universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

“Fault Tolerant Linear Least Squares Solvers and Matrix Multiplication in Parallel and Distributed Environments”

verfasst von / submitted by

Dipl.-Ing. Karl E. Prikopa, BSc

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

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik / Computer Science
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wilfried Gansterer, M.Sc.

Abstract

Faults are a ubiquitous companion in the world of computing. Generally, the possibility of faults occurring during a computation is neglected, but in mission critical applications fault tolerance has always been at the forefront of their development. The rise of exascale and future extreme-scale supercomputers has increased the importance of handling faults during long-running computations, from hardware faults to soft errors like message loss or silent data corruption. In this thesis the battle against bit-flips is handled at the algorithmic level, while never losing sight of high performance and high accuracy. We apply fault tolerance to two different linear algebra algorithms in two very different environments, while proving that fault handling does not have to come at the cost of high performance or high accuracy.

A core concept employed throughout this thesis is arbitrary precision arithmetic and iterative refinement. We analyse the generalisation of mixed precision iterative refinement to arbitrary precision, which is no longer bound by the standard IEEE floating-point precisions. In particular, we are interested in using significantly lower working precisions to improve the overall performance of the algorithm while still achieving double precision accuracy in the result. Furthermore, we develop an analytical performance model targeted at reconfigurable hardware (FPGAs) and demonstrate the feasibility of low working precisions with experimental evaluations of a software emulated implementation, showing significant performance gains compared to iterative refinement using standard precision levels. Iterative refinement is a naturally self-healing algorithm and therefore can be used to correct bit-flips. The inherently available fault tolerant properties are analysed and further improvements are introduced to expand its usage to practical applications.

Another result of this research is a truly distributed and fault tolerant linear least squares (LLS) solver for wireless sensor networks based on epidemic algorithms known as gossiping. Very few LLS solvers have considered a truly distributed approach, where nodes only communicate with their direct neighbourhood without the need for a fusion centre or clustering. Our approach is based on the semi-normal equations in combination with iterative refinement, which stabilises the otherwise numerically unstable method. The use of arbitrary precision reduces the amount of communication while still achieving high precision accuracy and increasing the performance. The strengths of our LLS solver are advantageous in unreliable, non-static, limited capacity networks. However, we show the applicability of our approach to high-performance systems, where significant performance increases can be achieved compared to established parallel algorithms and libraries, while at the same time protecting the computation from silent data corruption.

The knowledge gained during our research into our fault tolerant LLS solver led to the development of a fault tolerant communication-optimal 2.5D matrix multiplication. The well-known algorithm-based fault tolerance (ABFT) for matrix multiplication adds a small amount of redundant data, the checksums, to the input data to detect and correct erroneous results. In practice, classical ABFT cannot protect all exponent bits of a floating-point number and is restricted to the mantissa and some of the least significant bits of the exponent, an unrealistic limitation in real-world applications. We propose an improved version of ABFT, dABFT, which removes this constraint and can handle bit-flips at any position, a necessity for fault tolerant matrix multiplications on real-world systems. By integrating dABFT into the high-performance 2.5D matrix multiplication we demonstrate that the overhead of our fault tolerant variant is less than one percent, a very small price to pay for the protection of the valuable computational resources against bit-flips.

Zusammenfassung

In der Welt des Computers sind Fehler allgegenwärtig. Im Allgemeinen wird das mögliche Auftreten eines Fehlers während einer Berechnung aber vernachlässigt. Eine Ausnahme bilden missionskritische Systeme, bei denen die Fehleranfälligkeit und dessen Behandlung bereits bei der Entwicklung dieser Systeme eine zentrale Rolle spielt. Mit dem Aufstieg der Supercomputer in den exascale Bereich und darüber hinaus, gewinnt die Fehlerbehandlung bei lang-laufenden Berechnungen immer mehr an Bedeutung, angefangen bei Hardwarefehlern bis hin zum Verlust zu Nachrichten oder zur Datenkorruption. In dieser Dissertation liegt der Fokus auf die Erkennung und Behebung von Bit-Flips (auch bekannt als "silent data corruption"), ohne dabei die hohe Leistungsfähigkeit der Algorithmen und die hohe Genauigkeit der Ergebnisse jemals aus den Augen zu lassen. Dabei wenden wir verschiedene Methoden der Fehlertoleranz auf zwei verschiedene Algorithmen und auf zwei sehr unterschiedlichen Zielarchitekturen an und demonstrieren, dass Fehlerbehandlung nicht auf Kosten der Leistung oder der Genauigkeit erzielt werden muss.

Eine zentrale Komponente dieser Dissertation ist die Verwendung von Arithmetik in variabler Genauigkeit ("arbitrary precision") und der Methode iterativer Verbesserung ("iterative refinement"). Im ersten Schritt wird die Verallgemeinerung der Methode "mixed precision iterative refinement" zu variabler Genauigkeit analysiert, die dadurch nicht mehr an die Standard IEEE Genauigkeiten gebunden ist. Wir sind besonders daran interessiert, Genauigkeiten zu verwenden, die signifikant niedriger sind als IEEE Single Precision, um die Performance des Algorithmus zu verbessern, gleichzeitig aber die Zielgenauigkeit von IEEE Double Precision weiterhin zu erreichen. Um die Wirksamkeit der Verwendung von niedriger Präzisionen zu demonstrieren, wurde ein analytisches Performancemodell entwickelt, welches auf die Verwendung von rekonfigurierbarer Hardware (FPGAs) ausgelegt ist. Basierend auf einer Softwareemulation von "arbitrary precision iterative refinement" konnten signifikante Leistungsverbesserungen im Vergleich zur Verwendung von IEEE Standardpräzisionen erzielt werden. Iterative refinement ist ein selbst-heilender Algorithmus und eignet sich daher ideal, um Bit-Flips zu korrigieren. Die inhärent vorhandenen fehlertoleranten Eigenschaften des Algorithmus werden analysiert und weitere Verbesserungen eingebaut, um die Verwendung in der Praxis zu ermöglichen.

Ein weiteres Ergebnis dieser Forschungsarbeit ist ein verteilter, fehlertoleranter Algorithmus für das lineare Ausgleichungsproblem ("linear least squares", LLS) für drahtlose Sensornetzwerke, der auf den epidemiologischen Algorithmen namens "Gossiping" basiert. In der Literatur gibt es sehr wenige LLS Algorithmen, die wahrhaftig verteilt arbeiten, bei denen es keine Zusammenführung der Ergebnisse zu einer zentralen Stelle ("fusion centre") oder zumindest zu einer zentralen Stelle einer Gruppe von Sensoren gibt ("clustering") und die stattdessen nur mit ihren direkten Nachbarn kommunizieren. Unser Zugang bedient sich der Semi-Normalgleichungen ("semi-normal equations") in Kombination mit Iterative Refinement, welches die ansonsten numerisch instabile Methode stabilisiert. Durch die Verwendung von variabler Genauigkeit wird die Menge an benötigter Kommunikation zwischen den Knoten reduziert, wodurch die Performance des Algorithmus steigt und weiterhin die hohe Zielgenauigkeit erreicht wird. Die Stärken unseres Algorithmus können in nicht zuverlässigen, nicht statischen, ressourcenknappen Netzwerken ausgeschöpft werden. Allerdings kann unsere Methode auch in hochperformanten Systemen zum Einsatz kommen, auf denen wir signifikante Performancezuwächse im Vergleich zu etablierten parallelen Algorithmen und Programmbibliotheken vorzeigen können, während die Berechnung gleichzeitig vor Datenkorruption durch Bit-Flips oder Nachrichtenverlust geschützt wird.

