universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Scalability and Fault Tolerance of ABFT Methods for Dense Matrix Multiplication“

verfasst von / submitted by

Svetoslav Inkolov, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 940
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Scientific Computing UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, MSc.

Abstract

The idea of algorithm-based fault tolerance (ABFT) is not new, it has its origins in the early '80s. This technique is used in computations with matrices which form the basis of many computationally-intensive tasks. As supercomputers consist of more and more components, their overall complexity increases, and many challenges arise that need to be handled. Therefore, the need for comprehensive fault detection and error correction algorithms became increasingly important over the last few years.

This precarious situation is mainly due to loss of stability when many hardware components come together in one system. In a small system or an average supercomputer, hardware parts are reliable enough even over a long period of time (months/years). On the other hand, current supercomputers (petaflop range) have tens of thousands of computational nodes with a mean time to interrupt (MTTI) of about one day. If we expand the calculations to a system at exaflop scale (next-gen supercomputers), this will eventuate in an MTTI of about 1 hour. Since in exascale platforms, there can be millions of nodes, the possibilities and scenarios of failure should be thoroughly tested before launching such systems in reality. The focus of this thesis is to examine how efficient and reliable ABFT methods can be implemented to work on dense matrix operations and to estimate their behavior towards exascale systems. This investigation is done by concentrating on the Local ABFT method where a general matrix multiplication (GEMM) is performed and where it is tested against insertions of bit flips during and after the GEMM. As an essential basis for the results DPLASMA, a highly optimized library for distributed hybrid systems was used. As an outcome, we have that a Local ABFT algorithm should be used in future supercomputers.

Another part of this thesis is concentrating on simulators. Nowadays exist simulators which can represent >100,000 of computational nodes with several million processors on a system which consists only of a few dozens of nodes. Of course, not all simulators are capable of simulating all possible situations, so the study focuses on the summary of their benefits and drawbacks in high-performance computing (HPC) context.

Zusammenfassung

Die Idee der „algorithm-based fault tolerance“ (ABFT) ist nicht neu, sie hat ihren Ursprung in den frühen 80er Jahren. Diese Technik wird bei Berechnungen mit Matrizen angewendet, welche die Grundlage für vieler rechenintensive Aufgaben bilden. Da Supercomputer aus immer mehr Komponenten bestehen, nimmt ihre Komplexität insgesamt zu, und es entstehen viele Herausforderungen die bewältigt werden müssen. Daher wurde in den letzten Jahren die Notwendigkeit von umfassenden Fehlererkennungs- und Fehlerkorrekturalgorithmen immer wichtiger.

Diese prekäre Situation ist hauptsächlich auf den Verlust der Stabilität, wenn viele Hardwarekomponenten in einem System zusammenkommen, zurückzuführen. In einem kleinen System oder einem durchschnittlichen Supercomputer sind Hardwareteile sogar über einen langen Zeitraum (Monate / Jahre) zuverlässig genug. Auf der anderen Seite, besitzen aktuelle Supercomputer (Petaflop-Bereich) Zehntausende von Rechenknoten, bei einer „mean time to interrupt“ (MTTI) von etwa einem Tag. Wenn wir die Berechnungen auf ein System im Exaflop-Maßstab (Supercomputer der nächsten Generation) erweitern, würde das in einer MTTI von etwa 1 Stunde resultieren. Da es in Exascale-Plattformen Millionen von Knoten geben kann, sollten die Möglichkeiten und Szenarien eines Systemausfalls gründlich getestet werden, bevor solche Systeme in der Realität gestartet werden. Der Fokus dieser Arbeit liegt auf der Untersuchung, wie effizient und zuverlässig ABFT-Methoden für dicht besetzte Matrizen implementiert werden können und ihr Verhalten gegenüber Exascale-Systemen abzuschätzen. Diese Untersuchung wird durchgeführt indem man sich auf die lokale ABFT-Methode konzentriert, bei der eine allgemeine Matrizenmultiplikation (MM) durchgeführt wird und mit Einfügungen von Bitflips während und nach der MM überprüft wird. Als wesentliche Grundlage für die Ergebnisse wurde DPLASMA, eine hochoptimierte Bibliothek für verteilte Hybridsysteme verwendet. Als Ergebnis haben wir, dass ein lokaler ABFT-Algorithmus in zukünftige Supercomputer verwendet werden sollte.

Ein weiterer Teil dieser Arbeit konzentriert sich auf Simulatoren. Heutzutage existieren Simulatoren welche > 100.000 Rechenknoten mit mehreren Millionen Prozessoren, auf einem System das nur aus ein paar Dutzend Knoten besteht, darstellen können. Natürlich sind nicht alle Simulatoren in der Lage, alle möglichen Situationen zu simulieren, daher fokussiert sich die Studie auf die Zusammenfassung ihrer Vor- und Nachteile im Zusammenhang mit „High Performance Computing“ (HPC).

Contents

Abstract I

Zusammenfassung III

Table of Contents V

List of Figures IX

List of Tables XIII

List of Listings XV

1 Introduction 1

1.1 Objective . . . 1

1.2 Motivation . . . 1

1.3 Synopsis . . . 7

2 Related Work 11

2.1 The Beginnings of ABFT and the First Modifications . . . 11

2.2 First Experiments on Systems with Multiple Processors . . . 12

2.3 Different Detection and Correction . . . 12

2.3.1 Other Detection Methods . . . 12

2.3.2 Improved ABFT Detection and Correction . . . 13

2.4 Outer Product Matrix Multiplication in combination with ABFT . . . 13

2.4.1 Experiments in ScaLAPACK . . . 13

2.4.2 Correction of Soft Errors On-The-Fly . . . 14

2.5 Other ABFT Applications . . . 14

2.5.1 ABFT in Cloud Computing . . . 14
2.5.2 ABFT in FPGAs and GPUs . . . 15
2.5.3 Blocked ABFT with Disk-Less Periodic Checkpointing . . . 15
2.6 Comparison of Master Thesis to Other Works . . . 16

3 Fault-Tolerant Methods 17
3.1 Errors, Faults, Fault Tolerance and Resilience . . . 17
3.2 ABFT Technique . . . 21
3.3 ABFT Methods for Matrix Multiplication . . . 24
3.3.1 Local ABFT . . . 26
3.3.2 Global ABFT . . . 27
3.3.3 Local vs Global ABFT . . . 29
3.4 Further Techniques used in HPC . . . 30
3.4.1 Checkpointing Techniques . . . 30
3.4.2 Composite Approach: ABFT & Checkpointing . . . 35
3.4.3 Fault-Tolerant MPI Approaches . . . 36

4 Simulators and other Tools in HPC 39
4.1 PDES Simulators . . . 39
4.1.1 X-Sim . . . 40
4.1.2 MuPI . . . 42
4.1.3 SST . . . 43
4.1.4 OMNeT++ . . . 44
4.2 Non-PDES Simulators . . . 45
4.2.1 Charm++ BigSim . . . 45
4.2.2 JCAS . . . 46
4.2.3 SimGrid . . . 47
4.2.4 GridSim . . . 47
4.2.5 DIMEMAS . . . 47
4.2.6 ns3 . . . 48
4.2.7 NetSim . . . 49

4.3 Tools for Using in Combination with Simulators or as Assistance 49
4.3.1 PARAVER 50
4.3.2 Vampir 51
4.4 Summary of Simulators and Tools 52
4.5 Towards Exascale Simulation 54
4.5.1 Exaflop PDGEMM 54
4.5.2 Simulators in Scientific Problems 57
4.6 Issues Using Simulators and HPC Libraries 59

5 Implementation **61**
5.1 DPLASMA 62
5.2 Local ABFT PDGEMM 65
5.2.1 Structure 66
5.2.2 Conditions 67
5.2.3 Correction Algorithm 68
5.2.4 Tolerance Value 70
5.2.5 Local ABFT Space Analysis 70
5.3 Fault Injector 71
5.3.1 Spatial Data Distributions 73
5.3.2 Temporal Data Distributions 73

6 Experiments **75**
6.1 Test Cases General 77
6.1.1 Value Distributions 77
6.1.2 No Errors Overhead 81
6.1.3 Manual Sign Bit Flips 83
6.2 Test Cases using Fault Injector 85
6.2.1 Sign Bit Flips with Fault Injector 85
6.2.2 Bit Flips in Mantissa 86
6.2.3 Bit Flips in Exponent 88
6.2.4 Bit Flips Everywhere 90

6.3 Conclusion to Bit Flips 92

7 Conclusion **93**

7.1 Future Work and Open Issues 95

Appendices **99**

A.1 Test Environment 99

A.2 Extended Space Analysis Local ABFT 102

A.3 Project Files 103

A.4 Software and Library Versions 104

B.1 Glossary 105

B.2 List of Acronyms 109

Bibliography **111**

List of Figures

1.1 Dennard scaling and evolution of processors [53] 4

1.2 Top 500 supercomputers performance development [100] 4

1.3 Error distribution for 3 processors with equally likely MTBF over a period t [20] 6

1.4 Error distribution of the platform for 3 processors over a period t [20] . 6

3.1 Tripartite graph and its decomposition to bipartite graphs for ABFT systems 23

3.2 Detection and correction in a Local ABFT approach on a (3×3)-process grid in the final computed matrix C^f 27

3.3 Detection and correction using a Global ABFT variant on a (3×3)-process grid in the final computed matrix C^f 28

3.4 Comparison of Local ABFT and Global ABFT for a (3×3)-process grid 29

4.1 xSim architecture and design [64] 41

4.2 SST algorithm scheme[141] 43

4.3 Combination of the tools Paraver and Dimemas [27] 50

5.1 Evolution of Dense Linear Algebra legacy software libraries[50] 63

5.2 DPLASMA software stack [50] 64

6.1 Different data distributions on various grids with fixed 200 sign flips per node and relative 1-norm illustration 78

6.2 Different data distributions on various grids with fixed 200 sign flips per node and verification of the correction time 79

6.3 Different data distributions on various grids with fixed 200 sign flips per node and performance evaluation in GFlop/s 79

6.4 Different data distributions on various grids with fixed 200 sign flips per node total time comparison 79

6.5 Different data distributions on various grids with fixed 200 sign flips per node relative overhead compared to DPLASMA_dgemm 80

6.6 Comparing ABFT_PDGEMM to DPLASMA_dgemm when no faults occurred during runtime measuring 81

6.7 Comparing the ABFT matrix-matrix multiplication algorithm to the DPLASMA_dgemm when no faults occurred with focus on relative overhead 82

6.8 ABFT_PDGEMM relevant time in seconds for various grids when no faults are injected 82

6.9 Relative 1-norm for various manual sign bit flips 83

6.10 Various number of sign flips per node for relative overhead in percent . 83

6.11 Time comparison with focus on the matrix size for various number of sign flips per node 84

6.12 Time comparison to the DPLASMA_dgemm routine for various number of sign flips per node with emphasis on the node size 84

6.13 Box plot for Fault Injector bit sign flips 85

6.14 Relative 1-norm of 5 independent test runs when using Fault Injector for bit sign flips 85

6.15 Comparing ABFT_PDGEMM to DPLASMA_dgemm when sign bit flips are inserted on different grid sizes 86

6.16 Box plot for bit flips in the mantissa using the own-threaded Fault Injector 87

6.17 Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips in mantissa 87

6.18 Relative overhead when bit flips are inserted in the mantissa for different grid sizes 87

6.19 IEEE 754 Double-precision floating-point format [143] 88

6.20 Example of a small matrix-matrix multiplication with fault injection in the exponent on a (2×2)-processor grid 88

6.21 Box plot for bit flips in the exponent using the Fault Injector 89

6.22 Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips in the exponent of a value 90

6.23 Relative overhead when bit flips are inserted in the exponent for different grid sizes 90

6.24 Box plot for bit flips all over the bit mask using the Fault Injector . . . 91

6.25 Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips everywhere in the bit mask of a value 91

6.26 Relative overhead when bit flips are inserted truly randomly for different grid sizes 91

A.1 Repeal2 hardware topology from execution of lstopo from HWLOC part 1 (NUMA-node 0-1) 100

A.2 Repeal2 hardware topology from execution of lstopo from HWLOC part 2 (NUMA-node 2-3) 101

A.3 Local ABFT PDGEMM extended space analysis for # of elements and # of bytes plus regarding legend 102

List of Tables

1.1 Towards Exascale Computing Roadmap [20, 53] 2

3.1 Errors, faults and failures in a system 17
3.2 Advantages and disadvantages of Local ABFT and Global ABFT . . . 30
3.3 Checkpointing techniques for HPC applications [74, 20, 21] 30

4.1 Summary of simulators and tools towards exascale 52
4.2 Summary of simulators and tools towards exascale continued 53
4.3 Minimum amount of RAM required for simulating an Exaflop machine without communication and calculation overhead 55
4.4 Minimum amount of RAM required for simulating an Exaflop machine (not fault-tolerant) 56
4.5 Supercomputer and equipped physical RAM in terabyte [106] 56
4.6 Minimum amount of RAM required for simulating an Exaflop machine (fault-tolerant version) 57

5.1 Summary of DPLASMA's capabilities [50] 62
5.2 Setting the Fault Injector with different MTBF 74

6.1 Verification of the tile size responsible for unusual behaviour 78
6.2 Summary of relevant values for matrix value data distribution [−1E4, 1E4] 80
6.3 Summary for overhead on a (10×10)-processor grid when no errors occurred 82
6.4 Correction time for various number of errors per node 84

A.1 Testing environment details 99
A.2 Total system metrics 99

A.3 Summary and description of relevant project files 103
A.4 Summary and description of used libraries and packages 104

Listings

5.1 Matrix Matrix Multiplication using DPLASMA 65
5.2 Definition of Fault Injector 72
6.1 Generator for uniformly distributed double precision values 75
6.2 How FLOPS for a DGEMM are defined and Gflop/s calculated 76

Chapter 1

Introduction

1.1 Objective

This master thesis emphasise the problems occurring when trying to reach extreme-scale computing platforms, especially in the high performance and scientific computing sector. The primary goal was to examine in this context algorithm-based fault tolerance (ABFT-) methods deeply and provide additionally an overview of fault tolerance techniques and methods which can be used for High-Performance Computing (HPC) purposes in supercomputers. Further, a proposal of which tools and methods can act as a helping hand to the programmers and which are not so appropriate to use towards exascale computing is shown. The results have been investigated using the highly optimized library DPLASMA which stands for Distributed Parallel Linear Algebra Software for Multicore Architectures [18]. For producing plausible data, a parallel double precision valued general matrix-matrix multiplication (PDGEMM) with an embedded *Local* ABFT method was performed. Supplementary, different types of errors were spatially and temporally inserted by two specific *Fault Injectors* to inspect the fault tolerance of the algorithm. With this particular combination of tools and methods, it was possible to investigate the advantages and the disadvantages of ABFT methods for dense matrix operations thoroughly.

1.2 Motivation

When we look at supercomputing history, mainly when computational-intensive tasks have to be performed, fault tolerance had always been playing an important role. The first techniques used in this context were based on TMR (Triple Modular Redundancy) [127, 3]. One of the first known fault-tolerant computer using TMR was SAPO (in 1950s), where its basic design was based on magnetic drums connected via relays, with

a voting method of memory error detection [3]. In general, the early efforts at fault-tolerant designs were mainly focused on internal diagnosis by an operator/technician doing monitoring and reacting to signals [114]. The actions which are usually taken into account had mainly to do with replacing a module or a circuit card. Later efforts showed that to be fully actual, the system had to be self-repairing and diagnosing which leads to the N-modular redundancy technique [144].

On the other hand for the average home user, the components of a personal computer are so resilient that there is usually no need for a data backup even over a long period. This leads us to one of the crucial points in this master thesis which is how long a PC component can remain without failures? Here especially pointing out to the term MTBF (Mean Time Between Failure) of a single hardware component used in such a way that no additional fault-tolerant technique is needed.

In a single PC, every hardware device has its term in where it is specified that no failures should happen within this particular time window. An excellent and often used example in this context is a hard disk. Let us assume that a hard drive has an MTBF of around 100 years [126] so it might take up to 100 years before an error occurs. It's obvious that if such a situation is the case, the user of this PC with this hard disk doesn't have to bother or think neither of fault tolerance nor of some error-tolerant techniques.

Systems	**2009**	**2011**	**2016** (Sunway TaihuLight) [55]	**2020** (may be 2023)
System peak	2 PFlop/s	20 PFlop/s	125.4 PFlop/s	1 EFlop/s
System memory	0.3 PB	1.6 PB	1.31 PB	32 - 64 PB
Node performance	125 GF/s	200 GF/s	3.06 TF/s	1.2 or 15 TF/s
Node memory BW	25 GB/s	40 GB/s	32 GB/s	200 - 400 GB/s
Node concurrency	12 cores	32 cores	260 cores	$\mathcal{O}$(1k)–$\mathcal{O}$(10k)
Interconnect BW	1.5 GB/s	22 GB/s	16 GB/s	50 GB/s
System size(nodes)	18,700	100,000	40,960	$\mathcal{O}(100,000)$ or $\mathcal{O}$(million)
Total concurrency	225,000	3,200,000	10,649,600	$\mathcal{O}$(billion)
Storage	15 PB	30 PB	20 PB	$>$ 100 PB
IO	0.2 TB/s	2 TB/s	$\sim$ 10 TB/s	$>$ 10 TB/s
MTTF	4 days	19 h 4 min.	Few / day	$>$ 5 per / day
Power	6 MW	$\sim$ 10 MW	$\sim$ 15 MW	$\sim$ 20 MW

Table 1.1: Towards Exascale Computing Roadmap [20, 53]

GB = Giga Bytes TB = Tera Bytes BW = Bandwidth PB = Peta Bytes
GF = Giga Flop/s TF = Tera Flop/s MW = Mega Watts

Supercomputers and other big servers usually have thousands or can even have millions of components. In Table reftab:tow-exa-table, we can see the development of such supercomputers over the last decade and what are the expectations from the next generations' system, the exascale machine (10^{18} Flop/s). First, when we take a look at the system size row, we can see that in a current system we have a number of nodes in the order of about 100k, whereas for an Exaflop machine $\mathcal{O}$(million) is expected. For the total concurrency row, we can see that for supercomputers in 2009 the total number of components were about 200,000 and an exascale system will probably have billions of parts. With this massive amount of devices in such a system, there are also many other problems arising. Besides building of resilient hardware there are among others software problems (redesigning of algorithms), machine design problems (lightweight / hybrid machine) and optimizing the performance per watt which have to be considered.

This whole situation has partly to do of course with work distribution and efficiency but mainly with the fact that it is not possible just to make the programs faster, for instance by raising the frequency of one component (speaking of CPUs). Although Moore's Law, that the number of transistors doubles approximately every two years still holds, Dennard scaling with the assumption that voltage and current should be commensurate to the linear dimensions of a semiconductor device was not the case [53]. The crucial point was the power that has to be handled later on. As the power is defined to be proportional to the frequency to the power of 3, so if we have a chip of running at 3.0 GHz with 100 watts and want to double its hertz rate to 6.0 GHz the power which has to be managed by the cooling will be of about 800 watts. Thus, as a consequence, the power consumption will rise rapidly and secondly, the cooling will fail. The power cost situation of frequency [53] nowadays looks like:

$$\begin{aligned} \text{Power} &\propto \text{Voltage}^2 \times \text{Frequency} \quad (V^2F) \\ \text{Frequency} &\propto \text{Voltage} \\ \text{Power} &\propto \text{Frequency}^3 \end{aligned} \tag{1.1}$$

Summarized we can say that we can gain 50 percent of performance by using a multicore architecture which consumes 20 % less power because the voltage and the frequency is decreased.

For CPUs, the point where Dennard scaling could not meet its properties anymore was in 2004 (Figure 1.1). From this moment on the current leakage depict more significant challenges, and also causes the chip to heat up to a scale that can not easily be handled with knowing cooling methods. The heat situation goes that far, that if we take the last example of the 6.0 GHz from the point of Watts/cm^2 the CPU

will create a heat dissipation compared to a rocket nozzle [93]. Thus today's cooling mechanisms have no chance to get rid of the overwhelming heat dissipation which will occur when rising the clock of the processors even further. Therefore the multicore era was introduced.

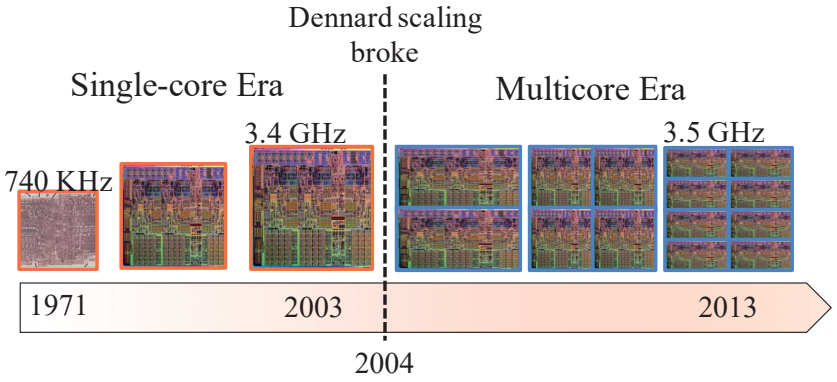


Figure 1.1: Dennard scaling and evolution of processors [53]

If we take a look at how supercomputers have developed over the past few decades, we can observe that they are on the brink to reach the next ultimate level of computational power (Figure 1.2). For the exascale level, it is expected with high probability

Figure 1.2: Top 500 supercomputers performance development [100]

that the built-in components which have to be controlled here are going in the range of millions or billions. In Table 1.1 we can further see in which order the hardware have to change and with which other challenges the future engineers will have to get rid of, making the situation even more exhaustive and complicated. Figure 1.2 shows the floating point operations performance of the slowest supercomputer (#500), the fastest supercomputer (#1) and the summed up performance in the corresponding year. From this view, we can also see that in 1998 the TFlop/s border and in 2009 the Petaflop border passed, whereas in the extended projection up until 2020 the expected exascale should be reached.

This brings us to the essential point why fault tolerance is needed. Again in Table 1.1 we can find the term MTTF which stands for mean time to failure and gives us a boundary of how stable the system is when significant errors occur [74, 53, 20]. MTTF is defined as:

$$\text{MTTF} = \frac{\text{Uptime}}{\#\text{Failures}} \tag{1.2}$$

In 2009 the machine used in the table had an MTTF of about four days. This means that one significant error, where the whole system/computation has to be interrupted and data restored from the previous checkpoint or backup, happened every 4 days. In 2011 this occurs approximately every 19 hours and in 2016 every 4-6 h. The problem which arises here is that for an exascale machine it is expected that more then 5 such interruptions occur per day or that the associated MTTF value is even worse. The MTBF value of a single component plays a significant role here. Let us assume we have an MTBF of 100 years for a computing node, which of course is a neater value but when we have to calculate the MTBF_p[74] which means the MTBF for the whole platform or system then it's getting pretty exciting. The MTBF of a platform , where μ_{ind} is the MTBF of one individual processor or node and N is the number of identical components, is then expected to be:

$$\text{MTBF}_p = \frac{\mu_{ind}}{N} = \frac{\text{MTBF of 1 component or node}}{\#\text{components or nodes}} \tag{1.3}$$

Further assume an exascale machine with 1 million nodes. From that, it follows that if we have for all involved components the same MTBF we can calculate the MTBF for the whole platform or system using this example calculation:

$$\text{MTBF}_p = \frac{\text{100 years}}{\text{1,000,000 nodes}} = \frac{\text{876,000 h}}{\text{1,000,000 nodes}} = 0.876 \text{ h} \approx 53 \text{ min.} \tag{1.4}$$

This comes from the intuition that if we have 100 years for one component this could happen 1 million times faster for 1 million nodes. The situation can be explained by the following example where we have 3 processors with identical MTBF for each

processor and 20 faults within a time window t (see Figure 1.3 and 1.4). The decisive

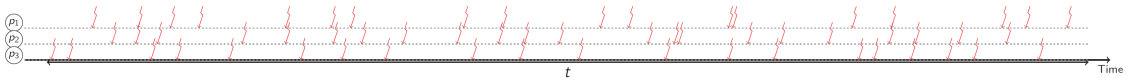


Figure 1.3: Error distribution for 3 processors with equally likely MTBF over a period t [20]

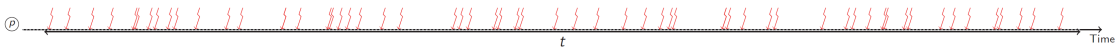


Figure 1.4: Error distribution of the platform for 3 processors over a period t [20]

point is that the 20 faults can happen arbitrarily over a period t and so when the error occurings of the 3 processors are overlapped we see the result in Figure 1.4. It turns out that seeing it arbitrarily the faults which can occur in a particular time window are summing up to 60 and therefore the MTBF_p is derived $\frac{1}{3}$ of the MTBF of one component (Equation 1.3). Hence we can see in the exascale example (from Equation 1.4) a sudden drop of the MTBF to 53 minutes which is dangerously low for current checkpointing and backup mechanisms. Nowadays without optimization, we can say that a checkpoint-restart needs about 1h. This value comes from the time for doing a checkpoint (∼30 minutes) plus the time for doing the restart operation which also lasts about 30 minutes [20].

That's the point where additional fault tolerant techniques like ABFT can come into action. Besides *fault resilient algorithms* and methods there are also further critical issues [53, 54] which have to be taken into consideration for an exascale machine to work correctly and to be maintainable. In this context the following ones are usually mentioned:

- *Communication-reducing algorithms*: the idea here is to formulate algorithms in other ways such that these methods achieve a lower bound on communication. Further, it has to do with observation of the algorithms and defining what the minimum communication for a particular algorithm can be.

- *Synchronization-reducing algorithms*: using a different kind of synchronization instead of the Fork-Join model, e.g., apply directed acyclic graphs as managing instance in the setup and execution of parallel programs and so avoid the majority of synchronous processing (like it is done in PaRSEC).

- *Mixed precision methods*: using a lower precision e.g., 16-bit instead of 32-bit or 32-bit instead of 64-bit for a particular part of the application and so speeding up operations and data movement of this section.

- *Auto-tuning*: the idea of putting an intelligent auto tuner into the software so that the user do not have to worry about the complexity and the multiple levels of a hardware environment. Further, the tuner should not be static (predefine parameters and use for all future computations) but should run within an application and adjust the required parameters dynamically.

- *Reproducibility of results*: nowadays it is difficult to guarantee that running the same problem on the same machine in a successive order produce the same results (e.g., because of round-off errors or bit flips etc.). Therefore there exists, on the one hand, some error bounds and other guide values which should help to evaluate the correctness of an execution but on the other hand, there are still applications which need to produce always the same result. In this context, the demanded reproducibility can only be achieved by more computationally expensive operations. Furthermore, if someone wants to do debugging this replicability also plays a major role [53, 54].

Another important part of this master thesis is dealing with simulators and other tools in HPC (Chapter 4). The decision which leads to the investigation of such tools was that there are many situations where the simulation of a specific problem can save a lot of costs and further can reveal design issues. When we think of the hardware design of next-gen supercomputers, it might be that another network topology is needed to assure the best communication between the computational nodes. In this case, a simulation tool (like presented in 4.2.6) can estimate such a situation and calculate the needed facts for building the desired supercomputer. On the other hand, it might be necessary that engineers want to know if their application can be adapted to the upcoming generation of supercomputers. Such a situation can be simulated on a tool using on-line simulation like in (4.1.1). Another application might be that someone wants to analyze large or multiple tracing files. Here besides the simulators, an additional tool (e.g., Paraver 4.3.1) can be used to achieve this goal. Therefore such helping tools are of great interest, especially for the HPC sector.

1.3 Synopsis

Chapter 2 is dealing with the related work. The summary was primarily made by the historical occurrence of ABFT methods in the literature. First, the early days of the ABFT technique are discussed and supplemented by the different encoding schemes of the ABFT methods (weighted or partitioned encoding). Then, the first investigation on a multiple processors machine and other fault detection methods is given. After that, a popular variant regarding ABFT and matrix operations, the outer product

version of the matrix multiplication is discussed. Furthermore integrating ABFT methods on different devices and a unique correction technique is mentioned. Then a combination approach of ABFT and Checkpointing is stated. Last but not least an outline of the differences to other research activities is given.

In Chapter 3 the emphasis lies on the fault-tolerant methods. Firstly an introduction to errors, faults, fault tolerance, and resilience is given. Then it is continued with the focus on the ABFT technique and ABFT methods for matrix multiplication. Therefore the *Local-* and the *Global* ABFT method is presented and compared in detail. Furthermore, the techniques which are commonly used in high-performance computing are discussed. Therefore there exist a variety of checkpointing techniques with their benefits and drawbacks which are used to achieve a resilient computation. Finally, the two common fault-tolerant MPI approaches are stated.

In Chapter 4, it is continued with simulators and other tools for HPC purposes. Here the focus was to find a simulator which can be used towards simulation of a problem in exascale range. Further, it is investigated if one of the tools can directly work with one of the two highly optimized libraries for linear algebra (DPLASMA or ScaLAPACK). Unfortunately, the study revealed that there is no simulation tool yet available which is capable of executing an application without extensive modification of the source code. For instance in the case of xSim (subsection 4.1.1) the inner functions of a main routine in an HPC library have to be extracted and executed without nesting, whereas in the case of Charm++ (subsection 4.2.1) additionally handler functions like `BgSendPacket(int x, int y, int z, int threadID, int handlerID, WorkType type, int numbytes, char* data)` has to be used [89]. The handler function `BgSendPacket` sends a portion of data to the Node[`x,y,z`] and also specifies the responsible handler function for this message. Further, a `threadID` has to be given (pointing on the desired thread to handle the message). Other simulation tools have even other specific expectations and thus using simulators in combination with applications is not straightforward. Further, in this chapter, an assessment of how a *Local ABFT* PDGEMM simulation would look like in an exascale computation is presented. The chapter is then concluded by the issues when using simulators and HPC libraries together.

Chapter 5 is mainly about the implementation of the testing application. Here the DPLASMA library, the *Local ABFT* method, and the fault injecting tool are discussed. The structure and the conditions of this particular PDGEMM are given in Section 5.2. In this section also the correction algorithm and the detection depending on the tolerance value is stated. Further, a space analysis of the *Local ABFT* method is given. The last part of this chapter deals with the value manipulations in space and in time. Therefore a special *Fault Injector* which is capable of inserting bit flips in the

bit mask of a matrix element was used. The insertion is based on a uniform discrete distribution in the sub-matrix array and on an exponential time distribution with a MTBF-value which can be specified. Furthermore, with this own-threaded tool, it is possible to manipulate the bit-mask of a value regarding the mantissa, the exponent or a random bit flip.

The next big part consists of the experiments and numerical results. They are located in Chapter 6 and can be summarized as follows:

- The first part of the test cases is about different value distributions. Here values of various uniform distributions are analyzed. There were two scientific finding. One was that some particular tile-size can cause a numerical instability and thus it might happen that not all bit flips are corrected in such a situation (Table 6.1). Another recognition was that the unique distribution of values between 0 and 1 had better correction results compared to the relative 1-norm of other distributions.

- The second test case includes a comparison to the standard `DPLASMA_dgemm` routine when no errors occur. Here it was shown that the ABFT specific part remains relatively constant when the matrix size grows.

- The third point is about sign bit flips where two different fault injectors were used. The scientific insights from this experiments is that sign bit flips can be corrected in almost all of the cases and that they are having least impact on the whole computation.

- Further, the fourth test case is analyzing bit flips in the mantissa, whereas the fifth test case has to do with bit flips in the exponent. If we can envision a ranking for the effects on a matrix multiplication, besides the sign bit flips, bit flips in the mantissa are the ones that are the second most difficult to detect and correct, and then the bit flips in the exponent are the most difficult.

- The last part of the test cases is investigating the random bit flips situation (including the opportunity of producing errors in the mantissa, exponent or sign bit flips in the values' bit mask). Here the situation regarding detection and correction is a kind of between the bit flips in the mantissa and in the exponent.

Furthermore, in Chapter 6, there exist tables showing the GFlop/s performance, relative overhead and time comparison accompanied by figures. The results are showing a clear trend, which is also emphasized in other works from the literature. As long as the number of computational nodes is growing the total overhead against the standard

`DPLASMA_dgemm` is shrinking. This situation is because of the structure of the *Local* ABFT method which has the advantage that the correction can be done locally by each node or processing unit. When a large matrix multiplication is calculated on a few nodes, the sub-matrices are big, and the overhead for each node or processor is significant. On the other hand, when there are more nodes available for the computation, the sub-matrices are small, and the overhead for the recomputation of the faulty values is decreasing. If the matrix problem size also increases, this profitable situation is even more noticeable.

In the last chapter, the conclusion- and future work points are listed. Especially in the future work section there exist several suggestions on how this work can be extended and investigated afterward. The topics go from integrating a checkpointing technique from Section 3.4 in the ABFT method, to extending one chosen simulation tool by an ABFT technique or implementing a fully fault-tolerant version of the matrix multiplication (where all involved matrices can tolerate errors).

Chapter 2

Related Work

Over the last three decades, there has been much research on ABFT for matrix operations. Initially, in the first publication about ABFT [76] by Kuang and Hua, the dimensions of the matrices, ergo the problem size was not playing such a prominent role at that time. What they want to emphasize especially in the context of matrix multiplication was the redundancy ratio, and of course the fault tolerance which was provided when using this technique. The ABFT technique and its various scientific applications is discussed in Section 3.2.

2.1 The Beginnings of ABFT and the First Modifications

Kuang and Hua showed at the very beginning experiments with matrix sizes between 10 and 1000 and further stated that the redundant hardware needed in higher dimensions is very low. This is, of course, dependent on the desired resilience, if many errors should be corrected more hardware components are necessary, and if the security should withstand only a few faults then only a small percentage of redundant parts are needed. We are speaking here of about 10 percent of overhead, whereas the most used techniques at that time TMR and quaded logic, require an additional amount of hardware at least of factor 2 or 3. Shortly afterwards, the authors of [83] developed a weighted encoding scheme, which extended the original ABFT encoding by the possibility that many occurring errors can be corrected by using more than one additional vector with checksums. This was another milestone, which also influences the method used in this thesis and on many other research projects. At this point, the partitioned encoding scheme by [123] has to be remarked too, which introduced another great feature where each node or processor can have its sub-matrix with local

checksums, one of the primary advantages of *Local ABFT* [1].

2.2 First Experiments on Systems with Multiple Processors

One of the first experimentation on a multiple processors machine was done by the authors of [6]. The experiments were done a hypercube multiprocessor system, where 4, 8 and 16 processors were compared together. It was shown that even for relatively small matrix sizes larger than (96×96) the overhead regarding the execution time of ABFT with full checking is below 20 percent. The results were evaluated with single precision and also double precision values in the matrices. Further different types of errors like transient bit- and transient word errors, and permanent bit- and permanent word errors were tested on their detection ability. A similar result was published in [125] where a single fault location and including a recovery method was observed. In their algorithm-based scheme, they achieve a 20 percent of time overhead for a (192×192)-matrix multiplication, where an error was successfully detected and corrected.

2.3 Different Detection and Correction

2.3.1 Other Detection Methods

The authors in [71] try to measure the overhead by using different detection methods. They present in their work a right-sided-, left-sided- and a two-sided error detection method, where first a matrix D is computed by $D = A \times B$. The right-sided detection method has to fulfil the condition of the computed $\tilde{D}$ with w as a checksum vector and where $||\tilde{D}w - A(Bw)||_\infty < \tau||A||_\infty||B||_\infty$. If this is valid then $C = \alpha D + \beta C$ is performed, if not D is recomputed again. The left-sided detection method on the contrary has the condition: $||v^T\tilde{D} - (v^TA)B||_\infty < \tau||A||_\infty||B||_\infty$, where v^T is a checksum vector. Here again if this is true then $C = \alpha D + \beta C$ is performed, if not D is recomputed. Finally, there is the two-sided error detection, which is a combination of the two techniques before and and should help to find all undetected errors. With this techniques, the authors performed matrix products up to a (512×512)-matrix, where at the biggest dimension the right-sided variant has the highest correction overhead, and the two-sided approach the highest detection overhead, but all methods achieve below six percent overhead.

