually how the ULFM MPI semantics could be used are not clearly presented. The key objective of this chapter is to present the idea in a brief and managed way so that it could be applied to different applications of this area with a less effort.

In order to achieve this objective, this chapter presents some high-level/mid-level algorithms for the detection, identification, and recovery of process and node failures.

Algorithm 3 is the main algorithm which uses Algorithm 2 as an error handler, Algorithm 5 as a detector and identifier of the failed processes, Algorithm 4 as a part of repairing the broken communicator via spawning the replacement processes or shrinking the broken communicator, and Algorithm 6 as a part of ordering the ranks in the reconstructed communicator for the spawning based approach.

A low-level implementation with documentation of these algorithms is done in C. It is available at https://github.com/mdmohsinali/ULFM-Process-Failure-Recovery to download, test, and apply as a toolkit for the targeted applications.

Function `void mpiErrorHandler`(MPI_Comm * comm, int *error_code, $\cdots$)

Input: A communicator (comm).
Output: Error handler of communicator comm.

```
1: MPI_Group failedGroup;
2: OMPI_Comm_failure_ack(*comm);
3: OMPI_Comm_failure_get_acked(*comm, &failedGroup);
   /* Sometimes a delay of at least 10 milliseconds (with usleep(10000);) is
   needed here */
```

Algorithm 2: Procedure for handling errors.

```
Function MPI_Comm communicatorReconstruct(MPI_Comm myWorld,
                                          bool shrinkMode)
Input: Broken communicator (myWorld).
Output: Reconstructed communicator (reconstructedComm).
1:  iterCounter ← 0;
2:  MPI_Comm_create_errhandler(mpiErrorHandler, &newErrHand); // Pass Algorithm 2
3:  MPI_Comm_get_parent(&parent);
4:  do
6:      failure ← 0;
7:      returnValue ← MPI_SUCCESS;
8:      if parent = MPI_COMM_NULL then // Parent
9:          if (iterCounter = 0) then
10:             reconstructedComm ← myWorld;
11:         MPI_Comm_set_errhandler(reconstructedComm, newErrHand);
12:         OMPI_Comm_agree(reconstructedComm, &flag); // Synchronize
13:         returnValue ← MPI_Barrier(reconstructedComm); // To detect failure
14:         if returnValue ≠ MPI_SUCCESS then
15:             tempIntracomm ← repairComm(&reconstructedComm, shrinkMode); //
                Call Algorithm 4
16:             failure ← 1;
17:         else
18:             failure ← 0;
19:     else if not shrinkMode then // Child and not shrinkMode
20:         MPI_Comm_set_errhandler(parent, newErrHand);
21:         OMPI_Comm_agree(parent, &flag); // Synchronize (child part)
            // Merging intercommunicator (child part)
22:         MPI_Intercomm_merge(parent, true, &unorderIntracomm);
            // Receiving rank information from parent
23:         MPI_Recv(&oldRank, 1, MPI_INT, 0, MERGE_TAG, unorderIntracomm,
            &mpiStatus);
            // Ordering ranks in the new intracommunicator
24:         MPI_Comm_split(unorderIntracomm, 0, oldRank, &tempIntracomm);
25:         returnValue ← MPI_ERR_COMM ;
26:         failure ← 1;
27:     if returnValue ≠ MPI_SUCCESS then
28:         reconstructedComm ← tempIntracomm;
29:     if returnValue ≠ MPI_SUCCESS and parent = MPI_COMM_NULL then // Parent
        was failed
30:         parent ← reconstructedComm;
31:     if returnValue ≠ MPI_SUCCESS and parent ≠ MPI_COMM_NULL then // Child
        was failed
32:         parent ← MPI_COMM_NULL;
33:     iterCounter++;
34: while (failure);
35: return reconstructedComm;
```