Die durch unsere Forschung an fehlertoleranten LLS Algorithmen gewonnenen Erkenntnisse führten weiters zur Entwicklung unserer fehlertoleranten 2.5D Matrixmultiplikation ("fault tolerant communication-optimal 2.5D matrix multiplication"). Der bekannte fehlertolerante Algorithmus ABFT ("algorithm-based fault tolerance") für Matrixmultiplikationen fügt kleine, redundante Datenmenge zur Eingabematrix hinzu, die Prüfsummen der Matrix beinhalten, um fehlerhafte Ergebnisse zu erkennen und zu korrigieren. In der Praxis kann klassisches ABFT aber nicht alle Positionen des Exponenten einer Gleitkommazahl schützen und ist auf die Mantisse und einige der niedrigwertigsten Stellen des Exponenten begrenzt, eine unrealistische Einschränkung in alltäglichen Anwendungsfällen. Um diese Restriktionen zu entfernen, haben wir eine verbesserte Version von ABFT entwickelt, dABFT, die Bit-Flips an jeder Position einer Gleitkommazahl beheben kann, eine absolute Notwendigkeit, um die fehlertolerante Matrixmultiplikation in echten Anwendungen verwenden zu können. Durch die Kombination von dABFT mit der hochperformanten 2.5D Matrixmultiplikation haben wir bewiesen, dass der Overhead unserer fehlertoleranten Variante des Algorithmus weniger als ein Prozent ausmacht. Dies ist ein sehr geringer Preis, um die wertvollen Ressourcen von Supercomputern gegen Bit-Flips zu schützen.

Contents

Chapter 1

Introduction

Fault tolerance has been a serious concern in computing systems since the advent of computers in mission critical applications, such as aerospace, medical equipment, military and the infrastructure networks for traffic control, power supply or communication. A malfunction in one of these systems could lead to catastrophic consequences, in the worst case to the loss of life. Long-term space exploration missions have to survive the harsh environment of space with little to no protection against the majority of cosmic threats our atmosphere normally shields us from. Even on earth, our atmosphere cannot protect us from all forms of cosmic radiation. These cosmic rays, e. g. neutron radiation, can affect our computational components and introduce faults, so called *silent data corruptions* (also known as single event upset or bit-flips), into our computations and corrupt our results.

Historically, many highly customised computers and processing chips were developed to support fault tolerance in critical systems [Sie91], each with their own emphasis on a specific task. One of the first fault-tolerant computers, the SAPO computer, was built in the 1950s in Czechoslovakia and used triplication of the components and voting on the correct output with automatic retries in the event of an error to ensure receiving the correct result. The STAR (Self-Test and Repair) computer developed in the 1960s as a satellite-control computer used functional-unit redundancy, voting and task-level rollbacks as some of its fault-tolerant techniques. The Voyager space probes used standby redundancy for entire subsystems. As one can see by these examples, fault tolerance mainly focused on using redundant hardware components to detect and handle faults, a very costly endeavour especially considering the highly specialised components required for these applications.

The susceptibility of petascale and exascale high-performance systems to faults has already been identified and recognised as an important challenge [SWA*13, CGG*14]. Large applications can run for days or weeks on such systems and it is important to ensure that the valuable computing time is not wasted due to faults during the computation. Furthermore, it would not be feasible to use triple or N-times the amount of resources to handle faults through redundant computations. There are a variety of permanent or transient fault types which have to be considered, from hardware faults to node crashes as well as soft errors like message loss or bit-flips. In this thesis, we will focus on handling silent bit-flips in the registers, the cache as well as in the main memory. We will also consider the effects of message loss and message corruption through bit-flips in loosely coupled distributed systems. Performance and scalability of algorithms have to take into consideration the efficient handling of faults. The approach we pursue in this thesis is fault tolerance at the algorithmic level, which incorporates the detection and correction of faults within the algorithm itself, ideally with limited impact on performance, and delivers the correct result even in the presence of faults during the computation.

The primary focus of this thesis will be the development of fault tolerant algorithms to handle silent data corruptions at the algorithmic level while not compromising on high performance and high accuracy. We will focus on applying fault tolerance to two different linear algebra algorithms in two very different environments. We will develop a truly distributed linear least squares solver for wireless sensor networks using iterative refinement and epidemic algorithms known as gossiping. The second algorithm is a parallel matrix multiplication for high-performance supercomputers, the core component of many numerical algorithms, where we improve the fault tolerant properties of algorithm-based fault tolerance for practical use.

1.1 Problem Statements and Motivation

In this section, we will define the problem statements for both algorithms and revise the basic properties of wireless sensor networks and the challenges arising when developing algorithms for truly distributed networks.

1.1.1 Linear Least Squares Solvers on Wireless Sensor Networks

Scientists often face the problem to find the best fit for measurements subject to errors to the parameters of a model. Naturally with an increased number of measurements the accuracy of the model can be improved, resulting in the need to solve an overdetermined linear or non-linear system of equations. Of course, the overdetermined system of equations does not have a unique solution. This requires finding a solution which minimises a weighted sum of the squares of the residual, also known as the *least squares problem* [Bjo96]

$$\min_x \|b - Ax\|_2 \tag{1.1}$$

where $A \in \mathbb{R}^{n \times m}$ with $n \geq m$, $b \in \mathbb{R}^n$ and $x \in \mathbb{R}^m$. Least squares problems arise in many different fields, for example in signal processing, photogrammetry, geodetics and statistics. There are many different approaches for solving least squares problems. As summarised in [Bjo96], the available numerical methods include direct methods, for example applying the pseudo-inverse, a QR decomposition, or forming the normal equations and solving a linear system, and iterative methods.

The need for *distributed* least squares solvers arises when being applied to wireless sensor networks. Wireless sensor networks (WSN) consist of a large number of inexpensive sensor nodes which cooperate with each other to achieve a common goal. The sensor nodes are normally constrained in terms of their resources, primarily their energy consumption and computation capabilities. Wireless sensor networks can be deployed in a wide range of applications. A WSN can monitor a wide area of interest and measure specific physical properties, for example temperatures, chemical levels or seismic activity. WSN can be used to measure levels of chemical substances in the air or in the water, detect any abnormal behaviour and alert the authorities to act on this information. Another example for the application of WSNs is the protection of critical infrastructures like the "smart grid" [MRAMBJ12], the next evolution of the power grid. The smart grid monitors the power usage of the consumers and acts based on the collected data. The goal is to improve the efficiency and reliability of the power grid and sustain the on-demand delivery of electricity.