[1] For a detailed description of the encoding schemes see Section 3.3.

2.3.2 Improved ABFT Detection and Correction

In [110] an improvement of the standard ABFT correction capabilities was developed, such that every possible bit flip can be restored. Usually, bit flips in the exponent of an integer value or floating-point value can be fatal when they occur at a certain place in the bit mask, where they cause a large numerical change. Since the used variable can be defined with different precision, this decision determines the numerical boundary. In a correction of an error or bit flip the limit is given by the corresponding machine epsilon. A value may be changed by internal or external manipulation from 2^1 to 2^{100}, and therefore it cannot be corrected by standard checksum comparison. Their approach goes in the direction that the erroneous value 2^{100} is changed to 0 (zero). Thus a correcting algorithm efficiently can recompute the value from 0 to the origin 2^1. Of course, this is also valid for randomized bit flips which are part of real-life applications. Therefore with their optimized variant, it is ensured that even values like `NaN` and `infinity` can be corrected with no remarkable overhead.

2.4 Outer Product Matrix Multiplication in combination with ABFT

The outer product version of the matrix multiplication, on the contrary, is a popular variant which has been used in several research activities for enabling the feature of handling a failure at any time during the multiplication. This matrix multiplication technique uses an iteration based update of the final full checksum matrix C^f.

2.4.1 Experiments in ScaLAPACK

One of the first experiments on a 2-D block-cyclic matrix data structure [2] with the high-performance library ScaLAPACK were done by [36]. Here the authors included a *Global* ABFT approach (subsection 3.3.2) in combination with the fault tolerant MPI API (subsection 3.4.3) such that on the one hand single node failures can be handled and on the other erroneous data caused by bit flips can be recalculated. The ScaLAPACK library makes it possible to work with a big variety of parallel architectures [9]. Thus their work refers also to square matrices of dimension $(n \times n)$ which are distributed over a mesh of P times Q processes, like in this master thesis. The difference is of course in the using of the *Global ABFT* method, where in their variant a single process failure can be recovered at each matrix update iteration step, whereas in the *Local ABFT* only bit flips are manageable. Further in their work, the overhead

[2] This is also the major data scheme in this thesis.

for calculating, performing computations and recovery on the encoded matrices was analysed. They have in all three analysing points a time overhead below 10 percent, where the overhead decreases with rising matrix problem size (in the direction to $n = 32000$). Last but not least a remarkable result was achieved on time to recover the FT-MPI (single node failure). In all three matrix cases, it was below 1%.

A bit later [37] show that their approach can be effectively used for fail-stop failures (stopped processes where all data to the corresponding failed process are lost). There the experiments were extended up to a (10×10)-process grid with similar results to their previous research [36].

The outer dot product was also used in [22]. Bosilca et al. also show the sloping curve when the number of processes rise. In their publication, the overhead of the ABFT version of the matrix multiplication against the PBLAS PDGEMM from ScaLAPACK falls below 10 percent at about 400 processes. Their further research shows even a drop to about 5 percent at 1024 cores.

2.4.2 Correction of Soft Errors On-The-Fly

An approach to correct soft errors on-the-fly was presented by Wu et al. in their research about an online fault-tolerant matrix-matrix multiplication [148]. The online variant was achieved by using an extension of the outer dot product, where the part of using rank one updates in the algorithm, was replaced by a changed version with rank k updates. Further, a checking of the checksum relationship of the partial result is done periodically (e.g. time interval of 10 seconds) to guarantee that it is online. With their extension, they showed that the performance does not drop by more than 5% for up to 20 soft errors compared to an ATLAS DGEMM. This also helps with error propagation and avoiding corrupted computations.

2.5 Other ABFT Applications

2.5.1 ABFT in Cloud Computing

The authors of [4] tested an ABFT matrix multiplication on cloud computing. As a testing environment the Amazons EC2 was chosen, and they try to find out if it is suitable for scientific computing purposes. Unfortunately, at that time the implemented fault-tolerant method was at least an order-of-magnitude worse performance than the same implementation which was executed on a dedicated scientific cluster. This was mainly due to service offering constraints, and because the priority of EC2 was on web-based- and business applications. Further, the tuning of the application to

run on virtualized resources can be challenging. In the meanwhile, Amazon is offering so-called *Spot Instances* [3] which can be used for scientific needs.

2.5.2 ABFT in FPGAs and GPUs

Other research projects with ABFT methods are going in the direction of integrating into FPGAs, GPUs or a combined version with checkpointing. In [15] a composite ABFT & Periodic Checkpointing approach was analysed and it was shown that it wastes much fewer hardware resources than using only *Periodic Checkpointing* or *Bi-Periodic Checkpointing* for fault tolerance. This approach is discussed more detailed in subsection 3.4.2.

A highly efficient blocked outer product matrix multiplication algorithm on GPGPUs was presented by [47]. This algorithm copes with memory soft errors on-line as described in previous on-line variants. They achieve an average overhead of 20 percent, where the blocked ABFT update variant worked much better compared to the traditional ABFT update like in [148] for CPUs. Their GPU blocked method achieves significantly better results compared to the CPU variant because the algorithm is differently parallelized (internal cache, shared memory) and the time for one floating-point arithmetic operation is also different on the various devices.

An FPGA approach based on ABFT was shown in [150]. They provide a lightweight concurrent AES (Advanced Encryption Standard) error detection scheme with exploiting the properties of ABFT. There again two versions of error detection were considered. The first deals with the whole AES process and the detection occurs at the end (after all rounds passed and, whereas in the second version in every round an error detection is performed. Furthermore, the results have been done on a Xilinx FPGA board.

2.5.3 Blocked ABFT with Disk-Less Periodic Checkpointing

At this point, it has to be remarked that a combination where a *Local ABFT* method was used with DPLASMA (like in this thesis) has only a closer relation to the work done in [32]. There has also been used a sub-matrix extended structure, where the checksums are distributed over all processing nodes. They presented two application-level mechanisms, including a sub-DAG (Directed Acyclic Graph), sub-DAG & checkpointing approach and a task level mechanism (Local ABFT). Since the tasks in DPLASMA or PaRSEC are executed in a DAG-based order, the sub-DAG mechanisms are responsible that failures can be handled by re-executing a minimum of the

[3] For details see: https://aws.amazon.com/ec2/spot/spot-and-science/.

required tasks again to restore the flow-based execution of the framework. Furthermore, the extension they also provided for this sub-DAG mechanism was a disk-less periodic checkpoint strategy. With this extension and by using a constant interval, it is possible to diminish the number of recomputed tasks further. Since a final task may require up to 100 percent of the predecessor tasks (whole re-computation from scratch), this additional checkpoints can help to avoid such a situation. Finally, their task-based approach with Local ABFT PDGEMM and a silent data corruption detector achieve an overhead of 5 to 6 percent, which is also near to the presented theoretical overhead. The difference to their publication with the results of this thesis are the extensive bit flips experiments in Chapter 6, including different value distributions in the matrix, various correction analyses, and the research in trying to integrate into a simulator environment.

2.6 Comparison of Master Thesis to Other Works

In this master thesis, the successful fault correction plays a major role. Depending on which types of bit flips have occurred, fault detection and the corresponding correction may fail (see conclusion over bit flips in Section 6.3). Other publications in the literature have mainly to do with node failures and designing of algorithms for online fault correction, whereas in this publication one of the primary focuses was to investigate a variable number of faults and estimate how many checksums should be used for the *blocked* ABFT method. The empirical research showed that a d-value (d-many additional rows or columns for checksums) of between 16 and 64 per submatrix is not having a significant impact on the total time overhead for this particular ABFT method (subsection 6.1.3). This further means that with such settings between 16 and 64 errors can be corrected locally per computational node. Furthermore, in this scientific work, it is studied what effect the different types of bit flips have on the detection- and correction procedure. Therefore bit flips in the mantissa, in the exponent and everywhere in the bit mask of a value are analyzed and discussed thoroughly in Section 6.2. What's more, the challenges towards an exascale simulation in combination with a fault-tolerant matrix multiplication are mentioned in Section 4.5. Therefore an assessment of how the *Local ABFT* method would look like in an exascale computation is outlined.

Since the main topic of this thesis is fault tolerance, other fault-tolerant techniques are presented and discussed in the next chapter, and respectively in Section 3.4. Furthermore, arguments about scalability and simulators used in HPC are located in Chapter 4.

Chapter 3

Fault-Tolerant Methods

In this chapter, there will be fault-tolerant methods and techniques considered. It is starting with Section 3.1 where errors, failures, fault tolerance and resilience will be introduced. Secondly in Section 3.2 the main emphasis of the master thesis will be discussed. In Section 3.3 the theory of the practical part of this thesis will be showed. And finally Section 3.4 further fault-tolerant techniques for High Performance Computing (HPC) will be pointed out.

3.1 Errors, Faults, Fault Tolerance and Resilience

Basically we can distinguish between two main categories [74, 21]:

errors	-	**faults**	**failures**
in software	-	software error	fail-stop
in hardware	-	hardware malfunction	partial
environmental influences	-	memory corruption	single process or simultaneous process
silent, transient or unrecoverable			unrecoverable

Table 3.1: Errors, faults and failures in a system

When we speak of errors, we can say that they can be silent, transient or unrecoverable, where this statement is also valid for faults. Begining with the explanation of silent errors or silent data corruptions (SDC), which are more or less undetected faults and are usually classified as bit flips [74]. These bit flips can occur in storage (volatile memory or nonvolatile disk) or during computations in the processing units. A *Error-Correcting Code* (ECC) memory, on the contrary, is only able to correct and detect a single bit flip when such memory modules are used in a system. Since double bit flips are roughly detected and cannot be handled by the ECC, such a situation

forces unfortunately to an instant reboot of the process or node. By different studies at the Cray XT5 system from Oak Ridge National Laboratory it was estimated that this could happen about one time per day for 75,000+ DIMMs [66]. Extending this situation to an exascale system (800,000 - 8 million DIMMs [1]) would result in an double bit flip rate of 10 - 100 times a day which is then remarkable when a longer computation is executed.

Transient faults or errors are acting a bit different. Here power fluctuations and alpha particle strikes are caused by the shrinking progress of the feature size of transistors [94]. The problem is that all kind of processors (GPUs, CPUs, SoCs, etc.) are becoming vulnerable to transient faults with an increasing transistor policy. Therefore such devices have to be augmented with fault-tolerant mechanisms that can guarantee a correct computation. What should also be remarked at that point, are the results of the authors in [118]. They investigate the effect of cosmic rays especially on the soft error rate (SER) of a DRAM. The SER was measured at various locations and altitudes, and the results showed that even at sea level there could be soft errors caused by cosmic radiation. Therefore choosing the right location is not enough to help with the problem of external influences.

On the other hand, we have the unrecoverable errors where a severe fault or exception is raised by a software error or a hardware component. The problem here is that they can freeze the system and no attempts can correct or undo the error. Thus this mostly ends in a situation where the system must be rebooted to work again. They are usually caused by programs or applications that run in user-mode on a computer [132]. User-mode executed processes use a virtual space assigned to them by the system, and so they don't have direct access to the memory. Since they are isolated, such processes do not interfere with the resources and hence do not compromise of the integrity of a system. They are at an enormous potential to cause an unrecoverable error if they try to read or write anything from the system memory. The result is an exception call which ends either by freezing the system or doing a reboot.

Fail-stop process failures are bad scenarios where the failed process stops working at all and the correspoinding data associated with the event are lost [37]. This type of failures is prevalent in today's large computing systems including high-end clusters where thousands of nodes are involved and computational grids with hybrid and dynamic computing resources. In order to prevent against fail-stop failures, a globally consistent state of the application like it is achieved by a checkpoint technique is common. By using a checkpoint, a recovery of the lost data can be extracted, or a recomputation can be executed. The checkpoint/restart procedure contains a point in

[1] Assuming an exascale machine having between 100k and 1 million of nodes and 8 DIMMs per node (like in Table 1.1).

time of an execution where all process states of the fault tolerant application are saved into resilient storage periodically. Thus it is able to withstand a failure of the computation or even of the whole system. However in a checkpoint/restart approach, if one process fails, all surviving processes' states are cancelled and the entire application is recovered from the last available checkpoint.

Furthermore there exist partial failures which can be managed with reduced overhead. In this case, they are usually handled by diskless checkpointing where saving to a stable storage is avoided by using memory and processor redundancy [37]. This is especially interesting for applications which modify small chunks of memory between checkpoints. Therefore in such cases, even multiple simultaneous process failures can be corrected without much overhead by using diskless checkpointing. However, the problem in the context of this master thesis is, that during matrix operations like in the matrix product a large amount of memory is modified between the checkpoints. This leads to the conclusion that diskless checkpointing produces then a checkpoint of a significant size which ends in an exceptional overhead, requiring of other fault-tolerant mechanisms.

Then we have the term fault-tolerant. A HPC application can be made fault-tolerant either by a forward recovery mechanism like ABFT or by a backward recovery mechanism (checkpointing) [21]. The main difference is the point where the computation has to take place. In a forward recovery mechanism, the additional calculation is initiated when a failure occurs, whereas in a backward recovery mechanism the last checkpoint data is taken and states already reached are re-computed, and therefore the backward term. Another interesting point in a checkpoint-restart approach is that it takes a certain amount of time (about 30 min.) to re-execute the computations. This overhead, on the other hand, makes a forward recovery mechanism attractive for a massively parallel system towards exascale when we involve the predicted MTTF rate in Table 1.1.

The last point which should be remarked here is about resilience and the definition of the RAS model introduced by [130] which has to do with measuring the reliability, availability, and serviceability of a supercomputer. The RAS terminology includes:

- **Reliability** - The probability that a system functions without failure under stated conditions over a specified amount of time. It is often calculated using a constant failure rate based on an exponential random variable model $R(t) = e^{-\lambda t}$ with $\lambda = \frac{1}{MTBF}$. Here some associated metrics are *Mean Time Between Job Interrupt* ($MTBI_J$), *Mean Time Between System Interrupt* ($MTBI_S$) and *Mean*

Time Between System Failures ($MTBF_S$).

$$MTBI_J = \frac{\text{uptime}}{\#\text{ of job interrupts}} \tag{3.1}$$

$$MTBI_S = \frac{\text{availability time}}{\#\text{ of system interrupts}} \tag{3.2}$$

$$MTBF_S = \frac{\text{availability time}}{\#\text{ of system failures}} \tag{3.3}$$

- **Availability** - The fraction of a time period that a system is in a condition to perform its destined functionality (expressed as a probability). Important metrics here are:

$$\text{Availability}_S(\%) = \frac{\text{uptime} * 100}{\text{operations time}} \tag{3.4}$$

$$\text{Production Availability}_S(\%) = \frac{\text{production time} * 100}{\text{operations time}} \tag{3.5}$$

$$\text{Production Utilization}_S(\%) = \frac{\text{productive node hours} * 100}{\text{production node hours}} \tag{3.6}$$

- **Serviceability** - The probability that a system will be retained in, or restored to, operable condition within a specified period of time. Relevant metrics here are *Mean Time To Repair* ($MTTR$), *Mean Nodehours To Repair* ($MNTR$) and *Mean time to Boot System* ($MTTB_S$).

$$MTTR = \frac{\text{unscheduled downtime}}{\#\text{ of failures}} \tag{3.7}$$

$$MNTR = \frac{\text{unscheduled downtime nodehours}}{\#\text{ of failures}} \tag{3.8}$$

$$MTTB_S = \frac{\text{sum of wallclocktime to boot system}}{\#\text{ of boot cycles}} \tag{3.9}$$

- *Maintenance* - The act of sustaining a system in or what is needed to restore it to a condition where its planned functionality can be successfully performed.

There also exists a current extension presented by [77] where the RAS model is expanded by the terms safety, performance and integrity to redefine the term resilience. Among other things, it is also emphasized that the most commonly used platform reliability metrics, including the MTTF (and its variants such as mean time to in-

terrupt (MTTI) and MTTR), are not so suitable for evaluating and quantifying the resilience of a whole system. Therefore [77] provide two new outcome-based metrics [2] for measuring HPC resilience like the *resilience factor* (RF) and *resilience factor yield* (RY).

3.2 ABFT Technique

The idea of algorithm-based fault tolerance (ABFT) is not new, and it was first proposed in 1984 by Huang and Abraham [76]. At that time the VLSI (very-large-scale integration) technology has been advanced to the point where the costs for hardware components, especially processors, could be decreased properly. It was possible to acquire a large number of computational units for an acceptably low price. With this progress, high-performance computations were made accessible for a broader range of scientists, whom on the other hand needed a high-reliability technique for long lasting calculations. That was the incentive to develop ABFT.

The main goal of the ABFT technique is to correct errors from permanent or transient failures with possibly low overhead. At the beginning, it was developed to detect and correct errors on a variety of matrix operations such as addition, scalar product, multiplication, LU-decomposition, and transposition. Later on this technique has also been applied to FFT [82], matrix equation solvers [95], recursive least squares [122], singular value decomposition [35], sorting algorithm [38], QR decomposition [97], Cholesky factorization [73] and ZU factorization [24]. However back to the approach by Huang and Abraham, where the fault masking (the idea of TMR) versus the detection of errors by testing (ABFT) was a major topic. As the usually used fault masking approach by TMR or quaded logic is expensive due to replicating the hardware, therefore a new way with detection and correction schemes was proposed.

Algorithm-based fault tolerance [76] has three main characteristics:

1. Encoding of the data (obtained from the two-dimensional product code)
2. Redesign of the algorithm for processing the encoded data
3. Distribution of computation steps among processors / computation units

The encoded matrices are called checksum matrices and contain encoded data at the word [3] level instead of at bit level. This is established in a higher level manner because a faulty module could have an impact on all of the bits in a word. The redesign is

[2] The theory of this master thesis does not rely on the new proposed metrics because most of the work was done before the publication of [77] and most research and publications rely on the older model [130].

[3] Means here a variable that can take the form of integers or floating point numbers.

needed because of the information part of the encoded data must be made simple in the recovery procedure, and further the distribution of the computation should prevent that a failure of a hardware component can affect the whole calculation or data. Another substantial point which has to be remarked is the term *algorithm-based*, which means that it's applied directly to an algorithm and further to the source code (within a program). This property further excludes additional (replication) of the hardware and also the need of checkpointing technique with writing to an external media e.g., to a hard disk drive (HDD). One limitation [4] is given by the total amount of additional information (in this case the checksums) to provide the desired fault tolerance. As this checksum values are written directly into the physical memory (RAM) it should be pre-calculated of how much additional checksum vectors are willing to be used not surpassing the available system memory. This concern is especially relevant when targets with huge problem sizes are to be achieved and to prevent performance issues coming from writing to slower memory (hard disks).

Diagnosability and Diagnosis of ABFT Systems

As ABFT becomes more and more popular since the first publication the need of a diagnosability algorithm for ABFT systems has increasingly become of interest. In [140] an algorithm to analyze an ABFT system for its fault diagnosability was developed. Therefore in their publication properties of graphic-theoretic and matrix-based models to investigate ABFT system were presented. From a theoretical point of view, a tripartite graph can be decomposed into two bipartite graphs. Since the numerical representation of a bipartite graph is usually a matrix, thus a tripartite graph can be represented as two matrices. An example for such graphs is given in Figure 3.1.

The idea of the graphic-theoretic model is based on a tripartite PDC graph [5]. Each letter in the PDC term stands for an individual set of vertices. The P denotes the processors in the system, D the output data elements and C stands for the checksums in the system. The concept in connection with the tripartite graph can be described as follows: nodes C1, C2, C3 contains the row- or column checksums (C), these are connected to the data values (D) sitting in nodes D6, D7, D8 and D9, and the nodes P1 to P5 are the processors (P) which can affect the data in D6 to D9 (see also Figure 3.1). Therefore we have the PDC structure, and we can visually see which values can be handled on which processors with which associated checksums. The final output of the PDC graph shows only faulty data elements and the processors by which they are covered.

[4] Additional info to this limitation in this thesis is discussed in Section 3.3.

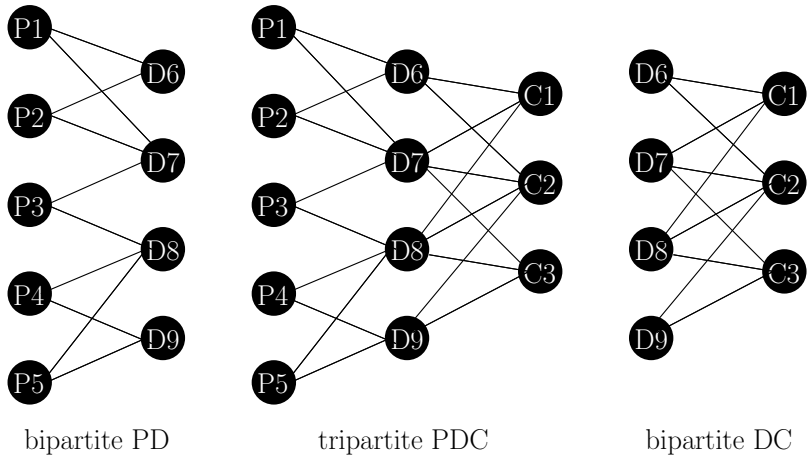


Figure 3.1: Tripartite graph and its decomposition to bipartite graphs for ABFT systems

On the other hand, there is the matrix-based model which works with the PDC graph [111]. The graph is split into two bipartite graphs (see example in Figure 3.1), the processor data (PD) graph and the data-check (DC) graph with corresponding matrices. By doing multiplication of the two matrices, a processor-check (PC) matrix can be gained, which later represents a weighted bipartite graph [5] (PC-graph). If we have an error in the system, then the element in the i-th row and j-th column in the PD-matrix is set to one. This further means that if $PD(i, j) = 1$ then processor i has a fault and data element j is erroneous. Whereas when there is a one in the PC-matrix, the faulty processor has a supporting checksum. Besides, a comparison of the row of the PC-matrix to the row of the PD-matrix can help to denote the faulty processor.

From the two previously presented models, we can extend the analysis of t-fault detectable and t-fault diagnosable. Generally, a set of faulty processors can be summarized to a fault pattern and a set of erroneous data elements affected by this fault pattern to an error pattern. This error pattern is then classified as *detectable* if in the corresponding check-vector [6] (extracted from the PC-matrix) exists a value with one. In the fault pattern manner, on the contrary, it is detectable *iff* for every error pattern it produces is classified as detectable. To conclude the term diagnosable, we can say that a system is diagnosable if a fault pattern can be located from its check-vector. Further, if a system contains such characteristics, then an analysis of its diagnosability can be done. Thus finally a t-fault detectable system contains fault patterns of size $\leq t$, and is t-fault diagnosable if each such fault pattern can be located from the number of check-vector(s) it produces. A t-diagnosable system provides the features that, if a particular check-value in the vector is zero this could have two reasons: firstly

[5] Graph where each edge has a weight associated with it.
[6] States of the checks represented as a q-bit binary vector where the q is # of checks in the system.

no data item in the error set of the check-value is erroneous or secondly the error detectability of the check-values has been exceeded. The detailed diagnosis algorithm can be found in [140] and should help to isolate a fault pattern from a given problem in an ABFT system.

3.3 ABFT Methods for Matrix Multiplication

The classic variant of ABFT_PDGEMM fulfils a fault-tolerant matrix-matrix multiplication with the structure of $C^f = A^r \cdot B^c$. This notation for the matrices is used through the whole section, where A^r denotes matrix A with row-wise added checksums, B^c matrix B with column-wise added checksums, and C^f the resulting matrix C with checksums in additional rows and columns (full checksums) [76]. In Section 5.2 the development of the variant used in this thesis is described. This section, on the contrary, contains matrix encoding schemes and the corresponding computations.

Back to the original definition from [76] where the checksum matrices initially only have one additional column summation vector and one additional row summation vector. The matrices were of following dimensions:

$$A^r = (n+1) \times m$$
$$B^c = n \times (m+1)$$
$$C^f = (n+1) \times (m+1)$$

The above-shown checksum matrix technique has a difference from the previous checksum codes commonly used up to that time. They were based on bit level correction, whereas now we have floating point numbers or integer values to be detected and corrected. The detection has been done by computing the sum of elements in each row and column and compare the result to the corresponding checksum value. The location, on the contrary, is achieved by finding inconsistent row or columns and then pointing at their intersection. When an erroneous element was detected, the value was adjusted by adding the difference of the sum to the affected value or by replacing the wrong checksum.

An optimized approach, which is also more frequently used, is to have a variable value of additional columns or row checksums. In this thesis an extended variant was used, giving the possibility that more than one error (namely d errors) can be detected and corrected per column or row. The new matrix dimensions resulting from this fact

are:

$$A^r = (n+d) \times m \qquad A^r = (n + (P * d)) \times (n + (Q * d))$$
$$B^c = n \times (m+d) \qquad B^c = (n + (P * d)) \times (n + (Q * d))$$
$$C^f = (n+d) \times (m+d) \quad C^f = (n + (P * d)) \times (n + (Q * d))$$

The left group shows how the general dimensions changed due to this extended encoding scheme, and on the right-hand side the dimensions like used in this thesis (section 5.2). Further, the additional P and the Q variables are important for the special partition encoding scheme described in next subsection and denote the processor node structure. To provide this capability, where even all values of the resulting matrix can be locally restored, firstly a weighted checksum encoding scheme is needed as presented in [83]. This scheme consists of a weight matrix W and a negative identity matrix, which together build up the correction matrix H. The idea is that with this matrix H the correction of a faulty matrix can be done by solving a least squares problem later on (see subsection 5.2.3). First, a weight matrix has to be built [7] with ones in the first column, and e.g., uniformly distributed values (udv) in the range between zero and one otherwise. It is usually a $(n \times d)$-matrix, but for the illustration of the weight matrix on the next page, the indices n_b and d like in the *Local ABFT* variant were used instead. The n_b marks the block-size of the sub-matrices per computational node, as each node or process has its checksums in its sub-matrix.

$$W^T = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \text{udv}_{11} & \text{udv}_{12} & \dots & \text{udv}_{1d} \\ \text{udv}_{21} & \text{udv}_{22} & \dots & \text{udv}_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \text{udv}_{n_b 1} & \text{udv}_{n_b 2} & \dots & \text{udv}_{n_b d} \end{pmatrix}, -I = \begin{pmatrix} -1 & 0 & \dots & 0 \\ 0 & -1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

The resulting correction matrix where the contained $d-1$ encoding vectors have to be linearly independent is then $H = (W^T - I)$. By using this variant of ABFT, usually only the final computed matrix (matrix C) can be corrected. If it is necessary the matrices A and B can also be made fault tolerant by calculating the full checksum variants of them. This is done by multiplication with the weight matrix such that:

$$A^f = \begin{pmatrix} A & A \cdot W^T \\ W^T \cdot A & W^T \cdot A \cdot W^T \end{pmatrix} \text{ and } B^f = \begin{pmatrix} B & B \cdot W \\ W \cdot B & W \cdot B \cdot W \end{pmatrix}.$$

There exist also a variant in [22] where the outer product version of the matrix-matrix product is used. As in this approach the end matrix C is updated in a loop, a failure at any time after the update procedure can be recovered.

Additionally, such an implementation can theoretically require up to 4-times [8] more

[7] This is only one proposal how the weight matrix can be built, in the literature there are also other variants available.

[8] Extended matrix dimensions of $(n+n) \times (n+n) = 4n^2$ vs. origin matrix size $(n) \times (n) = n^2$.

RAM as of the original matrix size. Hence this has to be considered as a limitation for a computational system when dealing with huge matrix problem sizes. It can be explained with the additional rows- and columns checksums which are controlled by the d-value. If it is required that all values of the end-matrix should be detected and corrected, then the d-value has to be set to n the actual size of the matrix. This results in $(n+n) \times (n+n)$-matrices instead of $(n) \times (n)$ and would be visualized as:

$$C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right) \Longrightarrow C^f = \left(\begin{array}{cc|cc} C_{11} & \text{cksm}_{11_2} & C_{12} & \text{cksm}_{12_2} \\ \text{cksm}_{11_1} & \text{cksm}_{11_3} & \text{cksm}_{12_1} & \text{cksm}_{12_3} \\ \hline C_{21} & \text{cksm}_{21_2} & C_{22} & \text{cksm}_{22_2} \\ \text{cksm}_{21_1} & \text{cksm}_{21_3} & \text{cksm}_{22_1} & \text{cksm}_{22_3} \end{array}\right)$$

As an example explanation let C be a (16×16)-matrix and C_{11} be a (4×4)-sub-matrix, then the corresponding checksums $\text{cksm}_{11_1}, \text{cksm}_{11_2}$, and cksm_{11_3} are also (4×4)-matrices. Therefore the dimensions are doubled vertically and horizontally, resulting in an 4-times larger matrix (C^f of dimension (32×32)) compared to the original size without any checksums.

In the next subsections, the emphasis lies on the two major variants used for matrix multiplication *Local ABFT* and *Global ABFT*. They rely on different weight- and correction matrices and therefore have their specific advantages and disadvantages which will also be discussed and summarized. Nonetheless, the previously described encoding technique with weighted checksums is used as the basis in both approaches.

3.3.1 Local ABFT

For a *Local* or *Blocked* ABFT approach it is recommended to use the partitioned encoding scheme by [123]. This kind of encoding should help to reduce inaccuracy issues in the computation and should simultaneously assist for a better comparison of the checksums in the detection phase. After preparing the weight matrices as described in the previous section, the next step is calculating the checksums. From the definition of the partitioned encoding scheme it follows that the matrices A^r and B^c for a (2×2)-process grid are constructed as follows:

$$A^r = \begin{pmatrix} A_{11} & A_{12} \\ W^T A_{11} & W^T A_{12} \\ A_{21} & A_{22} \\ W^T A_{21} & W^T A_{22} \end{pmatrix} \quad B^c = \begin{pmatrix} B_{11} & B_{11} \cdot W & B_{12} & B_{12} \cdot W \\ B_{21} & B_{21} \cdot W & B_{22} & B_{22} \cdot W \end{pmatrix}$$

In Figure 3.2 the detection and correction process of *Local ABFT* is illustrated. We can see that an error or bit flip (black block) is detected by two corresponding checksums (green blocks). As each computational node has its checksum values (orange blocks) the detection and correction are done locally (by each processor itself).

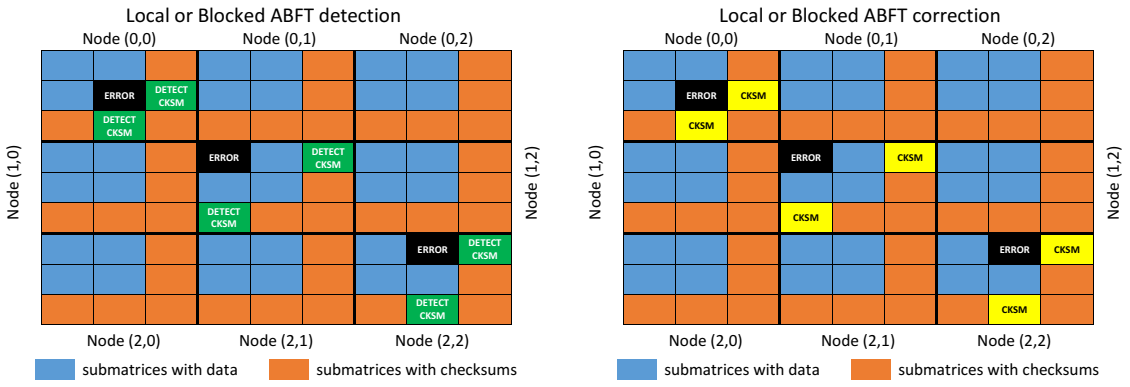


Figure 3.2: Detection and correction in a Local ABFT approach on a (3×3)-process grid in the final computed matrix C^f

For one thing, this variant is attractive when we have large matrices with many processors so during the restoring of the data only a small correction matrix is involved and each process restore its faulty data. Further, from a numerical point of view using such an encoding scheme results in a more stable maximum round-off error. On the other hand, no node failures can be handled, because the checksums are only designed to act locally and directly on the faulty process' data. There is no information about other processes generated, and therefore a failure of a process cannot be recomputed.

3.3.2 Global ABFT

The *Global ABFT* variant, on the contrary, is more widely used in research then *Local ABFT*, because node failures can be handled effectively by using a fault tolerant MPI implementation like *Fault-Tolerant MPI* or *User Level Failure Mitigation* (see also subsection 3.4.3). In [12] an explanation of how ABFT and ULFM can be combined is presented and further in [37] it is discussed how to use FT-MPI with ABFT. Firstly, again a weight matrix W and a correction matrix H is computed and distributed globally over all CPU ranks. Then the matrices can be equipped with checksums by:

$$A^r = \begin{pmatrix} A \\ W^T A \end{pmatrix} \quad B^c = \begin{pmatrix} B & B \cdot W \end{pmatrix} \quad C^f = \begin{pmatrix} A \cdot B & A \cdot B \cdot W \\ W^T A \cdot B & W^T A \cdot B \cdot W \end{pmatrix}$$

We can see from the first sight that the weight matrix, in this case can, be as large as the original matrices A and B. The size shrinks of course with the number of nodes because we only have to specify a weight matrix as large as one sub-matrix located on one node if we want to guarantee that a node can be fully restored. It is on the other hand also possible to specify only a few additional checksums to be calculated, but this will result that only a partial data of a node failure can be reconstructed. This situation can happen because of not enough checksums in the final matrix C^f.

Another difference to the *Local ABFT* approach is that the checksums have to be calculated globally by multiplication with the transposed weight matrix W^T. In a big problem size, this can use much more computation time, whereas in the local variant each node or processor calculates its part with the weight matrix. Furthermore, the comparison of the faulty values with the checksums and the correction of C^f are global operations. In the figure below, we have a (3×3)-node system where the checksum parts are saved at nodes (2,0), (2,1), (0,2), (1,2) and respectively (2,2) and the origin data at nodes (0,0), (0,1), (1,0) and (1,1).

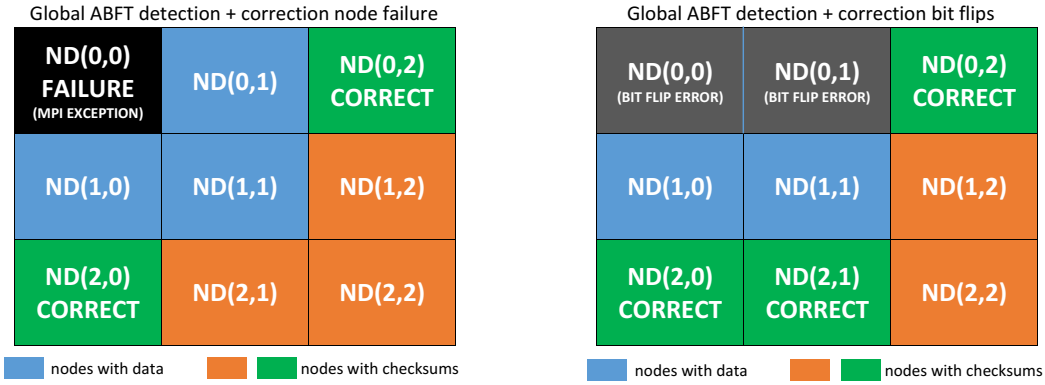


Figure 3.3: Detection and correction using a Global ABFT variant on a (3×3)-process grid in the final computed matrix C^f

The checksums, in this case, are 1/3 of the size of the whole matrix C^f but with equal size of sub-matrices at each node. Figure 3.3 shows the situation which is generated when a *Global* ABFT approach is used. Here the recognition mechanism can be activated by a raised MPI exception from the MPI error handler (FT-MPI) [37]. This message is then distributed globally such that the application knows which node has been shut-down (failed). Another way is described in [12] where the failed processes are dropped, and the calculation is continued (data which was on this node may be lost, but computation completes). On the contrary, if complete fault tolerance is needed, the MPI communicator can be changed such that a new process is reintegrated and the data can be restored on this new process from the checksums (ULFM approach). Back to Figure 3.3, where on the left-hand side we have a node failure of the process (0,0). The re-computation of the lost data can either be done by node (0,2) or (2,0) because these processes comprise the weighted checksums data for the restoration. Furthermore, we have on the right-hand side a different condition, with an occurrence of faulty data (bit flips). In this case, the errors are detected by comparing the checksums globally (over several nodes). For the correction on the contrary there exist several possibilities which can successfully recompute the corrupted data:

1. Node (0,2) restores faulty data of node (0,1) and (0,0)
2. Node (0,2) restores faulty data of node (0,0) and ND(2,1) of node (0,1)
3. Node (0,2) restores faulty data of node (0,1) and ND(2,0) of node (0,0)
4. Node (2,0) restores faulty data of node (0,0) and ND(2,1) of node (0,1)

Concluding the *Global ABFT* method, if we want to have a fault tolerant system which is also capable of dealing with node failures, then this variant is strongly recommended. Furthermore, there are also other possibilities like the composite ABFT & Checkpoint approach described in subsection 3.4.2 which can also be used for such situations.