Algorithm 3: Procedure for reconstructing the broken communicator due to process failures.

```
Function MPI_Comm repairComm(MPI_Comm * brokenComm,
                             bool shrinkMode)
Input: Broken communicator (brokenComm).
Output: Part of repaired communicator (repairedComm or shrunkenComm).
 1: SLOTS ← 16; // Suppose slots count in each host (node) is 16
 2: OMPI_Comm_revoke(&brokenComm); // Revoke the communicator
 3: OMPI_Comm_shrink(*brokenComm, &shrunkenComm);// Shrink the communicator
 4: if shrinkMode then
 5:     return shrunkenComm;
 6: else // not shrinkMode
 7:     (failedRanks, totalFailed) ← failedProcsList(brokenComm); // Call Algorithm 5
 8:     The list failedRanks with size totalFailed contains ranks of the failed processes that are happened due
        to both process and node failures. Process and node failure detection algorithm is applied to split this
        list into two lists: failedRanksDueToProcFail with size totalFailedDueToProcFail and
        failedRanksDueToNodeFail with size totalFailedDueToNodeFail. ;
 9:     if process failure then
10:         for i ← 0; i < totalFailedDueToProcFail; i++ do
11:             hostfileLineIndex ← ⌊failedRanksDueToProcFail_i /SLOTS⌋;
12:             Read hostfileLineIndex^th entry (0^th is the starting) without slots information from rankmap
                file (hostfile) to hostNameToLaunch_i;
13:             appLaunch_i ← "./ApplicationName";
14:             argvLaunch_i ← argv;
15:             procsLaunch_i ← 1;
16:             MPI_Info_create(&hostInfoLaunch_i);
                // host information where to spawn the processes
17:             MPI_Info_set(hostInfoLaunch_i, "host", hostNameToLaunch_i);
18:     else // node failure
19:         for j ← 0; j < totalFailedDueToNodeFail; j++ do
20:             Set hostfileLineIndex with ⌊j/SLOTS⌋^th unused line index of hostfile reserved for spare nodes.
                Initially, all the line indices reserved for spare nodes are flagged as unused. After mapping a
                total of SLOTS processes to the same unused line index, the unused flag of that line index is
                changed to used. ;
21:             Read hostfileLineIndex^th entry without slots information from hostfile to
                hostNameToLaunch_{i+j};
22:             appLaunch_{i+j} ← "./ApplicationName";
23:             argvLaunch_{i+j} ← argv;
24:             procsLaunch_{i+j} ← 1;
25:             MPI_Info_create(&hostInfoLaunch_{i+j});
                // host information where to spawn the processes
26:             MPI_Info_set(hostInfoLaunch_{i+j}, "host", hostNameToLaunch_{i+j});
        // Spawn new processes on the same host which experiences process failures
27:     MPI_Comm_spawn_multiple(totalFailed, appLaunch, argvLaunch, procsLaunch,
                hostInfoLaunch, 0, shrunkenComm, &tempIntercomm, MPI_ERRCODES_IGNORE);
        // Merging intercommunicator (parent part)
28:     MPI_Intercomm_merge(tempIntercomm, false, &unorderIntracomm);
29:     OMPI_Comm_agree(tempIntercomm, &flag); // Synchronize (parent part)
30:     MPI_Comm_size(shrunkenComm, &shrunkenGroupSize);
31:     for i ← 0; i < totalFailed; i++ do
32:         child_i ← shrunkenGroupSize + i;
33:     MPI_Comm_rank(unorderIntracomm, &newRank);
34:     MPI_Comm_size(unorderIntracomm, &totalProcs);
        // Sending rank information to child
35:     if newRank = 0 then
36:         for i ← 0; i < totalFailed; i++ do
37:             MPI_Send(&failedRanks_i, 1, MPI_INT, child_i, MERGE_TAG, unorderIntracomm);
        // Ordering ranks in the new intracommunicator
38:     rankKey ← selectRankKey(newRank, shrunkenGroupSize, failedRanks, totalProcs); // Call
        Algorithm 6
39:     MPI_Comm_split(unorderIntracomm, 0, rankKey, repairedComm);
40:     return repairedComm;
```

Algorithm 4: Procedure as a part of repairing the broken communicator.

```
Function int * failedProcsList(MPI_Comm * brokenComm)
Input: Broken communicator (brokenComm).
Output: List of failed processes (failedRanks) and number of failed processes
        (totalFailed).
1:  MPI_Comm_group(*brokenComm, &oldGroup);
2:  MPI_Comm_group(shrunkenComm, &shrinkGroup);
3:  MPI_Comm_size(*brokenComm, &oldSize);
4:  MPI_Group_compare(oldGroup, shrinkGroup, &result);
5:  if result ≠ MPI_IDENT then
6:      MPI_Group_difference(oldGroup, shrinkGroup, &failedGroup);
7:  MPI_Comm_rank(*brokenComm, &oldRank);
8:  MPI_Group_size(failedGroup, &totalFailed);
9:  for i ← 0; i < oldSize; i++ do
10:     tempRanks_i ← i;
11: MPI_Group_translate_ranks(failedGroup, totalFailed, tempRanks, oldGroup,
    failedRanks);
12: return failedRanks and totalFailed;
```