While there are many possible applications for wireless sensor networks, such networks also pose many challenges. The typical sensor nodes have very limited capacities in terms of energy and computing power, limitations that have to be considered when designing and implementing algorithms aimed at these kinds of networks. One of the sources of high power consumption is the communication. The energy required by the nodes to communicate with other nodes is directly proportional to the communication range [SLS*12]. This implies that communicating with the immediate neighbourhood of a node is significantly cheaper than communicating with very distant nodes.

Another uncertainty is the wireless network connection, which is unreliable in nature and can lead to link failures or message losses. The entire network can be variable and unpredictable. Sensor nodes do not have to be stationary and mobile nodes lead to a non-static network topology. Nodes can join or leave the network at any time. Managing the information about arrivals and departures of nodes, for example due to defects, incurs additional overhead.

The literature proposes many so-called "distributed" algorithms for wireless sensor networks. Many of these approaches employ the use of a "fusion centre", a central processing node normally more powerful than the individual sensor nodes. The fusion centre aggregates the data from all sensor nodes, computes the required solution and then distributes the result to all nodes. This method comes with many drawbacks. The nodes are distributed over a wide area and the cost of communication with a central unit over long distances is very high. Another source of a costly overhead is the use of routing tables. Due to the dynamic network, the routing tables have to be updated continuously incurring more expensive communication. The network controller of the fusion centre can become a communication bottleneck due to the vast number of messages received from the entire network. Last but not least, the fusion centre is a "single point of failure". If the central unit fails, the entire network becomes inoperable. To avoid all these shortcomings, the computation should take place in a truly distributed nature, decentralised on the sensor nodes themselves.

As already mentioned, the algorithms have to be designed keeping in mind the constrained environment of the sensor nodes. Preserving energy increases the lifespan of the nodes and in turn of the entire network. Limiting the communication is one important aspect of distributed algorithms on WSNs. Aside of limiting the amount of messages, the range of the communication should be limited to the immediate neighbourhood. Minimising the communication costs allows for an increased amount of computations to take place on the sensor nodes. An example is given by Panigrahi et al. [PPPM12], where 1 KB of data being sent over 100m, operating at 1GHz, uses 3J of energy. The same amount of energy can be used for 300 million instructions on a 100 MIPS/watt general-purpose processor.

The limited computation capacity can further be improved through the use of lower precisions within the computations. One may believe that this would come with a decrease in accuracy, but methods like iterative refinement [Wil65] can be employed to achieve the same or even higher accuracy using higher precisions for specific, low complexity operations, while computing the majority of operations in a lower precision. Iterative refinement is a core component of this thesis and will be applied to the distributed least squares solver to decrease the amount of messages required and to reduce the computational strain on the sensor nodes.

The unreliable network and communication require algorithms to be robust against failures. These include temporary node or link failures, for example the loss of messages, as well as permanent node or link failures, which can lead to significant loss of information depending on when the failure occurs. In addition, soft errors like bit-flips also have to be considered. These aspects have to already be considered during the design phase of a distributed algorithm.

There are many different ways to achieve fault tolerance. The lowest-level approach targets the hardware level directly. This can range from special hardware components that are more resilient against failures to redundancy through duplication of the entire system. Obviously, these measures come at a high economic cost. Another approach would be to target the communication protocols. Fault resilient/tolerant protocols, e. g. [AFB*06], exist, but incur an overhead in maintaining additional information to ensure fault tolerance or additional messages to ensure correct delivery of the information. Including fault tolerance at the algorithmic level leads to an approach independent of the underlying hardware components or protocols. This can either be included in the complex high-level operations of the algorithm or in the low-level operations which are used as building blocks of the algorithm. An algorithm consisting of lower fault tolerant operations will itself be fault tolerant.

The goal of this research is a fully distributed least squares solver without the need of a centralised fusion centre using gossip algorithms. The algorithm has to be fault resistant in terms of temporary and permanent node or link failures. The use of reduced accuracy and using iterative refinement to improve the approximate result, leading to less communication and computation time will be a central aspect of this thesis. The distributed least squares solver developed within the scope of this research will not be limited to wireless sensor networks, but will be applicable to any type of distributed network. However, its strengths will be advantageous in unreliable, non-static, limited capacity networks.

1.1.2 ABFT for Matrix Multiplication

While considering different approaches to fault tolerance for linear least squares solvers, we came across algorithm-based fault tolerance (ABFT) for matrix multiplication [HA84]. However, on closer examination of the existing literature, we noticed that ABFT was limited to specific bits in a floating-point representation, specifically to the bits in the mantissa and only the lowest bits in the exponent. Such restrictive fault locations are impractical and are preventing us from using ABFT in real-world applications. We therefore sought out to remove these restrictions.

Fault tolerance is often associated with high overheads in order to protect a computation, e. g. N modular redundancy (NMR), which requires N times the amount of resources, or the widely used checkpoint/restart (C/R) method [SPD*05, CGG*14]. Checkpoints generate a significant amount of I/O traffic and often block the progression of the application [CGG*14]. The efficiency of C/R decreases with increasing system size [FME*12]. The benefit of these methods is that they are generally applicable and do not require special implementations of the algorithms. However, the drawbacks, such as the high resource costs and overheads, outweigh their advantages in many scenarios.

For a crucial component like the matrix multiplication, the building block of so many linear algebra algorithms, any performance impact due to fault tolerance has to be kept at a minimum. Fault tolerance and high performance do not have to contradict each other, as we will demonstrate throughout this thesis. The goal of our research of ABFT for matrix multiplications is to improve the method and remove artificial limitations, while keeping the performance impact at a minimum.

1.2 Thesis Outline

This thesis is organised into three larger, yet equally important parts:

1. The study of iterative refinement and the expansion to arbitrary precision focusing on precisions lower than the double precision target precision to improve performance and reduce communication costs (Chapters 2-4).

2. The search for a truly distributed linear least squares (LLS) solver by employing gossiping in combination with the beforementioned properties of IR and arbitrary precision to improve performance and recover from silent data corruption and message loss (Chapters 5-7).

3. The practical use of algorithm-based fault tolerance in the context of high-performance matrix multiplication to recover from faults at any position in a floating-point representation, partly inspired by the recovery properties of iterative refinement (Chapters 8-10).

1.3 Overview and Contributions

Chapter 2 provides an introduction to iterative refinement (IR) [Wil63] which is an essential building block of this thesis. IR is a strategy for improving the accuracy of a computed solution by reducing the round-off errors. The method iteratively computes a correction term to an approximate solution by solving a linear system using the residual of the result as the right-hand side. In the related work, we discuss the variations and applications of IR. Additionally, we derive a model for the number of iterations required by IR to reach a desired target accuracy.