3.3.3 Local vs Global ABFT

In this subsection, the most important features of the two previously discussed ABFT methods for matrix multiplication are summarized. In Figure 3.4 the blue blocks denotes data which has been generated during the computation of $A^r \times B^c$, whereas the orange blocks contain the corresponding calculated checksums. Further, we can see on the left-hand side that each node in the final matrix C^f has its checksums (locally), whereas on the right-hand side the checksums are distributed over several nodes (globally).

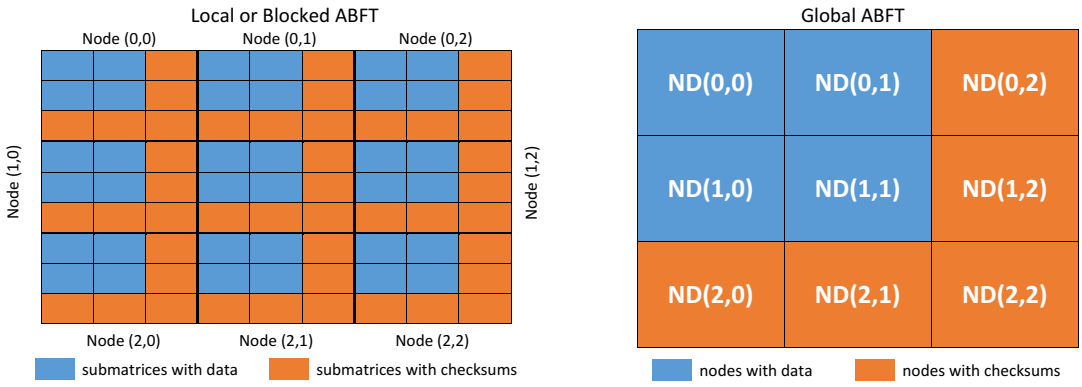


Figure 3.4: Comparison of Local ABFT and Global ABFT for a (3×3)-process grid

The following table shows a quick overview of the pros and cons of the two variants discussed in the last two subsections:

	Local ABFT		**Global ABFT**
+	Bit flips can be corrected	+	Bit flips can be corrected
+	Smaller weight matrix and local correction	+	Node failures can be restored and data recalculated
+	Good numerical stability (regarding max. round-off error)	−	Checksums are calculated in a global operation
−	Node failures because of MTBF not handled	−	Bigger weight matrix and correction executed globally (communication of nodes)

Table 3.2: Advantages and disadvantages of Local ABFT and Global ABFT

3.4 Further Techniques used in HPC

Despite the mainly discussed fault-tolerant method in this thesis, the ABFT, there are also checkpointing techniques, composite approaches, *Fault-Tolerant* MPI and user level failure mitigation (ULFM). This section is dealing with the most important and frequently used ones in context with HPC.

3.4.1 Checkpointing Techniques

Setting up checkpoints where data is written to a medium for providing a backup and resilience of a system has been widely used in the last few decades. There exist different types of checkpoint techniques and approaches which can be used to guarantee a fault-tolerant and robust system. For HPC applications we can distinguish mainly between protocols for checkpointing and probabilistic models for checkpointing (Table 3.3).

Checkpointing Protocols	**Probabilistic Models for Checkpointing**
• Process checkpointing	• Periodic checkpointing
• Coordinated checkpointing	• Incremental checkpointing
• Uncoordinated checkpointing	• Multi-level checkpointing
• Application-level checkpointing	• Checkpointing with fault prediciton
• Hierarchical checkpointing	• Checkpointing with replication

Table 3.3: Checkpointing techniques for HPC applications [74, 20, 21]

Process Checkpointing

The first fault-tolerant protocol is *process checkpointing*. The primary goal is to save the current state of a process by momentarily interrupting the execution of all kind of corresponding threads before saving [74]. A process itself can be seen as a parallel application because it contains many user-level or system-level threads. Hence by process checkpointing the problem of saving the state is reduced to a sequential problem. It can be characterized by the level of the software stack, how the checkpoint is generated, and how it is stored.

Techniques used in combination with the characterization of process checkpointing are user-level checkpointing, system-level checkpointing, blocking call, and asynchronous call [20]. In user-level checkpointing the main function has to do with creating a serialized view of the application where an application-specific routine can, later on, restore a meaningful state of the process. On the other hand system-level checkpointing can have different possible implementations like OS syscall, dynamic library or compiler assisted. Here the important thing is that a serial file is created in such a way that it can be loaded in a process image. It has the advantage that it can be restored on a machine which has the same architecture, same operating system, and same software environment but the checkpoint size is consequently large because of memory footprint. Furthermore, a checkpoint can be generated by a blocking call that has a termination event once the serial file of the process checkpoint is complete. Here synchronization of the threads is needed, or each thread for itself must do a checkpoint. The limitation is that no process activity has to be done during the whole checkpoint operation. Otherwise, an asynchronous call can be achieved by duplicating the entire process, allowing the parent process to continue its execution. In [21] another approach was presented called staging checkpointing, where the memory hierarchy is used to reduce the checkpoint time. It can be used as an alternative to the asynchronous call checkpointing. Last but not least the checkpoints have also to be stored. This procedure is accepted to be successful once the saving to a non-corruptible space is done, where the storage can be local or to a remote device. Depending on how important or risky the data is, we can choose between libraries which can save the data in-memory, in NVRAM, disk-less, or to a remote file system [74]. This is then done asynchronously in the background.

Coordinated Checkpointing

In coordinated checkpointing, we can distinguish between a blocking and a non-blocking variant of it [74]. The main idea is to create a consistent view of the application. There are notifications with messages to evaluate if a checkpoint wave has

completed and the checkpoint is consistent. For instance, in the blocking variant, the spreading of application messages are delayed after entering the checkpointing wave, so that the state of communication channels is saved. Whereas in the non-blocking variant the communications after the beginning and before the end of a checkpoint are added to its receiver. Further messages which are inside of a checkpoint are pushed back at the beginning of the queues.

Uncoordinated Checkpointing and Message Logging

Whereas the processes have to rollback to the last valid checkpoint wave, when a failure is detected (coordinated checkpointing protocols), in uncoordinated checkpointing the restart of a minimal set of processes is forced. In fact, only the processes which are corresponding to the failure should be restarted. The disadvantage in a coordinated checkpointing is that all checkpoint taken randomly may be invalidated by missing messages, forcing the application to restart from scratch. Therefore in the uncoordinated variant, the piecewise deterministic assumption (PWD) and the concept of message logging are introduced to avoid such a situation.

In PWD the behaviour of a sequential process can be managed from a given state to another deterministic state by constraining each non-deterministic choice between them. The order of the messages and the required actions are captured by the MPI library. The role of message logging is to provide a tool which can capture and replicate message receptions or nondeterministic events. In this context, the right order and a suitable content are playing a crucial role. Message logging prepares a log of the event and a log of the message's content. This can help in a case of failure to restart the failed process from its last checkpoint and successfully to enter in the replay mode. Once the history has been entirely restored, the distributed application can continue its progress according to the valid PWD [74].

Application-Level Checkpointing

For application-level checkpointing the synchronization at a specific point takes the major role [21]. A programmer has to specify when to execute a checkpoint within an application. The limitation here is that the calls can happen only at a certain point in the source code whereas system-level checkpoints can be executed anywhere. They can be triggered by an automation tool (library) or by a function call specified by the user.

Helping libraries are for instance the scalable checkpoint restart (SCR), which is based on files written to the local storage. Advantages of this library are managing

the reliability of storage and atomic commits. Another library is the fault tolerance interface or FTI. It adds an additional option to the SCR features which includes handling of transparent restarts. Here the storage hierarchy for checkpoints is beginning with memory, then local file and finally distributed file system. Last but not least another option to use is the global view resilience (GVR) library. The difference to previous helping tools is that the data backup is done entirely in memory, in a reliable tuple-space. Therefore independent processes are involved which can be accessed through the GVR API.

Hierarchical Checkpointing

There exist hierarchical protocols as well as probabilistic models for hierarchical checkpointing [20, 74]. In the protocol variant, a group of processes is co-ordinating its checkpoint. Therefore a combination of uncoordinated protocols and event logging is, where the concept of message logging is used between the groups. On the other hand for the probabilistic model approach, specific parameters are introduced to refine the model. These parameters have firstly to do with the impact of message logging on execution, secondly with re-execution, and finally with the checkpoint image size.

Periodic Checkpointing

In this approach, a blocking model is used [21]. This means while a checkpoint is taken, there can not be an execution of any computations in the background. With the provided framework the *waste*-value due to failures can be defined and calculated. This value should help to find an optimal checkpointing interval, and this is also the primary purpose of it. Expressed in words, we can say that a high *waste*-value, denote that too many checkpoints are made, and we have a big loss of computation. Otherwise, when the value is too low, this can be an indicator of using only a few number of checkpoints. This setting, on the other hand, can result in a high risk of losing data when a failure occurs. The exact definition and calculation can be found in [74].

Incremental Checkpointing

A half a year later the same team from the ABFT & *Periodic Checkpointing* approach [15] have developed another resilience technique, the *incremental checkpointing* which should compete to the bi-periodic-, pure periodic-, and ABFT & *Periodic Checkpoint* one [19]. In bi-periodic the checkpoint interval may change, whereas in the pure periodic it remains constant. Hence there was an investigation of a novel dynamic

programming approach with which checkpoints can be optimally placed. The results have been researched on a QR factorization, and it was shown that the incremental checkpointing could easily beat the bi-periodic checkpoint- and the pure periodic checkpoint approach, and further remains competitive with the ABFT & *Periodic Checkpointing* for up to 100,000 nodes. Another benefit, which the new technique brings along is the time gain and I/O gain when using it over the bi-periodic checkpointing. The expected gain on completion time for one million nodes was in the range of 10,000 seconds which are about 3 hours of saving time when executing a QR-routine. This result is mainly because of the checkpoints are smaller which make them less costly, and the dynamic programming reduces the risk of re-execution in case of failure. Here again, the *waste*-value calculation is used for comparing the results.

Multi-Level Checkpointing

This probabilistic model can combine multiple technologies like local memory/solid state drive (SSD), partner copy/XOR, Reed-Solomon coding or parallel file system to get rid of different failure types [21]. Further help libraries including *Scalable Checkpoint/Restart* (SCR) and *Fault Tolerance Interface* (FTI) may also be involved. For k levels an optimal pattern length, number of checkpoints at level l, and pattern overhead can be constructed. These equations can later be used to calculate what scheme or combination of the different levels would achieve an optimal multi-level checkpointing.

Checkpointing with Fault Prediciton

In a fault prediction model, all efforts are about trying to predict an error or fault before they arise [20]. Therefore a fault predictor which should warn the user about possible upcoming faults on the platform is used. The predictor is characterized by the recall r (# of faults indeed predicted) and precision p (fraction of correct predictions). The optimal approach depends on the time which is needed for executing a proactive checkpoint and on the predictor's precision. Bringing the *waste*-value from [74] in combination with the predicted and unpredicted faults together, can lead to the result that an optimal time interval for a checkpoint can be calculated. Furthermore, the decision at what point in the period of the execution the prediction to be considered can have a substantial impact on the optimal value. It can happen that if a fault is predicted, there is no time left to take preemptive actions because there is already a checkpointing in progress. In this case, such a prediction must be ignored and not used for optimality calculations.

Checkpointing with Replication

Another possible way to withstand system failures is to replicate all the computations. In this model firstly the processors are grouped by pairs in a way that for example two processors have identical work to do. It is obvious that if a failure happens on one of the processors from the group, the other one still can do its correct computations. The idea, at first sight, seems to be expensive because the processors are split up at least in groups of two, but a different checkpoint strategy can resolve this. In a system where the computations are replicated only one half of the processors need to have a checkpointing technique running. By introducing the term of MNFTI (mean number of faults to interruption) and comparing the standard approach with checkpointing to the one using replication, a crossing point can be calculated. Further, with considering the corresponding MTBF of both variants the checkpoint time where the replication method is more efficient than a standard execution with checkpointing can be calculated. In an example calculation, it has been shown that for a parallel system with 2^{20} (about one million processors) the replication variant is more computationally effective, if the time needed for a checkpoint is greater then 6 minutes [74]. Conclusively with the output of this model it can be decided which variant to prefer in which particular situation.

Another direction where current investigations are done is in *partial replication*. The idea behind this approach is that to replicate only critical processes or a smaller part of all processes. This can help to save computational resources and to optimize the mean time between failure.

3.4.2 Composite Approach: ABFT & Checkpointing

On the road to exascale computing, the engineers are confronted with many challenges. One of them is the huge number of usually fault-tolerant hardware components where the fault tolerance is fighting against the time to failure (meaning the MTTF). The fault tolerance cannot be held constant when the number of components rises (see Table 1.1). When there should be a long-lasting computation executed a fault-tolerant technique like ABFT or checkpoint/restart is indispensable. In [15] a solution was presented which has to do with a composite protocol, which alternates between ABFT and checkpoint/restart. This should provide effective protection of an iterative application where ABFT is involved.

So the composite approach of *ABFT & Checkpointing* combines ABFT in library phases and periodic checkpointing in general phases. For the procedure, this means that in sections that can be protected by an ABFT method, a partial checkpoint is

taken and so the rest of the dataset is protected. The first advantage is that the library dataset is not contained in that partial checkpoint because the ABFT algorithm will reconstruct it. Every time a call returns, a partial checkpoint of the library's modification is added to the partial checkpoint. The data is split but complete by the coordinated checkpointing mechanism. Thus if a failure is detected the crashed process within the library call can be recovered using the combination of the roll-back recovery and ABFT. The distribution is as follows: the ABFT thread recovers the library dataset and the partial checkpoint the rest of the data. The restoring happens before quitting the library routine. An important issue is setting the right checkpoint interval. Setting a more frequent checkpoint interval results in an overhead but reduces the amount of time lost, whereas a long period without checkpoint decreases the overhead and increases the loss of already computed data in case of a system failure.

The results were shown in comparison to a bi-periodic checkpoint- and a pure periodic checkpoint algorithm, at which in the bi-periodic the interval may change, and in the pure variant the checkpoint interval remains constant. As an advantage of the ABFT & *Periodic Checkpointing* approach it was observed that the ABFT technique appears to stay at almost constant waste [9] rate when the number of nodes increases. From the point of total waste, and especially when the computational nodes exceed 100k, the new approach is highly efficient. Comparing bi-periodic- and a pure periodic checkpoint to ABFT & *Periodic Checkpointing* there exists a factor five of better efficiency (low wasting) by using the new provided approach at one million nodes.

3.4.3 Fault-Tolerant MPI Approaches

As actually the current MPI Standard v3.1 from 2015 still does not provide mechanisms for dealing with communication failures software engineers have to develop own fault-tolerant techniques to support them. Since 1999 the Message Passing Interface (MPI) community is trying to extend MPI by some interesting feature for HPC applications called *Fault-Tolerant MPI* (FT-MPI). Until then the first attempts were including a checkpointing and rollback mechanisms. Later a master-slave model variant (MPI-FT) was presented, where extra spare processes were assigned to grids, and utilized when there was a failure. An observer process was able to restore the lost messages between the master and slaves [65]. FT-MPI is going more in the direction of fine tuning of the application, as the level of correctness for individual communicators can be changed to the own needs.

[9] Means here the fraction of time when on the platform resources no advancement of the computation is done by the application.

FT-MPI has been developed in the scope of the HARNESS project [57]. The main goal of FT-MPI is to provide a communication library with an MPI API for the end-user. It was tightly coupled to the fault-tolerance in the HARNESS system. The HARNESS (Heterogeneous Adaptive Reconfigurable Networked SyStem) project, on the contrary, is on the first sight an experimental meta-computing system. The main advantage is the highly dynamic plug-in modularity and the fault-tolerant computing environment for HPC applications. The first extension of FT-MPI has to do with new, additional MPI communication states. These are: {`FT_OK | FT_DETECTED | FT_RECOVER | FT_RECOVERED | FT_FAILED`}. Further, additional attributes indicating how many processes have failed are handled by `FTMPI_NUM_FAILED_PROCS` and `FTMPI_ERRCODE_FAILED`.

Besides all the extensions *Fault Tolerant MPI* also has integrated four different error modes which can be specified at the beginning of an application in the MPI communicator. Depending on the chosen mode the API can react differently when errors are detected. This is the list of the error modes:

1. *ABORT*: When error occurs simply abort
2. *BLANK*: Failed processes are not replaced, remaining processes have the same rank, and `MPI_COMM_WORLD` remains the same size (communicator can contain gaps to be filled later)
3. *SHRINK*: Failed processes are not replaced, and processes might have a changed rank after recovery, communicator is reduced $\Longrightarrow$ contiguous data structure, `MPI_COMM_RANK` recalled
4. *REBUILD*: Default mode, failed processes are respawned, other (surviving processes) have the same rank as previously

The modes as mentioned earlier are accompanied by a message mode which can be either *NOP* or *CONT*. In *NOP*(E) all ongoing messages are dropped (no operation on error), while *CONT* denotes that all communications which are not affected by an error can continue to work as normal until the communicator's state is reset. All in all, FT-MPI is an attempt to give different methods for dealing with system failures in a MPI application.

User Level Failure Mitigation

Since 2012 a new optimistic approach called *User Level Failure Mitigation* (ULFM) should help against failures in MPI applications [11]. The main goal is to resume communication capability for MPI. Basically, if errors occur, they are exposed through MPI exceptions to the application. In the case of MPI, we are speaking of middleware

which should be made fault-tolerant. Back to the features of ULFM. Once a failure condition has been reported by the MPI error handlers an exception is raised. In the MPI Standard this exception is usually handled by aborting the whole application, so the idea of ULFM is to avoid the cancellation by replacing the standard error handler by a user specified one. For handling different failure conditions ULFM provides several recovery functions [11, 74]:

1. `MPI_Comm_failure_ack(comm)`: Helps the application to resume `MPI_ANY_SOURCE` point-to-point operations between non-failed processes
2. `MPI_Comm_failure_get_acked(comm, &group)`: To determine which communicator corresponding processes have failed
3. `MPI_Comm_revoke(comm)`: Interrupts all operations on the communicator; Communicator becomes improper for further communications with other CPU ranks and at every attempt the `MPI_ERR_REVOKED` is raised
4. `MPI_Comm_shring(comm, &newcomm)`: All failed processes are excluded, and a new communicator without them is created; Collective operation $\Longrightarrow$ all participating processes must complete it
5. `MPI_Comm_agree(comm, &mask)`: Collective operation which relies on a binary mask; Failed processes are ignored such that computation can be continued e.g., reliable AllReduce

Another advantage of ULFM is the backward compatibility to previous MPI Standards. It can be applied to make an application fault-tolerant if needed. On the other hand, there is still an open issue with shared-memory windows supported by MPI. These are MPI objects or file objects with a similar *revoke* function, and in this case, only failures of MPI processes are addressed. More precisely by ULFM, only issues that may disrupt collective operations on a file are handled.

Irrespective of the interesting features of ULFM, the current version ULMF 2.0, is even integrated into the Open MPI master branch [23]. This has the result that Open MPI and ULFM from now on are developed together. There exist also a specification of the current ULFM which should be by and by merged with the Open MPI Standard.

Chapter 4

Simulators and other Tools in HPC

There exist actually many different simulators for a variety of applications, but finding the right one, especially for HPC purposes is not always trivial. Some simulators are not capable of simulating real-time applications, and they may act as network topology emulators or as supporting tools for the simulators, such that engineers can effectively have a profound knowledge at how future supercomputers shall be designed. In this master thesis, the emphasis lies on showing which PDES- and non-PDES simulators are capable of simulating a system where the scale goes towards an exaflop machine and also to look at their pros and cons. Further a summary of the simulators' capabilities and associated tools is given in Table 4.1 and continued in Table 4.2.

4.1 PDES Simulators

First a brief journey into the world of parallel discrete event simulation. In PDES primarily the execution of an individual discrete event simulation with simultaneous threads is the crucial point which should be highlighted [67, 64]. Typically each thread simulates one logical process or in the oversubscription case multiple logical processes. The situation with logical processes is that they are simulated all together at different points in simulated time, and the simulation needs to guarantee the ancestry between logical processes such that the correctness of the simulated system is sustained. There exist commonly two approaches for guaranteeing causality, the conservative approaches, and the optimistic approaches. In conservative approaches, the main task is to determine when it is not dangerous to process an event which means where causality errors can be strictly avoided. Optimistic approaches on the contrary work the other way round, they firstly detect causality errors and restore the situation by using rollback mechanisms.

Historically, the first distributed simulation mechanisms were based on conservative approaches, but they have the risk that a distributed deadlock can occur. In such a case, one particular logical process is holding up all the progress of different processes then it, while it is waiting for the progress of another. By doing so, a distributed deadlock can be resolved. That happens in a simulated point of time, where it is safe for the other processes to execute an event. There also exists a situation where frequently distributed deadlocks may result in a lock-step simulation. The positive point here is that this is not taking advantage of the concurrency. The occurrence of distributed deadlocks can be decreased or exclusively avoided by transmitting null messages with timestamps between logical processes [34]. These messages are then used to communicate information about the progress and further supports, that each logical process obtains an independent view of the global simulated time. From that point, the logical process can determine if processing an event would end in a violation of the causality. Moreover, in optimized PDES solutions it is common in a simulated system to utilize the context of an event, such that it can be determined if it is safe to process [64].

Optimistic approaches, on the other hand, require knowing periodically of the state of each logical process, such that rollback mechanisms can be activated to the right time. They do not need to determine when it is safe to proceed because as an error occurs the procedure to recover is invoked. Because of this openness, it can also happen that in a worst-case scenario, all logical processes are rolled back. Advantages here are that a simulator with an optimistic approach can exploit parallelism in some situations where a causality error is expected and that a dynamic creation of logical processes can be easily adapted. Therefore in a PDES, an optimistic protocol is chosen, which can identify causality violations by event time stamps. The authors of [67] presented, for instance, the *Time Warp* protocol, which is based on the *Virtual Time* [81] paradigm and which is commonly used for PDES solutions.

4.1.1 X-Sim

In this thesis one essential part was to find out if there exists a simulation tool which is usable towards exascale simulation and if it can be used in combination with the two highly optimized libraries for linear algebra (DPLASMA, ScaLAPACK). One major tool which has been analyzed more in detail and with which we tried to achieve this goal was xSim [1].

The xSim simulator [64, 63, 113, 62, 58, 13, 61] is a performance investigation toolkit that allows running a HPC application in a monitored environment with MPI.

[1] http://www.christian-engelmann.info/?page_id=1804.

Software developers can write applications, that can be executed on an HPC system where millions of concurrently executing threads are simulated. In [59] it was shown that with xSim it is possible to simulate up to 2^{27} or about 134 million MPI processes (each with its process context). During the simulation, xSim collects performance information of the application, which is displayed summarized at the end of the run. There exist many environment variables that can be preconfigured to the user's needs. The network topology can also be configured so that there can various networks be simulated. Further, it uses a lightweight, conservative PDES algorithm, so that it can execute a MPI application on a relatively small system by high oversubscription of the processes. The architecture and design are shown in Figure 4.1. We have simulated processes (SP) which are running under a PDES with MPI on native processors (P). Further on the application is getting access and communicates over the simulated MPI.

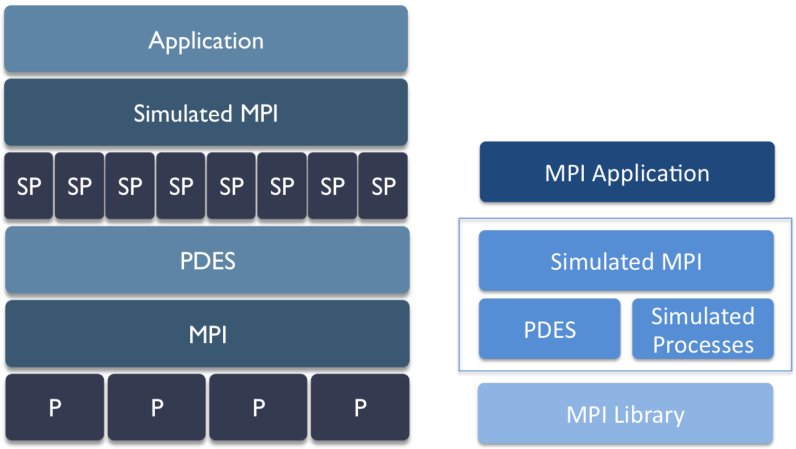


Figure 4.1: xSim architecture and design [64]

As the execution with xSim uses a virtual wall clock time, performance data can be inferred based on a network- and a processor model. The design of xSim is implemented to work as an interposition library located between MPI application and MPI library. MPI calls are virtualized, by the MPI performance tool interface (PMPI). The simulation tool xSim currently does not support [64] execution models with threading (OpenMP [45]), accelerators (GPGPUs) and task-based execution models (High Performance ParalleX (HPX) [46, 85]). Besides it also has a fault injection feature which makes an advantage when we want to analyze failures in a simulated environment. Furthermore, propagation, detection, notification, and handling capabilities are integrated to allow more detailed investigations [62].

With all the mentioned features before, xSim is one of the most attractive tools when we want to simulate real-case scenarios of an extreme-scale computation. Although at the time it is not capable to fully work with DPLASMA or ScaLAPACK,

which is mainly because the internal structure of their functions, xSim still could be used to simulate a matrix-matrix multiplication in an exascale range when the algorithm for the multiplication is written and optimized by hand. This procedure includes that the subfunctions of the complex main routines need to be extracted and the compilation has to be done without nesting routines.

Further there is a need of e certain space where the information of the additional virtual MPI processes has to be stored. When xSim should be used as a preferred simulator, the following additional MPI context [2] has to be considered:

- Virtual MPI process stack per native MPI process = 512 KB
- Virtual MPI process heap per native MPI process = 32 Bytes
- Virtual MPI process table per native MPI process = 184 Bytes

4.1.2 MuPI

MuPI ($\mu\pi$) [119] has nearly the same architecture like xSim but instead the simulated MPI and simulated processes (SP), the concept is based on virtual MPI and virtual processes (VP). It is, therefore, a process-oriented simulator where each task or thread is represented as a logical process. MuPI supports both PDES approaches (optimistic and conservative execution) which are based on the μsik PDES engine. These capabilities imply that, e.g., null messages and *Time Warp* as features are supported. MuPI can also distribute multiple virtual MPI ranks by multiplex onto each available real processor. Furthermore, within a simulated environment, unmodified MPI codes are supported [120].

Since a limited amount of processing resources are available, a strict hierarchical structure is adapted to accommodate a significant amount of virtual MPI ranks across them. Firstly, each node may consist of multiple processor sockets, and each processor may have multiple cores integrated. Further, each physical core can handle a pre-assigned number of virtual MPI ranks, which are time-multiplexed on their associated processing core. In [120] there is shown a successful run of 16 million virtual ranks in a discrete event fashion on as few as 16,128 real processors (scaling factor 1,024) executed on the Cray XT5 supercomputer. With 216,000 compute cores on the same machine a prototype of $\mu\pi$ even achieved 221 million simulated MPI processes, where each has a separate thread context, and where all processes were synchronized by simulated time [64]. Summarized we can say that the scalability in $\mu\pi$ has a per core limitation because per CPU there cannot be scaled much more than 1,024 simulated MPI processes. This leads to the result that if we want to simulate an extreme-scale

[2] A similar MPI context has to be taken into account when using other simulators too.

system (exascale), we would need another extreme-scale supercomputer (petascale), but it will be possible.

Unfortunately $\mu\pi$ is not developed any more and current works is going in direction of PDES on cloud/virtual machine platforms [149] and simulations on GPGPUs [91] including simulating cell biological systems.

4.1.3 SST

SST (Structural Simulation Toolkit)[124] offers an innovative simulation [3] for current architectures, including latest processor, memory and network support. It is a modular PDES framework using MPI and scales up to a few hundred simulated multi-core nodes (512+ processors)[141]. The SST Core framework copes with independent time-scale simulations including micro-, meso-, macro-scale simulations. Further, it provides many interfaces and various utilities for simulation models and is beyond compatible with external models like Gem5, DRAMSim2, and others. The simulator supports checkpointing using Boost Serialization Library to convert the core's state and the state of each component into a binary format which is dumped to a file and can be later used to restart the simulation. The SST algorithm works according to the

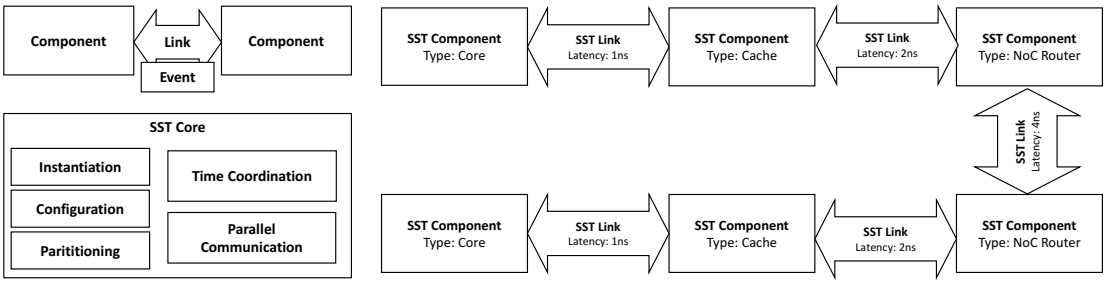


Figure 4.2: SST algorithm scheme[141]

two illustrations in Figure 4.2. Simulations are comprised of components connected by links and components interact by sending events over links, where each link has a minimum latency. Components can load *subComponents* and modules for additional functionality and perform the simulation.

SST uses a Python configuration file to define global parameters for the simulation, determine and configure components, and specify links and link latencies between components.

[3] http://sst-simulator.org/SSTPages/SSTMainDownloads/.

To summarize the compatibility to this master thesis main emphasis, the SST is not very suitable for the application of exascale simulations. Here the requirements of simulating millions of cores, support of C++11 and using it without much code changing of the source code is not provided, just to mention some. Further, it is complicated to use, and the developer using SST should well know the language Python. SST does not support over-decomposition and relies on deterministic execution without the possibility that distributed deadlocks can occur [64]. On the other hand, SST provides a standard interface to various power estimation libraries including Orion, McPAT, and Sim-Panalyzer and hooks are included in the interface to allow thermal modeling tools (HotSpot) [124].

4.1.4 OMNeT++

OMNeT++ [109, 64] is an expandable, modular and component-based framework with a combination of a C++ PDES library. Primarily it is used for building network simulators and has a generic architecture. The features of OMNeT++ are compatible with various problem domains, including modeling of wireless and wired communication networks, messaging-based hardware and software systems and queueing networks. Oversubscription is not offered on its own since OMNeT++ uses a conservative PDES with null messages [34] to avoid distributed deadlocks.

The latest available version [4] has extended features in the data export tools, toolchain and libraries on Windows including 64-bit Windows support, and to the Eclipse [30] base for the IDE.

So actually OMNeT++ [138] is not an online simulator but provides infrastructure and tools for writing simulations. Besides it can validate hardware architectures, evaluate performance aspects of complex software systems, models multiprocessors and other distributed hardware systems, and supplies also an accessibility to the Eclipse IDE. Again this network simulation framework is not suitable for testing the practical aspects of this master thesis. This assumption is made mainly due to the several limitations mentioned in the OMNeT++ manual. There are inter alia the constraints that no direct method call or member access can occur (unless they are mapped to the same processor), no global variables can be defined, and sending to a submodule of another module is not allowed. However, millions of cores/nodes can be simulated as a mesh topology, but not in the manner of a fault-tolerant matrix-matrix multiplication including testing of occurrence of soft errors. This is also because the strengths of network simulators lie in simulating *network communications* in particular scenarios or situations where no *real* machines or networks are configured.

[4] https://omnetpp.org/.

4.2 Non-PDES Simulators

4.2.1 Charm++ BigSim

From the beginning on, the BigSim [151] project has to do with programming issues in large-scale HPC systems [64]. The BigSim Emulator is developed for testing applications (including debugging) at large scale machines and is based on the Charm++/AMPI [87], which is a machine independent parallel programming system. Charm++ has an intelligent runtime system, which applies the idea of overdecomposition or more precisely processor virtualization based on migratable objects. The *AMPI* stands for adaptive MPI, which is an implementation of MPI on top of Charm++ [5].

With BigSim emulator we can run for example an MPI program with 100,000 simulated MPI processes and their context spread over 2,000 processor cores [64]. On the other hand, the BigSim emulator does not provide time-accurate simulation and no PDES. Here, oversubscription is achieved by using Charm++, where MPI process contexts are encapsulated in objects. Although it has succeeded in improving the performance of many scientific applications, the emulator has limited scalability (it is designed for simulating PetaFLOPS supercomputers but not up to an exascale range). One of the major research done with Charm++ is going in the direction of NAMD (simulating a molecular dynamics code) [86]. It was also successfully used in [80] to evaluate HPC networks (torus, fat-tree, and dragonfly) which are typically used in large supercomputers via simulation. If the emulator should be used for computations, developers have to define additionally initialization details (e.g., `BgSetNumWorkThread(int num)` or `BgSetNumCommThread(int num)` for setting the number of worker threads or communication threads per node) and handler functions like `BgSendPacket(int x, int y, int z, int threadID, int handlerID, WorkType type, int numbytes, char* data)` has to be used [89]. This particular handler function sends, for instance, a portion of data to the Node[`x,y,z`] and also specifies the responsible handler function for this message. Further, a `threadID` has to be given (pointing on the desired thread to handle the message) or with which the desired thread category can be specified. These are also arguments why this tool cannot be directly used without modification of the source code.

The second tool (BigSim simulator), in contrast, was developed to identify performance bottlenecks in various applications and uses a trace-driven PDES [64]. That means that with its concept, architecture specific parameters of HPC systems can be modeled. For instance, time-accurate simulations are done by a variable-resolution

[5] http://charm.cs.illinois.edu/.

processor model and an extensive network model. While it uses a restrained PDES to maintain accuracy, the support range includes only post-mortem trace replay without the possibility to run applications. Because a trace replay is completely deterministic, BigSim's PDES solution does not mind of distributed deadlocks for example. That is also a reason why the BigSim is classified as a non-PDES simulator. Finally, the simulator allows a variable-resolution model in the range from simple scale factors to the more advanced interpolation factors based on performance counters (e.g., cycle-accurate simulators) [88]. Furthermore, it can be nicely used for analysing the performance of communication networks. The analysis is done by plugging in either a straightforward latency model or with a detailed model of the whole communication fabric. Further, it is possible to evaluate huge networks with it.