Algorithm 5: Procedure for creating list and number of failed processes.

```
Function int selectRankKey(int mpiRank, int shrunkenGroupSize,
                           int * failedRanks, int totalProcs)
Input: MPI rank (mpiRank), shrunken communicator size
       (shrunkenGroupSize), list of failed processes (failedRanks), and MPI
       global communicator size (totalProcs).
Output: Key value of a rank (key) to be used for splitting the communicator to
        order the ranks.
1: j ← 0;
2: for i ← 0; i < totalProcs; i++ do
3:     if i ∉ failedRanks then
4:         shrinkMergeList_j ← i;
5:         j++;
6: for i ← 0; i < shrunkenGroupSize; i++ do
7:     if mpiRank = i then
8:         key ← shrinkMergeList_i;
9: return key;
```

Algorithm 6: Procedure for selecting the keys of ranks to be used for splitting the communicator to order the ranks.

Appendix B

Application Input Parameters

This chapter presents some parameters which are used to initialize different applications for performing different experiments.

```
&parallelization
!n_procs_s   =   1
!n_procs_v   =   1
!max_npv     =   4
!n_procs_w   =  16
!n_procs_x   =   4
!n_procs_y   =   1
!max_npy     =   2
!n_procs_z   =   2
!n_procs_sim = 128
/

&box
n_spec =    1
nx0    =  160
nky0   =   24
nz0    =   16
nv0    =   32
nw0    =   16

kymin =   0.6064E-01
lv    =     3.00
lw    =     9.00
lx    =   147.760
x0    =   0.5000
n0_global =    4
mu_grid_type = 'equidist'
/

&in_out
diagdir = '/out'

read_checkpoint  = F
write_checkpoint = F

istep_field =   100
istep_mom   =   100
istep_nrg   =    20
istep_vsp   =     0
istep_schpt =  5000

write_std = T
momentum_flux = F
/
```

```
&general
nonlinear =   T
x_local   = F
comp_type = 'IV'
!perf_vec = 1 1 2 1 1 1 2 1 1
!nblocks  =  256
arakawa_zv =  T
arakawa_zv_order = 4
hypz_opt = F

timescheme = 'RK4'
dt_max     =  0.2110E-01
courant    =  0.30
timelim    =      86000
ntimesteps =      100
underflow_limit = 0.1000E-09
beta       =    0.0000000
debye2     =    0.0000000
collision_op = 'none'

init_cond = 'db'

hyp_x =    2.000
hyp_z =    1.000
hyp_v =   0.2000
/

&nonlocal_x
dealiasing = F
l_buffer_size =   0.4000E-01
u_buffer_size =   0.2500E-01
rad_bc_type =  1
/
```

```
&geometry
magn_geometry = 'circular'
trpeps   = 0.18270000
major_R  = 1.0000000
major_R  = 1.0000000
minor_r  = 0.36550000
q_coeffs = 0.854, 0.00, 2.184
mag_prof = T
rhostar  = 0.54140000E-02
norm_flux_projection  = F
/

&species
name   = 'ions'
prof_type = 4
kappa_T   = 6.910000
LT_center = 0.500000
LT_width  = 0.40E-01

kappa_n   =  2.21800
Ln_center = 0.500000
Ln_width  = 0.40E-01

delta_x_T  = 0.80000
delta_x_n  = 0.80000

mass   = 1.0000000
temp   = 1.0000000
dens   = 1.0000000
charge = 1
/

&units
/
```

Table B.1: Parameters in `testsuite/big/parameters_6` file used in GENE experiments.

parameter	2D	3D
-flow_relaxation_mode	0 (SRT)	0 (SRT)
-discretization	d2q9	d3q19
-ncomponents	2	2
-component1_name	outer	outer
-mm_outer	1.0	1.0
-tau_outer	1.0	1.0
-component2_name	inner	inner
-mm_inner	1.0	1.0
-tau_inner	1.0	1.0
-g_11	0.0	0.0
-g_22	0.0	0.0
-g_12	0.1	0.1
-g_21	0.1	0.1
-rho_outer	0.97,0.03	0.97,0.03
-rho_inner	0.03,0.97	0.03,0.97
-bc_periodic_x	enabled	enabled
-bc_periodic_y	enabled	enabled
-bc_periodic_z	disabled	enabled
-walls_type	2	2

Table B.2: Parameters of Taxila LBM experiments from `tests/bubble_2D/input_data` and `tests/bubble_3D/input_data` files.