In Chapter 3, we explore the possibilities of mixed precision IR for solving eigenvalue problems. Mixed precision approaches exploit the performance benefits of executing the majority of the operations in a lower working precision (e. g. single precision) and only performing critical operations in the higher target precision (e. g. double precision) while still obtaining a result that is accurate to the target precision [BDK*08]. We return to the origin of IR, Newton's method, and investigate various approaches to solve the resulting linear system, including matrix splitting techniques and solvers for saddle point problems. A complexity analysis shows that for our method the number of floating-point operations is lower than the previously described iterative improvement method for eigenvalue problems by Dongarra, Moler and Wilkinson [Don82].

In Chapter 4, we introduce *arbitrary precision iterative refinement* (APIR) for solving dense linear systems based on LU factorisation. In APIR, the precisions levels used are no longer restricted to the standard IEEE 754-2008 [IEE08] floating-point formats. We are specifically interested in working precisions which are below single precision and will therefore lead to increased performance benefits. Analytical performance models based on arbitrary precision floating-point arithmetic on reconfigurable hardware (FPGAs) in combination with experimental evaluations of a software emulated implementation illustrate that this approach can achieve significant performance gains over double precision direct LU-based solvers and over classical double/single precision LU-based iterative refinement.

The search for a truly distributed linear least squares (LLS) solver for large loosely connected distributed networks (such as wireless sensor networks) begins with Chapter 5. As discussed in subsection 1.1.1, a *truly distributed* algorithm should ideally require very little coordination between the nodes. This favours algorithms which do not require a fusion centre, cluster heads or any multi-hop communication. First, we review the existing literature for solving LLS problems in distributed environments and categorise them based on their communication pattern. We also introduce PSDLS, a truly distributed LLS solver using the gossip algorithm push-sum [KDG03] as its aggregation function, limiting its communication to only the neighbouring nodes. In the subsequent chapters, PSDLS will form the basis of the LLS algorithms developed in this thesis, where further performance optimisations and fault tolerant techniques will be implemented. We analytically compare the communication cost of PSDLS to truly distributed algorithms existing in the literature and illustrate that our novel PSDLS solver requires significantly fewer messages per node than the previously existing methods to reach a predefined solution accuracy.

In Chapter 6, we present the novel parallel linear least squares solvers ARPLS-IR and ARPLS-MPIR for dense overdetermined linear systems. All internode communication of our ARPLS solvers arises in the context of all-reduce operations across the parallel system and therefore they benefit directly from efficient implementations of such operations. Our approach is based on the semi-normal equations, which are in general not backward stable. However, the method is stabilised by using iterative refinement. We show that performing IR in mixed precision also increases the parallel performance of the algorithm. We consider different variants of the ARPLS algorithm depending on the conditioning of the problem and in this context also evaluate the method of normal equations with iterative refinement. For ill-conditioned systems, we demonstrate that the semi-normal equations with standard iterative refinement achieve the best accuracy compared to other parallel solvers. We discuss the conceptual advantages of ARPLS-IR and ARPLS-MPIR over alternative parallel approaches based on QR factorisation or the normal equations. Moreover, we analytically compare the communication cost to an approach based on communication-avoiding QR factorisation. Numerical experiments on a high-performance cluster illustrate high speed-ups on 2048 cores for ill-conditioned tall and skinny matrices over state-of-the-art solvers from DPLASMA or ScaLAPACK.

Our truly distributed linear least squares solver GLS-IR for overdetermined linear systems on wireless sensor networks is presented in Chapter 7. Like ARPLS-IR, the solver is based on the semi-normal equations or normal equations combined with IR in mixed precision. In this case, IR does not only stabilise the method but also decreases the communication cost. In GLS-IR, all communication between nodes is contained within a gossip-based algorithm for distributed aggregation, which limits the communication of each node to its immediate neighbourhood. Therefore, GLS-IR benefits directly from efficient and fault-tolerant implementations of such operations. We use a fault-tolerant alternative to the push-sum method, the push-flow algorithm [GNSSG13], which is able to recover from silent message loss and temporary or permanent link failures. We analytically compare the communication cost of GLS-IR to existing truly distributed algorithms. Since the theoretical analysis contains problem-dependent parameters, numerical experiments are needed in order to get a complete picture. Our simulation experiments illustrate a significantly reduced communication cost of GLS-IR compared to other existing truly distributed least squares solvers. We also illustrate that due to the properties of iterative refinement and push-flow, GLS-IR can achieve a result accurate to machine precision even if a high amount of message loss occurs.

In Chapter 8, we discuss the existing related work on algorithm-based fault tolerance (ABFT). In this approach, a small amount of redundant data, the checksums, is added to the input data to detect and correct erroneous results. Many variations have been described in the literature for many different linear algebra methods, including LU factorisation and matrix multiplication. We present *FTAPIR*, a fault tolerant version of APIR using ABFT to detect and recover from faults during the iterative refinement process.

Analysing the properties and the resilience of fault tolerant algorithms requires the simulation of bit-flips. Therefore, we develop a thread-based bit-flip fault injector in Chapter 9, FaITh (**Fa**ult **I**njector **Th**reads), which mimics the effects of real bit-flips, which sooner or later will mainly result in a memory corruption. A bit-flip fault injector has to have a low overhead to not impact the performance of the main algorithm. It should ideally require minimal code modifications and work with existing libraries without the need for recompilation. Furthermore, to analyse the resilience of the fault tolerant techniques, fine-grained control concerning the bits of a floating-point representation, which shall be affected by bit-flips, is required. Our fault injector fulfils all these requirements and is purely written in the C++11 standard. FaITh has a very low overhead and therefore is ideal to examine our fault tolerant matrix multiplication on thousands of cores of a supercomputer.

In Chapter 10, we illustrate that in practice classical algorithm-based fault tolerance (ABFT) cannot protect all exponent bits of a floating-point number. Consequently, we extend the method to recover from bit-flips in all positions without additional overhead. We also derive fault detection conditions suitable for multiple checksum encoding vectors. Moreover, we show how to efficiently employ ABFT to protect communication-optimal parallel 2.5D matrix multiplication against bit-flips occurring silently during the computation. Furthermore, we show that for very low fault rates the overhead of fault tolerance in the context of the 2.5D matrix multiplication algorithms can be reduced even further. Numerical experiments on a high-performance cluster illustrate the high scalability and low overhead of our algorithms. We demonstrate the fault tolerance of our approach with randomly and asynchronously injected bit-flips (using our fault injector FaITh) and illustrate that our method can also handle bit-flips occurring at high frequencies. Like in classical ABFT, the overhead per correctable bit-flip of our approach decreases with increasing error rate.

1.4 Publications

The results summarised in this thesis have been presented in the following peer-reviewed publications at conferences and in journals:

Prikopa K. E., Gansterer W. N.: On Mixed Precision Iterative Refinement for Eigenvalue Problems. *Procedia Computer Science 18* (2013), 2647–2650. International Conference on Computational Science (ICCS).
DOI: 10.1016/j.procs.2013.06.002

Prikopa K. E., Mücke M., Gansterer W. N.: Arbitrary Precision Iterative Refinement. under revision, 2021.