4.2.2 JCAS

JCAS [60] or Java Cellular Architecture Simulator was a predecessor to xSim(4.1.1), which was developed in 2001. The goals at that time were to research the scalability and fault-tolerance of HPC algorithms for systems with about 100,000 processor cores [64]. As a testing application, a fast Fourier transform (FFT) algorithm which has the ability of self-healing and was based on peer-to-peer diskless checkpointing was observed. The JCAS prototype was capable of running $< 500,000$ virtualized processes on a Linux cluster with only five processor cores (where one core was reserved for visualization). Further, it was primarily used for solving standard mathematical problems and has implemented several network topologies(nearest neighbor, torus, mesh, random) for testing [60]. Furthermore, it has a GUI with the support of some failure modes. It was possible to kill a selected node, percentage of nodes in a region or a block of nodes directly in the GUI. JCAS can execute algorithms at scale but has some weak points which are required when we want to simulate the next-gen supercomputing systems. These disadvantages are a time-accurate simulation, high-performance capabilities, running the simulator on an MPI basis, and a complete functional virtualized MPI. Further JCAS does not implement a PDES capability because it relies only on Java threads. The oversubscription feature is achieved in this case by encapsulating simulated MPI process contexts, but only in JAVA objects, which also brings a certain drawback along. Therefore it is not suitable for a computationally and spatially intensive matrix multiplication towards exascale based on MPI. Although, the tool might be configured to simulate a small matrix product problem.

4.2.3 SimGrid

SimGrid [33, 131, 64] does not use a PDES. Instead, it uses a central simulation engine. This toolkit provides the necessary capabilities which are needed in heterogeneous distributed environments for simulating distributed applications. According to this, the distributed systems that can be simulated are Grid computing infrastructures, HPC systems running MPI, peer-to-peer computing environments, and Clouds. SimGrid [6] offers MPI and online simulation for running the applications. Oversubscription is done by UNIX98 contexts, which is provided by an OS support for cooperative threading. On the other hand by the use of a central simulation engine, a global synchronization and scalability problem is established. Since SimGrid partially covers the MPI functionality [128] at the moment and as only non-multithreaded applications are supported (neither Pthreads [7] nor OpenMP [8]) it can't be directly used with the HPC libraries (DPLASMA or ScaLAPACK).

4.2.4 GridSim

GridSim [39, 31] is a grid simulation toolkit for resource modeling and further for application scheduling of parallel and distributed computing. Hence it investigates the characteristics of computing resources with various network configurations. Instead of an online approach like in xSim(4.1.1) that indeed executes an application, GridSim, on the contrary, executes models of applications, interconnect networks and computing resources [64]. GridSim [9] also uses a centralized simulation engine called *Distributed SimJava* for providing a discrete event simulation, whereas MPI is not supported. Moreover, the GridSim toolkit has extensive support for modeling and simulation of several heterogeneous resources (single or multiprocessors, workstations, SMPs, shared and distributed memory machines, and clusters with different capabilities and configurations) [31]. As it is based on Java, it is also not suitable for the HPC libraries mentioned in this thesis.

4.2.5 DIMEMAS

Dimemas [27, 70] is a trace-driven PDES solution for investigating application performance with the specialization on predicting parallel performance using a single CPU machine. In other words, it enables a developer to tune parallel applications (message-

[6] http://simgrid.gforge.inria.fr/.

[7] Execution model that works independent of a language; supporting also a parallel execution model.

[8] API that supports multi-platform shared memory multiprocessing.

[9] http://www.cloudbus.org/gridsim/.

passing programs) on a workstation, while providing an objective prediction of their performance on a desired parallel machine. From a predefined set of performance parameters, the Dimemas simulator can reconstruct the time behavior of a parallel application. Supported target architecture classes are networks of workstations, distributed memory parallel computers, single and clustered symmetric multiprocessors (SMPs), and heterogeneous systems which gives a certain degree of freedom.

Dimemas also generates trace files that are suitable for Paraver, since they are both developed by the team at the Barcelona Supercomputing Center (BSC) [7]. This combination enables the user to examine any performance problems indicated by a simulator run conveniently. Especially the analysis of application execution on non-existing machines makes Dimemas interesting for the HPC community. With this tool, good quality of prediction can be achieved and further the costs can be kept low because of the high abstraction. Nonetheless, it gives useful results, but for an extreme-scale application execution, Dimemas is not suitable. However, it might be used as an additional tool to analyze the performance bottlenecks of an application without actually running the whole context. A use case would be to run the application on a smaller parallel machine (with an average problem size) and then use the produced tracing files in the simulation engine to estimate how the application will behave if it is done for instance on an exascale system. Therefore, for example, the behavior of random bit-flips or node failures in a huge parallel system cannot be observed with this tool. This performance analysis tool and also the tool Paraver (4.3.1) can be obtained from the BSC main site [10].

4.2.6 ns3

The latest release of the ns-3 network simulator is at [117]. NS-3 is a discrete-event network simulator for detailed network architecture, where the core of the simulation and models are implemented in C++ [116]. It is built as a library which can be statically or dynamically linked to an application written in the C++ language. This program defines then the simulation topology and executes the simulation. NS-3 has a built-in pseudo-random number generator, which can easily be accessed by instances of `ns3::RandomVariableStream`. Conceptually, the simulator keeps records of many events that are scheduled to execute at a predefined simulation time and further it can be seen as a real-time network emulator. It has a tracing API and a data collection framework where the raw data can be transformed by the collector and where the *Aggregator* marshals data into plots, files or databases. Furthermore, a framework called common open research emulator (CORE) consisting of a Python-

[10] https://tools.bsc.es/downloads.

based framework using ns-3 Python bindings, a distributed computing library, and an ns-3 TapBridge framework is also provided. For visualization and configuration the Eclipse IDE [30] is used. Moreover, the ns-3 simulation CORE gives the opportunity that both IP and non-IP based networks can be investigated. Therefore the vast majority of its users concentrates on wireless/IP simulations where models for Wi-Fi, LTE, or WiMAX are involved. Furthermore, a mixture of dynamic or static routing protocols such as AODV and OLSR for IP-based applications can be used in combination with ns-3. On the other hand, it is not suitable for real-time simulations of virtual MPI processes needed for simulating towards exascale HPC systems. However, for engineers, it can be interesting to simulate an exascale machine from the number of nodes and different network topologies with the ns-3 simulator.

4.2.7 NetSim

NetSim [134, 135] is a detailed network architecture simulator and is primarily used as software for protocol modeling and simulation, for network R&D purposes and defense applications. It supports the analysis of computer networks and is flexible to use. There exist three versions of NetSim [11], whereas the *Pro* version is suited for commercial (enterprise/defense) customers while the standard and academic version are for education customers.

Its features include data center simulation, simulation of LTE covering, cellular networks, cognitive radio networks, military radio, and others. It also has an exhaustive tracing capability for performance reporting, packet- and event trace and dynamic metrics. For high-performance purposes especially the data center simulation is relevant. Unfortunately, there is a limitation to about 100,000 nodes [133], and therefore it is not suitable to be used in exascale range. As it is a non-PDES simulator, it does not support live MPI-threads.

4.3 Tools for Using in Combination with Simulators or as Assistance

In this section, there are tools presented which can be used in combination with the previous simulators. These tools can act as a tuning support, as trace file analyzers or managers and as visualization help for developers.

[11] http://tetcos.com/index.html.

4.3.1 PARAVER

Paraver [121, 29, 28] is a compelling performance visualization and analysis tool which work with trace files. These files can be used to analyze any information from its input trace format. The tool is developed by engineers at the BSC in Barcelona [12].

The main features of Paraver are opening the support for concurrent comparative analysis of several traces, detailed quantitative analysis of program performance, the customizable semantics of the visualized information, cooperative work, sharing views of the trace-file and building of derived metrics [29]. Further Paraver accepts any programming model as long as the used model can be mapped to the parallelism expressed in the Paraver trace which consists of three levels. An example of two-level parallelism in the meaning of Paraver would be hybrid MPI + OpenMP applications. Among other things programming interfaces like MPI, OpenMP, Pthreads, OmpSs[13] and CUDA [14] are supported by the runtime measurement system *Extrae* [26] that also generates the Paraver traces. In the following figure we can see how the two previously described tools (Dimemas and Paraver) can be used and tuned together by pursuing the sequence paths:

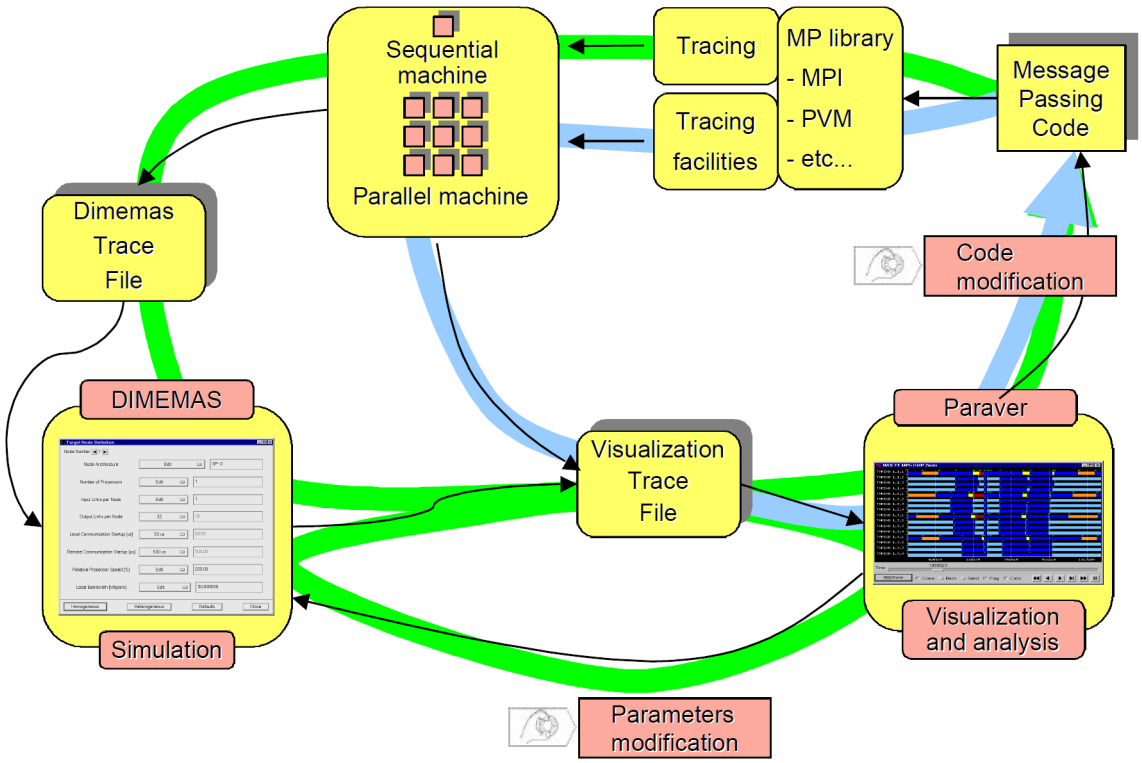


Figure 4.3: Combination of the tools Paraver and Dimemas [27]

[12] https://tools.bsc.es/paraver.

[13] Programming model based on OpenMP to support asynchronous parallelism and heterogeneous architectures (GPUs).

[14] API by Nvidia.

With the unique multiple traces feature of Paraver, it is possible to compare two versions of code, the difference between two runs, behavior on two machines, the influence of problem size or application scalability. The multiple view options (graphical-, textual- and analysis view), large trace file support and the universality makes the tool attractive when somebody wants to analyze extreme-scale systems.

4.3.2 Vampir

The Vampir [90, 72] tool consists of the instrumentation and measurement component VampirTrace and the visualization applications Vampir and VampirServer. It is a trace-driven PDES performance analysis tool for parallel applications. Since 2014 it works strongly coupled together with the Scalable Performance Measurement Infrastructure for Parallel Codes tool (SCORE-P) [139].

The current version can be found at Vampir's home page [15]. The feature list consists of a comprehensive performance analysis framework, including a graphical data representation for a detailed understanding of the dynamic processes on massively parallel machines, identification of performance problems and bottlenecks and support for Linux-based PCs and clusters. The Score-P collaboration allows the tool to works natively with Scalasca, TAU, and Periscope [139]. The whole support for instrumentations and further details can be found in a table [16] from the developers' web site.

From a high-performance point of view, this tool can be classified to be a good performance bottleneck analyzing tool. It supports MPI, OpenMP, PAPI [17], Pthreads, and can be used to analyze the tracing data of millions of cores.

[15] https://www.vampir.eu/.
[16] https://www.vampir.eu/support/comparematrix.
[17] Portable library to hardware performance counters.

4.4 Summary of Simulators and Tools

We have here an overview of the advantages and disadvantages [18] of the different simulators and tools studied in this master thesis:

Simulator or Tool	Based on		Advantages / Support		Disadvantages / Not supported
X-Sim (4.1.1)	PDES	+	Real-time simulations, millions of simulated MPI processes, distributed deadlocks, fault injection feature	–	Not directly working with DPLASMA, ScaLAPACK, OpenMP, High Performance ParalleX, GPGPUs
MuPI (4.1.2)	PDES	+	Online PDES engine, optimistic and conservative execution supported	–	Not developed any more, Scalability per core ∼1,024
SST (4.1.3)	modular PDES	+	MPI support, checkpointing, power estimation libraries, time-scale independent simulations (micro-, meso-, macro-scale) & thermal modelling tools	–	Can only simulate 512+ processors, Python understanding is required, no distributed deadlocks supported
OMNeT++ (4.1.4)	Component-based PDES library	+	Tools for writing and building network simulators, Eclipse IDE, validate hardware architectures, simulating wired and wireless communication networks, deals with distributed deadlocks	–	Oversubscription not offered
Charm++ Bigsim (4.2.1)	Trace-driven PDES	+	Evaluation of HPC networks, performance investigation of scientific applications, simulating a molecular dynamics code (NAMD)	–	No time-accurate simulation, oversubscription in objects, limited scalability, models only architectural parameters of HPC systems

Table 4.1: Summary of simulators and tools towards exascale

[18] For details and further explanations see the corresponding subsections in the brackets and list of acronyms.

Simulator or Tool	Based on		Advantages / Support		Disadvantages / Not supported
JCAS (4.2.2)	Java threads	+	Simulation of $\sim 500,000$ processes	−	MPI, no time-accurate simulation
SimGrid (4.2.3)	Central simulation engine (CSE)	+	Simulating grid computing infrastructure, HPC systems and clouds	−	global synchronization and scalability problem, partially covers MPI, non-multithreaded applications only
GridSim (4.2.4)	Models of applications (CSE)	+	Toolkit for resource modeling, application scheduling, for parallel and distributed computing, grid computing	−	No online execution, no MPI, based on Java (scalability)
DIMEMAS (4.2.5)	Trace-driven PDES	+	Predicting parallel performance on a single CPU machine, simulate several architecture classes	−	No real-time execution, no full simulation of all threads with contexts
ns3 (4.2.6)	DES	+	DES for detailed network architecture, IP and non-IP based networks, Wi-Fi, LTE	−	Simulation core and models only C++, no MPI
NetSim (4.2.7)	DES	+	Network architecture simulator, protocol modelling and simulation, LTE, cellular networks, military radio	−	No live MPI-threads, simulate $<$ 100,000 nodes
Paraver (4.3.1)	Tracing files	+	Comparative analysis of several traces, many programming models accepted, multiple views	−	Older GUI
Vampir (4.3.2)	Trace-driven PDES analysis tool	+	Instrumentation, measurement components and visualization applications provided, Score-P tool	−	A few compiler instrumentations and performance counters not supported

Table 4.2: Summary of simulators and tools towards exascale continued

4.5 Towards Exascale Simulation

To develop an optimal simulator for future systems is not trivial. One crucial point for achieving the exascale goal is the scalability of the simulators. It should be possible that per real core the simulated core capability (MPI process) is nearly infinite. With infinite, we want to express that it should be only limited by the physical memory (RAM) which is needed for the simulated process' context. This feature is crucial if we want to simulate next-gen supercomputers on small servers. As described in subsection 4.5.1 we have to expect > 10 million cores for an exascale supercomputing center. With a scaling ratio of 1:100, for example, we would need a simulating machine with at least 100,000 cores. That is apparently not very efficient because simulations usually take much time and running a supercomputer with 100k cores cannot be occupied by a single user easily. Further, the financial aspect when running massively parallel systems over a longer period should be considered.

Another point expected from engineers is that the simulator is providing accurate results. As we are talking about a dozen million of cores the used tools and simulating libraries should give precise results for the time, architecture, design and network topologies. The budget for such systems is estimated to be > $200 million [20, 53] and therefore these tools should give good evaluations because they also can help to save probably millions of dollars regarding the hardware costs. Besides the main challenges at exascale [52] which are power, extreme concurrency, limited memory, data locality and resilience, the predicting tools should work accurate and give good results. A proposal of using a good combination of the presented tools can be to use xSim (4.1.1) for an online simulation of the desired application, then use ns3 (4.2.6) for modeling and examining the network architecture and finally use Paraver (4.3.1) to analyze the tracing files and to find performance bottlenecks. Thus actually a way to exascale can be a combination of the available tools and simulations for the own required application. However, in most cases, the developers have to reprogram the source code to the specific requirements of the chosen simulator.

Simulators in scientific problems and difficulties and issues are discussed in (4.5.2) and (4.6). In the coming subsection, an example is given of how a matrix-matrix multiplication would look like when using xSim (4.1.1) to simulate an exascale situation.

4.5.1 Exaflop PDGEMM

The following example was compiled based on the best theoretical computer performance (Rpeak) from the current TOP500 supercomputer list [106]. Therefore the Rpeak performance was divided by the total number of cores and then sorted in de-

scending order. The number of cores needed for an Exaflop range was calculated by using the highest GFlop/s per core rate from all systems without additional accelerators (no GPUs, acceleration cards, ...). From this it follows, that to simulate an exascale machine with current CPU power capabilities we are in need of a system with following setup (for instance):

1. TOP500 supercomputer with best FLOPS performance / core (called HPE SGI 8600) [103], equipped with Xeon Gold 6154 18C 3GHz and Intel Omni-Path. Rpeak approx. 1 Petaflop, total cores 10,368, about 100 Gflop/s per core.
2. For 1 Exaflop performance approx. 10 million cores needed (10 M $\times$ 100 Gflop/s)
3. For simplicity and quadratic property of the local ABFT PDGEMM algorithm: 12.25 M cores assumed here (comes from a of 3500 $\times$ 3500-grid of CPU cores)
4. Submatrix size of dimension: 200 $\times$ 200 needed for good FLOP rate (on 1 core)
5. Overall matrix dimension: $700,000 \times 700,000$ or (700K $\times$ 700K)
6. Simulator: xSim (capable of simulating millions of cores)
7. Calculation for 1 Exaflop peak performance:

Simulation of 12.25 M requires	6.2	TB (0.5 MB/virtual MPI process)
700 K $\times$ 700 K matrix A	3.5	TB
700 K $\times$ 700 K matrix B	3.5	TB
700 K $\times$ 700 K matrix C	3.5	TB
Total RAM required	**$\sim$16.7**	**TB RAM** (as a lower bound)

Table 4.3: Minimum amount of RAM required for simulating an Exaflop machine without communication and calculation overhead

The following conclusion from the example above is that it is not easy to simulate an exascale computation which theoretically will reach a peak performance of 1 Exaflop/s. Especially when using a matrix-matrix multiplication which is expensive on space when the dimensions of the matrices are huge. However simulating a petascale computation might be a way more simple to arrange. In such a case, for a system of about one million cores with an estimated performance of about 100 Petaflops/s the total RAM requirement will range from 1.5 to 2.5 TB.

Instead, back to the exascale example, when the calculations should be done with DPLASMA and a non-fault-tolerant PDGEMM, approx. **25 TB RAM** is needed. This estimate is shown in Table 4.4 where a realistic scenario was calculated based on empirical investigations. Further, it should be remarked, that these values were not

Simulation of 12.25 M requires	6.2	TB (0.5 MB/virtual MPI process)
700 K × 700 K matrix A	3.5	TB
700 K × 700 K matrix B	3.5	TB
700 K × 700 K matrix C	3.5	TB
MPI communication and calculation overhead for DPLASMA (approx.)	8.5	TB
Total RAM required	**∼25.2**	**TB RAM**

Table 4.4: Minimum amount of RAM required for simulating an Exaflop machine (not fault-tolerant)

measured in a real test run, they should only give an approximation to an exascale test case.

To get a good sense of supercomputer and the amount of RAM they are equipped, in Table 4.5 there are some chosen machines presented from the current TOP500 list.

Supercomputer	**Rank**	**Total Amount of RAM**
No. 1 Supercomputer in the world - Sunway TaihuLight (China)	1	∼1300 TB RAM [105]
No. 1 Supercomputer of Austria - Vienna Scientific Cluster 3 (VSC 3)	460	∼150 TB RAM [142]
No. 1 Supercomputer of Germany - Hazel Hen	19	∼988 TB RAM [75]
No. 1 Supercomputer of Japan - Gyoukou	4	∼575 TB RAM [102]
No. 1 Supercomputer of Switzerland - Piz Daint (CSCS)	3	∼340 TB RAM [104]
Last supercomputer in the list - Discover SCU11	500	∼73 TB RAM [101]

Table 4.5: Supercomputer and equipped physical RAM in terabyte [106]

According to Table 4.6 (on the next page), where a fault-tolerant variant is presented [19], we can say that one of the current TOP 500 supercomputers is actually needed to simulate an Exaflop system under such conditions. Further we can see that for an Exaflop *Local* ABFT PDGEMM, the required system for simulation should have at least 30 TB of RAM.

[19] A detailed description can be found in the implementation section under (5.2.1) and (5.2.3).

Simulation of 12.25 M requires	6.2	TB (0.5 MB/virtual MPI process)
720 K × 720 K matrix A	3.8	TB ($d = 5$)
720 K × 720 K matrix B	3.8	TB ($d = 5$)
720 K × 720 K matrix C	3.8	TB ($d = 5$)
720 K × 720 K help matrix D	3.8	TB ($d = 5$)
Additional space for fault tolerance arrays H1,H2,S1,S2,I1,I2	0.3	TB ($d = 5$)
MPI communication and calculation overhead for DPLASMA (approx.)	8.6	TB
Total RAM required	**∼30.3**	**TB RAM**

Table 4.6: Minimum amount of RAM required for simulating an Exaflop machine (fault-tolerant version)

4.5.2 Simulators in Scientific Problems

In the last section, we show that simulating a matrix-matrix multiplication towards exascale will cost many resources as it is pretty heavy-weight from the number of computations, but there exist problems in science which are not so heavy and can be used to show the scalability of a simulator or tool.

Charm++ has several applications which are using it. NAMD is probably the most important one. It is a program which can be used in parallel systems for simulating biomolecular assemblies consisting of proteins, cell membranes, DNA molecules, water molecules, and so forth [87]. Here, two chosen effective Charm++ features especially useful for parallelizing NAMD are the adaptive overlap of communication and calculation across modules and dynamic load balancing. It was used to scale NAMD to petascale machines. Another application in combination with the Charm++ is the Car-Parrinello ab initio molecular dynamics (CPAIMD) method which can model complex systems with nontrivial bonding, chemical bond forming events and breaking events. The idea behind is first to improve the methods and sampling algorithms to allow greater scale simulations of higher accuracy, and second to employ the next-gen software engineering tools to design and implement efficient algorithms for CPAIMD which scale to more than thousands of processors. Another project is the ChaNGa (Charm++ N-body Gravity solver), which is used for cosmological simulations of the formation of galaxies and massive-scale structures. It has to do mainly with the N-body problem where hierarchical methods for N-body simulations are investigated. What's more, there is the use in other applications like ParFUM and POSE, where ParFUM provides a framework for the efficient parallelization of unstructured meshes and POSE is a general-purpose optimistically synchronized PDES environment built with Charm++. The POSE environment was explicitly designed to handle simulation models which can be problematic (where fine computation granularity and a slight

degree of parallelism is involved). POSE works with the BigSim simulator and supports virtual topologies like N-dimensional meshes, tori, and hypercubes and K-ary N-trees and hybrid topologies.

In the literature of extreme-scale simulator xSim also several applications were observed. In [13] the primary goal was to enable the execution of the complete NAS Parallel Benchmark (NPB) suite by [112] in a simulated HPC environment. The NPB suite is described as a set of MPI benchmarks (in C and Fortran) that utilize some chosen, more complex MPI calls, which are common in today's parallel applications. Therefore this is more about seeing if the simulator is competitive with scaling and different weight categories of computations. Further, the NPB suite from NASA Advanced Supercomputing Division (NAS) includes several benchmarks (mini-applications) which have different classifications of how computationally intensive they are. Here are some of them, which were investigated by xSim [64]:

- CG, a conjugate gradient solver including an irregular memory and communication patterns (class [20] C)
- EP, an embarrassingly parallel kernel (class D)
- MG, a multi-grid solver on an order of meshes, long- and short-distance communication patterns, (mostly memory intensive operations (class D))
- FT, a discrete 3D fast Fourier transform introducing all-to-all communication patterns (class C), and
- IS, an integer sort concentrating on random memory access patterns (class D)

In [58] conversely, a Monte Carlo solver was successfully executed up to 2^{24} or about 16 million simulated MPI processes and also on different network topologies. This example can be seen as a lightweight scientific computation which is used for computing or estimating pi (π). There is not much context which has to be shared over all simulated MPI processes and therefore simulators can easily cope with such situations. Such lightweight computations are also good for showing the scalability of a simulator.

Another direction of investigations is showed in [80]. Here the evaluation of HPC networks via simulation of parallel workloads was observed. The TraceR [1], which is a trace replay tool working on the basis of the ROSS-based CODES simulation framework was used. In this paper an evaluation and a comparison of the three most used network topologies for building interconnection networks in great-scale supercomputers: torus, fat-tree, and dragonfly were presented. The evaluation was done by the scalable packet-level network simulator, TraceR and for a simulation of

[20] The classes here have mainly to do with the problem size (Class A smallest problem size to Class D biggest problem size).

a 46K node system. This research should mainly help to estimate operational costs and performance efficiency for future systems.

Further scientific work is going in the direction of simulating a particular time of a protocol like P2P or simulating a binomial broadcast, which is a specialization of SimGrid [128] (presented in 4.2.3). NetSim (4.2.7) is also used for protocol modeling and simulation, and further for network R&D and in the defense sector. NetSim can be used moreover for investigations in wireless sensor networks, cognitive radio and LTE networks [134].

4.6 Issues Using Simulators and HPC Libraries

One of the most significant issues is the time for simulation. Especially when the simulated application is very computationally intensive, and the amount of simulated MPI processes are exceeding the one million-boundary. We can see from xSim investigations [64] for matrix-matrix multiplication that the simulation time with 2^{12} or 4096 virtual MPI processes goes in the range of 1E+04 seconds or about 3 hours of execution time. When we project this results on an exascale case with 2 million or 2^{21} MPI processes, the wall-clock time will result in about 1E+07 seconds or 2800 h or 117 days. Here it should be remarked that this is only an estimation and the matrix-matrix multiplication is not a well-optimized version (not using one of the HPC libraries). It should only visualize that applications which are computationally intensive have an enormous impact on the execution time of the simulation.

Another point is the required physical memory for the simulation. As the simulation needs space for the simulated MPI contexts or objects and also the problem size pulls strongly on the available memory, this can also be a limitation issue for simulating an individual problem. From the current point of view, we can say that a supercomputer of current TOP 500 list [106] is needed [21] for simulating a next-generation supercomputer as stated in subsection 4.5.1.

Last but not least there is the issue of subroutines of the libraries not knowing of the actual network topology (also extensively described in Section 7.1). This situation also comes from the fact that online simulators like xSim (4.1.1) are using own MPI contexts and thread grid structures which overlap the provided grid information of the HPC libraries. Hence DPLASMA and ScaLAPACK are working with own grid information acquiring routines which give the available hardware topology to the library. DPLASMA further uses HWLOC which gives extended information like in (Figure A.1) and (Figure A.2). Based on this information the MPI context and further

[21] When matrix operations should be simulated.

the application's grid structure is set up to provide optimal performance for all library functions. Therefore to achieve the execution of a simulator with one of the both HPC libraries, we have to break the functions into the subroutines and extract the needed code for a successful run. That also implies that no simulator was found in the study of this thesis which can be directly executed without disassembling the computing routines. This issue [22] has not been tried and tested in a real case scenario and is therefore given as an open issue for future work (Section 7.1).

[22]The issue of application not knowing the execution environment has only been recognized, and one possible solution was proposed as a theoretical thought, but no test cases or application has been evaluated with reprogramming.

Chapter 5

Implementation

For the implementation DPLASMA, a highly optimized library for HPC was chosen. The advantages of this library are that it can easily cope with various hybrid hardware combinations, including CPU + GPU accelerators, CPU + Xeon Phi accelerators or other combinations like it is the case in supercomputer machines. The tricky part of the implementation was to get rid of the special 2-D block-cyclic storing of the matrix values as the used ABFT algorithm does not support this special structure. Therefore transpositions, in the sense of reordering the matrix values were used before and after the correction of the erroneous data. To assure that there are no performance bottlenecks these transpositions were measured and classified to have no impact on the algorithms' performance. Further space optimizations including, for instance, using only one additional matrix for special purposes (transpositions, checking, validation) were applied (see subsection 5.2.5). There are also special output functions for checking the correctness against the numerical computing environment MATLAB (matrix laboratory) and for writing values to disk. To make the application more flexible, besides the DPLASMA-own command line arguments, several preprocessor possibilities were implemented. This means by using `-D name` [129, 3.11 Options Controlling the Preprocessor] during compilation functions for changing the output precision, output extended details, writing values to disk, setting MTBF, tolerance value, activating/deactivating Fault Injectors, number of faults or runs and other functionalities can be set.

In the first part of this chapter the emphasis will lie on the HPC library, then the algorithm used for the results in the master thesis will be analyzed thoroughly and finally, the Fault Injector which is running as an own thread will be presented.

5.1 DPLASMA

DPLASMA stands for Distributed Parallel Linear Algebra Software for Multicore Architectures [50, 14, 16, 17, 18] and is a collection of functions for dense linear algebra problems. It concentrates on providing a novel approach to address load balance by using a DAG (direct acyclic graph) structure. This DAG representation has several advantages which are: independence of the problem-size, overlapping of communication and computation, prioritized tasks, automatic extraction of the communication info from the dependencies, and architecture-aware dynamic scheduling and management of tasks [8]. In the following table we can see the two main scopes of DPLASMA:

Dense Linear Algebra Problems	**Dense Matrices**
• linear systems of equations	• real arithmetic | • complex arithmetic
• matrix inversion	• single precision | • double precision
• least squares problems	• mixed precision
• condition number estimation	• general
• singular value problems	• symmetric | • hermitian
• eigenvalue problems	• symmetric positive defnite
• generalized eigenvalue problems	• hermitian positive defnite

Table 5.1: Summary of DPLASMA's capabilities [50]

For the linear systems, there is primarily a focus on the LU factorization (with Gaussian elimination), LLT (Cholesky decomposition) and LDLT decomposition which is mainly done by the operations *factor the matrix*, *solve the system*, *iteratively refine*, *solve in lower precision and refine in higher precision* and *compute matrix inverse* [50]. In the least squares or minimum norm, there are the following problems supported: QR decomposition, LQ factorization, QP3 (QR factorization with pivoting). Here the procedure is again similar to *factor the matrix*, *solve the system*, *iteratively refine*, *solve with lower precision and refine in higher precision*, *apply* Q and finally *generate* Q. The scope for singular values is defined by reducing to condensed form where it can be done either by reduction to band-bidiagonal or reduction to proper bidiagonal. After the reduction phase, singular values are found and forwarded to the computation of the singular vectors (all | subset). For handling eigenvalues there are functions provided for the symmetric- and non-symmetric matrices or eigenvalue problems. The symmetric problems are handled by reduction to condensed form (band-tridiagonal | proper tridiagonal), find eigenvalues and compute eigenvectors (all | subset). Whereas the non-symmetric case is done by reduction to condensed form (by Hessenberg reduction), find eigenvalues and again find eigenvectors (all | subset supported).

On the next figure (5.1) we can see how legacy software libraries have developed

over the years. First, we have LINPACK which was designed for supercomputers in use in the 1970s and early 1980s [48]. LINPACK has mainly and only to do with vector operations (Level 1 BLAS operations [92]) and been superseded mainly by LaPACK in the 80's. There also exists the LINPACK benchmarks which appeared initially as part of the LINPACK user's manual. The parallel LINPACK benchmark implementation called HPL (High-Performance Linpack) is nowadays used to benchmark and rank supercomputers for the TOP500 list [98, 100].

LaPACK, however, has been designed to run efficiently on shared-memory, vector supercomputers and was the next step to support Level 3 BLAS operations including block operations [2]. It consists of Fortran 77 subroutines with support for solving the most general problems in numerical linear algebra. There are three main routines, the driver routines, computational routines and auxiliary routines. The driver routines are capable of solving standard types of problems, the computational routines to perform a discrete computational task, and auxiliary routines are used to perform a particular subtask or general low-level computation. Besides the high efficiency on vector processors, shared memory multiprocessors and high-performance workstations, LAPACK further can also be used on all types of scalar machines (simple workstations, PC's, mainframes).

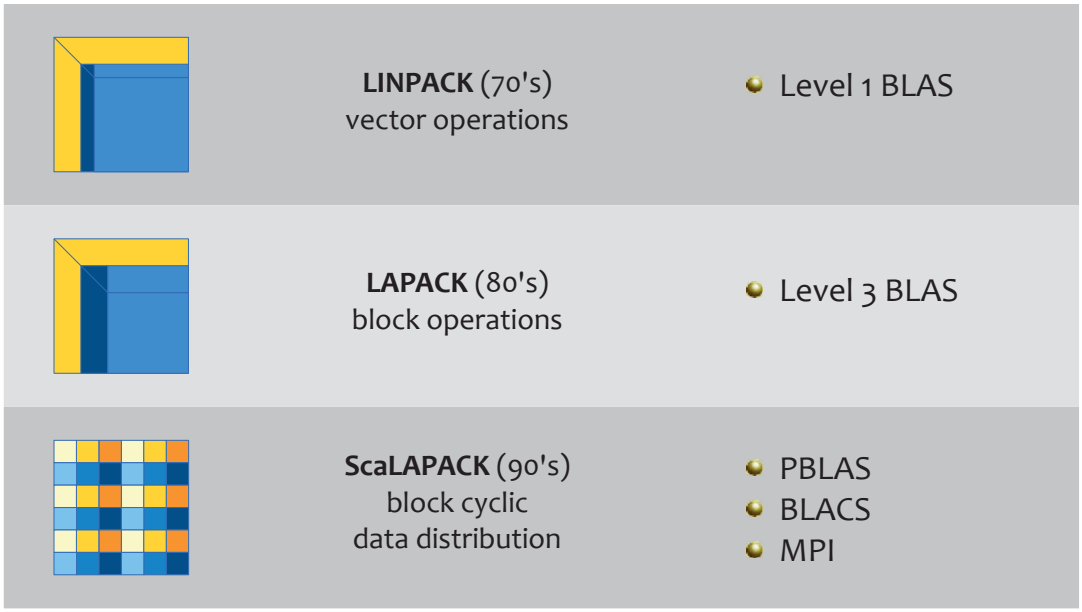


Figure 5.1: Evolution of Dense Linear Algebra legacy software libraries[50]

The next step in the history was to develop a distributed-memory version of LAPACK, which then was called ScaLAPACK [9]. It has been developed for other kinds of parallel architectures (for instance, distributed memory machines or massively parallel SIMD machines) and also the new block cyclic (see Appendix B.1) data distribution was introduced [2]. This library now extends the capabilities of the previous

libraries by high-performance linear algebra routines which can be nicely used for distributed-memory message-passing (MIMD) computers [9]. Furthermore, networks of workstations with the support of Parallel Virtual Machines (PVM) and/or MPI were added, and two exceptional libraries Basic Linear Algebra Communication Subprograms (BLACS) and Parallel Basic Linear Algebra Subprograms (PBLAS) were included. The main idea behind the ScaLAPACK library and its building blocks is that it uses distributed-memory versions of the Level 1, 2, 3 of BLAS called (PBLAS), and a set of BLACS for communication tasks that usually arise in parallel linear algebra computations. This behavior leads to that the majority of interprocessor communication happens within the PBLAS environment.