Bibliography

IPM: Integrated performance monitoring. http://ipm-hpc.sourceforge.net/. (cited on pages 24, 77, and 79)

NCI: National computational infrastructure. http://nci.org.au/~raijin/. (cited on page 23)

2015a. Solving the Bratu (SFI - solid fuel ignition) problem in a 2D rectangular domain. http://www.mcs.anl.gov/petsc/petsc-2.2.0/src/snes/examples/tutorials/ex5f90.F.html. (cited on page 59)

2015b. Solving the Bratu (SFI - solid fuel ignition) problem in a 3D rectangular domain. http://www.mcs.anl.gov/petsc/petsc-2.2.0/src/snes/examples/tutorials/ex14.c.html. (cited on page 59)

2015. Taxila LBM website. https://software.lanl.gov/taxila/. (cited on page 54)

Agbaria, A. M. and Friedman, R., 1999. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing (HPDC 1999)*, 167–176. doi:10.1109/HPDC.1999.805295. (cited on page 11)

Ajima, Y.; Sumimoto, S.; and Shimizu, T., 2009. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42, 11 (November 2009), 36–40. (cited on page 1)

Ali, M. M.; Southern, J.; Strazdins, P. E.; and Harding, B., 2014. Application level fault recovery: Using Fault-Tolerant Open MPI in a PDE solver. In *Proceedings of the IEEE 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2014)*, 1169–1178. Phoenix, USA. doi:10.1109/IPDPSW.2014.132. (cited on pages iii and 27)

Ali, M. M. and Strazdins, P., 2013. Algorithm-based master-worker model of fault tolerance in time-evolving applications. In *Proceedings of the Third International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO 2013)*, 40–47. (cited on page 2)

Ali, M. M.; Strazdins, P. E.; Harding, B.; and Hegland, M., 2016. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *International Journal of High Performance Computing Applications (IJHPCA 2016)*, 30, 3 (2016), 335–359. doi:10.1177/1094342015628056. (cited on pages iii, 5, and 43)

Ali, M. M.; Strazdins, P. E.; Harding, B.; Hegland, M.; and Larson, J. W., 2015. A fault-tolerant gyrokinetic plasma application using the sparse grid combination technique. In *Proceedings of the 2015 International Conference on High Performance Computing & Simulation (HPCS 2015)*, 499–507. Amsterdam, The Netherlands. (cited on pages iii, 5, and 43)

Allan, B. A., 2014. Memory reliability and performance degradation: Hunting rabbits with an elephant gun? In *Proceedings of the Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA 2014), IEEE Cluster 2014*. Madrid, Spain. (cited on page 85)

Ashby, S.; Beckman, P.; Chen, J.; Colella, P.; Collins, B.; Crawford, D.; Dongarra, J.; Kothe, D.; Lusk, R.; Messina, P.; Mezzacappa, T.; Moin, P.; Norman, M.; Rosner, R.; Sarkar, V.; Siegel, A.; Streitz, F.; White, A.; and Wright, M., 2010. The opportunities and challenges of exascale computing. Technical report, U.S. Department of Energy, Office of Science. (cited on page 6)

B. Harding and M. Hegland, 2013. A parallel fault tolerant combination technique. In *Proceedings of the International Conference on Parallel Computing (ParCo 2013)*, 584–592. Garching, Germany. doi:10.3233/978-1-61499-381-0-584. (cited on pages 16, 18, and 35)

Balay, S.; Abhyankar, S.; Adams, M. F.; Brown, J.; Brune, P.; Buschelman, K.; Eijkhout, V.; Gropp, W. D.; Kaushik, D.; Knepley, M. G.; McInnes, L. C.; Rupp, K.; Smith, B. F.; and Zhang, H., 2014. PETSc Web page. http://www.mcs.anl.gov/petsc. (cited on pages 23, 24, 54, and 59)

Bebernes, J. and Eberly, D., 1989. *Mathematical problems from combustion theory*. Springer-Verlag, New York, USA. (cited on page 59)

Benk, J. and Pfluger, D., 2012. Hybrid parallel solutions of the black-scholes PDE with the truncated combination technique. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2012)*, 678–683. doi:10.1109/HPCSim.2012.6266992. (cited on pages 15 and 17)

Bernstein, P. A. and Goodman, N., 1981. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13, 2 (June 1981), 185–221. doi:10.1145/356842.356846. (cited on page 10)

Bland, W., 2013a. User level failure mitigation in mpi. In *Proceedings of the 2012 Parallel Processing Workshops (Euro-Par 2012)*, vol. 7640 of *Lecture Notes in Computer Science*, 499–504. Springer Berlin Heidelberg. doi:10.1007/978-3-642-36949-0_57. (cited on pages 2, 13, and 28)