Prikopa K. E., Straková H., Gansterer W. N.: Analysis and Comparison of Truly Distributed Solvers for Linear Least Squares Problems on Wireless Sensor Networks. In *Euro-Par 2014 Parallel Processing, vol. 8632 of Lecture Notes in Computer Science*. Springer, 2014, pp. 403–414.
DOI: 10.1007/978-3-319-09873-9_34

Prikopa K. E., Gansterer W. N., Wimmer E.: Parallel Iterative Refinement Linear Least Squares Solvers based on All-Reduce Operations. *Parallel Computing 57* (2016), 167–184.
DOI: 10.1016/j.parco.2016.05.014

Prikopa K. E., Gansterer W. N.: Fault-tolerant Least Squares Solvers for Wireless Sensor Networks based on Gossiping. *Journal of Parallel and Distributed Computing 136* (2020), 52–62.
DOI: 10.1016/j.jpdc.2019.09.006

Moldaschl M., Prikopa K. E., Gansterer W. N.: Fault Tolerant Communication-Optimal 2.5D Matrix Multiplication. *Journal of Parallel and Distributed Computing 104* (2017), 179–190.
DOI: 10.1016/j.jpdc.2017.01.022

Part I

Arbitrary Precision Iterative Refinement

Chapter 2

Iterative Refinement

A methodological core component of this thesis is iterative refinement (IR), a method for iteratively reducing a round-off error and thereby improving the accuracy of an already computed solution. The method iteratively computes a correction term to an approximate solution by solving a linear system using the residual of the result as the right-hand side.

Iterative refinement was first analysed in detail by Wilkinson [Wil65, Wil63], but had already been used in desk calculators and computers in the 1940s [Hig97]. Wilkinson first described the process for solving linear systems and using a scaled fixed-point arithmetic. The analysis was later expanded by Moler [Mol67] to cover floating-point arithmetic. The use of floating-point arithmetic especially affects the calculation of the residual, the first step of the iterative refinement. The author analysed the round-off errors and convergence and included the additional factors introduced by the usage of floating-point arithmetic.

Iterative refinement can be used in combination with many solvers to improve the accuracy of a result. Aside from linear systems, eigenvalue problems [SW80, Don82, DMW83, DHT01] or linear least squares problems [Gol65, GW66, Gul94, DHRL09] have been combined with iterative refinement. Iterative refinement can also be used to stabilise otherwise unstable methods, as demonstrated by Higham [Hig91] for a QR factorisation with poor row scaling.

In this thesis, we will explore the possibilities of iterative refinement in combination with many different solvers. We will improve the performance of iterative refinement to solve eigenvalue problems using mixed precision. We extend the linear systems solver to arbitrary precision and demonstrate the performance benefits of using precisions outside of the range provided by the standardised IEEE definitions. In Chapters 6 and 7 we apply our newly gained knowledge to linear least squares solvers and use arbitrary precision to increase the performance and reduce the communication cost of distributed algorithms. Furthermore, we use another property of iterative refinement to make our algorithms fault tolerant. These are the beginnings of our journey to achieve all these goals.

2.1 Algorithmic Background of Iterative Refinement

Iterative refinement is based on the Newton-Raphson method, which finds a root of a function $f(x)$ using the following iterative process.

$$x_{k+1} = x_k + \Delta x_k \quad \textit{with} \quad \Delta x_k = -\frac{f(x)}{f'(x)} \tag{2.1}$$

In higher dimensions, the derivative of $f(x)$ is the Jacobian matrix $J_f(x)$ which results in finding the solution to a linear system of equations.

$$J_f(x_k)\Delta x_k = -f(x_k) \tag{2.2}$$

In iterative refinement, the function $f(x)$ is the residual of the solution which is being improved. For solving linear systems of the form $Ax = b$, the residual is

$$f(x) = Ax - b \quad .$$

The steps of the iterative refinement process for improving the solution of linear systems are as follows:

1. Solve $A\widehat{x} = b$ with $\widehat{x}$ being an approximation of x
2. For $i = 0, 1, 2, \ldots$ with $x_0 = \widehat{x}$
   (a) Compute the residual $r_i = b - Ax_i$
   (b) Solve $A\Delta x_i = r_i$
   (c) Update $x_{i+1} = x_i + \Delta x_i$

IR uses an approximate initial solution and increases the accuracy of the solution by computing the residual of the result and using the residual as the right-hand side to solve a linear system for the correction term Δx. Finally, the correction term is added to the result to correct the solution of the linear system. These steps are repeated until the requested accuracy is reached. The cost of the iterative improvement is very low compared to the cost of the factorisation but it results in a solution which can be accurate to machine precision.

The literature describes many different termination criteria for iterative refinement, which use different measures to check if the convergence is complete. For example, the process can be halted if the norm of the residual $\|r_i\|_2$ or the norm of the correction term $\|\Delta x_i\|_2$ is under a described tolerance, which can be the machine epsilon ε or a tolerance which also includes the condition number of the input matrix. Other approaches check if the correction term is changing the solution significantly enough.

The algorithmic details, the error analysis and the convergence of iterative refinement in general and our novel method *arbitrary precision iterative refinement* (APIR) in particular will be discussed in detail in Chapter 4.

2.2 Variations of Iterative Refinement

A wide range of variations of iterative refinement exist which mainly differ in the precisions used for computing the different steps in the process. The standard iterative refinement method (SIR) uses the same precision to compute both, the initial solution and the correction term for the improved result. The Extra Precise Iterative Refinement (EPIR) [DHK*06] uses a higher precision to compute the residual and the correction term of the solution to compensate for slow convergence and ill-conditioned systems.

Mixed precision iterative refinement (MPIR) [BDK*08] is a special performance-oriented case of IR for solving linear systems of equations, where the majority of operations, mainly the matrix decomposition, is computed in a lower precision, usually IEEE single precision (SP). Only computing the residual, which is critical to the accuracy of the solution, is performed in a higher precision, usually IEEE double precision (DP), an operation of low complexity compared to the decomposition. Due to iterative refinement, the result is accurate to the targeted precision while the performance has been greatly improved because the iterative process incurs only a very low additional cost. As long as the system is not too ill-conditioned, MPIR achieves the same or higher accuracy than a DP direct solver while achieving a performance benefit by predominantly operating at the lower precision. This is particularly attractive on chip architectures where lower precision operations exhibit significant performance benefits over higher precision operations, e. g. GPUs or FPGAs.