In Figure 5.2 we can see an overview of the DPLASMA software stack. In order to build an optimized DPLASMA library, there exist various ways of coming to a result. For the BLAS building block, there can be used several versions of preconfigured libraries including the fast commercial ones (MKL, ACML, ESSL, VecLib, LibSci), the fast academic one (ATLAS) or the unoptimized, reference, implementation from Netlib to build this block [50]. Thus here a certain freedom of optimizing the whole package from scratch is given. The next step is to setup the Netlib LaPACK (FORTRAN implementation), Netlib CLAPACK – C source (FORTRAN API) and the LAPACKE (C API) which all goes into the PLASMA component. The thread

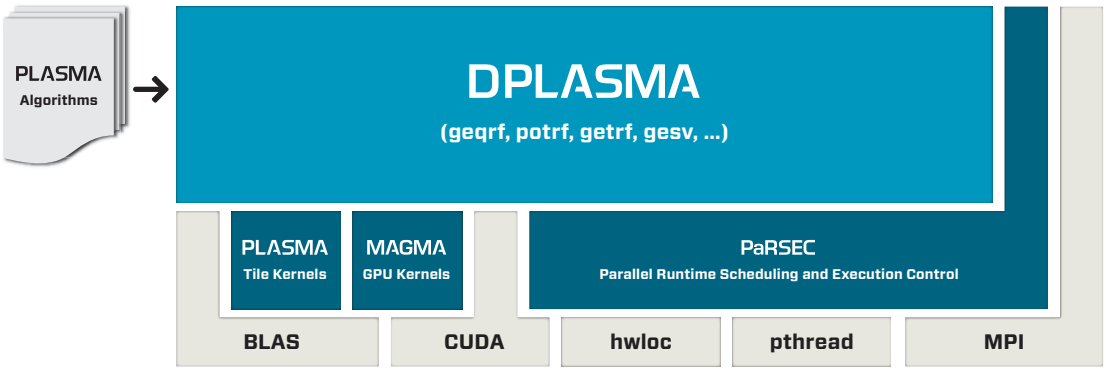


Figure 5.2: DPLASMA software stack [50]

affinity (sockets, cores) and the memory affinity (NUMA nodes) are acquired from the hwloc block. Furthermore, we have the hybrid computing with accelerators which is supported by the MAGMA component. Finally the generic framework PaRSEC is taking care of architecture aware scheduling and administration of micro-tasks on the heterogeneous architectures [18]. The PaRSEC runtime is further capable of porting the highly efficient PLASMA algorithms to the distributed memory realm. There is also a software hierarchy present which consists of the grey blocks and the other colored blocks. The grey ones are acting locally whereas the other ones are responsible for global communications and handling parallel routines.

The DPLASMA library used in this thesis was prepared and compiled with BLAS, LaPACK 3.7.1, OpenBlas, PLASMA 2.8.0, HWLOC 1.11.4 and OpenMPI 2.1.1 (for further details see Appendix A.3).

5.2 Local ABFT PDGEMM

Here we can see an extract of the local ABFT PDGEMM function using DPLASMA:

```
491         }
492         #ifdef FAULTINJECTOR
493            FaultInjector::run();
494         #endif
495         t[0] = std::chrono::steady_clock::now();
496         for(i=0; i<RUNS; i++){
497            /* Create DAGuE */
498            PASTE_CODE_ENQUEUE_KERNEL(parsec, dgemm,
499                                      (tA, tB, alpha,
500                                       (tiled_matrix_desc_t *)&ddescA,
501                                       (tiled_matrix_desc_t *)&ddescB,
502                                       beta,
503                                       (tiled_matrix_desc_t *)&ddescC));
504
505            /* execute kernel! */
506            PASTE_CODE_PROGRESS_KERNEL(parsec, dgemm);
507            elapsed_times[i] = elapsed_t = sync_time_elapsed;
508            dplasma_dgemm_Destruct( PARSEC_dgemm );
509
510            MPI_Reduce(&elapsed_t, &elapsed_t_overall_maxes[i], 1,
                  MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
511            MPI_Reduce(&elapsed_t, &elapsed_t_overall_mins[i], 1,
                  MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
512         }
513         t[1] = std::chrono::steady_clock::now();
514         FaultInjector::terminate();
```

Listing 5.1: Matrix Matrix Multiplication using DPLASMA

In Listing 5.1, there can be seen several building blocks. The first one is a preprocessor macro `#ifdef FAULTINJECTOR` which is responsible for activating the own-threaded *Fault Injector* if defined during the compilation. Then for calculating the time intervals for inserting the errors is set by `t[0]` and `t[1]`. The matrix-matrix multiplication is done in a loop for getting accurate timing results. Within the loop, there is a `PASTE_CODE_ENQUEUE_KERNEL` for creating the DAGuE/PaRSEC environment, `PASTE_CODE_PROGRESS_KERNEL` for executing the kernel, a destructor for the

PDGEMM and a reduce operation to get the minimum and maximum times of each processor for the operation. Further, the number of errors for each processor rank are gathered by `FaultInjector::getNrInjected()`. The corrections in this version are done after the multiplication has been successfully executed and after some faults were injected.

5.2.1 Structure

The implemented Local ABFT_PDGEMM algorithm [123] is build up to the following key facts:

- Input matrices: $A, B \in \mathbb{R}^{n \times n}, n \in \mathbb{N}^+$ and n = real matrix dimension (space considered without additional row/columns for the checksums)
- The default matrix values distribution is $A_{ij}, B_{ij} \in [-(10^4), 10^4]$ which is uniformly distributed (this range is the default used in the experiments, other possible ranges can also be specified)
- Weight matrix:
  - $W \in \mathbb{R}^{n_b \times d}, d \geq 1, n_b = \frac{n}{P} \vee \frac{n}{Q}$
  - P and $Q \equiv \#$ of nodes of a quadratic $(P \times Q)$-process grid
- Matrices A^r, B^c and C^f have extended dimension of $(n + (P * d)) \times (n + (Q * d))$ or $(N \times N)$, N = extended matrix dimension including the space for checksums
- The checksums in the matrices A^r and B^c are created by multiplying the weight matrix with the corresponding sub-matrix part to the scheme below. This is a sample notation for a (2×2)-node grid:
  - $A^r = \begin{pmatrix} A_{11} & A_{12} \\ W^T A_{11} & W^T A_{12} \\ A_{21} & A_{22} \\ W^T A_{21} & W^T A_{22} \end{pmatrix} B^c = \begin{pmatrix} B_{11} & B_{11} \cdot W & B_{12} & B_{12} \cdot W \\ B_{21} & B_{21} \cdot W & B_{22} & B_{22} \cdot W \end{pmatrix}$
- Matrix-matrix multiplication is done by: $C^f = A^r \cdot B^c$
- $C^f = \left(\begin{array}{cc|cc} & & \multicolumn{2}{c}{\text{node 1 } (C^f_{11})} \\ C_{11} & \text{cksm}_{11_2} & C_{12} & \text{cksm}_{12_2} \\ \text{cksm}_{11_1} & \text{cksm}_{11_3} & \text{cksm}_{12_1} & \text{cksm}_{12_3} \\ \hline C_{21} & \text{cksm}_{21_2} & C_{22} & \text{cksm}_{22_2} \\ \text{cksm}_{21_1} & \text{cksm}_{21_3} & \text{cksm}_{22_1} & \text{cksm}_{22_3} \\ \multicolumn{2}{c}{\text{node 3 } (C^f_{21})} & \multicolumn{2}{c}{\text{node 4 } (C^f_{22})} \end{array}\right),$

where $\text{cksm}_{11_1}, \text{cksm}_{11_2}$, and cksm_{11_3} are corresponding checksum arrays of C_{11} calculated by the sub-matrices of A^r and B^c etc.

- Correction matrix: $H = (W^T \ -I)$, where $I \in \mathbb{R}^{d \times d}$, $H \in \mathbb{R}^{d \times (n_b + d)}$
- Correction of sub-matrix C_{ij}^f by:
  1. $S1 = H \cdot C_{ij}^f$ and $S2 = C_{ij}^f \cdot H^T$
  2. $H(:, I2) \cdot C1 = S1(:, I1)$ or $H(:, I1) \cdot C2^T = S2(I2, :)^T$
  3. $\Rightarrow C_{ij}^f(I2, I1) -= C1$ or $C_{ij}^f(I2, I1) -= C2$

  $S1, S2 \equiv$ checksum sub-matrices which are gained by multiplication of the correction matrix H with the appropriate sub-matrix C_{ij}^f. $I1$ contains faulty columns in $S1$ and $I2$ contains faulty rows in $S2$. $C1$ and $C2$ are the calculated correction values by solving the least squares problem in point (2) for $C1$ or respectively $C2$.

5.2.2 Conditions

These are the special conditions which have to be fulfilled so that there can be a successful pass of the application:

- The first condition is the execution which has to be strictly in this order:
  `mpirun -np <NP> [-hosts <HOSTS>] ./labft_pdgemm_opt -N <matrix_size> -T <tile size> <d-value> <#faults> -x (for checking)...[further arguments]`
- Only $P = Q$ is supported (the node grid or processor grid has to be quadratic)
- T (tile size) has to be a divisor of N (extended matrix dimension)
- T has to be a divisor of the block size $n_b = \frac{N}{P}$
- T has to be $\geq \sqrt{(\frac{N}{P})}$ for optimal internal DPLASMA tiling performance
- d value has to be: $\leq \frac{N}{(P*2)}$
- P and Q have to be a divisor of N
- Double precision values with a range of $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$ and with a precision of approximately 16 decimal digits [78, 84] at least are to be preferred so that there is a leeway for the matrix correction and the corresponding solving of the least squares problem.

All these conditions are considered in the program and caught during the execution with appropriate exceptions.

5.2.3 Correction Algorithm

The correction algorithm [110] has the following detailed structure:

Algorithm 1: Correction algorithm of the ABFT_PDGEMM

Initialization:

$Num_{rows} \leftarrow 0, Num_{cols} \leftarrow 0$, init W, // W = weight matrix

set T_{val} // T_{val} = tolerance value for correction

Arrays: $S1[d*(N+d)]$, $S2[(N+d)*d]$, $H1[d*d]$

| $H2[d*d]$, $I1[d]$, $I2[d]$;

1: **for** All submatrices C_{ij}^f **do**
2: $S1 \leftarrow C_{ij}^f$ // copy corresponding row checksums from C_{ij}^f
3: $S2 \leftarrow C_{ij}^f$ // copy corresponding column checksums from C_{ij}^f
4: $S1 = (W^T \times C_{ij}^f) - C_{ij}^f$; $S2 = (C_{ij}^f \times W) - C_{ij}^f$
5: **for** All checksums in $S1$ **do**
6: **if** $S1 \neq 0$ **then**
7: **if** $!(|S1/\max\{\text{checksum}_1, \text{checksum}_2\}| < T_{val})$ **then**
8: **if** $Num_{cols} == d$ **then**
9: **return** -1 // correction with column checksums not possible
10: **end if**
11: $I1[Num_{cols}] = i$ // set to current index i
12: $Num_{cols}++$ // increment number of corrected columns
13: **end if**
14: **end if**
15: **end for**
16: **for** All checksums in $S2$ **do**
17: **if** $S2 \neq 0$ **then**
18: **if** $!(|S2/\max\{\text{checksum}_1, \text{checksum}_2\}| < T_{val})$ **then**
19: **if** $Num_{rows} == d$ **then**
20: **return** -2 // correction with rows checksums not possible
21: **end if**
22: $I2[Num_{rows}] = i$ // set to current index i
23: $Num_{rows}++$ // increment number of corrected rows
24: **end if**
25: **end if**
26: **end for**
27: Solve least squares problem: minimize $||S2 - H2^{**}T * X||$.
28: $C_{ij}^f[I2, I1] -= S2$
29: **return** $Num_{rows} * 10000 + Num_{cols}$
30: **end for**

Before the errors are corrected, it is verified if any of the processors have registered some fault injection. The first part of the correction algorithms contains setting up the required arrays and matrices. After that is ready and verified for all sub-matrices from the calculated matrix C, the values which are needed for correction are calculated locally. This part is done efficiently because the sub-matrices are usually small enough and can be processed pretty fast on each node or processor. Compared to the *Global* ABFT variant the communication is kept pretty low because no other processors or nodes have to communicate together for restoring the data of a whole node.

The inner part of the algorithm deals with finding the indices (for the column and the row) of the faulty value. The first loop goes column-wise and looks if there are errors in the column when by doing a division (in line 7) and comparing the given tolerance value(the T_{val} is described in the next section). If this is passed successfully and the number of columns is not exceeding the value of d then an error has been detected, and the index in $I1$ is set to the current index of the column, and the number for the corrected columns is increased by 1. When this number exceeds the value of d then "-1" is returned because it is not possible to correct more faulty values than the set checksums. For instance, 4 faulty values cannot be corrected with only three corresponding checksums. The same procedure is done row-wise for all checksums in $S2$. Here only the return value is different. It is "-2" if there are too many errors present.

After this two inner loops, the faulty rows and columns are aggregated and the remaining part is to solve the least squares problem according to $S2$. The notation minimize $||S2 - H2^{**}T * X||$ was taken from the documentation of the LaPACK function `LAPACK_dgels` [2, 56]. The last step is correcting the error in the sub-matrix by subtracting the correction value from the erroneous value and replace it by the result. What's more, the returning value in line 29 gives the verification of how many values have been detected and corrected in the column-wise and row-wise loop. An example value of "70003" will mean that seven errors were detected in the row section and three errors in the column section and further that they were also corrected. Nevertheless, this standard ABFT approach of correcting the value by computing the difference is not stable for large numerical differences like in the case of bit flips in the exponent. In [110] an improved approach called *dABFT* is developed, where the faulty values are replaced by 0 (zeroes) for minimizing the large numerical differences and so here nearly all bit flips can be corrected.

5.2.4 Tolerance Value

Here the tolerance value T_{val} which is an essential part of deciding whether a value will be corrected or not is described. The location where it is used can be looked up in line 7 or 18 of Algorithm 1 and it has the following meaning: For instance if absolute value of $S1$ divided by the maximum of the 2 checksum values is not smaller then the tolerance value then there is an error. Let T_{val} be (10^{-3}), $S1 = 12$ and $\text{checksum}_1 = 12$ and checksum_2 be 0. This will result in a division of $12/12 = 1$ and $1 < 0.001$ is not true so in this column there must be an error and the algorithm will correct this value. The index of $I1$ is set to the current column and the Num_{cols} value is incremented. Let us assume an example to show the opposite: Let T_{val} be (10^{-3}), $S1 = 12$ and $\text{checksum}_1 = 12$ and checksum_2 be 24000. This will give $12/24000 = 0.0005$, and $0.0005 < 0.001$ becomes true, and the algorithm will not correct the error because the tolerance boundary has been set too high. As a consequence setting the right tolerance value can become a tricky part. From the above example we can also see that if there are bit flips which brings major changes in the values (like bit flips in the exponent) the algorithm has no chance to repair the values from numerical point of view. The corresponding result can be seen in subsection 6.2.3.

There are some additional notes: Usually, the tolerance value T_{val} varies between (10^{-14}) for common problems and (10^{-7}) if the sub-matrices are big. However, this cannot fully close the machine-epsilon problem, because we only have a limited numerical representation of the chosen precision. This limitation means in the case of double precision that it is valid only up to 16 digits. Therefore this variant of correction recognition is not convenient for large numerical differences and so for correcting bit flips in the exponent. In the last subsection, a modified variant by [110] was discussed. Further, the checksum_2 value is calculated by the checksum in the sub-matrix C^f_{ij} + the value at the same index at $S1$ and the same is valid for setting checksum_2 of $S2$.

5.2.5 Local ABFT Space Analysis

In the appendices under Figure A.3, we can see the detailed space analysis of the *Local* ABFT algorithm used in this thesis. Compared to a standard matrix-matrix multiplication this will look like:

$$\text{Total elements std. GEMM} = 3 \text{ matrices} \times (n \times n) \text{ elements} = 3n^2 \text{ elements}$$

$$\text{Total elements labft PDGEMM} = dnP + 2(d^2 + dNP + dP^2 + 2dP + 2N^2 + P2)$$

When we remember the exascale example where we have 12.25 million of cores

(example approximated from the best GFlops/core performance over all current supercomputer in Section 4.5), this will lead to the following problem size with key parameters:

$$N = 717,500;\ P = 3,500;\ Q = 3,500;\ n = 700,000 \text{ and } \ d = 5$$

Using this values, we will get 2.1×10^{12} elements with using the fault tolerant variant versus 1.5×10^{12} elements for a standard matrix-matrix multiplication which results in a 42.7% overhead of space (RAM) for the *Local ABFT* method used in this thesis. If we subtract the extra matrix which is needed for value transformations and correctness verification, the overhead shrinks to **∼18%**. On the other hand, with this configuration, the algorithm can correct five errors per processor or node locally. This also means that over the whole matrix with 12.25 M cores there can be at most 61.25 million of fault injections corrected over the whole end-matrix C^f before the correction algorithm refuses. In another case where $d = 1$ and this is the minimum possible $d-$value for the algorithm, the overhead is leveling off at about 10%.

5.3 Fault Injector

For the results of this thesis, two fault injectors were used. This section will mainly deal with the own-threaded *Fault Injector* [110] which runs in the background and can be activated at different times in the execution process.

The results in Section 6.1 comes from the more straightforward and manual injector, whereas for the figures in Section 6.2 the own-threaded *Fault Injector* was used. The simple injector method is activated by the preprocessor macro `#if INJECTION` and also accepts an additional `injectionvalue` and a macro `#if INJECTRANK`. With this three predefined values (during the compilation) we can test a particular value to be inserted at a particular CPU-rank at the main diagonal of a sub-matrix. The other option is that at every sub-matrix of each CPU the sign of the value along the main diagonal is changed. Here the preprocessor macro `FAULTS` plays a role, where according to the setting of this value the sub-matrices are manipulated along the main diagonal. A setup with `FAULTS=5` means that the first five values along the main diagonal of every sub-matrix are getting their signs flipped (an example is shown below).

$$C_{ij}^{f} = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \text{ to } C_{ij}^{f} = \begin{pmatrix} -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdot & \cdots \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

In contrast stays the own-threaded *Fault Injector* which is far more complex and provides many functionalities for testing or approximating real-life scenarios. Further worst-case scenarios and little-affected scenarios can be examined by setting the appropriate MTBF. It supports randomized manipulation of the values by flipping a sign, flipping bits in the exponent, in the mantissa or a combination of all. The Fault Injector depends on the spatial and temporal data distributions which are playing a role during the error insertion. First, we start with the definition of the Fault Injector which is showed in Listing 5.2.

```
58  FaultInjector::setDefaultFlipRange(fliprange::FLRANGE);
59  //FaultInjector::setDefaultFlipRange(fliprange::SIGN);
60  //FaultInjector::setDefaultFlipRange(fliprange::EXPONENT);
61  //FaultInjector::setDefaultFlipRange(fliprange::MANTISSA);
62  //FaultInjector::setDefaultFlipRange(fliprange::ALL);
63
64  UniformDataDistribution* dataDistribution = new
        UniformDataDistribution();
65  FaultInjector::setDataDistribution(dataDistribution);
66  ExponentialTimeDistribution* timeDistribution = new
        ExponentialTimeDistribution(mtbf);
67  FaultInjector::setTimeDistribution(timeDistribution);
```

Listing 5.2: Definition of Fault Injector

Here again, the `FLRANGE` acts as preprocessor macro which is set during the compilation and can accept the names `SIGN, EXPONENT, MANTISSA` or `ALL`. Furthermore, it is set up with uniform data distribution and an exponential time distribution explained in subsection (5.3.1) and (5.3.2).

5.3.1 Spatial Data Distributions

In [42] we have the explanation for the uniform discrete distribution which follows the probability mass function in Equation 6.2. Basically, this gives a random integer number between the first and the last index of a sub-matrix. This index is then used for manipulating this specific value. The random number generator is set by `srand` [41], and produces random numbers in a range of $[a, b]$ where a is the first index of the sub-matrix and b the last one minus one (because indices of arrays start at 0) [42]. Further, in the composed range each possible value has an equal likelihood of being produced.

5.3.2 Temporal Data Distributions

According to [74] in section faults and failures, the frequency at which unrecoverable failures that interrupt the execution of the application occurs are in a manner which is similar to exponential probability distributions. Moreover, in a fault-prone environment, the time-steps at which faults are happening are non-deterministic, and so they vary between each execution. Therefore as temporal data distribution in this thesis, the exponential time distribution was used, which is defined in [44]. Further, it uses the default_random_engine [40] for generating pseudo-random numbers and the following probability density function:

$$p(x|\lambda) = \lambda e^{-\lambda x}, \ x > 0 \tag{5.1}$$

The important part here is that the returned value is the time/distance until the next random event. If random events occur, they are independent but statistically defined at a constant rate λ per unit of time/distance. Additionally also all requirements of the C++ `RandomNumberDistribution` concept [43] are satisfied. The random number generator is here again set by `srand` [41].

Mainly the *Fault Injector* produces errors according to a timing value which can be set at execution time. This value is the mean time between failure (MTBF) which can be given to the application as a value of days. The MTBF as given argument of 1 equals to the value of 1×86400 seconds (the time of seconds for 1 day). Therefore the minimum value at which can be examined is an MTBF of 1 second by using the value 0.00001 as an argument. Moreover, the Fault Injector algorithm is processing according to the scheme:

$$\text{MTBF} = 7 \implies 7 \text{ seconds per failure per bit}$$

This situation is achieved by calculating the data in bits which is also needed when the data is malformed at a particular bit. Therefore to accomplish a particular fault rate we also need to consider the matrix-values as a number of bits. Assume an example where we want to achieve a value manipulation of every 7 seconds in a (64×64)-matrix then the following setup will be required:

1. Main matrix dimension = (64×64) = 4096 elements = 32768 bytes = 262144 bits
2. Sub-matrix = $8 \times 8 = 64$ elements = 512 bytes = 4096 bits
3. Processor grid $P \times Q = 8 \times 8$
4. Data calculated in 7 seconds **not** exceeds 262144 bits
5. MTBF set to the value of 1,835,008 seconds or 21.24 days

That means concretely that if there are 262144 bits processed over a time window of 7 seconds the average fault injection rate will result in one value manipulation every 7-th second. This value comes from the fact that the set MTBF value for the *Fault Injector* is also dependent on how many bits the data have and therefore we can determine the MTBF argument which is needed as $(7 \times 262144 = 1,835,008)$ or 21.24 days. On the other hand, we can see that the situation of every 7 seconds one value manipulation is not realistic because with this setting many more bits then 262144 will be processed within such a long time of computation for current processors. In a real case the multiplication for a (64×64)-matrix would last only about 0.01 seconds and the MTBF value has to be only $(0.01 \times 262144 = 2621.44)$ about 2600 seconds. Therefore it is difficult to achieve a certain fault rate. The execution time has to be estimated with several runs and after that the MTBF value can be calculated. Below, in Table 5.2 some investigations with real values, where the MTBF in days for one value manipulation per execution time was calculated can be found.

M	**N**	**# of Elements**	**# of Bits**	**Exec. Time (sec)**	**MTBF (sec)**	**MTBF days**	**Grid P × Q**
640	640	409600	26214400	0.6	15.73 M	182.05	4 × 4
1280	1280	13107200	104857600	0.8	83.89 M	970.91	4 × 4
3200	3200	81920000	655360000	3	1.97 B	22756	4 × 4
6400	6400	327680000	2621440000	13	34.08 B	394430	4 × 4

Table 5.2: Setting the Fault Injector with different MTBF

Concluding on the fault injection algorithm, we can say that it is useful to test and simulate a specific failure rate regarding the MTBF of a hardware component or for instance of a node.

Chapter 6

Experiments

To verify that ABFT methods make sense when hardware components, especially number of cores and nodes are growing there were several test cases chosen. The test cases are described in the next section. For the verification of the error between the calculated data and the manipulated data besides the norms provided by DPLASMA the relative 1-norm was mainly used:

$$\text{rel 1-norm} = \frac{||C - C'||_1}{||C||_1} \tag{6.1}$$

where C argues as the correct matrix-matrix multiplication without bit flips and C' as the calculated matrix with bit flips. Actually, two separated matrix products are made to verify the correctness. For generating the values of matrices A and B the function `initDP_local` shown below in detail (description is continued on the next page) was used.

```
925 void initDP_local(int m, int n, double a, double b, double* A,
        unsigned long seed, int rank)
926 {
927     int i = 0, j = 0;
928     std::uniform_real_distribution<double> distribution(a,b);
929     std::default_random_engine generator;
930     generator.seed(seed+rank);
931     for (i=0; i<m; i++)
932     {
933         for (j=0; j<n; j++)
934         {
935             A[i+j*m] = distribution(generator);
936         }
937     }
938 }
```

Listing 6.1: Generator for uniformly distributed double precision values

This generator has an impact on the values for each processor because it produces different values depending on the seed of each rank which is set by seed + the rank ID which is a number between 0 and max number of processors in the system. Further, there exists an impact on the value generation performance as the values are generated for the sub-matrices of a given problem by each appropriate processor. The `std::uniform_int_distribution` [42] produces random integer values i, uniformly distributed on the interval $[a, b]$, with the probability function:

$$P(i|a,b) = \frac{1}{b-a+1}, \ a \leq i \leq b \tag{6.2}$$

and satisfies all requirements of the C++ `RandomNumberDistribution` concept [43].

The runtime was measured for each major part of the algorithm separately. There is a measurement for the `DPLASMA_dgemm` (the DPLASMA own double precision matrix-matrix multiplication) which is done 5-times and the minimum over all maximal values is logged, for the correction time and for the ABFT relevant parts. The correction time and the ABFT relevant parts are measured only once but here also the maximum time over all MPI-processes is taken. The total ABFT runtime is then calculated by summing up all three values. This total ABFT overhead time value [1] is then compared to a `DPLASMA_dgemm` standard execution with the same problem- and matrix block size to get the relative time comparison and relative ABFT overhead.

GFlop/s information is gathered and calculated by the own build in DPLASMA function `FLOPS_DGEMM( M, N, K )` and then later rescaled to GFlop/s by the calculation in Listing 6.2. The `elapsed_t_overall_min` value is the measured runtime for the extended DPLASMA matrix multiplication, M, N, and K are the dimensions of the matrices, whereas the `flops` value comes from the steps showed in the listing below.

```
#define FMULS_GEMM(__m, __n, __k) ((double)(__m) * (double)(__n) *
    (double)(__k))
#define FADDS_GEMM(__m, __n, __k) ((double)(__m) * (double)(__n) *
    (double)(__k))
#define FLOPS_DGEMM(__m, __n, __k) (      FMULS_GEMM((__m), (__n),
   (__k)) +        FADDS_GEMM((__m), (__n), (__k)) )
flops = FLOPS_DGEMM( M, N, K );
gflops = (flops/1e9)/elapsed_t_overall_min;
```

Listing 6.2: How FLOPS for a DGEMM are defined and Gflop/s calculated

Detailed and more exhausted information about the project files and test environment can be found under (section A.1).

[1] The total overhead includes the time for preparing the checksums, doing necessary computations on them, communication and transposition, error detection and correction.

6.1 Test Cases General

This thesis concentrates on the following test cases to witness the fault tolerance and efficiency of ABFT methods for dense matrix operations:

1. Different value distributions - Values of various uniform distributions
2. No errors overhead - For comparison when no errors occur
3. Sign bit flips - Using two different fault injectors
4. Bit flips in mantissa - Using an own thread for producing errors in the mantissa
5. Bit flips in exponent - Same as before but producing errors in the exponent
6. Bit flips everywhere - Same as before, but including the opportunity of producing errors in the mantissa, exponent or sign bit flips in the values' bit mask.

The bit flips were initiated by using two different fault injectors, a special sign flip injector and an own threaded *Fault Injector* capable of feeding in bit flips in mantissa, exponent, everywhere and random sign flips. The first fault injector is manually set up and can only perform sign flips at a certain time in the execution and along the main diagonal. This injection can be initiated before the calculation of matrix C has started or instantly after the calculation of the matrix-matrix multiplication. Otherwise, the second fault injector uses an own thread and can be executed at any time during the computation of the application and everywhere in the bit mask of a matrix value (for details see Section 5.3).

The range for the examined matrix dimensions was $N = [1000, 12560]$ where N stands for the extended matrix dimension (matrix dimension plus extended rows/-columns for the checksums). The dimensions were in some cases matched to the processor grids, e.g. (3000×3000)-matrix on a (3×3)-process grid or (10000×10000)-matrix on a (10×10)-process grid like in the experiments for no errors overhead (subsection 6.1.2).

6.1.1 Value Distributions

Here the aim was to test various uniform distributions and how this matrix value initialization affects the relative 1-norm described at the beginning of Chapter 6. The first figure on the next page shows how the ABFT correction algorithm performs when there are different initial values.

In Figure 6.1 we can see that there is indeed a difference between the various distributions. To point out on the distribution with values between 0 and 1, we can see that the correction of the injected sign flips has been evaluated with a very low overall error, this is also applicable over all possible matrix sizes. The other four

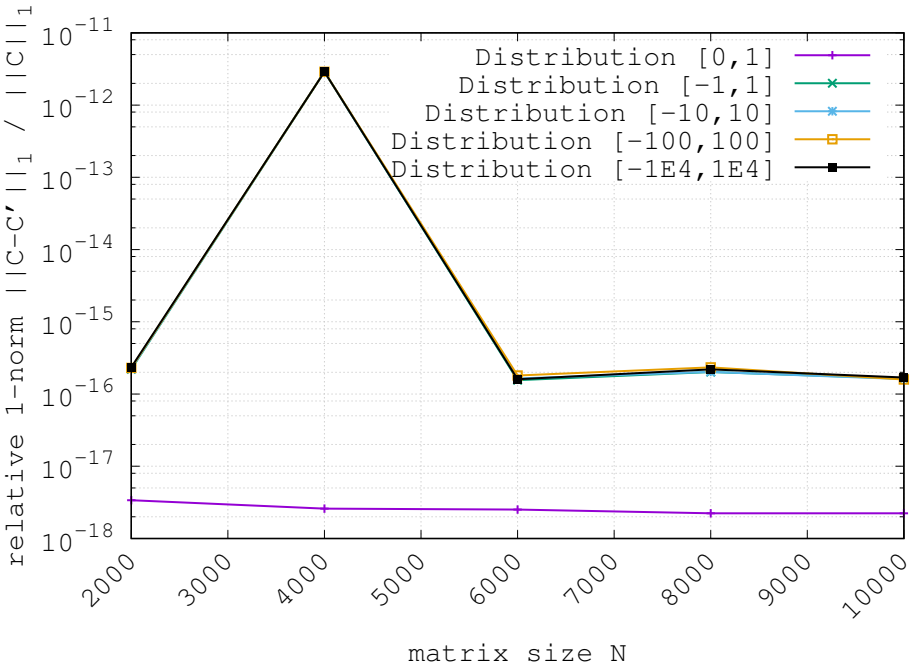


Figure 6.1: Different data distributions on various grids with fixed 200 sign flips per node and relative 1-norm illustration

distributions are acting nearly identical. There exists one outlier at the (4000×4000)-matrix dimension which further belongs to a (4×4)-node grid. Here the simulated (4×4)-process grid produces a much higher relative error as in all other processor grid variants. The other grids including $(2 \times 2), (6 \times 6), (8 \times 8)$ and (10×10) are having a relative 1-norm of about 10^{-16}, whereas the (4×4)-grid results in 10^{-12}. This outlier has been analyzed, and it came out that it has something to do with the tile size value T of the given matrix. When setting the tile size to 200, the 200 sign flips per node are producing some unusual value constellation for the whole matrix, whereas if the tile size was different, e.g., 100, 250, 500 or 1000 the behavior was normal and as expected (Table 6.1). Furthermore, this particular configuration leads to the situation that the matrix correction algorithm does not succeed in repairing all erroneous values. As the tile size also have an impact on how the matrix is partitioned into 2-D block-cyclic sub-matrices this further modifies the results of the correction algorithm, and the relative 1-norm varies for the different configurations.

NODES	M	N	Tile Size	relative 1-norm	Faults Injected	Faults Corrected
16	4000	4000	100	1.79E-16	3200	3200
16	4000	4000	200	2.89E-12	3200	3199
16	4000	4000	250	2.23E-16	3200	3200
16	4000	4000	500	1.84E-16	3200	3200
16	4000	4000	1000	2.15E-16	3200	3200

Table 6.1: Verification of the tile size responsible for unusual behaviour

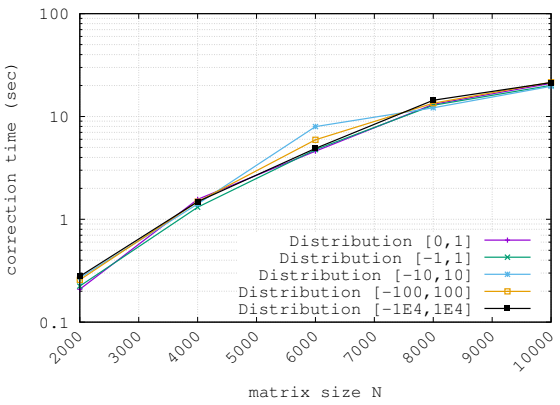


Figure 6.2: Different data distributions on various grids with fixed 200 sign flips per node and verification of the correction time

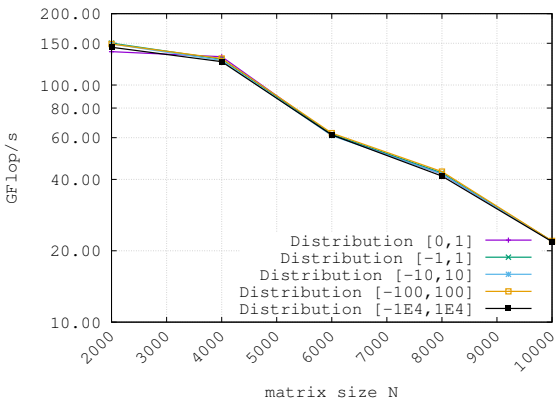


Figure 6.3: Different data distributions on various grids with fixed 200 sign flips per node and performance evaluation in GFlop/s

In Figure 6.2 the time which is needed to correct the inserted sign flips is visualized. We can see of course a time increase as the matrix dimensions are rising, but this trend seems to flatten out when the problem size becomes large enough. The only outlying event which can be found is at (6000×6000)-matrix on a (6×6)-grid where the correction time varies between 8 and 10 seconds, but this has no considerable leverage on the whole runtime.

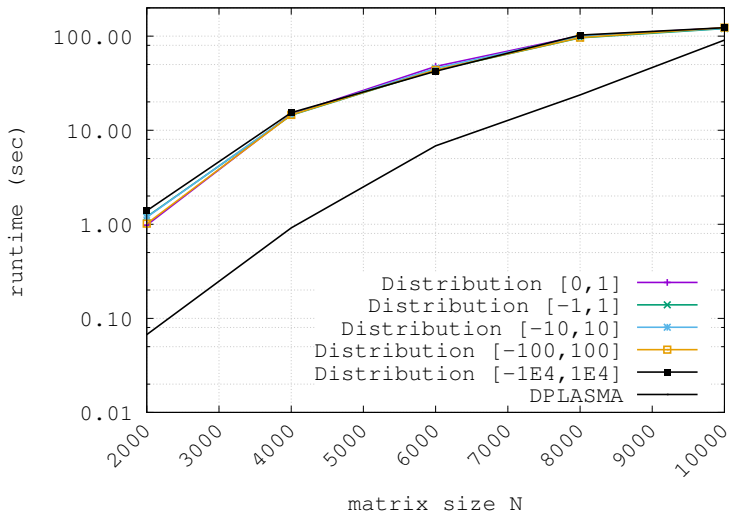


Figure 6.4: Different data distributions on various grids with fixed 200 sign flips per node total time comparison

What's more (Figure 6.3) is dealing with the performance of the ABFT_PDGEMM algorithm. As according to the appendix in Section (A.1) the system environment consist of a real (2×2)-NUMA-node grid the full performance can be found on the plot in the (2000×2000)-matrix size section which equivalently stands for a (2×2)-node grid.