Bland, W.; Bouteiller, A.; Herault, T.; Hursey, J.; Bosilca, G.; and Dongarra, J. J., 2012. An evaluation of user-level failure mitigation support in

MPI. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI 2012)*, 193–203. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-642-33518-1_24. (cited on pages 17 and 28)

Bland, W. B., 2013b. *Toward Message Passing Failure Management*. Ph.D. thesis, University of Tennessee. (cited on pages 2, 13, 17, and 28)

Bosilca, G.; Bouteiller, A.; Cappello, F.; Djilali, S.; Fedak, G.; Germain, C.; Herault, T.; Lemarinier, P.; Lodygensky, O.; Magniette, F.; Neri, V.; and Selikhov, A., 2002. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of the ACM/IEEE 2002 Conference of Supercomputing*, 1–18. doi:10.1109/SC.2002.10048. (cited on page 12)

Bouteiller, A.; Lemarinier, P.; Krawezik, G.; and Capello, F., 2003. Coordinated checkpoint versus message log for fault tolerant MPI. In *Proceedings of the IEEE International Conference on Cluster Computing (IEEE Cluster 2003)*, 242–250. doi:10.1109/CLUSTR.2003.1253321. (cited on page 9)

Bungartz, H.-J. and Griebel, M., 2004. Sparse grids. *Acta Numerica*, 13 (2004), 147–269. (cited on page 13)

Chakravorty, S.; Mendes, C. L.; and Kalé, L. V., 2006. Proactive fault tolerance in MPI applications via task migration. In *Proceedings of the 13th International Conference on High Performance Computing (HiPC 2006)*, vol. 4297 of *Lecture Notes in Computer Science*, 485–496. Springer Verlag, Berlin, Germany. http://www.springerlink.com/content/9q840u6310467255/. (cited on page 10)

Coon, E. T.; Porter, M. L.; and Kang, Q., 2014. Taxila LBM: a parallel, modular lattice Boltzmann framework for simulating pore-scale flow in porous media. *Computational Geosciences*, 18, 1 (2014), 17–27. doi:10.1007/s10596-013-9379-6. (cited on page 54)

Dean, J. and Ghemawat, S., 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, 1 (January 2008), 107–113. doi:10.1145/1327452.1327492. (cited on page 10)

Eriksson, K.; Estep, D.; Hansbo, P.; and Johnson, C., 1996. *Computational Differential Equations*. Studentlitteratur, Lund, Sweden. (cited on page 46)

Fagg, G. E.; Bukovsky, A.; and Dongarra, J. J., 2001. HARNESS and fault tolerant MPI. *Parallel Computing*, 27, 11 (October 2001), 1479–1495. doi:10.1016/S0167-8191(01)00100-4. (cited on page 13)

Fagg, G. E. and Dongarra, J., 2000. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 346–353. Springer-Verlag, London, United Kingdom. (cited on pages 2, 12, and 28)

Fault Tolerance Working Group. Run-through stabilization interfaces and semantics. svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization. (cited on pages 13 and 28)

Ferreira, K.; Stearley, J.; Laros, J. H., III; Oldfield, R.; Pedretti, K.; Brightwell, R.; Riesen, R.; Bridges, P. G.; and Arnold, D., 2011. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2011, 1–12. ACM, New York, NY, USA. doi:10.1145/2063384.2063443. (cited on page 9)

Foster, I., 1995a. *Designing and building parallel programs*. Addison Wesley Publishing Company. (cited on page 71)

Foster, I., 1995b. *Designing and Building Parallel Programs*. Addison-Wesley. http://www.mcs.anl.gov/~itf/dbpp/text/book.html. (cited on page 72)

Gainaru, A.; Cappello, F.; Snir, M.; and Kramer, W., 2013. Failure prediction for HPC systems and applications: Current situation and open issues. *International Journal of High Performance Computing Applications (IJHPCA 2013)*, 27, 3 (August 2013), 273–282. doi:10.1177/1094342013488258. (cited on page 10)

Garcke, J. and Griebel, M., 2000. On the computation of the eigenproblems of hydrogen and helium in strong magnetic and electric fields with the sparse grid combination technique. *Journal of Computational Physics*, 165, 2 (2000), 694–716. doi:10.1006/jcph.2000.6627. (cited on page 16)

Garcke, J. and Griebel, M., 2001. On the parallelization of the sparse grid approach for data mining. In *Proceedings of the Third International Conference on Large-Scale Scientific Computing (ICLSSC 2001)*, 22–32. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/3-540-45346-6_2. (cited on page 75)

Geist, A. and Engelmann, C., 2002. Development of naturally fault tolerant algorithms for computing on 100,000 processors. http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf. (cited on page 7)