The authors in [KBD08] implemented a linear solver using the iterative refinement method on the CELL processor, where single precision computations are performed at a much higher rate than double precision computations and which is therefore an interesting target platform for mixed precision iterative refinement. The implementation focuses on symmetric positive definite systems of linear equations and therefore uses the Cholesky factorisation to solve the system. The main goal of the author's implementation was to exploit the thread-level parallelisation and fine-grained task granularity to reach the peak performance of the CELL processor. On the tested CELL processors, the mixed precision solver only produces a relatively small overhead between 9 and 15% compared to the single precision implementation, but due to iterative refinement delivers the result in double precision accuracy. The speedup compared to the double precision peak performance of the CELL processors is on average 10 and also includes the benefit of delivering the result in double precision accuracy.

2.3 Related Work

Demmel et al. [DHK*06] define reliable error bounds for the solution of the iterative refinement process which also require very low additional computational cost. With the help of scaling matrices, a component-wise error bound can also be defined. They transform the input matrix A by applying two diagonal matrices R and C and by choosing R to equilibrate the rows of A the condition number can be significantly reduced and reach the Skeel condition number, which can be significantly smaller than the condition number of A. The authors also address the issue of extremely ill-conditioned input matrices and have found an approach which in some cases can lead to a successful convergence by storing the solution vector x using extra precision.

Kiełbasiński [Kie81] proposed an iterative refinement method which is not bound to fixed precisions. The process determines the lowest sufficient precision for the computation of the residual vectors taking into consideration, among other factors, the condition number of the input matrix A. If certain values required by the *binary cascade iterative refinement* (BCIR) are not known a priori, the algorithm can be modified to adapt the length of the mantissa based on observations of the iterative process. The author also analyses the algorithm in terms of accuracy and time-cost and shows that a maximum precision can be determined to stop the iteration and reach a requested accuracy for the solution vector x. An extended analysis of the binary cascade iterative refinement was performed in [Pri11]. In terms of accuracy of the solution, BCIR was shown to deliver the best results compared to other iterative refinement methods. This is largely due to the adaptive choice of the working precisions. However, these working precisions were very often significantly higher than the target precision. A performance model presented in [Pri11] shows, that BCIR cannot compete with the other iterative refinement methods. The differences in the relative residual of the results compared to standard iterative refinement are not significant enough to justify the modelled high computational costs.

Alkurdi and Kincaid [AK06] focused on applying iterative refinement to solve sparse linear systems and on exploiting the benefits of a sparse input matrix A. Normally, the LU-decomposition using iterative refinement to increase the accuracy of the solution requires more storage and more computing time, but in this case the sparsity of the matrix A can be exploited to reduce these requirements. The sparsity pattern of the LU factors of A is determined by using the powers of a Boolean matrix strategy, which tries to find two Boolean matrices which fulfil $B^{2m} = B^{2m+1}$, where $1 \leq 2m \leq n - 1$ with n being the dimension of the matrix to determine the fill-in positions in A. Through the use of this strategy, the factorisation is inaccurate, but the following use of the iterative refinement compensates for the lost accuracy and achieves a result with an increased accuracy compared to the direct solution without iterative refinement in addition to the reduction of the amount of storage required and the computing time.

In [OOR09], the authors have proposed an iterative refinement algorithm for ill-conditioned linear systems and have proved the forward and backward stability of this algorithm. They further show that with the new method even matrices with high condition numbers can be solved in a very small number of iterative refinement steps. For this purpose they have chosen Hilbert matrices and Rump's matrices, which have extremely high condition numbers and in their examples can still be solved in 3 to 5 iterations.

Anzt et al. [ARH10] investigated mixed precision iterative refinement on hybrid computing platforms, using a GPU as a co-processor. The authors came to the conclusion that hardware-aware algorithms lead to increased performance and also reduces energy requirements due to the shorter computation times of the single precision operations.

Businger and Golub [BG65] developed an iterative refinement method for linear least squares solutions using Householder transformations. The authors show that a correction vector can be obtained using the residual, leading to a successive solution to a linear least squares problem. Using the already computed decomposition and transformation for the initial solution, the residual and correction vector can be computed at a very low cost, continuing the iterative process until convergence. Golub and Wilkinson [GW66] provide an extensive error analysis for the least squares iterative refinement method with resemblance to the linear equation case.

Iterative refinement for linear least squares problems is the main focus of Chapters 6-7, where the method will be applied to parallel and distributed solvers using mixed and arbitrary precisions. The related work relevant to linear least squares solvers will be discussed therein.

2.4 A Model for the Number of Iterations Required by IR

The condition number $\kappa(A)$ provides a means to estimate the accuracy of the solution to a linear system. This property can also be used to estimate the number of iterations required by IR to achieve a given target precision p. The following is based on the explanations by Rice [Ric81], where the number of iterations required by IR is used to roughly estimate the condition number of the linear system $Ax = b$.

The logarithm to base b of $\kappa(A)$ returns an estimate of the number of base-b digits that are lost while solving the linear system. Let s denote the number of correct base-b digits obtained by solving the linear system, then the accuracy of the solution can be increased by s digits in each iteration. In order to reach the base-b target precision p, the required number of iterations until convergence is therefore described by $i_{\text{conv}} = p/s$, gaining s digits of accuracy in each iteration. This leads to the following estimate for $\kappa(A)$:

$$\kappa(A) \approx b^{p-s} = b^{p-p/i_{\text{conv}}} \quad . \tag{2.3}$$

Using this estimate, the following model can be defined to determine the number of iterations required by IR:

$$i_{\text{conv}}(p, \kappa) \approx \frac{p}{p - log_b(\kappa)} \quad . \tag{2.4}$$

2.5 The Use of Iterative Refinement in this Thesis

The approach used by MPIR can further be improved by reducing the lower working precision below single precision, leading to an *arbitrary precision* iterative refinement (APIR). Arbitrary precision is not bound to IEEE standard precisions and has a wide range of applications, some of which are in use in everyday life. Arbitrary precision plays a great role in cryptography [KA11] and is present in modern web browsers using public-key cryptography. Other common applications of arbitrary precision include calculating mathematical constants like π or the ability to prevent overflows and underflows by increasing the precision of computations. Of course, arbitrary precision is used to increase the accuracy of computations.

APIR allows for the use of working precisions well below single precision. Although the number of iterations is increased with a decrease in working precision, the performance benefit of the lower working precision is the dominant factor and therefore the entire process benefits from a performance increase. The minimum precision possible to apply depends on the input data, on the condition number and size of the matrix. Arbitrary precision allows for faster computations due to the reduction of storage costs and the amount of data being transferred between the memory hierarchies. The smaller floating-point numbers also reduce the message sizes in distributed environments and therefore the communication cost between nodes.

In the next chapter, we first investigate the possibilities of using mixed precision iterative refinement in combination with eigenvalue solvers. In Chapter 4, we will extend iterative refinement for linear systems to arbitrary precision and present a performance model to predict the performance benefits of our approach on field programmable gate arrays (FPGAs). After studying these effects extensively, we apply our findings to linear least squares solvers in parallel and distributed environments (Chapters 6 and 7, respectively). The computational limitations on sensor nodes can greatly benefit from the reduction of the precision of the computations by reducing the bit-width of the floating-point values.