The GFlop/s performance is dropping as the grid size, and consequently, the problem size is growing. The explanation for these heavy losses of efficiency is that the other simulated node-grids do not overlap with the real hardware structure and therefore there is an increasing amount of communication overhead which can be clearly verified by the produced curves in the plot. There is a continuous drop from 150 GFlop/s (on a (2×2)-grid) to about 20 GFlop/s (on a (10×10)-grid). More information on the system's peak performance can also be found in Appendix A.1.

Figure 6.4 concentrates on showing that the runtime gap is getting smaller and smaller as the matrix size and ergo the node-size is growing. We have on a (10000×10000)-matrix, with corresponding (10×10)-process grid a time difference of about 30 seconds in comparison to 78.92 sec for the (8000×8000)-matrix. Furthermore, the distribution of the matrix values does not give any reasonable discrepancy. The more precise data can be obtained from Table 6.2 located below.

NODES	M	N	GFlop/s	ABFT tot Time (sec)	DPLASMA Time (sec)	Time difference
4	2000	2000	144.32	1.39	0.07	1.33
16	4000	4000	125.49	15.37	0.92	14.46
36	6000	6000	61.62	42.37	6.82	35.55
64	8000	8000	41.31	102.63	23.71	78.92
100	10000	10000	21.85	123.15	90.83	32.32

Table 6.2: Summary of relevant values for matrix value data distribution $[-1E4, 1E4]$

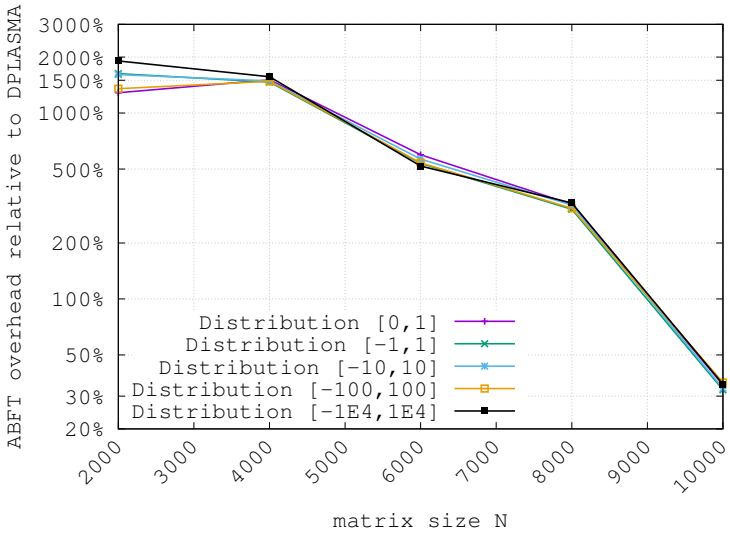


Figure 6.5: Different data distributions on various grids with fixed 200 sign flips per node relative overhead compared to DPLASMA_dgemm

The described situation before can also be verified with the results in Figure 6.5. It shows that the overhead is shrinking as the matrix size N is growing. Further, the

condition with different matrix value distributions remains the same as before (no significant differences).

6.1.2 No Errors Overhead

The interesting part in this subsection focuses on how much the ABFT_PDGEMM algorithm overhead usually is when there are no faults injected at all. Thus for analysis, several runs have been prepared to measure and visualize it. As there are no faults present, a plot for the relative 1-norm like in Figure 6.1 was not prepared because it is apparently 0 everywhere. Therefore we are jumping straight to the total runtime reflected in Figure 6.6. Paying attention to the upper two curves in the graph, we can see that the DPLASMA variant and the ABFT variant of the matrix-matrix multiplication are nearly overlapping. Sometimes the DPLASMA_dgemm is in the lead, sometimes the ABFT_PDGEMM, although there are logged several runs of both, the difference in the execution time varies. There is also the explanation that the block size in this experiments is fixed and there exists for every matrix dimension a specific optimal block size so that it can be adapted to reach optimal performance and therefore an optimal curve progression. As there was executed an automatic script with increasing dimensions in a loop, unfortunately in this particular case only one measurement was successful for the (8×8)-grid. This situation has mainly to do with the conditions mentioned in subsection 5.2.2 which all have to be fulfilled to achieve a successful run. There is a specific matrix size, sub-matrix block size, node processor grid, additional rows/columns for the checksums and other conditions which have to be satisfied. We can see of course that again the (2×2)-process grid versions are the fastest with DPLASMA_dgemm as a leader.

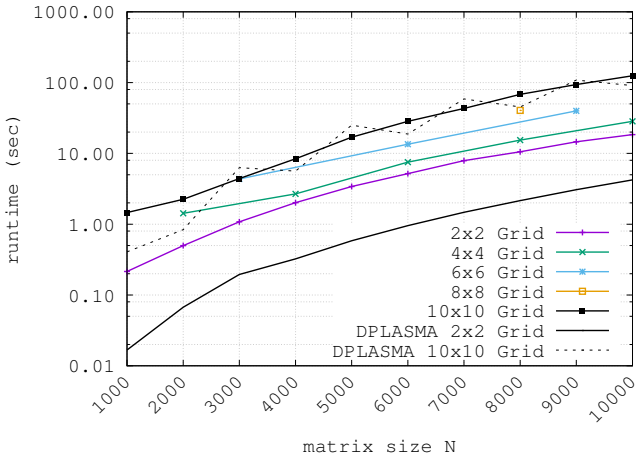


Figure 6.6: Comparing ABFT_PDGEMM to DPLASMA_dgemm when no faults occurred during runtime measuring

In Table 6.3 the relative overhead comparing to the standard DPLASMA_dgemm routine for a (10×10)-processor grid is emphasized. Since the matrix size increases, the relative overhead drops to a rate of about 10%. This progression can also be observed in Figure 6.7 and there is a clear trend that the overhead declines when the matrix size and likewise the processor grid size are growing.

NODES	M	N	GFlop/s	ABFT Overhead Time (sec)	DPLASMA Time (sec)	Relative Overhead
100	1000	1000	10.28	1.27	0.41	312.10%
100	2000	2000	13.36	1.05	0.84	125.01%
100	3000	3000	16.39	1.12	6.30	17.74%
100	4000	4000	20.64	2.19	5.62	39.05%
100	5000	5000	17.75	2.83	25.05	11.30%
100	6000	6000	17.91	4.21	18.80	22.42%
100	7000	7000	18.23	5.49	58.73	9.34%
100	8000	8000	17.49	9.69	45.30	21.39%
100	9000	9000	17.47	10.32	108.62	9.50%
100	10000	10000	17.35	9.72	90.83	10.70%

Table 6.3: Summary for overhead on a (10×10)-processor grid when no errors occurred

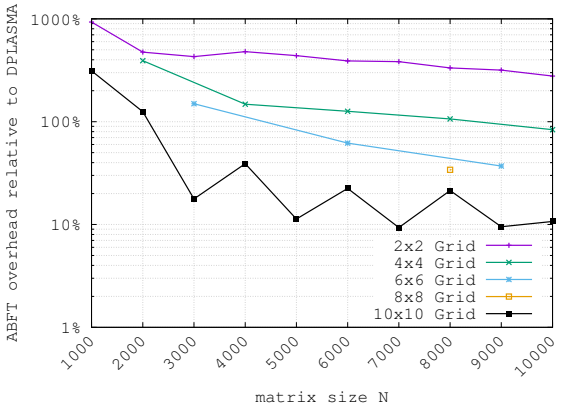


Figure 6.7: Comparing the ABFT matrix-matrix multiplication algorithm to the DPLASMA_dgemm when no faults occurred with focus on relative overhead

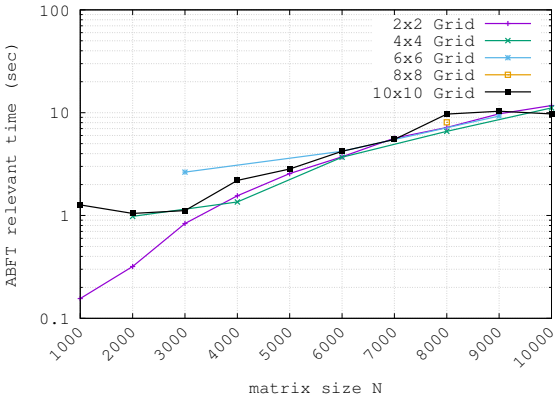


Figure 6.8: ABFT_PDGEMM relevant time in seconds for various grids when no faults are injected

An interesting observation can be seen in Figure 6.8. The time for the ABFT specific part remains relatively constant when the matrix size is big enough, and so the algorithm shows stability independently from the processor grid size and the matrix dimension.

6.1.3 Manual Sign Bit Flips

In the first series of testing, there was developed a manual fault injector. This fault injector was used to determine if the detection and correction algorithm is working correct when sign bit flips are inserted at a particular point of the execution. The following plots show the results of this examination. We have at the beginning the analysis of the relative 1-norm (Figure 6.9). The behavior is like it is expected when a single bit flip is involved the error is very low (at about 10^{-18}) while the first curve in the plot represents the case where 256 bit flips were initiated per node. It is obvious that the correction of 256 bit flips has a more significant effect on the whole matrix error than if there was only a single sign bit flip per node, and this is also clearly represented in this figure.

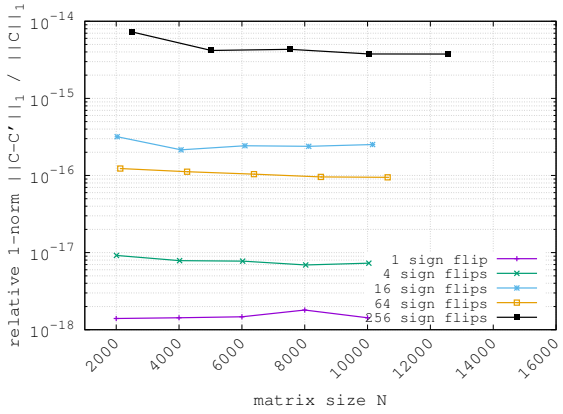


Figure 6.9: Relative 1-norm for various manual sign bit flips

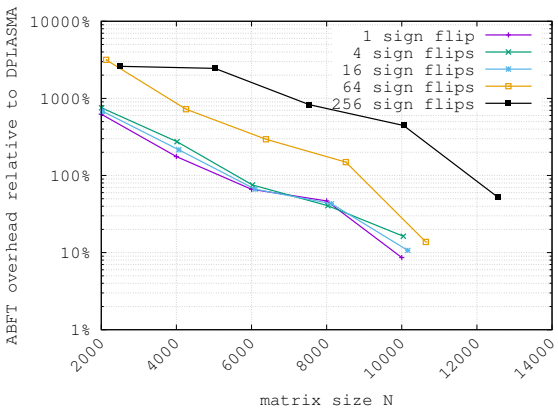


Figure 6.10: Various number of sign flips per node for relative overhead in percent

Next to the relative 1-norm chart in Figure 6.10 the relative overhead in relation to the DPLASMA_dgemm is exemplified. The higher matrix dimensions come from the additional rows/columns which are needed to repair the errors in matrix C successfully. For the case with 256 bit flips per node on a (10×10)-nodes machine hence it implies that the extended matrix dimension has to be set to $N = 12560$ because 10 nodes $\times 256 = 2560$ additional rows/columns and further equals a $10000 + 2560 = 12560$ of matrix size. This size is needed if we want to guarantee that there can be (10000×10000) elements stored in the matrix and additionally up to 256 bit flips can be corrected per computing node. What's more, as the matrix size keeps growing, the overhead is subsiding which is one of the nice features of parallel ABFT methods.

So in Figure 6.11 and 6.12 the runtime is again the relevant point. We can see that the runtime for DPLASMA is leading anew, but that is clear because there is no overhead for correction and no ABFT specific quota measured. The interesting part of this charts is that the gap and consequently the overhead between the DPLASMA

variant and the ABFT variant is getting smaller and smaller. Besides the only difference here is in the x-axis, where on the left the matrix size and on the right, the node size is considered. When we take a closer look at the standard deviation we can see that up to 64 bit flips can be corrected per node without rising the time overhead for the correction too much. This is verified in Table 6.4.

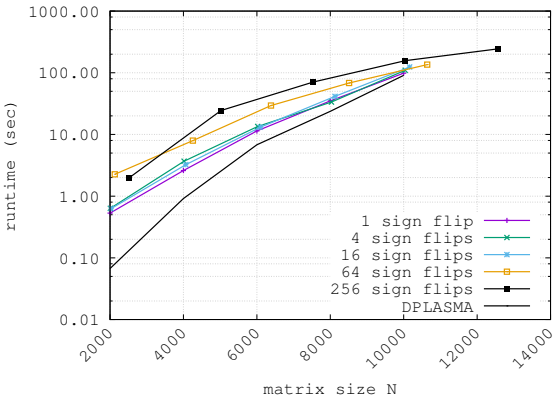


Figure 6.11: Time comparison with focus on the matrix size for various number of sign flips per node

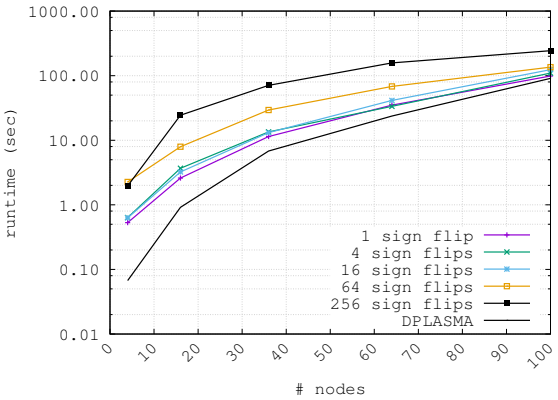


Figure 6.12: Time comparison to the DPLASMA_dgemm routine for various number of sign flips per node with emphasis on the node size

The formula for the used standard deviation [107] in Table 6.4 is defined by the following equation:

$$s = \sqrt{\frac{1}{(N-1)} \sum_{i=1}^{N} (x_i - \overline{x})^2} \tag{6.3}$$

where $\overline{x}$ = mean value of the observed values$\{x_1, x_2, \ldots, x_N\}$

If no more than 64 faults occurred, the standard deviation is pretty low, and so

	Correction time for **x** faults in (sec)						
NODES	**1 fault**	**4 faults**	**16 faults**	**64 faults**	**256 faults**	**Standard Deviation till 64**	**Standard Deviation incl. 256**
4	0.04	0.03	0.09	0.17	0.35	0.06	0.13
16	0.13	0.16	0.28	0.39	2.89	0.12	1.19
36	0.31	0.38	0.55	1.00	8.01	0.31	3.34
64	0.53	0.56	0.86	1.55	16.40	0.47	6.96
100	0.60	0.70	1.42	2.28	25.04	0.78	10.66

Table 6.4: Correction time for various number of errors per node

this configuration can be used without loosing of significant performance. Instead, if

we want to assure that 256 bit flips can be corrected, this might end in an essential overhead for correcting them all.

In real life, we can assume that this 64 faults per node value should be enough at least for a supercomputing system going towards exascale. As it is expected that such supercomputers will consist of million of nodes [20], each node then would get the capability to repair up to 64 bit flips without having that much effect on the total runtime of the ABFT algorithm.

6.2 Test Cases using Fault Injector

In this section the testing cases using the own-threaded *Fault Injector* [110] described in Section 5.3 are visualized. Beginning with the simplest case of a sign bit flipping and evolving to the general case where everywhere in the bit mask of a matrix value a modification can occur. The emphasis will lie on showing which bit flips are the most dangerous for the correcting algorithm and which can affect the result of the whole matrix-matrix multiplication.

6.2.1 Sign Bit Flips with Fault Injector

The attendant results here should show how frantically the own-threaded *Fault Injector* acts during the insertion of sign bit flips.

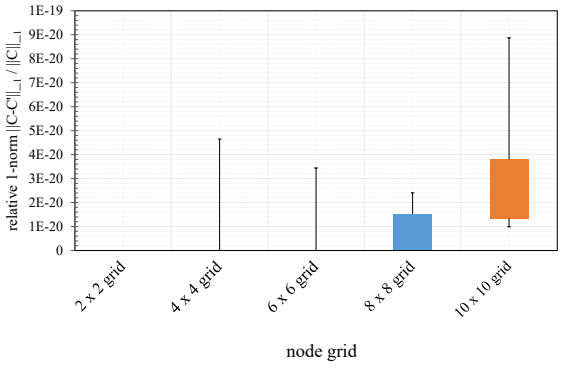


Figure 6.13: Box plot for Fault Injector bit sign flips

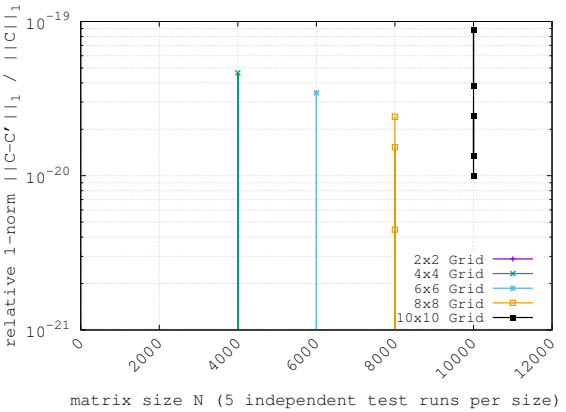


Figure 6.14: Relative 1-norm of 5 independent test runs when using Fault Injector for bit sign flips

From Figure 6.13 and 6.14 we can identify the relative 1-norm. They show clearly that the norm is below 9E−20 which leads to the conclusion that all sign bit flips were detected and corrected successfully. The explanation for the first empty result

at (2000 × 2000) is that there was produced such a little error or no error at all. Although there were five independent measurements each, there were cases where no bit flips occur. On the other hand, we can see from Figure 6.14 that for the (10000 × 10000)-case there are five data points on the vertical line. This illustration means that in every test run sign bit flips were done and further as the norm stays below 10^{-19} all of them were successfully corrected.

The third plot (Figure 6.15) shows that the overhead is relatively high when the matrix size and accordingly the processor grid size is small. It also shows that with growing size the relative overhead to the DPLASMA execution drops. At the (10000 × 10000)-test cases we have an average overhead of about 15%. All bars are showing the mean over the five independent test runs for each size.

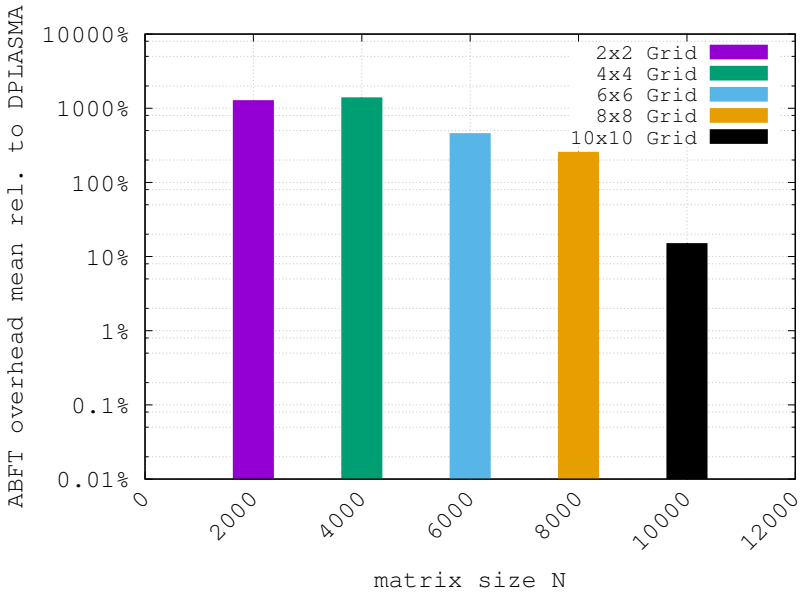


Figure 6.15: Comparing ABFT_PDGEMM to DPLASMA_dgemm when sign bit flips are inserted on different grid sizes

6.2.2 Bit Flips in Mantissa

In the previous subsection we saw the situation with sign bit flips, and when the ABFT algorithm has to correct them, now the emphasis lies on how severe bit flips are when the injection is in the mantissa of some matrix values. A good overview can be shown again with a box plot (Figure 6.16). The conclusion from the graph is that there were injected errors in all grid sizes and later corrected. The worst case can be found on the (10 × 10)-grid where the relative 1-norm (Equation 6.1) reaches below 10^{-11}. Despite the fact that all errors were corrected the impact on the whole matrix was fairly measurable. Further the (2 × 2)-grid hows the most stable results from the five measurements.

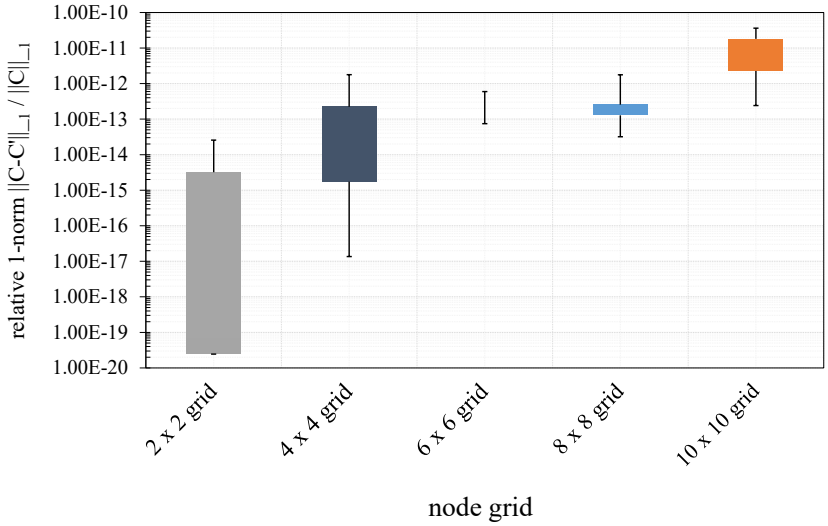


Figure 6.16: Box plot for bit flips in the mantissa using the own-threaded Fault Injector

Figure 6.17 show the results with lines and data points. From this plot additionally, we can determine how many of the five independent test runs were passed. To summarize from left to right we have on the 2000s matrix dimension 4 out of 5, then 5 out of 5, 2 out of 5, 4 out of 5 and finally the (10000 × 10000)-test cases 5 out of 5. The 4 out of 5 in the (2000 × 2000) and the 2 out of 5 in the (6000 × 6000)-matrix dimension means that there can also be cases that no faults were injected at all. In Figure 6.18 we can see again a scenario of an overhead that can be compared to the previous subsection. The crucial point is the greater the grid size, the lower the overhead.

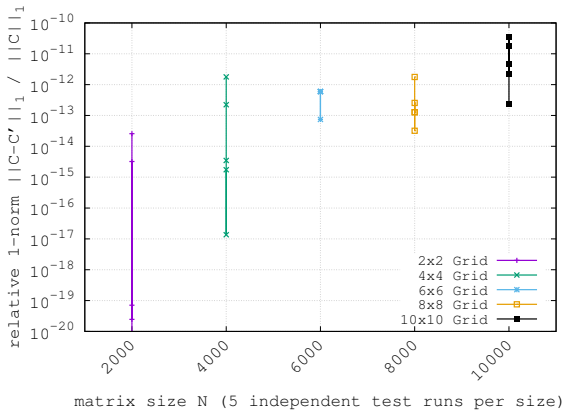


Figure 6.17: Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips in mantissa

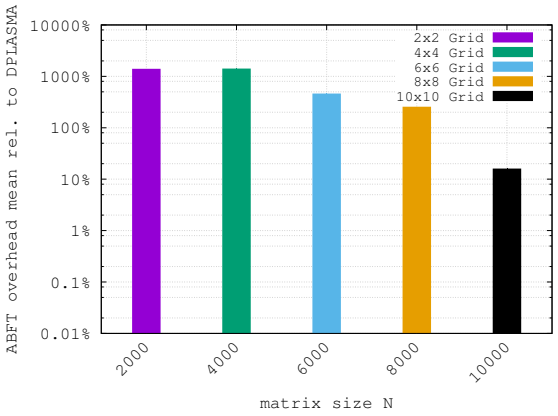


Figure 6.18: Relative overhead when bit flips are inserted in the mantissa for different grid sizes

6.2.3 Bit Flips in Exponent

The situation when injecting bit flips in the exponent of a matrix element can evolve in the most severe direction. This has actually to do with the properties of the exponent of a value. In Figure 6.19 we can see how the Double-precision floating-point format (DPFPF) looks like in the IEEE 754 standard [84, 143]. A number example for the decimal representation as binary and exponent representation can be the number $0.15625_{10} = 0.00101_2 = 1.01_2 \times 2^{-3}$. The first is the decimal representation, the second the binary and then the shifted binary and exponent representation. Now we can analyze the fraction and the exponent where $.01_2$ is the fraction part and -3 the exponent [145]. For the DPFPF we have an exponent range of $[-1022, +1023]$.

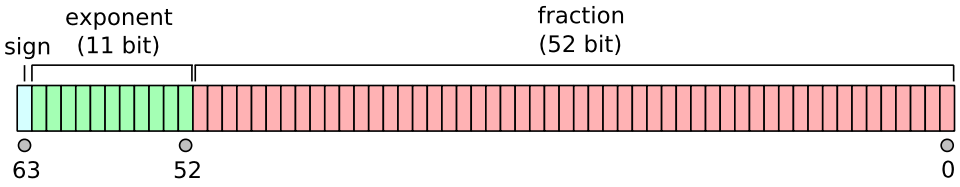


Figure 6.19: IEEE 754 Double-precision floating-point format [143]

In the following real case example, it's shown why bit flips in the exponent are so dangerous. The essential point is the massive difference in the values which can be easily created by changing the exponent arbitrary. We can see the green marked values in the left-most box and how they are changed in the middle box. The right-most box shows the difference between original values and changed values by exponent bit flipping.

matrix_C without_injections	on	rank	0
0.327535	0.258814	0.586349	0.247416
0.643272	0.792656	1.435928	0.678956
0.970807	1.05147	2.022277	0.926373
0.508411	0.594776	1.103187	0.515094

matrix_C without_injections	on	rank	1
0.251349	0.421653	0.673002	0.341281
0.378770	0.577783	0.956553	0.475172
0.630120	0.999436	1.629556	0.816452
0.312169	0.484561	0.796730	0.397304

matrix_C without_injections	on	rank	2
1.049830	1.406764	2.456594	1.184872
0.529480	0.704400	1.233880	0.594127
1.579310	2.111164	3.690474	1.778999
0.589315	0.786216	1.375531	0.662770

matrix_C without_injections	on	rank	3
0.725272	1.064052	1.789323	0.881152
0.356160	0.501837	0.857996	0.418664
1.081431	1.565889	2.647320	1.299816
0.400589	0.573663	0.974252	0.477152

matrix_C_with exponent_injection	on	rank	0
0.020471	0.258814	0.586349	0.247416
0.643272	0.792656	1.435928	0.678956
0.970807	1.051470	0.000000	0.926373
0.508411	0.000009	1.103187	0.002012

matrix_C_with exponent_injection	on	rank	1
0.251349	0.421653	0.673002	0.341281
0.378770	0.577783	0.956553	0.475172
0.630120	0.999436	1.629556	0.816452
0.312169	0.484561	0.796730	0.397304

matrix_C_with exponent_injection	on	rank	2
1.049830	1.406764	2.456594	1.184872
0.529480	0.704400	1.233880	0.594127
1.579310	2.111164	3.690474	1.778999
0.589315	0.786216	1.375531	0.662770

matrix_C_with exponent_injection	on	rank	3
0.725272	1.064052	1.789323	0.881152
0.000000	0.501837	0.857996	0.418664
1.081431	1.565889	3.06539E+77	0.081238
0.000000	0.573663	0.974252	0.477152

difference_C	on	rank	0
0.307064	0	0	0
0	0	0	0
0	0	2.02228	0
0	0.594767	0	0.513082

difference_C	on	rank	1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

difference_C	on	rank	2
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

difference_C	on	rank	3
0	0	0	0
0.35616	0	0	0
0	0	-3.065E+77	1.218578
0.400589	0	0	0

Figure 6.20: Example of a small matrix-matrix multiplication with fault injection in the exponent on a (2×2)-processor grid

A severe problem in Figure 6.20 is the huge difference on `rank 3` where the values is changed from `2.647320` to `3.06539E+77`. This modification is the point where the correction algorithm refuses and reaches its limits. The difference in the two values is so extreme that the corresponding checksums do not have enough numerical capability to correct this value to its origin, mainly because of numerical possibilities and the machine epsilon. As the machine epsilon gives an upper bound on the relative error due to rounding in floating point arithmetic thus, this limitation also applies to the correction algorithm. Moreover, the exponent range of $[-1022, +1023]$ make the things worse. Therefore it is possible that a value changes e.g., from 10 to 10^{100} simply by an exponent bit flip which makes a correction of the matrix value impossible with standard checksum algorithms on a machine limited by double precision machine epsilon.

The following figures represent the results which have been examined on the Repeal2 server machine. First again the box plot (Figure 6.21) with showing completely different results compared to the previous types of bit flips. There is the situation that as long as small value changes are caused by exponent bit flips nothing spectacular happens with the relative 1-norm results, like for the (2×2)-grid. On the other hand, if there is a big value change, the correction algorithm just don't manage to keep the overall error low which can also be due to tolerance boundary (subsection 5.2.4) for correcting a value. Besides, it can also be that the error inserting interval is set to high (too many bit flips occur). How to determine a suitable spacing for bit flips is discussed in subsection 5.3.2. The significant differences for the maximum and mini-

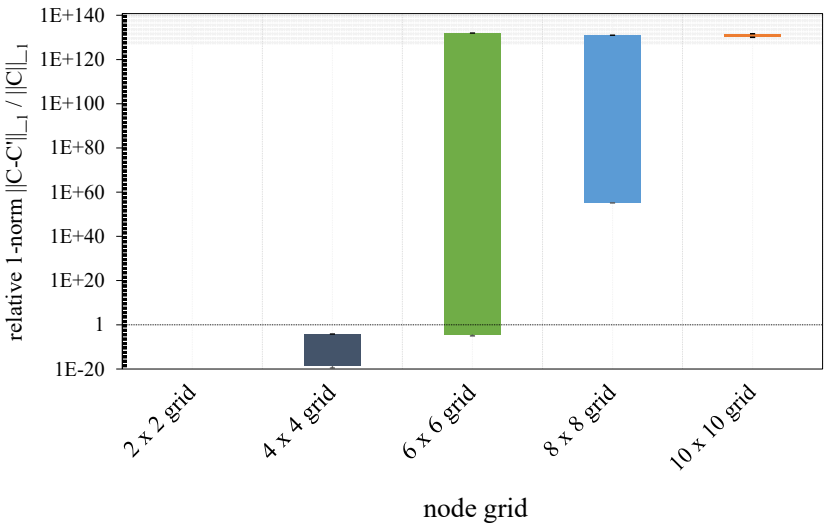


Figure 6.21: Box plot for bit flips in the exponent using the Fault Injector

mum of each bar come from the fact that the algorithm is not correcting the erroneous values and after the elements of the matrices are multiplied the error is adding up in

total by each multiplication. Additional info for each test case can be found in Figure 6.22. Because most of the data points for the (10×10)-grid are between 10^{120} and 10^{140}, the box plot shows this bad range for this grid size.

In Figure 6.23 nothing regarding the overhead has changed to previous bit flip variants but just for comparison and emphasizing on the trend when grid size grows.

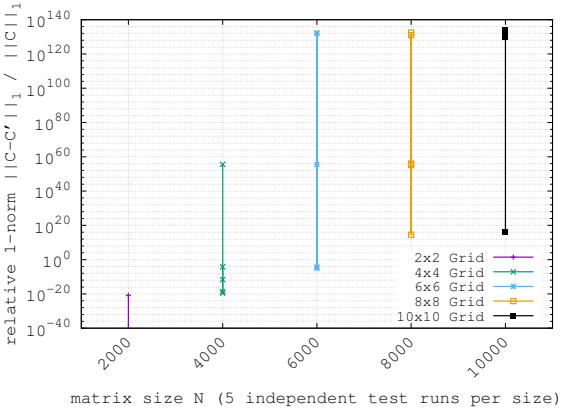


Figure 6.22: Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips in the exponent of a value

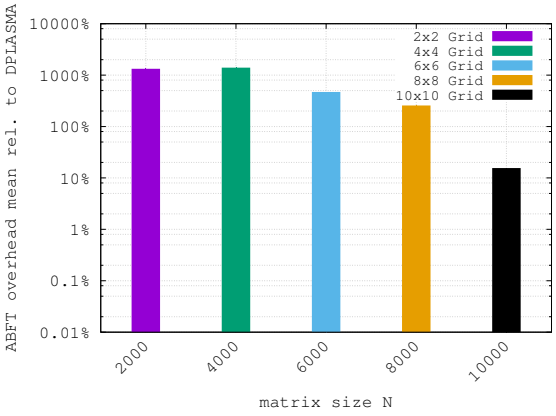


Figure 6.23: Relative overhead when bit flips are inserted in the exponent for different grid sizes

6.2.4 Bit Flips Everywhere

Last but not least the utterly arbitrary bit flips are investigated. Thus all possible value distortions including sign bit flips, mantissa bit flips, or exponent bit flips can be existent. The type of bit flips and how many of them are injected is the attention of this subsection.

The results from Figure 6.24 can be summarized as follows. As long as the own-threaded *Fault Injector* is active and do many different bit flips the results become worse and worse with the duration of its active status. On the other hand, when the matrix multiplication is fast enough like in the (2×2)-grid or (4×4)-grid, there is not much time for the injector to mess up with the matrix elements, so the norm results remain stable. For the (10×10)-grid actually the multiplication costs enough time so that there can be even exponent bit flips injected. Thus this enormously affects the results for this particular case of matrix-matrix multiplications. When the multiplication lasts long enough, there exists a high probability that it is not going to be corrected successfully. This complication also comes from the adjustment of the *Fault Injector*, which is difficult to set up with the appropriate MTBF-value (see subsection 5.3.2).