Gibson, G.; Schroeder, B.; and Digney, J., 2007. Failure tolerance in petascale computers. Software Enabling Technologies for Petascale Science, 3, 4 (November 2007), 4–10. (cited on page 1)

Görler, T., 2009. *Multiscale Effects in Plasma Microturbulence*. Ph.D. thesis, Universität Ulm. (cited on page 48)

Görler, T.; Lapillonne, X.; Brunner, S.; Dannert, T.; Jenko, F.; Merz, F.; and Told, D., 2011. The global version of the gyrokinetic turbulence code GENE. *Journal of Computational Physics*, 230, 18 (August 2011), 7053–7071. doi:10.1016/j.jcp.2011.05.034. (cited on pages 45 and 46)

Griebel, M., 1992. The combination technique for the sparse grid solution of PDE's on multiprocessor machines. *Parallel Processing Letters*, 02, 01 (1992), 61–70. doi:10.1142/S0129626492000180. (cited on pages 14 and 18)

Griebel, M.; Huber, W.; Rüde, U.; and Störtkuhl, T., 1992a. The combination technique for parallel sparse-grid-preconditioning or -solution of pdes on workstation networks. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing (CONPAR 1992)*, CONPAR 92 - VAPP V, 217–228. Springer-Verlag, London, UK, UK. doi:10.1007/3-540-55895-0_416. (cited on page 75)

Griebel, M.; Huber, W.; Störtkuhl, T.; and Zenger, C., 1993. On the parallel solution of 3d pdes on a network of workstations and on vector computers. In *Proceedings of the Parallel Computer Architectures: Theory, Hardware, Software, Applications*, 276–291. Springer-Verlag, London, UK, UK. doi:10.1007/978-3-662-21577-7_20. (cited on page 75)

Griebel, M.; Huber, W.; and Zenger, C., 1996. Numerical turbulence simulation on a parallel computer using the combination method. In *Proceedings of Flow Simulation on High Performance Computers II, Notes on Numerical Fluid Mechanics 52*, 34–47. (cited on page 18)

Griebel, M.; Schneider, M.; and Zenger, C., 1992b. A combination technique for the solution of sparse grid problems. In *Proceedings of Iterative Methods in Linear Algebra*, 263–281. IMACS, Elsevier, North Holland. (cited on page 14)

Harding, B. and Hegland, M., 2013. A robust combination technique. *ANZIAM Journal*, 54, 0 (2013). http://journal.austms.org.au/ojs/index.php/ANZIAMJ/article/view/6321. (cited on pages 35 and 73)

Harding, B.; Hegland, M.; Larson, J.; and Southern, J., 2014. Scalable and fault tolerant computation with the sparse grid combination technique. *ArXiv e-prints*, (April 2014). (cited on page 75)

Harding, B.; Hegland, M.; Larson, J.; and Southern, J., 2015. Fault tolerant computation with the sparse grid combination technique. *SIAM Journal on Scientific Computing*, 37, 3 (2015), C331–C353. doi:10.1137/140964448. (cited on pages 17 and 35)

Heene, M.; Kowitz, C.; and Pflüger, D., 2013. Load balancing for massively parallel computations with the sparse grid combination technique. In *Proceedings of the International Conference on Parallel Computing (ParCo 2013)*, 574–583. Amsterdam. (cited on pages 19 and 75)

Herlihy, M.; Eliot, J.; and Moss, B., 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 289–300. (cited on page 10)

Huang, K.-H. and Abraham, J. A., 1984. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33, 6 (1984), 518–528. doi:10.1109/TC.1984.1676475. (cited on page 10)

Hupp, P.; Jacob, R.; Heene, M.; Pflüger, D.; and Hegland, M., 2013. Global communication schemes for the sparse grid combination technique. In *Proceedings of the International Conference on Parallel Computing (ParCo 2013)*, 564–573. Garching, Germany. (cited on page 19)

Hursey, J. and Graham, R., 2011. Building a fault tolerant MPI application: A ring communication example. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW 2011)*, 1549–1556. (cited on page 3)

Hursey, J.; Squyres, J. M.; Mattox, T. I.; and Lumsdaine, A., 2007. The design and implementation of Checkpoint/Restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE Computer Society. (cited on pages 2 and 8)

Jenko, F. and the GENE development team, 2014. The GENE code. http://www.gene.rzg.mpg.de. (cited on pages 45 and 46)