Chapter 3

Mixed Precision Iterative Refinement for Eigenvalue Problems

We consider the eigenvalue problem

$$Ax = \lambda x$$

with symmetric $A \in \mathbb{R}^{n \times n}$, the eigenvalue λ and the corresponding eigenvector x. In this chapter, we investigate different approaches to solve the eigenvalue problem using iterative refinement. We use a mixed precision approach to improve the performance of the algorithms while still achieving the required high accuracy. Our algorithms are derived from Newton's method, the origin of iterative refinement, which leads to non-constant linear systems needing to be solved for each eigenpair, consisting of the eigenvalue and eigenvector. The complexity of solving the resulting linear systems directly would be $O(n^4)$, which is too high for an efficient eigenvalue iterative refinement method. Therefore, we evaluate iterative linear solvers to reduce the complexity of solving the linear systems. Iterative matrix splitting methods and solvers for equilibrium problems are used to find an efficient solution for the continuously changing linear systems. Parts of the research presented in this chapter have been published in [PG13]. We consider additional approaches to reduce the high-complexity requirements and extend the analysis of the methods mentioned in the publication.

After summarising the related work on iterative refinement for eigenvalue problems in section 3.1, we derive our approach from Newton's method in section 3.3. The possibilities of solving the resulting systems for the correction term are analysed in section 3.4 using matrix splitting methods and we introduce a method using the Jacobi iteration. An alternative approach is to view the Jacobian matrix from Newton's method as an equilibrium system, which is the focus of section 3.5, leading to *mixed precision eigenvalue iterative refinement* methods. We discuss the number of operations for each mixed precision approach. section 3.6 concludes this chapter with numerical results from experiments on the behaviour of the iterative refinement methods introduced in this chapter.

3.1 Related Work on IR for Eigenvalues

Dongarra, Moler and Wilkinson [DMW83] describe a method for improving the numerical accuracy of eigenvalues and eigenvectors. The authors state that the algorithm is similar to Newton's method and can be used to improve approximate solutions. It also provides information on the numerical bounds of the eigenpairs. It is similar to iterative refinement for linear systems [Wil63] and improves eigenvalues and either improves or computes the corresponding eigenvectors.

The eigenvalue iterative refinement from [DMW83] is divided into two parts: the *pre-SICE* phase and the *SICE* phase. In the *pre-SICE* phase the matrix is factored using the Schur decomposition $A = QUQ^{-1}$, where U is an upper triangular matrix and Q is a unitary matrix. Due to U being similar to A, their eigenvalues are the same and because of the triangular nature of U, the eigenvalues are located on the diagonal of U. The *SICE* phase then uses the results from the Schur decomposition in combination with triangularisations using plane rotations to improve the approximate eigenvalues by iteratively solving a linear system for a correction term using the residual $r = \lambda x - Ax$ as the right hand-side. The algorithm is described in [DMW83] and a Fortran implementation can be found in [Don82].

3.2 Computational Cost of Existing Eigenvalue Iterative Refinement

In the LAPACK User's Guide [ABB*99], the LAPACK eigenvalue solver for general matrices `xGEEV` is described to have a floating-point operation count of $26.33n^3$ for computing the eigenvalues and the right eigenvectors. When computing only the eigenvalues, the flop count decreases to $10n^3$. As described in [Don82], the *pre-SICE* phase requires $10n^3 + 30n^2$ fused-multiply add operations, which corresponds well with the flop count of the LAPACK function for computing the eigenvalues only. The *SICE* phase requires $13n^2$ operations per iteration. The author states that an average of 3 iterations is needed to improve an eigenpair. Experiments have shown that while this is correct for small matrices with $n = 10$, the number of iterations required increases with the matrix size, for example, a matrix with $n = 1000$ requires on average 4.77 iterations to reach convergence.

The method described in [DMW83, Don82] computes the majority of operations in single precision and only a few operations use double precision to achieve the target accuracy of the eigenpairs. From $13n^2$ operations, n^2 operations are executed in double precision. To retrieve the error bounds of the improved eigenpairs, an almost complete additional run of the *SICE* phase is required, adding another $11n^2$ operations per eigenpair of which n^2 operations are again computed in double precision. To compute all eigenpairs to double precision accuracy, the total number of floating-point operations required by the eigenvalue iterative refinement presented in [DMW83], is

$$10n^3 + 13kn^3 + 30n^2 \approx (10 + 13k)n^3 \quad \text{operations,}$$

with k being the average number of iterations and kn^3 operations being executed using higher precision. Using the experimental observation $k \approx 5$, the number of operations would be $75n^3$. Estimating the performance difference between single and double precision to be a factor of 2, the algorithm would have

$$(10n^3 + 12kn^3 + 30n^2)/2 + kn^3 \approx (5 + 7k)n^3 \quad \text{double precision operations.}$$

Thus, for $k = 5$, $40n^3$ double precision floating-point operations are required for the entire process. This is higher than the flop count described for the LAPACK function. The algorithm can only improve one eigenvalue at a time, limiting the use of BLAS subroutines to level 2, which cannot exploit the benefits achieved with level 3 BLAS in terms of optimisation and memory access.

LAPACK also offers specialised functions for computing the eigenvalues of symmetric matrices. In this case, the LAPACK function `xSYEVD` requires $9n^3$ floating-point operations to compute the eigenvalues and eigenvectors and only $1.33n^3$ operations for computing the eigenvalues only.

3.3 Newton's Method for Iterative Refinement Eigensolver

As already discussed in section 2.1, iterative refinement is based on Newton's method for solving non-linear equations, which finds a root of a function $f(x)$ using the iterative process Equation 2.1 and solving a linear system of equations Equation 2.2 using the residual of the solution which is being improved. For the eigenvalue problems the residual can be expressed for each eigenpair as $Ax - \lambda x$. In [RÖ3], the function f is expanded by the additional condition $x^\top x - 1$ to normalise the eigenvector x. The correction term therefore consists of a correction Δx_k for the eigenvector and a correction $\Delta \lambda_k$ for the eigenvalue:

$$\Delta(x_k, \lambda_k) = \begin{pmatrix} \Delta x_k \\ \Delta \lambda_k \end{pmatrix} \quad .$$

For an eigenpair consisting of the eigenvector x and the eigenvalue λ the function is defined as follows

$$f(x, \lambda) = \begin{pmatrix} Ax - \lambda x \\ x^\top x - 1 \end{pmatrix} \quad .$$

$f(x, \lambda) = 0$ if and only if $Ax = \lambda x$ and $x^T x = 1$, which requires x to be normalised. The resulting Jacobian matrix, which is used for computing the correction term $\Delta(x_k, \lambda_k)$, is

$$J_f(x, \lambda) = \begin{pmatrix} A - \lambda I & -x \\ 2x^\top & 0 \end{pmatrix} \quad . \tag{3.1}$$

Alternatively to the function described in [RÖ3], a correction term $\Delta(x_k, \lambda_k)$ could be computed for $f(x, \lambda) = (Ax - \lambda x)$. This results in a rectangular Jacobian matrix

$$J_f(x, \lambda) = \begin{pmatrix} A - \lambda I & -x \end{pmatrix}$$

and can use other solvers to compute the correction term, for example a QR decomposition. However, this approach will not be considered in this thesis.