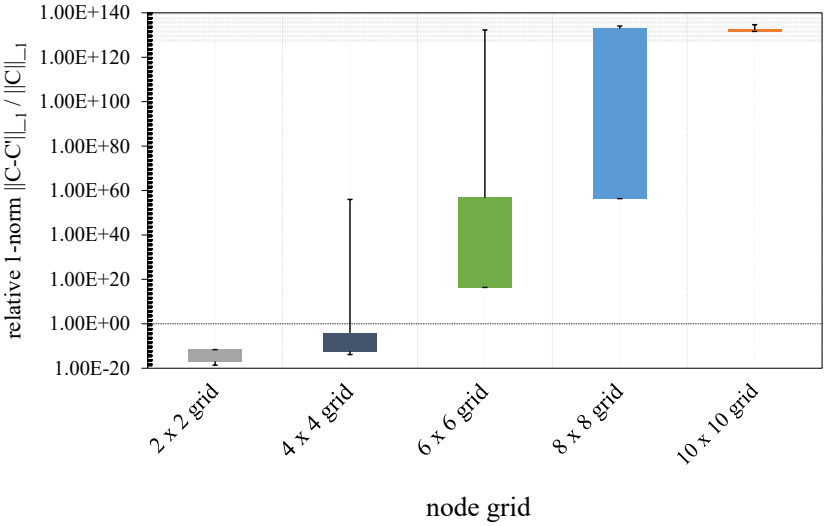


Figure 6.24: Box plot for bit flips all over the bit mask using the Fault Injector

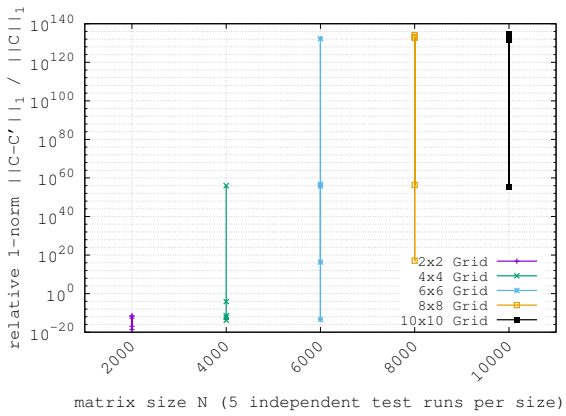


Figure 6.25: Relative 1-norm of 5 independent test runs when using Fault Injector for bit flips everywhere in the bit mask of a value

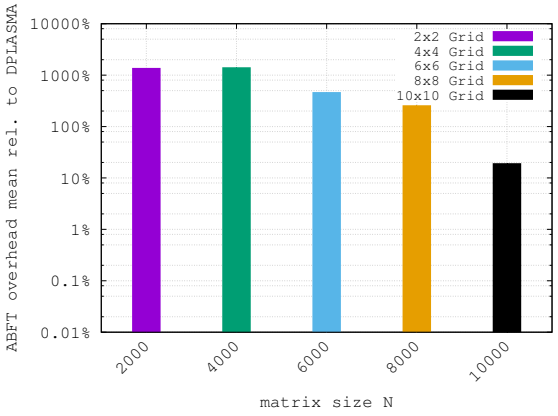


Figure 6.26: Relative overhead when bit flips are inserted truly randomly for different grid sizes

Again Figure 6.25 is showing the details for each of the five independent test runs regarding the relative 1-norm and Figure 6.26 the attendant overhead. The results in the left figure are a bit more distributed than they were in the previous bit flip variants, but that is also the expectation when we have randomly and uniformly distributed errors. Besides the manipulations are following an exponential time distribution. On the right side, we have a pretty similar overhead scheme as we have in the other bit flip variants (`bigger grid sizes` $\Longrightarrow$ `less overhead`).

6.3 Conclusion to Bit Flips

First of all sign bit flips are not so severe. As long as they are not more frequent then the available checksums, the algorithm can correct with high probability. The only fail which can happen in this case is when a particular tile size in a combination of as many bit flips as defined checksums are used. There it can take place that the algorithm does not correct all sign bit flips due to the described problem in subsection 6.1.1 coming from the 2-D block-cyclic structure of the sub-matrices.

Bit flips in mantissa can have a greater impact like shown in subsection 6.2.2. There it is time to take care of the number of mantissa bit flips injected. The correction of the erroneous values has still a reasonable probability to be done successfully, but if there are many of them, the algorithm reaches its limits fast. It can also be that the limitations given by the numerical issues in subsection 6.2.3 can cause the correction algorithm to fail.

Then we have the most dangerous ones. Bit flips in exponent can lead to a deplorable situation for the whole matrix-matrix computation. To emphasize the example where the value is changed from 10 to 10^{100} by an exponent bit flip, in such a case there is no chance that the correction can be prosperous. The detection algorithm records merely a severe numerical error and the correction is discarded and further logged as a failure. It can be of course that the exponent bit flip is not so severe ($1.0 \rightarrow 10$, $10 \rightarrow 100$, $0.001 \rightarrow 0.000001$) or other similar, but when a lot of them occur, later the total error is added up during every matrix element multiplication, resulting in a substantial relative 1-norm.

The bit flips everywhere observations are something in between. It can happen, that a few small bit flips from the type of sign bit flips, or mantissa bit flips occur, and which are easy to detect and further to correct but it can also happen that some bit flips in exponent continuously appear. When the latter one is the case, then the algorithm will fail with high probability. At this type of errors, every before mentioned weakness play a role, including the structure of the sub-matrices, numerical limitations, big value differences and the frequency setup of injecting errors. In summary, we can say that as long as not too many exponent bit flips in combination with mantissa bit flips are injected during the matrix multiplication, the implemented ABFT_PDGEMM algorithm will be capable of managing the error situation.

It should be noted that the ABFT algorithm used in this thesis is based on the classic ABFT approach (where values are compared) and cannot handle bit flips with large numerical differences. In [110] an advanced version called *dABFT* was presented, which is more stable to numerical changes and capable of withstanding any bit flips (including NaN and infinity).

Chapter 7

Conclusion

As the title of this master thesis says, the scalability of ABFT methods for dense matrix operations should be well examined, therefore we decided to use a relatively new method to do that, as by using the help of simulating tools. By studying the simulators thoroughly in their documentation materials and manuals and further by trying to implement a working parallel ABFT general matrix-matrix multiplication in combination with them, one major thing has been observed. The highly optimized libraries DPLASMA and ScaLAPACK which are commonly used in Scientific Computing, are not capable of directly working together with any of the mentioned simulators in Chapter 4 (for detailed explanation see the associated subsections). The main problem is that the libs are strongly coupled to hardware topology or hardware locality (e.g., using the information for their routines and sub-routines from the HWLOC software package). This means restricted virtual MPI processes freedom for simulators and the different MPI contexts of the application's library and the simulator's library are blocking or rather rivaling each other. Moreover, there is probably not a real blocking of the different contexts but a kind of unknowing of the real hardware topology for the subroutines of DPLASMA or ScaLAPACK, besides it can come from the unsupported MPI functions of the various simulators. Since there exists a deep structure of the high-performance libraries' routines, there is a need of other ways to implement that successfully, described in Section 7.1.

From the experiments in Chapter 6 it is apparent that ABFT though can be used to optimize, stabilize and make scientific routines like GEMM more tolerant against faults, bit flips, and even failures, especially node failures. In the results there can be found a definite trend pointing that as the components of a system, in particular cores and nodes, in this case, are growing, the overhead which is given by the additional effort for an ABFT PDGEMM is shrinking. That would be strongly recognized as we are going towards *Exascale Computing* where millions of cores or computational

nodes are involved. In such a situation one of the leading advantages of an ABFT method can come into action by splitting the work to each node or in other respects to each core so that they can do their correction of the values locally, and in one of the most efficient ways for an extreme-scale well parallelized system. Further, this can be expanded by the combination of Checkpointing & ABFT which then would lead to another point reported in the future work section.

What's more the tolerance value for the correction of uniformly and randomly distributed values is not easy to set. There exists a research at the University of Vienna, from the Faculty of Computer Science where this topic is analyzed mathematically [110]. However, their findings have not been used in this master thesis, instead, values based on empirical research and other factors have been used for getting the appropriate results. The main problem here is that setting a tolerance value that is too high can lead to unsuccessful detection or correction of the elements that were already manipulated and should be corrected. The other way round, working with a low value can also advance into a miscarried recovery. This situation also depends and varies with the chosen value distribution and precision which has further an impact on the matrix values. Therefore the range of the matrix values and the limits coming from the numerical precision can lead to a better or a worse detection and correction.

Simulating an exascale situation with current simulating tools is not easy, even if some of the simulators are capable of reproducing an environment with millions of cores or nodes. From the results and examples given in subsection 4.5.1, it follows the conclusion that there is a minimum amount of about 17 TB (terabytes) of RAM needed to simulate an exascale system doing a matrix multiplication. The calculation is based on executing such a simulation with the xSim simulating tool, using DPLASMA but without considering of any MPI communication and computation overhead. Therefore reproducing the situation of an exascale system, including all overheads and address spaces, will require at least a supercomputing machine from the `top500.org` list or for instance using one-fifth of the computational power of Austria's fastest supercomputer VSC-3 [99, 142] which provides about 130 TB of RAM.

About the question, if ECC devices should be used or can be replaced partly or entirely by falt-tolerant algorithms, the answer is that ECC correction is indispensable and will still be used as it was also further emphasized in [69]. In short, because the measured amount of ECC errors at their Cray XT5 system Jaguar exceeded over 100+ per minute and it's expected that this value will rise in future supercomputer systems, the need of ECC devices (in particular ECC memory modules) will still be substantial. Here the exciting part where ABFT algorithms can operate very well is that ECC can't detect and correct a double bit-flip error in one word of memory (typically 64 bits). If two bits are flipped in a word, ECC can recognize that the word

is corrupted, but cannot fix it because it's used only designed for single-bit errors. ABFT instead can manage the situation even if there is an occurrence of many or multiple bit flips.

Concluding the thesis from the study about this topic I can say that an ABFT algorithm in combination with a checkpointing mechanism can be truly considered for future supercomputing systems. Since the time for writing checkpoints to external devices and the mean time between failure (MTBF) of a supercomputing system is getting closer and closer, soon, there will be a need of such fault tolerant approaches like presented in [15]. When the time gap for recalculation and resuming the computation is not enough an ABFT algorithm has to be integrated. It can handle problems arising from that huge amount of components in large-scale supercomputers, especially when double- or more bit flips and node failures are involved. What's more ABFT can be easily adapted to the needs of a specific problem. The main idea behind combining approach is to use ABFT when we know about the application behavior (e.g., in functions, sub-functions, data which can be recomputed) and for the general system consistency, save the state into a checkpoint. With this combination, it is possible that if a severe error occurs, the global state can be reconstructed from the checkpoint, and if there are bit flips or node failures, the ABFT algorithm can recompute the lost data based on the additional checksums.

7.1 Future Work and Open Issues

Like mentioned in the conclusion section, there can be intended for implementation of various ABFT & Checkpointing approaches. This thesis goes only in the depth for a Local ABFT PDGEMM approach. Thus it will be interesting to see, how a *Local ABFT* & Checkpointing approach or a *Global ABFT* & Checkpointing approach will be working together, especially when the bit flips are getting more and more intensive. Also, an ascending number of node failures could be investigated. Further GPUs can be included in the computations when using DPLASMA library. Examining results will show which approach for which particular system will perform best. Since the two ABFT variants have their specific strengths it has to be decided, whether there is a need of being capable of restoring data of whole MPI nodes (accomplished with the *Global* approach), or if the communication overhead between the nodes should be kept as low as possible like it is common in *Local ABFT* methods.

Further, an ABFT PDGEMM can be specially programmed for different simulators. This would imply to choose one particular simulator and implement the ABFT algorithm based on the requirements, pros and contras of the chosen simulator. As there are many simulators which use simulator-based directives, the degree of which a

program has to be reimplemented can vary from adding a few lines of code to a whole restructuring of the application. Depending on the problem which has to be solved there can be made a decision between:

1. If a programming project should be simulated like it's running on a real supercomputing machine (with online simulation),
2. Only the network topology and the contained advantages of a specific topology of an exascale platform should be simulated,
3. If there should be an approximated emulation of the routines' behaviour.

All these options can be reached by different types of simulators which are available over the internet.

Another important point is solving the problem of the libraries' routines and subroutines not knowing the current hardware topology when additionally using a simulator. One way which can be tried is the extraction of the related DGEMM routines of DPLASMA or ScaLAPACK. To be more specific the subroutines of the *dgemm*-routine from the high optimized libraries should be extracted and executed with a particular simulator. There exists a suspicion that as the simulator is executing the *dgemm*-routine on a high level, moreover on an over-layered virtual MPI process, the subroutines do not really know or get the information on which MPI-grid-structure the execution is done actually. This behavior was tested and verified with both high-performance libraries.

In ScaLAPACK the verification was done by the subroutine `Cblacs_gridinit` and further by `Cblacs_gridinfo`. They both change the values required for the ScaLAPACK grid-structure to a negative value `"-1"`, where instead of in a normal run there have to be positive values containing the grid-info. Assuming there are eight processors in a (4×2)-processor grid present, then the correct values which have to be set up by the internal routines will be:

```
ictxt:  0, nprow:  4, npcol:  2, myrow:  0, mycol:  0, for rank 0
ictxt:  0, nprow:  4, npcol:  2, myrow:  1, mycol:  0, for rank 1
ictxt:  0, nprow:  4, npcol:  2, myrow:  2, mycol:  0, for rank 2
...
```

Instead the look of the erroneous output:

```
ictxt:  0, nprow:  -1, npcol:  -1, myrow:  -1, mycol:  -1, for rank 0
ictxt:  0, nprow:  -1, npcol:  -1, myrow:  -1, mycol:  -1, for rank 1
ictxt:  0, nprow:  -1, npcol:  -1, myrow:  -1, mycol:  -1, for rank 2
....
```

Further there is an error that in the subroutine `Cblacs_gridmap` pointing to that the

`MPI_Attr_get` function is not supported by xSim. Thus it has to be considered that some simulators did not support every MPI function.

In DPLASMA the situation is similar where the responsible routine `setup_dague` or respectively `setup_parsec` (in the newer version of DPLASMA) is not setting up the processor's grid structure and in another respect the MPI context correctly. The setup of the internal routines in DPLASMA is strongly managed by the provided information from the hardware locality software package, and so pointing on the real physical hardware. Apart from this fact, there exists an option where the hardware locality library (HWLOC) can be explicitly excluded during the compilation phase of DPLASMA, but this was not examined so far and would have with high probability to do with a loss of performance. Perhaps this could solve a part of the problem where the virtual number of processes is passed to the setup routine of DPLASMA instead of the real hardware data of the supercomputing system.

Another open issue is the implementation of a full and advanced ABFT PDGEMM. First on each node the weight matrix W and the correction matrix H according to the scheme in Section 3.3 is created, and matrix A and B distributed blockwise. The advanced variant includes that there are additional columns and rows with checksums (full-weighted) in matrix A (A^f) and B (B^f), and a matrix D (D^f) [83, 110]. The full weighted variants can also be computed to the presented structure in Section 3.3. The matrix D should be, on the contrary, computed by the row-wise checksums variant of A (A^r) and the column-wise checksums containing matrix B (B^c) such that:

$$D^f = A^r \cdot B^c. \tag{7.1}$$

With this extension it's possible to do failure detection and correction in matrices A^f, B^f and D^f. Furthermore, bit-flips and fault injections are allowed in all full weighted matrices. If a correction of D^f has failed the corrupt block or sub-matrix D^f can be recomputed again. The final matrix C^f which is also full-weighted will be computed by

$$C^f(b_i, b_j) = C^f(b_i, b_j) + D^f. \tag{7.2}$$

If there is an occurrence of some manipulation during the computation of C^f, after all calculations are complete, there can be done a failure detection and correction in C^f. All of the previously described steps would result in a thoroughly fault-tolerant matrix multiplication, where all values of all matrices can be recomputed.

Appendices

A.1 Test Environment

The machine, called Repeal2, on which the experiments in Chapter 6 was executed has the following key figures:

☐ 4× Intel Xeon E7-8867 v3 [79] (4-NUMA-nodes) each with:

# of Cores	**# of Threads**	**Processor Base Frequency**
16	32	2.5 GHz
Max Memory Size	**Max Memory Bandwidth**	**RAM per NUMA-node**
1.54 TB	85 GB/s	252 GB
L1 Cache	**L2 Cache**	**L3 Cache**
32 KB per Core	256 KB per Core	45 MB per Numa-node
(according to HWLOC, for hardware topology details see Figure A.1 and A.2 on pages 100 and 101)		

Table A.1: Testing environment details

# of Cores	**# of Threads**	**Processor Frequency**	**RAM**
64	128	2.5 GHz	1008 GB

Table A.2: Total system metrics

☐ Rpeak Performance in (GFlop/s) of whole System calculated by definition in [115, 68]:

$$\begin{aligned}\text{Node performance in GFlop/s} =& (\text{CPU speed in GHz}) \times (\#\text{ of CPU cores}) \times \\ & (\text{CPU instructions per cycle}) \times \\ & (\#\text{ of CPUs per node})\end{aligned} \tag{A.1}$$

$$\begin{aligned}\text{Rpeak Performance} &= (\text{Node performance in GFlop/s}) \times (\#\text{ of Nodes}) \\ \text{Rpeak GFlop/s} &= (2.5\text{ GHz}) \times 16 \times 16 \times 1 \times (4\text{ NUMA-nodes}) \\ &= \mathbf{2560\ GFlop/s}\end{aligned} \tag{A.2}$$

The CPU instruction per cycle for the `Xeon E7-8867 v3` is assumed to be `16` because of the successor generation presented in [68], the `Xeon E7-8867 v4` and with subject to [108, 115] describing that the `Intel Xeon E5-4600 v3` and `E5-2690 v3` which also belonging to the Haswell-EP processor family, already running with 16 DP FLOPS per cycle. Therefore the system's Rpeak of `2560 GFlop/s`.

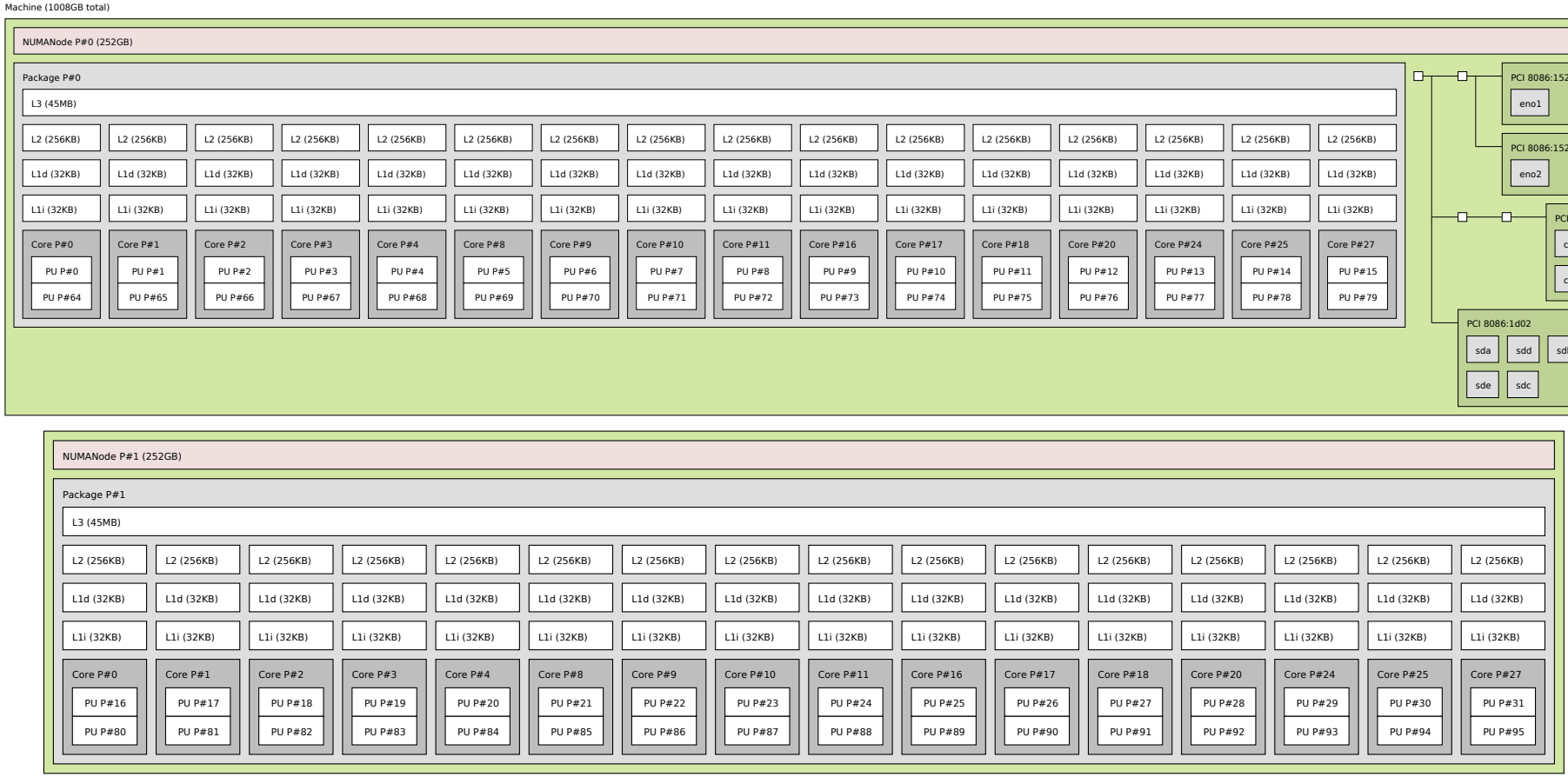


Figure A.1: Repeal2 hardware topology from execution of lstopo from HWLOC part 1 (NUMA-node 0-1)

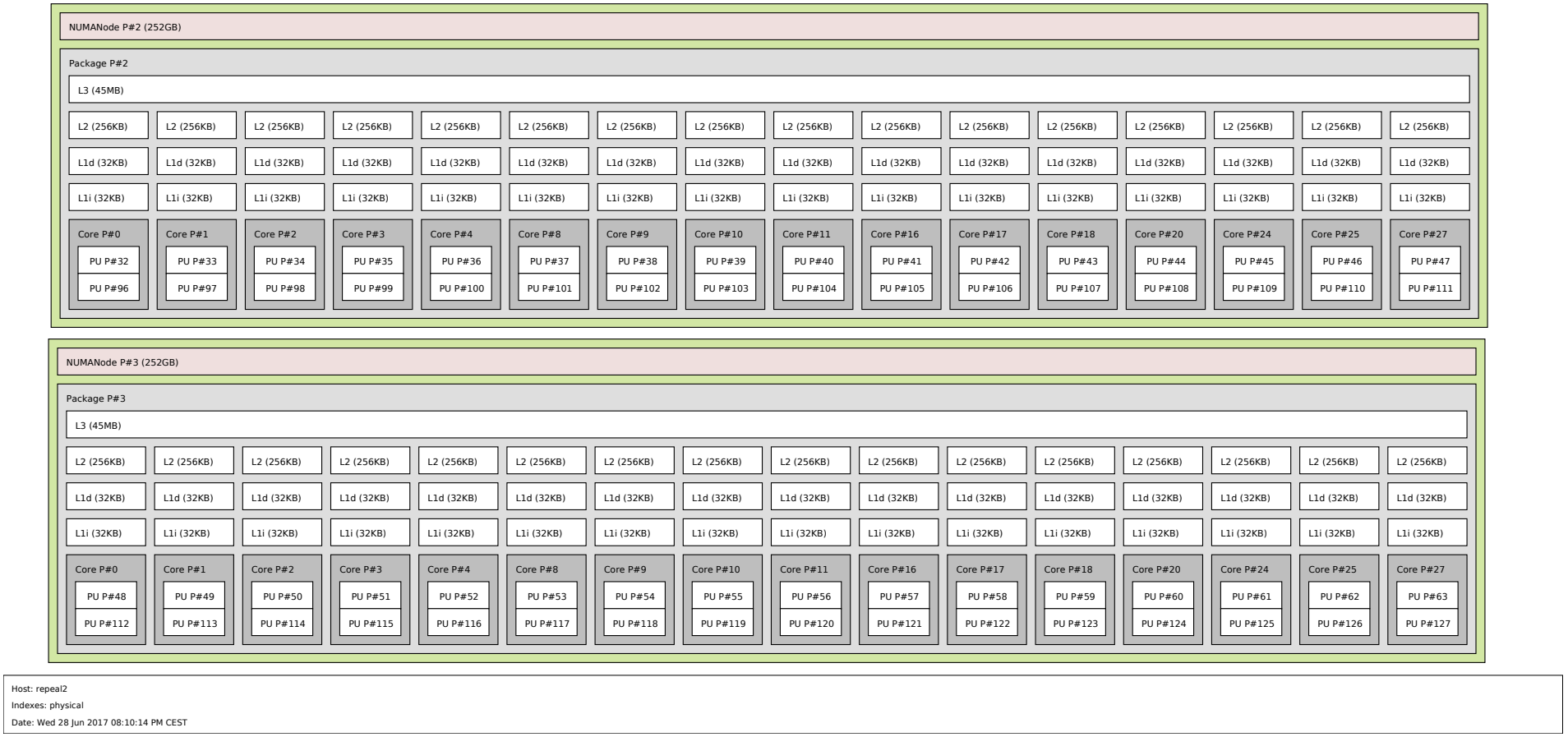


Figure A.2: Repeal2 hardware topology from execution of lstopo from HWLOC part 2 (NUMA-node 2-3)

A.2 Extended Space Analysis Local ABFT

This overview below contains analysis for all matrices and arrays used in the Local ABFT implementation.

Matrices / Arrays	Elements per submatix/ CPU rank	Elements for all CPU ranks	Elements for all CPU ranks (simplified)
weight matrix w:	$(\frac{n}{P}) \times d)$	$n \times (d * Q))$	$n \times (d * Q))$
matrix A:	$\left((\frac{n}{P}) + d\right) \times ((\frac{n}{Q}) + d)$	$(n + (P * d)) \times (n + (Q * d))$	$N \times N$
matrix B:	$\left((\frac{n}{P}) + d\right) \times ((\frac{n}{Q}) + d)$	$(n + (P * d)) \times (n + (Q * d))$	$N \times N$
matrix C:	$\left((\frac{n}{P}) + d\right) \times ((\frac{n}{Q}) + d)$	$(n + (P * d)) \times (n + (Q * d))$	$N \times N$
correction matrix H1:	$d \times d$	$(d * P) \times (d * Q)$	$(d * P) \times (d * Q)$
correction matrix H2:	$d \times d$	$(d * P) \times (d * Q)$	$(d * P) \times (d * Q)$
correction ind matrix S1:	$(d) \times ((\frac{n}{P}) + d))$	$(d * P) \times (n + (P * d))$	$(d * P) \times N$
correction ind matrix S2:	$(d) \times ((\frac{n}{Q}) + d))$	$(d * Q) \times (n + (Q * d))$	$(d * Q) \times N$
indices values I1:	d	$d \times (P * Q)$	$d \times (P * Q)$
indices values I2:	d	$d \times (P * Q)$	$d \times (P * Q)$
matrix ddescH:	$\left((\frac{n}{P}) + d\right) \times ((\frac{n}{Q}) + d)$	$(n + (P * d)) \times (n + (Q * d))$	$N \times N$

Total elements:	$dnP + 2(d^2 + dNP + dP^2 + 2dP + 2N^2 + P^2)$	
Total space for DP:	$8dnP + 16(d^2 + dNP + dP^2 + 2dP + 2N^2 + P^2)$ **bytes**	

N ...	extended dimension	$N = n + (P * d)$	
P ...	CPU ranks in P direction	$P = Q$	
Q ...	CPU ranks in Q direction		
d ...	additional rows / columns for checksums		
real size = n ...	extended dimension - additional rows / columns	$n = N - (P * d)$	

Figure A.3: Local ABFT PDGEMM extended space analysis for # of elements and # of bytes plus regarding legend

A.3 Project Files

The following table provides an overview of various files created and used in the project and provides a short description of their functions. Further down, the DPLASMA specific files are listed.

File	Description
`Makefile`	A makefile for compiling, running and testing the `ABFT_PDGEMM`. Further containing help of using the built-in functionalities.
`abft_pdgemm_opt.cpp`	Space and runtime optimized version of the ABFT matrix-matrix multiplication. This is the main file, containing the whole fault tolerant procedure.
`routines.cpp`	Containing all the extended functions, e.g., for print out matrix values and for debugging.
`routines.hpp`	Contains the headers for the `routines.cpp` and all special declarations and useful preprocessor macros.
`abft.cpp`	Functions for testing and correcting faulty matrices.
`abft.hpp`	Definitions of the function in `abft.cpp` and some preprocessor macros.
`abft_matrix.cpp`	Auxiliary functions for initialization and printing.
`abft_matrix.hpp`	Containing the default settings for the functions in `abft_matrix.cpp`. Defining the ABFT matrix class for `C++`.
`matrix.h`	This file is for the `C` definitions of the ABFT matrix class.
`abft_pdgemm.cpp`	The original and first version of the `ABFT_PDGEMM` (without special optimizations for a certain platform).
`faultinjector.cpp`	Contains the default settings for the own-threaded fault injector.
`faultinjector.hpp`	Contains the implementation of the own-threaded fault injector to be capable of injecting sign bit flips, bit flips in the mantissa and in the exponent of a matrix element.
`convert.sh`	This script is for producing plots and converting *.eps files to *.png and *.pdf.
`repeal2_labft.plt`	Contains all the information for Gnuplot to produce the scientific plots, with labeling and with or without title labels.
`checking_results.out`	To this file, all the measuring- and timing results are logged.
`faultcorrection.log`	This file logs the correction status of each processor.
`faultinjection.log`	This file logs the error initiation of the own-threaded fault injector with timestamps and further information.
`labft_times_logfile.out`	This file logs the runtimes for the multiple executions of the `ABFT_PDGEMM`.
`dplasma_times_logfile.out`	This file logs the runtimes for the multiple executions of the `DPLASMA_dgemm`.
DPLASMA Related File	**Description**
`common.c`	This is the common file for the DPLASMA interface. Including user interface and functions for the execution of DPLASMA routines.
`common.h`	There are all the definitions for the DPLASMA interface and extended preprocessor macros.
`testing_dgemm.c`	For measuring the runtime of executing the `DPLASMA_dgemm` routine.
`butterfly.c`	For special internal matrix tiling and splitting.
`common_timing.h`	For DPLASMA-own timing with integration for GFlop/s.
`flops.h`	This file contains multiple formulae of all possible FLOPS calculations as a preprocessor macro for Level 3 BLAS and some LaPACK routines.

Table A.3: Summary and description of relevant project files

A.4 Software and Library Versions

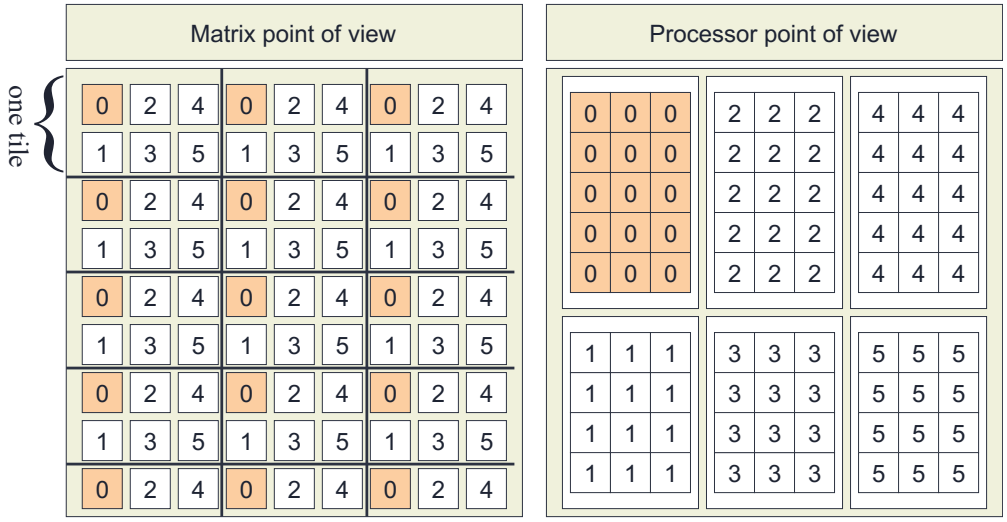
Names	Description
LaPACK 3.7.1	Linear Algebra PACKage library, needed for compiling DPLASMA. (can be found under http://www.netlib.org/lapack/#_lapack_version_3_7_1)
DPLASMA 2.0.0-rc2 / PARSEC 2.0.0-rc2	The official DPLASMA and PaRSEC site is located at http://icl.cs.utk.edu/parsec/news/index.html. The newest versions can also be found in the git repository with the direct cloning link: "git clone https://bitbucket.org/icldistcomp/parsec.git".
PLASMA 2.8.0	Parallel Linear Algebra Software for Multicore Architectures library, needed for compiling DPLASMA. (https://bitbucket.org/icl/plasma/overview)
HWLOC 1.11.4	Portable Hardware Locality library for hardware topology and processor details used in DPLASMA. (https://www.open-mpi.org/projects/hwloc/)
OpenMPI 2.1.1	OpenMPI is in the first place a High-Performance Message Passing Library. Besides, it is a Message Passing Interface implementation which is distributed as open source. OpenMPI is developed and maintained by a consortium of research, academic, and industry partners and it belongs to one of the required MPI libraries for DPLASMA. (https://www.open-mpi.org/software/ompi/v2.1/) [136]
OpenBLAS	OpenBLAS is an optimized BLAS(Basic Linear Algebra Subprograms) library based on GotoBLAS2 1.13 BSD version. It is the root linear algebra library needed by PLASMA, LaPACK, and DPLASMA.(https://github.com/xianyi/OpenBLAS/wiki)
Debian 8 with Debian Testing	The operating system used on the Repeal2 server machine. (https://wiki.debian.org/DebianJessie)

Table A.4: Summary and description of used libraries and packages

B.1 Glossary

Here a more comprehensive explanation of some relevant terms used in this thesis can be found:

2-D block-cyclic The 2-D block-cyclic structure is a unique matrix value decomposition where the matrix values are distributed in rectangle- or quadratic blocks such that an optimal load balancing can be achieved over the consecutive processors [49, 51, 10]. These conditions lead to the situation that on the tiles/submatrices BLAS-operations can be executed in a much better way than other matrix layouts. There are two main issues in choosing a 2-D block-cyclic data layout for dense matrix computations. The first one is load balance which gives the opportunity to split the work reasonably equally among the processors throughout the algorithm. The second one is, to make use of the Level 3 BLAS while computations are running on a single processor such that the memory hierarchy on each processor can be utilized. This particular structure is illustrated in the figure below .



DPLASMA Stands for (Distributed) Parallel Linear Algebra Software for Multicore Architectures and is a highly optimized library containing mainly linear algebra software for High-Performance Computing purposes [18]. The included functions are based on PLASMA with adding the possibility of also using GPU Kernels as accelerators. The MAGMA library provides this hybrid architecture option. For further details see (section 5.1).

GNUPLOT Gnuplot is a portable command-line using graphing utility for various operating systems including Linux, OS/2, MS Windows, VMS, OSX, and many

others. The source code has a copyright on it, but it is freely distributed. In the beginning, its goal was to allow scientists, students, and other users to visualize mathematical functions and data interactively. Meanwhile, it also supports many non-interactive applications such as web scripting and is also used in third-party applications (for instance Octave). Gnuplot has been supported and is under active development since 1986. Furthermore, it supports many types of plots (2D or 3D), and it can draw images by using lines, points, contours, boxes, surfaces, vector fields, and various associated text. It also supports several specialized plot types. Further direct output to pen plotters or current printers, and output to various file formats (e.g., emf, eps, fig, jpeg, pdf, LaTeX, png, postscript, and others) are supported [147, 146].

HWLOC The portable software package Portable Hardware Locality (hwloc) provides an abstraction (across OS, architectures, versions, ...) of the hierarchical topology of modern architectures. The support includes NUMA memory nodes, shared caches, sockets, cores and simultaneous multithreading. Furthermore, it has a collection feature where various system attributes like cache and memory information can be gathered. Further, the locality of input/output devices such as InfiniBand HCAs, network interfaces, or GPUs can be extracted. It primarily aims at helping applications with collecting information about increasingly complex parallel computing platforms and also acts as a portable CPU and memory binding API [137, 25].

NUMA Non-uniform memory access (NUMA) is a computer memory design which can be used in machines with multiple processors. Compared to a shared memory architecture where all processors share the same memory and handle it as a global address space, in a NUMA architecture we have processors with identical architecture connected to a scalable network, and each of them with a part of memory attached directly to it. Further, the memory access time in a NUMA architecture depends on the memory location which is relative to the processor. The main difference between a NUMA architecture and a distributed memory architecture is that no one of the processors can have mappings to the memory assigned to other processors. There also exists a classification of local memory and remote memory which is calculated according to the access latency of the different memory regions of each processor [96].

PaRSEC PaRSEC stands for Parallel Runtime Scheduling and Execution Controller and is a universal framework for architecture aware scheduling and administration of micro-tasks on many-core heterogeneous architectures. Applications, however, can be represented as a special graph, a so-called Directed Acyclic

Graph (DAG) of tasks where labeled edges designate data dependencies. These so-called DAGs have a compact problem-size independent format where data dependencies in a distributed fashion can be discovered if required. PaRSEC further has the ability to assign computation threads to the cores to arrange communications and computations and to use a dynamic, fully-distributed scheduler. This specific scheduler is working depending on architectural features like NUMA nodes and on algorithmic features such as data reuse. The framework brings libraries, a runtime system, and development tools with it. They should help application developers to get rid of the porting problems of applications especially when programs should be ported to highly heterogeneous- and diverse environment [18].