Kowitz, C.; Pflüger, D.; Jenko, F.; and Hegland, M., 2012. The combination technique for the initial value problem in linear gyrokinetics. In *Proceedings of Sparse Grids and Applications*, vol. 88 of *Lecture Notes in Computational Science and Engineering*, 205–222. Springer, Heidelberg. (cited on pages 18 and 46)

Larson, J. W.; Hegland, M.; Harding, B.; Roberts, S. G.; Stals, L.; Rendell, A. P.; Strazdins, P. E.; Ali, M. M.; Kowitz, C.; Nobes, R.; Southern, J.; Wilson, N.; Li, M.; and Oishi, Y., 2013a. Fault-tolerant grid-based solvers: Combining concepts from sparse grids and mapreduce. *Procedia Computer Science*, 18, 0 (2013), 130–139. doi:10.1016/j.procs.2013.05.176. 2013 International Conference on Computational Science (ICCS 2013). (cited on pages 14 and 18)

Larson, J. W.; Strazdins, P. E.; Hegland, M.; Harding, B.; Roberts, S.; Stals, L.; Rendell, A. P.; Ali, M. M.; and Southern, J., 2013b. Managing complexity in the parallel sparse grid combination technique. In *Proceedings of the International Conference on Parallel Computing (ParCo 2013)*, vol. 25, 593–602. doi:10.3233/978-1-61499-381-0-593. (cited on page 22)

Lastdrager, B.; Koren, B.; and Verwer, J., 2001. The sparse-grid combination technique applied to time-dependent advection problems. *Applied Numerical Mathematics*, 38, 4 (2001), 377–401. doi:10.1016/S0168-9274(01)00030-7. (cited on page 53)

Lax, P. and Wendroff, B., 1960. Systems of conservation laws. *Communications on Pure and Applied Mathematics*, 13, 2 (1960), 217–237. doi:10.1002/cpa.3160130205. (cited on page 35)

Louca, S.; Neophytou, N.; Lachanas, A.; and Evripidou, P., 2000. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10, 4 (2000), 371–382. doi:10.1142/S0129626400000342. (cited on page 11)

MacCormack, R., 2003. The effect of viscosity in hypervelocity impact cratering. *Journal of Spacecraft and Rockets*, 40, 5 (2003), 757–763. doi:10.2514/2.6901. (cited on page 35)

Meaney, P. J.; Swaney, S. B.; Sanda, P. N.; and Spainhower, L., 2005. Ibm z990 soft error detection and recovery. *IEEE Transactions on Device and Materials Reliability*, 5, 3 (September 2005), 419–427. doi:10.1109/TDMR.2005.859577. (cited on page 85)

Message Passing Interface Forum, 1993. MPI: A message passing interface. In *Proceedings of Supercomputing*, 878–883. IEEE Computer Society Press. (cited on page 2)

Narsale, A. and Huang, M. C., 2009. Variation-tolerant hierarchical voltage monitoring circuit for soft error detection. In *Proceedings of the 10th International Symposium on Quality Electronic Design (ISQED 2009)*, 799–805. doi:10.1109/ISQED.2009.4810395. (cited on page 85)

Ndai, P.; Agarwal, A.; Chen, Q.; and Roy, K., 2005. A soft error monitor using switching current detection. In *Proceedings of the 2005 International Conference on Computer Design (ICCD 2005)*, 185–190. doi:10.1109/ICCD.2005.15. (cited on page 85)

Parra Hinojosa, A.; Kowitz, C.; Heene, M.; Pflüger, D.; and Bungartz, H.-J., 2015. Towards a fault-tolerant, scalable implementation of GENE. In *Recent Trends in Computational Engineering (CE 2014)*, vol. 105 of *Lecture Notes in Computational Science and Engineering*, 47–65. Springer International Publishing. doi:10.1007/978-3-319-22997-3_3. (cited on page 19)

Pauli, S.; Kohler, M.; and Arbenz, P., 2013. A fault tolerant implementation of multi-level Monte Carlo methods. In *Proceedings of the International Conference on Parallel Computing, (ParCo 2013)*, 471–480. Garching, Germany. doi:10.3233/978-1-61499-381-0-471. (cited on page 17)

Pflüger, D.; Bungartz, H.-J.; Griebel, M.; Jenko, F.; Dannert, T.; Heene, M.; Parra Hinojosa, A.; Kowitz, C.; and Zaspel, P., 2014. EXAHD: an exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *Proceedings of the 2014 Parallel Processing Workshops (Euro-Par 2014)*, vol. 8806 of *Lecture Notes in Computer Science*, 565–576. Springer International Publishing. doi:10.1007/978-3-319-14313-2_48. (cited on page 19)