B.2 List of Acronyms

2DBC 2-D block-cyclic. 61, 78, 92, *Glossary:* 2-D block-cyclic

ABFT Algorithm-Based Fault Tolerance. 1, 6, 8–11, 13–16, 19, 21, 22, 24–30, 33, 35, 36, 55, 56, 61, 66, 69, 70, 75–77, 79, 81–86, 92–95, 97, 102, 103

AODV Ad hoc On-Demand Distance Vector Routing. 49

API Application Programming Interface. 13, 33, 37, 48

BLAS Basic Linear Algebra Subprograms. 63–65, 103–105

CUDA Compute Unified Device Architecture. 50

DES Discrete Event Simulation. 53

DP Double Precision (floating-point format). 100

DPLASMA Distributed Parallel Linear Algebra Software for Multicore Architectures. 8, 15, 40, 41, 47, 55, 59, 61, 62, 64, 65, 67, 75, 76, 81–83, 86, 93–97, 103, 104, *Glossary:* DPLASMA

ECC Error Checking and Correction. 94

FLOPS Floating Point Operations Per Second. 45, 55, 100, 103

GEMM General Matrix Matrix multiplication. 14, 70, 93, 96

GFlop/s Giga Floating Point Operations Per Second. 9, 55, 76, 80, 99, 103

Gnuplot Portable command-line driven graphing utility for Linux. 103, *Glossary:* GNUPLOT

GPGPUs General-Purpose Graphics Processing Unit. 15, 41, 43

HPC High Performance Computing. 7, 8, 16, 17, 19, 21, 30, 36, 39, 40, 45–49, 58, 59, 61

HWLOC Portable Hardware Locality. 59, 65, 93, 99, *Glossary:* HWLOC

IDE Integrated Development Environment. 44, 49

LaPACK Linear Algebra PACKage. 63, 65, 69, 103, 104

LINPACK LINear equations software PACKage. 63

LTE Long-Term Evolution. 49, 59

MAGMA Matrix Algebra on GPU and Multicore Architectures. 64, 105

MPI Message Passing Interface. 8, 13, 14, 27, 28, 30, 32, 36–38, 40–42, 45–47, 49–51, 54, 58, 59, 64, 76, 93–97

MTBF Mean Time Between Failure. 2, 5, 6, 9, 35, 61, 72–74, 90, 95

MTTF Mean Time To Failure. 5, 19, 20, 35

NUMA Non-Uniform Memory Access. 79, 99, 106, *Glossary:* NUMA

OLSR Optimized Link State Routing Protocol. 49

OpenMP Open Multi-Processing. 47, 50, 51

PAPI Performance Application Programming Interface. 51

PaRSEC Parallel Runtime Scheduling and Execution Controller. 6, 15, 65, 104, *Glossary:* PaRSEC

PDES Parallel Discrete Event Simulation. 39–42, 44, 45, 47, 51, 57

PDGEMM Parallel Double precision valued General Matrix Matrix multiplication. 8, 14, 16, 24, 55, 56, 66, 70, 79, 81, 92, 93, 95, 97

PLASMA Parallel Linear Algebra Software for Multicore Architectures. 64, 65, 104, 105

Pthreads POSIX Threads. 47, 50, 51

Rpeak Theoretical Peak Performance in (GFlop/s). 99, 100

ScaLAPACK Scalable Linear Algebra PACKage. 8, 13, 14, 40, 41, 47, 59, 63, 64, 93, 96

SST Structural Simulation Toolkit. 43

TMR Triple Modular Redundancy. 11, 21

WiMAX Worldwide Interoperability for Microwave Access. 49

xSim Extreme-Scale Simulator. 8, 40, 94, 97

Bibliography

[1] Acun, B., Jain, N., Bhatele, A., Mubarak, M., Carothers, C.D., Kalé, L.V.: "Preliminary Evaluation of a Parallel Trace Replay Tool for HPC Network Simulations". In: Euro-Par Workshops (2015)

[2] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: "LAPACK Users' Guide". Society for Industrial and Applied Mathematics, Philadelphia, PA, third edn. (1999)

[3] Avizienis, A.: "Fault-tolerance: The survival attribute of digital systems". Proceedings of the IEEE 66(10), 1109–1125 (Oct 1978)

[4] Ballard, G., Carson, E., Knight, N.: "Algorithmic-Based Fault Tolerance for Matrix Multiplication on Amazon EC2" (Dec 2009), http://www.cs.berkeley.edu/~knight/ballardcarsonknight_paper.pdf

[5] Banerjee, P., Abraham, J.A.: "Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems". IEEE Transactions on Computers C-35(4), 296–306 (April 1986)

[6] Banerjee, P., Rahmeh, J.T., Stunkel, C., Nair, V.S., Roy, K., Balasubramanian, V., Abraham, J.A.: "Algorithm-based fault tolerance on a hypercube multiprocessor". IEEE Transactions on Computers 39(9), 1132–1145 (Sep 1990)

[7] Barcelona-Supercomputing-Center: "Dimemas: predict parallel performance using a single cpu machine" (2017), https://tools.bsc.es/dimemas, [Online; accessed 10-May-2017]

[8] Berry, M.W., Gallivan, K.A., Gallopoulos, E., Grama, A., Philippe, B., Saad, Y., Saied, F.: "High-Performance Scientific Computing: Algorithms and Applications". Springer Publishing Company, Incorporated (2012)

[9] Blackford, L.S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.:

"ScaLAPACK User's Guide". Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997)

[10] Blackford, S.: "The Two-dimensional Block-Cyclic Distribution" (last visited: February 2017), http://www.netlib.org/scalapack/slug/node75.html

[11] Bland, W.: "User Level Failure Mitigation in MPI". In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par 2012: Parallel Processing Workshops. pp. 499–504. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[12] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: "Post-failure Recovery of MPI Communication Capability: Design and Rationale". Int. J. High Perform. Comput. Appl. 27(3), 244–254 (Aug 2013), http://dx.doi.org/10.1177/1094342013488238

[13] Böhm, S., Engelmann, C.: "xSim: The Extreme-Scale Simulator". In: Proceedings of the International Conference on High Performance Computing and Simulation (HPCS) 2011. pp. 280–286. IEEE Computer Society, Los Alamitos, CA, USA, Istanbul, Turkey (Jul 4-8, 2011), http://www.christian-engelmann.info/publications/boehm11xsim.pdf

[14] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA". In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. pp. 1432–1441 (May 2011)

[15] Bosilca, G., Bouteiller, A., Herault, T., Robert, Y., Dongarra, J.: "Assessing the Impact of ABFT and Checkpoint Composite Strategies". In: 2014 IEEE International Parallel Distributed Processing Symposium Workshops. pp. 679–688 (May 2014)

[16] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: "DAGuE: A generic distributed DAG engine for High Performance Computing". Parallel Computing 38(1), 37 – 51 (2012), http://www.sciencedirect.com/science/article/pii/S0167819111001347, extensions for Next-Generation Parallel Programming Models

[17] Bosilca, G., Bouteiller, A., Danalis, A., Hérault, T., Luszczek, P., Dongarra, J.J.: "Dense Linear Algebra on Distributed Heterogeneous Hard-

ware with a Symbolic DAG Approach". In: Scalable Computing and Communications: Theory and Practice (2012), http://www.netlib.org/lapack/lawnspdf/lawn264.pdf

[18] Bosilca, G., Bouteiller, A., Dongarra, J., et al.: "Parallel Runtime Scheduling and Execution Controller" (last visited: May 2017), http://icl.cs.utk.edu/parsec/index.html

[19] Bosilca, G., Bouteiller, A., Herault, T., Robert, Y., Dongarra, J.: "Composing resilience techniques: ABFT, periodic and incremental checkpointing". International Journal of Networking and Computing 5, 2–25 (Mar 2015)

[20] Bosilca, G., Bouteiller, A., Herault, T., Yves, R.: "Fault-tolerant Techniques for HPC: Theory and Practice 2016" (November 14 2016), http://fault-tolerance.org/downloads/sc16-tutorial.pdf, tutorial at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)

[21] Bosilca, G., Bouteiller, A., Herault, T., Yves, R.: "Fault-tolerant Techniques for HPC: Theory and Practice 2017" (November 12-17 2017), http://fault-tolerance.org/downloads/sc17-tutorial.pdf, tutorial at The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)

[22] Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: "Algorithm-based Fault Tolerance Applied to High Performance Computing". J. Parallel Distrib. Comput. 69(4), 410–416 (Apr 2009), http://dx.doi.org/10.1016/j.jpdc.2008.12.002

[23] Bosilca, G., et al.: "ULFM 2.0" (last visited: November 2017), http://fault-tolerance.org/2017/11/03/ulfm-2-0/

[24] Bouteiller, A., Herault, T., Bosilca, G., Du, P., Dongarra, J.: "Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy". ACM Trans. Parallel Comput. 1(2), 10:1–10:28 (Feb 2015), http://doi.acm.org/10.1145/2686892

[25] Broquedis, F., Clet Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications". In: IEEE (ed.) PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. Pisa Italie (Feb 2010), http://hal.inria.fr/inria-00429889/en/

[26] BSC: “Extrae Documentation - Release 3.5.2” (last visited: November 2017), https://tools.bsc.es/sites/default/files/documentation/pdf/extrae-3.5.2-user-guide.pdf, developed at Barcelona Supercomputing Center

[27] BSC: “Introduction to Dimemas - Performance Analysis Tool for Parallel Programs and Platforms” (last visited: November 2017), https://tools.bsc.es/sites/default/files/documentation/introduction_dimemas.pdf, developed at Barcelona Supercomputing Center

[28] BSC: “Parallel Program Visualization and Analysis Tool - Reference Manual” (last visited: November 2017), https://tools.bsc.es/sites/default/files/documentation/1364.pdf, devoloped at Barcelona Supercomputing Center

[29] BSC: “Paraver: a flexible performance analysis tool” (last visited: November 2017), https://tools.bsc.es/paraver, developed at Barcelona Supercomputing Center

[30] Burnette, E.: “Eclipse IDE Pocket Guide: Using the Full-Featured IDE”. O’Reilly Media (2005), https://www.amazon.com/Eclipse-IDE-Pocket-Guide-Full-Featured/dp/0596100655?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0596100655

[31] Buyya, R., Murshed, M.: “GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing”. Concurrency and Computation: Practice and Experience (CCPE 14(13)), 1175–1220 (2002)

[32] Cao, C., Herault, T., Bosilca, G., Dongarra, J.: “Design for a Soft Error Resilient Dynamic Task-Based Runtime”. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 765–774 (May 2015)

[33] Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. Journal of Parallel and Distributed Computing 74(10), 2899–2917 (Jun 2014), https://hal.inria.fr/hal-01017319

[34] Chandy, K.M., Misra, J.: “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”. IEEE Transactions on Software Engineering SE-5(5), 440–452 (Sept 1979), https://doi.org/10.1109/TSE.1979.230182

[35] Chen, C.Y., Abraham, J.A.: "Fault-Tolerant Systems For The Computation Of Eigenvalues And Singular Values". In: Proceedings Volume 0696, Advanced Algorithms and Architectures for Signal Processing I. vol. 0696, pp. 0696 – 0696 – 10 (1986), http://dx.doi.org/10.1117/12.936897

[36] Chen, Z., Dongarra, J.: "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources". In: Proceedings 20th IEEE International Parallel Distributed Processing Symposium. pp. 10 pp.– (April 2006)

[37] Chen, Z., Dongarra, J.: "Algorithm-Based Fault Tolerance for Fail-Stop Failures". IEEE Trans. Parallel Distrib. Syst. 19(12), 1628–1641 (Dec 2008), http://dx.doi.org/10.1109/TPDS.2008.58

[38] Choi, Y.H., Malek, M.: "A fault-tolerant systolic sorter". IEEE Transactions on Computers 37(5), 621–624 (May 1988)

[39] CLOUDS-Laboratory: "GridSim: A Grid Simulation Toolkit For Resource Modelling And Application Scheduling For Parallel And Distributed Computing" (2017), http://www.cloudbus.org/gridsim/, [Online; accessed 14-May-2017]

[40] cplusplus.com: "Default random engine" (last visited: May 2017), http://www.cplusplus.com/reference/random/default_random_engine/

[41] cplusplus.com: "srand" (last visited: May 2017), http://www.cplusplus.com/reference/cstdlib/srand/

[42] cplusplus.com: "Uniform discrete distribution" (last visited: May 2017), http://www.cplusplus.com/reference/random/uniform_int_distribution/

[43] cppreference.com: "C++ concepts: RandomNumberDistribution function object" (last modified: 10 May 2017), http://en.cppreference.com/w/cpp/concept/RandomNumberDistribution

[44] cppreference.com: "std::exponential_distribution numerics library" (last modified: 10 May 2017), http://en.cppreference.com/w/cpp/numeric/random/exponential_distribution

[45] Dagum, L., Menon, R.: "OpenMP: An Industry Standard API for Shared-Memory Programming". Computational Science & Engineering, IEEE 5(1), 46–55 (1998)

[46] Dekate, C., Anderson, M., Brodowicz, M., Kaiser, H., Adelstein-Lelbach, B., Sterling, T.L.: "Improving the scalability of parallel N-body applications with an event driven constraint based execution model". IJHPCA 26, 319–332 (2012)

[47] Ding, C., Karlsson, C., Liu, H., Davies, T., Chen, Z.: "Matrix Multiplication on GPUs with On-Line Fault Tolerance". In: 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications. pp. 311–317 (May 2011)

[48] Dongarra, J., Bunch, J., Moler, C., Stewart, G.: "LINPACK" (last visited: February 2017), http://www.netlib.org/linpack/

[49] Dongarra, J., Demmel, J., Heroux, M., Kurzak, J.: "Linear Algebra Libraries for High-Performance Computing: Scientific Computing with Multicore and Accelerators" (last visited: February 2017), http://www.netlib.org/utk/people/JackDongarra/SLIDES/sc2011-tutorial.pdf, tutorial at ACM/IEEE Conference on Supercomputing, Seattle, WA (SC'11)

[50] Dongarra, J., Demmel, J., Heroux, M., Kurzak, J.: "Tutorial on Distributed Parallel Linear Algebra Software for Multicore Architectures" (last visited: February 2017), http://www.icl.utk.edu/~kurzak/tutorials/SC13/SC13PlasmaDplasmaMagma.pdf, by University of Tennessee - Innovative Computing Laboratory Electrical Engineering and Computer Science Department

[51] Dongarra, J., van de Geijn, R., Walker, D.: "A look at scalable dense linear algebra libraries". In: Proceedings Scalable High Performance Computing Conference SHPCC-92. pp. 372–379 (Apr 1992)

[52] Dongarra, J., et al.: "Applied Mathematics Research for Exascale Computing". Tech. Rep. LLNL-TR-651000, Lawrence Livermore National Laboratory (2014), http://science.energy.gov/%7E/media/ascr/pdf/research/am/docs/EMWGreport.pdf

[53] Dongarra, J.: "Algorithms for future emerging technologies" (July 29 2016), http://dx.doi.org/10.24350/CIRM.V.19022503, at CEMRACS 2016 – Numerical Challenges in Parallel Scientific Computing

[54] Dongarra, J.: "Current Trends in High Performance Computing and Challenges for the Future" (Feb 7, 2017), https://learning.acm.org/webinars/hpc, ACM Webinar, (Online; accessed 16-January-2018)

[55] Dongarra, J.: "Report on the Sunway TaihuLight System" (June 2016), http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-

report-2016.pdf, tech report, University of Tennessee – Department of Electrical Engineering and Computer Science

[56] Dongarra, J., Gay, D., Grosse, E.: "SUBROUTINE DGELS" (last visited: February 2017), http://www.netlib.no/netlib/lapack/double/dgels.f, computer Science Dept, Univ. Tennessee and Oak Ridge National Laboratory

[57] Dongarra, J., et al.: "FT-MPI" (last visited: May 2017), http://icl.cs.utk.edu/ftmpi/overview/index.html

[58] Engelmann, C.: "Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale". Future Generation Computer Systems (FGCS) 30(0), 59–65 (Jan 2014), http://www.christian-engelmann.info/publications/engelmann13scaling.pdf

[59] Engelmann, C.: "Scaling to a Million Cores and Beyond: Using Light-weight Simulation to Understand the Challenges Ahead on the Road to Exascale". Future Gener. Comput. Syst. 30, 59–65 (Jan 2014), http://dx.doi.org/10.1016/j.future.2013.04.014

[60] Engelmann, C., Geist, G.A.A.: "Super-Scalable Algorithms for Computing on 100,000 Processors". In: Lecture Notes in Computer Science: Proceedings of the 5^{th} International Conference on Computational Science (ICCS) 2005, Part I. vol. 3514, pp. 313–320. Springer Verlag, Berlin, Germany, Atlanta, GA, USA (May 22-25, 2005), http://www.christian-engelmann.info/publications/engelmann05superscalable.pdf

[61] Engelmann, C., Lauer, F.: "Facilitating Co-Design for Extreme-Scale Systems Through Lightweight Simulation". In: Proceedings of the 12^{th} IEEE International Conference on Cluster Computing (Cluster) 2010: 1^{st} Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC). pp. 1–8. IEEE Computer Society, Los Alamitos, CA, USA, Hersonissos, Crete, Greece (Sep 20-24, 2010), http://www.christian-engelmann.info/publications/engelmann10facilitating.pdf

[62] Engelmann, C., Naughton, T.: "Toward a Performance/Resilience Tool for Hardware/Software Co-Design of High-Performance Computing Systems". In: Proceedings of the 42^{nd} International Conference on Parallel Processing (ICPP) 2013: 4^{th} International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI). pp. 962–971. IEEE Computer Society, Los Alamitos, CA, USA, Lyon, France (Oct 2, 2013), http://www.christian-engelmann.info/publications/engelmann13toward.pdf

[63] Engelmann, C., Naughton, T.: "Improving the Performance of the Extreme-scale Simulator". In: Proceedings of the 18^{th} IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT) 2014. pp. 198–207. IEEE Computer Society, Los Alamitos, CA, USA, Toulouse, France (Oct 1-3, 2014), http://www.christian-engelmann.info/publications/engelmann14improving.pdf, best paper candidate

[64] Engelmann, C., Naughton, T.: "A New Deadlock Resolution Protocol and Message Matching Algorithm for the Extreme-scale Simulator". Concurrency and Computation: Practice and Experience 28(12), 3369–3389 (Aug 2016), http://www.christian-engelmann.info/publications/engelmann16new.pdf

[65] Fagg, G.E., Bukovsky, A., Dongarra, J.J.: "HARNESS and fault tolerant MPI". Parallel Computing 27(11), 1479 – 1495 (2001), http://www.sciencedirect.com/science/article/pii/S0167819101001004, clusters and computational grids for scientific computing

[66] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: "Detection and correction of silent data corruption for large-scale high-performance computing". In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for. pp. 1–12 (Nov 2012)

[67] Fujimoto, R.M.: "Parallel Discrete Event Simulation". Commun. ACM 33(10), 30–53 (Oct 1990), http://doi.acm.org/10.1145/84537.84545

[68] Fujitsu Technology Solutions: "FUJITSU Server PRIMERGY Performance Report PRIMERGY RX4770 M3". Tech. rep., FUJITSU (Sep 2016), https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-performance-report-primergy-rx4770-m3-ww-en.pdf

[69] Geist, A.: "How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder" (last visited: March 2017), http://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder

[70] Girona, S., Labarta, J., Badia, R.M.: "Validation of Dimemas Communication Model for MPI Collective Operations". In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 39–46. Springer-Verlag, London, UK, UK (2000), http://dl.acm.org/citation.cfm?id=648137.746640

[71] Gunnels, J., Katz, D., Quintana-Orti, E., Van de Gejin, R.: "Fault-tolerant high-performance matrix multiplication: theory and practice". In: Dependable

Systems and Networks, 2001. DSN 2001. International Conference on. pp. 47–56 (July 2001)

[72] GWT-TUD GmbH: “Vampir - Performance Optimization” (last visited: November 2017), https://www.vampir.eu/

[73] Hakkarinen, D., Chen, Z.: “Algorithmic Cholesky factorization fault recovery”. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS). pp. 1–10 (April 2010)

[74] Herault, T., Robert, Y.: “Fault-Tolerance Techniques for High-Performance Computing”. Springer Publishing Company, Incorporated, 1st edn. (2015)

[75] HLRS University of Stuttgart: “CRAY XC40 (HAZEL HEN)” (last visited: November 2017), https://www.hlrs.de/systems/cray-xc40-hazel-hen/

[76] Huang, K.H., Abraham, J.: “Algorithm-Based Fault Tolerance for Matrix Operations”. Computers, IEEE Transactions on C-33(6), 518–528 (June 1984)

[77] Hukerikar, S., Ashraf, R.A., Engelmann, C.: “Towards New Metrics for High-Performance Computing Resilience”. In: Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale. pp. 23–30. FTXS ’17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3086157.3086163

[78] IEEE: “IEEE Standard for Floating-Point Arithmetic”. IEEE Std 754-2008 pp. 1–70 (Aug 2008)

[79] Intel Corporation: “INTEL® XEON® PROCESSOR E7-8867 V3 - Technical Specifications” (last visited: June 2017), https://www.intel.com/content/www/us/en/products/processors/xeon/e7-processors/e7-8867-v3.html

[80] Jain, N., Bhatele, A., White, S., Gamblin, T., Kale, L.V.: “Evaluating HPC Networks via Simulation of Parallel Workloads”. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC ’16 (to appear) (2016)

[81] Jefferson, D.R.: “Virtual Time”. ACM Trans. Program. Lang. Syst. 7(3), 404–425 (Jul 1985), http://doi.acm.org/10.1145/3916.3988

[82] Jou, J.Y., Abraham, J.A.: “Fault-tolerant FFT networks”. IEEE Transactions on Computers 37(5), 548–561 (May 1988)

[83] Jou, J.Y., Abraham, J.A.: “Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures”. Proceedings of the IEEE 74(5), 732–741 (May 1986)

[84] Kahan, W.: "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic". World-Wide Web document (Oct 1997), http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF, manuscript, University of California, Berkeley

[85] Kaiser, H., aka wash, B.A.L., Heller, T., Bergé, A., Biddiscombe, J., Bikineev, A., Mercer, G., Schäfer, A., atrantan, Serio, A., Habraken, J., Anderson, M., Brandt, S.R., Stumpf, M., Bourgeois, D., Copik, M., Huck, K., Amatya, V., Viklund, L., Khatami, Z., Bacharwar, D., Yang, S., Schnetter, E., Bcorde5, Brodowicz, M., Troska, L., Wagle, B., Upadhyay, S., Byerly, Z., Brakmić, H.: "STEllAR-GROUP/hpx: HPX V1.0: The C++ Standards Library for Parallelism and Concurrency" (Apr 2017), https://doi.org/10.5281/zenodo.556772

[86] Kale, L.V., Bhatele, A.: "Parallel Science and Engineering Applications: The Charm++ Approach". CRC Press, Inc., Boca Raton, FL, USA, 1st edn. (2013)

[87] Kale, L.V., Bohm, E., Mendes, C.L., Wilmarth, T., Zheng, G.: "Programming Petascale Applications with Charm++ and AMPI". In: Bader, D. (ed.) Petascale Computing: Algorithms and Applications, pp. 421–441. Chapman & Hall / CRC Press (2008)

[88] Kale, L.V., et al.: "BigSim - Simulating PetaFLOPS Supercomputers" (last visited: November 2017), http://charm.cs.uiuc.edu/research/bigsim

[89] Kale, L.V., et al.: "BigSim Parallel Simulator for Extremely Large Parallel Machines" (last visited: November 2017), http://charm.cs.illinois.edu/manuals/pdf/bigsim.pdf, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign

[90] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: "The Vampir Performance Analysis Tool-Set", pp. 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-68564-7_9

[91] Köster, T., Perumalla, K., Uhrmacher, A.: "Efficient Simulation of Nested Hollow Sphere Intersections: For Dynamically Nested Compartmental Models in Cell Biology". In: Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. pp. 173–183. SIGSIM-PADS '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3064911.3064920

[92] Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: "Basic Linear Algebra Subprograms for Fortran Usage". ACM Trans. Math. Softw. 5(3), 308–323 (Sep 1979), http://doi.acm.org/10.1145/355841.355847

[93] Lee, H.H.S.: "Lecture 13 Multithreading and Multicore Processors" (Aug 2015), https://de.slideshare.net/HsienHsinLee/lec13-computer-architecture-by-hsienhsin-sean-lee-georgia-tech-multicore, Course: ECE 4100/6100 Advanced Computer Architecture, Georgia Institute of Technology

[94] Loh, F., Ramanathan, P., Saluja, K.K.: "Transient Fault Resilient QR Factorization on GPUs". In: Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale. pp. 63–70. FTXS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2751504.2751505

[95] Luk, F.: "Algorithm-Based Fault Tolerance for Parallel Matrix Equation Solvers". Tech. Rep. EE-CEG-85-2, Cornell University (1985), to appear in Proc. SPIE, vol. 564; Real Time Signal Processing VIII

[96] Manchanda, N., Anand, K.: "Non-Uniform Memory Access (NUMA)". New York University 1 (May 2010), http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf

[97] Maslennikow, O., Kaniewski, J., Wyrzykowski, R.: "Fault tolerant QR-decomposition algorithm and its parallel implementation". In: Pritchard, D., Reeve, J. (eds.) Euro-Par'98 Parallel Processing. pp. 798–803. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

[98] Matlis, J.: "Sidebar: The Linpack Benchmark" (last visited: June 2017), https://www.computerworld.com/article/2556400/computer-hardware/sidebar--the-linpack-benchmark.html, ComputerWorld

[99] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "VSC-3 - Oil blade server" (last visited: July 2017), https://www.top500.org/system/178471

[100] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "TOP500.org - Statistics Performance Development" (last visited: June 2017), https://www.top500.org/statistics/perfdevel/

[101] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "Discover SCU11" (last visited: November 2017), https://www.top500.org/system/178529

[102] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "Gyoukou - ZettaScaler" (last visited: November 2017), https://www.top500.org/system/179102

[103] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "HPE SGI 8600 System" (last visited: November 2017), https://www.top500.org/system/179176

[104] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "Piz Daint (CSCS)" (last visited: November 2017), https://www.top500.org/system/177824

[105] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "Sunway TaihuLight System" (last visited: November 2017), https://www.top500.org/system/178764

[106] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "TOP 500 Supercomputers List - November 2017" (last visited: November 2017), https://www.top500.org/lists/2017/11/

[107] Microsoft Corporation: "STDEV function, Microsoft Office" (2017), https://support.office.com/en-us/article/STDEV-function-51fecaaa-231e-4bbb-9230-33650a72c9b0, (Online; accessed on 25 July 2017)

[108] Microway® Incorporated: "Detailed Specifications of the Intel Xeon E5-4600 v3 Haswell-EP Processors" (last visited: August 2017), https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-4600-v3-haswell-ep-processors/

[109] Minkenberg, C., Rodriguez, G.: "Trace-driven Co-simulation of High-performance Computing Systems Using OMNeT++". In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques. pp. 65:1–65:8. Simutools '09, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2009), http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5521

[110] Moldaschl, M., Prikopa, K.E., Gansterer, W.N.: "Fault Tolerant Communication-Optimal 2.5D Matrix Multiplication". Journal of Parallel and Distributed Computing 104, 179 – 190 (2017), http://www.sciencedirect.com/science/article/pii/S0743731517300412

[111] Nair, V.S.S., Abraham, J.A.: "A Model For The Analysis Of Fault-Tolerant Signal Processing Architectures". In: Proceedings Volume 0975, Advanced Algorithms and Architectures for Signal Processing III. vol. 0975, pp. 0975 – 0975 – 12 (1988), http://dx.doi.org/10.1117/12.948508

[112] National Aeronautics and Space Administration (NASA): "NAS Parallel Benchmarks" (last visited: November 2017), https://www.nas.nasa.gov/publications/npb.html

[113] Naughton, T., Engelmann, C., Vallée, G., Böhm, S.: "Supporting the Development of Resilient Message Passing Applications using Simulation". In: Proceedings of the 22^{nd} Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2014. pp. 271–278. IEEE Computer Society, Los Alamitos, CA, USA, Turin, Italy (Feb 12-14, 2014), http://www.christian-engelmann.info/publications/naughton14supporting.pdf

[114] Neilforoshan, M.R.: "Fault Tolerant Computing in Computer Design". J. Comput. Sci. Coll. 18(4), 213–220 (Apr 2003), http://dl.acm.org/citation.cfm?id=767598.767635

[115] NOVATTE: "How to calculate peak theoretical performance of a CPU-based HPC system" (last visited: August 2017), http://www.novatte.com/our-blog/197-how-to-calculate-peak-theoretical-performance-of-a-\cpu-based-hpc-system

[116] ns-3 project: "ns-3 Manual - Release ns-3.26" (last visited: May 2017), https://www.nsnam.org/docs/release/3.26/manual/ns-3-manual.pdf

[117] ns-3 project: "Releases" (last visited: May 2017), https://www.nsnam.org/releases/

[118] O'Gorman, T.J.: "The effect of cosmic rays on the soft error rate of a DRAM at ground level". IEEE Transactions on Electron Devices 41(4), 553–557 (Apr 1994)

[119] Perumalla, K.S., Park, A.J.: "Improving Multi-million Virtual Rank MPI Execution in [MUPI]". In: 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. pp. 454–457 (July 2011), https://doi.org/10.1109/MASCOTS.2011.45

[120] Perumalla, K.S.: "$\mu\pi$: A Scalable and Transparent System for Simulating MPI Programs". In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. pp. 62:1–62:6. SIMUTools '10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2010), https://doi.org/10.4108/ICST.SIMUTOOLS2010.8692

[121] Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., Girona, S., Computadors, D.D.D.: "PARAVER: A Tool to Visualize and Analyze Parallel Code". Tech. rep., In WoTUG-18: Transputer and occam Developments (1995)

[122] Reddy, A.L.N., Banerjee, P.: "Algorithm-based fault detection for signal processing applications". IEEE Transactions on Computers 39(10), 1304–1308 (Oct 1990)

[123] Rexford, J., Jha, N.: "Algorithm-based fault tolerance for floating-point operations in massively parallel systems". In: Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on. vol. 2, pp. 649–652 vol.2 (May 1992)

[124] Rodrigues, A.F., Hemmert, K.S., Barrett, B.W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., Jacob, B.: "The Structural Simulation Toolkit". SIGMETRICS Perform. Eval. Rev. 38(4), 37–42 (Mar 2011), http://doi.acm.org/10.1145/1964218.1964225

[125] Roy-Chowdhury, A., Banerjee, P.: "Algorithm-based fault location and recovery for matrix computations". In: Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing. pp. 38–47 (June 1994)

[126] Seagate Technology LLC: "Hard disk drive reliability and MTBF / AFR" (last visited: February 2017), http://knowledge.seagate.com/articles/en_US/FAQ/174791en?language=en_US

[127] Siewiorek, D., Bell, C., Newell, A.: "Computer structures: principles and examples". McGraw-Hill computer science series, McGraw-Hill (1982), https://books.google.at/books?id=CdlQAAAAMAAJ

[128] SimGrid Team: "SimGrid SMPI 101 - Getting Started with SimGrid SMPI" (last visited: May 2017), http://simgrid.gforge.inria.fr/tutorials/simgrid-smpi-101.pdf

[129] Stallman, R.M., DeveloperCommunity, G.: "Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3". CreateSpace, Paramount, CA (2009)

[130] Stearley, J.: "Defining and Measuring Supercomputer Reliability, Availability, and Serviceability RAS". In: In Proceedings of the Linux Clusters Institute Conference (2005)

[131] Suter, F.: "Bridging a Gap Between Research and Production: Contributions to Scheduling and Simulation". Habilitation à diriger des recherches, Ecole normale supérieure de Lyon (Dec 2014), https://hal.inria.fr/tel-01199185

[132] Techopedia Inc.: "Unrecoverable Error" (last visited: November 2017), https://www.techopedia.com/definition/13453/unrecoverable-error

[133] Tetcos: “How do the different versions of NetSim compare” (last visited: May 2017), http://tetcos.com/version-comparison.html

[134] Tetcos: “NetSim professional” (last visited: May 2017), http://tetcos.com/netsim-pro.html

[135] Tetcos: “NetSim experiment manual” (last visited: November 2017), http://tetcos.com/downloads/v10/NetSim_Experiment_Manual.pdf

[136] The Open MPI Project: “Open MPI: Open Source High Performance Computing” (last visited: August 2017), https://www.open-mpi.org/

[137] The Open MPI Project: “Portable Hardware Locality (hwloc)” (last visited: August 2017), https://www.open-mpi.org/projects/hwloc/

[138] Varga, A., OpenSim, Ltd.: “OMNeT++ Simulation Manual – Introduction” (last visited: May 2017), https://omnetpp.org/doc/omnetpp/manual/

[139] VI-HPS: “SCORE-P” (last visited: November 2017), http://www.vi-hps.org/projects/score-p/

[140] Vinnakota, B., Jha, N.K.: “Diagnosability and diagnosis of algorithm-based fault-tolerant systems”. IEEE Transactions on Computers 42(8), 924–937 (Aug 1993), http://doi.acm.org/10.1109/12.238483

[141] Voskuilen, G., Rodrigues, A., Hammond, S., Moore, B.: “Structural Simulation Toolkit (SST)” (June 13 2015), http://sst-simulator.org/SSTPages/SSTTutorialIscaTutorial/, structural Simulation Toolkit ISCA 2015 Tutorial (13th June 2015, Portland, OR)

[142] VSC - Vienna Scientific Cluster: “VSC-3” (last visited: July 2017), http://www.vsc.ac.at/systems/vsc-3/

[143] Wikipedia contributors: “Double-precision floating-point format — Wikipedia, The Free Encyclopedia” (2017), https://en.wikipedia.org/wiki/Double-precision_floating-point_format, (Online; accessed 22-June-2017)

[144] Wikipedia contributors: “Fault-tolerant computer system — Wikipedia, The Free Encyclopedia” (2017), https://en.wikipedia.org/w/index.php?title=Fault-tolerant_computer_system&oldid=802849427, (Online; accessed 19-December-2017)

[145] Wikipedia contributors: “IEEE 754-1985 — Wikipedia, The Free Encyclopedia” (2017), https://en.wikipedia.org/wiki/IEEE_754-1985, (Online; accessed 22-June-2017)

[146] Williams, T., Kelley, C., et al.: “gnuplot 5.0” (last visited: August 2017), http://www.gnuplot.info/docs_5.0/gnuplot.pdf

[147] Williams, T., Kelley, C., et al.: “gnuplot homepage” (last visited: August 2017), http://www.gnuplot.info/

[148] Wu, P., Ding, C., Chen, L., Gao, F., Davies, T., Karlsson, C., Chen, Z.: “Fault Tolerant Matrix-Matrix Multiplication: Correcting Soft Errors On-line”. In: Proceedings of the Second Workshop on Scalable Algorithms for Large-scale Systems. pp. 25–28. ScalA ’11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2133173.2133185

[149] Yoginath, S.B., Perumalla, K.S.: “Efficient Parallel Discrete Event Simulation on Cloud/Virtual Machine Platforms”. ACM Trans. Model. Comput. Simul. 26(1), 5:1–5:26 (2015), http://doi.acm.org/10.1145/2746232

[150] Zhang, C.N., Yu, Q., Liu, X.W., et al.: “An Algorithm Based Concurrent Error Detection Scheme for AES”, pp. 31–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-17619-7_3

[151] Zheng, G., Kakulapati, G., Kale, L.V.: “BigSim: a parallel simulator for performance prediction of extremely large parallel machines”. In: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. p. 78 (April 2004)