Plank, J. S.; Li, K.; and Puening, M. A., 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distribed Systems*, 9, 10 (1998), 972–986. doi:10.1109/71.730527. (cited on pages 9 and 18)

Porter, M. L.; Coon, E. T.; Kang, Q.; Moulton, J. D.; and Carey, J. W., 2012. Multicomponent interparticle-potential lattice Boltzmann model for fluids with large viscosity ratios. *Physical Review E*, 86 (September 2012), 036701. doi:10.1103/PhysRevE.86.036701. (cited on page 54)

Sáez, X. and Soba, A., 2008. GENE. Installation guide and performance analysis. Technical report, Barcelona Supercomputing Center. http://www.bsc.es/media/1799.pdf. (cited on page 79)

Sato, K.; Moody, A.; Mohror, K.; Gamblin, T.; Supinski, B. R. d.; Maruyama, N.; and Matsuoka, S., 2014. FMI: fault tolerant messaging interface for fast and transparent recovery. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*, 1225–1234. IEEE Computer Society, Washington, DC, USA. doi:10.1109/IPDPS.2014.126. (cited on page 18)

Schroeder, B. and Gibson, G. A., 2006. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006)*, 249–258. IEEE Computer Society, Washington, DC, USA. (cited on pages 1 and 6)

Schroeder, B. and Gibson, G. A., 2007. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 012022 (2007). (cited on page 8)

Schroeder, B. and Gibson, G. A., 2010. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7, 4 (October 2010), 337–350. doi:10.1109/TDSC.2009.4. (cited on page 7)

Shan, X. and Chen, H., 1993. Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E*, 47 (1993), 1815–1819. doi:10.1103/PhysRevE.47.1815. (cited on page 54)

Shende, S. S. and Malony, A. D., 2006. The TAU parallel performance system. *International Journal of High Performance Computing Applications (IJHPCA 2006)*, 20, 2 (May 2006), 287–311. doi:10.1177/1094342006064482. (cited on pages 24 and 75)

Snir, M.; Wisniewski, R. W.; Abraham, J. A.; Adve, S. V.; Bagchi, S.; Balaji, P.; Belak, J.; Bose, P.; Cappello, F.; Carlson, B.; Chien, A. A.; Coteus, P.; DeBardeleben, N.; Diniz, P. C.; Engelmann, C.; Erez, M.; Fazzari, S.; Geist, A.; Gupta, R.; Johnson, F.; Krishnamoorthy, S.; Leyffer, S.; Liberty, D.; Mitra, S.; Munson, T.; Schreiber, R.; Stearley, J.; and Hensbergen, E. V., 2014. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications (IJHPCA 2014)*, 28, 2 (2014), 129–173. doi:10.1177/1094342014522573. (cited on pages 1, 2, and 8)

Stellner, G., 1996. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, 526–531. doi:10.1109/IPPS.1996.508106. (cited on page 11)

Strazdins, P. E.; Ali, M. M.; and Debusschere, B., 2016a. Application fault tolerance for shrinking resources via the sparse grid combination technique. In *Proceedings of the IEEE 30th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2016)*, 1232–1238. Chicago, Illinois, USA. doi:10.1109/IPDPSW.2016.210. (cited on pages iii and 27)

Strazdins, P. E.; Ali, M. M.; and Harding, B., 2015. Highly scalable algorithms for the sparse grid combination technique. In *Proceedings of the IEEE 29th International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2015)*, 941–950. Hyderabad, India. doi:10.1109/IPDPSW.2015.76. (cited on pages 22, 44, 51, and 74)

Strazdins, P. E.; Ali, M. M.; and Harding, B., 2016b. Design and analysis of two highly scalable sparse grid combination algorithms. *Journal of Computational Science. Special Issue on Recent Advances in Parallel Techniques for Scientific Computing (JCS 2016)*, (2016), 41 pages. doi:10.1016/j.jocs.2016.06.004. http://www.sciencedirect.com/science/article/pii/S1877750316301053. (In Press). (cited on pages iii, 22, 51, and 71)

Teranishi, K. and Heroux, M. A., 2014. Toward local failure local recovery resilience model using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA 2014, 51:51–51:56. ACM, New York, NY, USA. doi:10.1145/2642769.2642774. (cited on page 18)

Wright, N. J.; Pfeiffer, W.; and Snavely, A., 2009. Characterizing parallel scaling of scientific applications using IPM. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*. (cited on pages 24, 77, and 79)

Young, J. W., 1974. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17, 9 (September 1974), 530–531. doi:10.1145/361147.361115. (cited on pages 33 and 66)