An alternative approach is the definition of $f(x, \lambda)$ as

$$f(x, \lambda) = \begin{pmatrix} Ax - \lambda x \\ -\frac{x^\top x - 1}{2} \end{pmatrix}$$

The additional factor -0.5 introduced to $x^T x - 1$ does not change the residual if x is the exact solution. Introducing this factor leads to a symmetric Jacobian matrix (if A is a symmetric matrix):

$$J_f(x, \lambda) = \begin{pmatrix} A - \lambda I & -x \\ -x^\top & 0 \end{pmatrix} \tag{3.2}$$

This leads to properties which can be exploited by special system solvers, as will be shown in section 3.5 for symmetric saddle point matrices.

The correction term $\Delta(x_k, \lambda_k)$ is found by solving the linear system

$$J_f(x_k, \lambda_k)\Delta(x_k, \lambda_k) = f(x_k, \lambda_k) \tag{3.3}$$

and the approximate solution from the previous iteration is then updated according to

$$\begin{pmatrix} x_{k+1} \\ \lambda_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ \lambda_k \end{pmatrix} + \Delta(x_k, \lambda_k) \quad .$$

The eigenvalue iterative refinement takes an approximate eigenvalue and a random eigenvector as its input and each eigenpair is refined separately. Any random vector x_0 can be used as an initialisation for the process. To significantly improve the rate of convergence, the eigenvector x_k should be normalised before constructing and solving the linear system. The convergence for close eigenvalues has not yet been investigated and could pose a problem if the process converged to the same eigenpair starting with marginally different approximate eigenvalues.

It is not possible to compute an improvement for all eigenpairs at the same time, because this would lead to the function $f(x, \lambda)$ being a matrix consisting of all independent residuals for each eigenpair and would lead to the Jacobian being a 3-dimensional hypermatrix. A straightforward way to solve such a hypermatrix would be to iterate through all n z-dimensions, which results in iterating through each eigenvalue independently. Each eigenpair can be improved independently and therefore allows for the workload to be distributed leading to a task parallelisation for up to n processes, not considering further distributed solvers for each eigenpair improvement.

3.3.1 Approaches Pursued in this Chapter

For each eigenpair the linear system Equation 3.3 has to be solved in each iteration because the improved eigenvalue and eigenvector are part of the function $f(x, \lambda)$ and its Jacobian. Therefore, the system of equations changes in each iteration. If the linear system was factorised, this would lead to a complexity of $O(n^3)$ for improving a single eigenpair, resulting in a complexity of $O(n^4)$ for improving all eigenpairs. The complexity is multiplied by the number of iterations required to reach a target accuracy.

Initial experiments have shown that it is sufficient to use a LU factorisation computed in a lower working precision once an approximate eigenvector is available. If an approximate eigenvalue is provided, the LU factorisation only needs to be performed once and the LU factorisation in the working precision can then be used to solve the systems in the subsequent iterations of this eigenvalue. However, if no approximate eigenvector is available, then the first random x_0 has to be refined before being able to reuse a computed LU factorisation. This method only reduces the preceding factor, but not the overall complexity of $O(n^4)$.

The complexity of factorizing the resulting linear systems is too high for an efficient eigenvalue iterative refinement method. Therefore, other solvers have been investigated to reduce the complexity of solving the linear systems. The Jacobian matrices Equation 3.1 and Equation 3.2 are saddle point matrices [Vav94, BGL05, GSU03], offering a wide range of solution methods. Iterative linear solvers and exploiting the properties of saddle point matrices are the focus of the following sections.

3.4 Approach 1: Matrix Splitting

To avoid the repetitive LU decomposition of the Jacobian matrices, which continuously change due to the eigenvectors and eigenvalues being improved in each iteration, a solution based on matrix splitting [Var59] is considered. The ideal solution would be a splitting into a constant part, which can either be computed once for all eigenvalues and all iterations or will only be used in matrix-vector operations, and a non-constant part, which is mainly comprised of the eigenvalue and eigenvector which are being improved and can easily be inverted. As seen in the previous section, the last element of the Jacobian matrix is 0. This has to be taken into consideration when splitting the matrix to ensure the resulting matrices do not become singular.

There are different splitting methods, the most general being a regular splitting.

$$A = B - C$$

Under certain conditions, the system $Ax = b$ can then be solved with the iterative method:

$$x_{k+1} = B^{-1}Cx_k + B^{-1}b, \quad k = 0, 1, 2, \ldots$$

However, the split matrices B and C have to fulfil special requirements in order for the iterative process to converge, as described in [Var59]. As usual, an arbitrary $n \times n$ matrix M, which has only non-negative entries, is denoted as $M \geq 0$. Analogously, all entries of M being positive is denoted as $M > 0$. Using this notation, $A = B - C$ is a *regular splitting* if and only if $B^{-1} \geq 0$ and $C \geq 0$. The convergence also depends on the spectral radius of $B^{-1}C$. The spectral radius $\rho(A)$ of a matrix $A \in \mathbb{C}^{n\times n}$ with eigenvalues $\lambda_1, \dots, \lambda_n$ is

$$\rho(A) = \max_{1\leq i\leq n} |\lambda_i| \quad .$$

Well known matrix splitting methods include the Jacobi method, the Gauss-Seidel method or the method of successive over-relaxation (SOR). In these methods, the matrix A is split into three parts

$$A = D - L - U$$

with D being the diagonal of A, U and L being the strictly upper and lower triangular matrices of A. The Jacobi method sets B as the easily invertible D and $C = U + L$. The Gauss-Seidel method uses $D - L$ as B and sets $C = U$. SOR is a variation of the Gauss-Seidel method and introduces a relaxation factor ω to improve the rate of convergence with $B = D - \omega L$ and $C = (1 - \omega)D + \omega U$. For these special matrix splittings, $\rho(B^{-1}C) < 1$ is sufficient for convergence.

One possible regular splitting would be to keep only the system matrix A in the first split matrix B and use the non-static components, the eigenvalue and the eigenvector, as the matrix C, i. e.

$$\begin{pmatrix} A - \lambda_k I & -x_k \\ -x_k^\top & 0 \end{pmatrix} = \begin{pmatrix} A & 0 \\ 0^\top & \mu \end{pmatrix} - \begin{pmatrix} \lambda_k I & x_k \\ x_k^\top & \mu \end{pmatrix} \tag{3.4}$$

μ is introduced to avoid the split matrices becoming singular. B has to be inverted in the iterative method, but due to its static nature, this inversion can be performed for all eigenvalues and eigenvectors before starting the iterative process. Even though the matrix A only has to be inverted once, a better construction of B can be found which can be inverted more easily and has a lower operations count.

A Jacobi matrix splitting of the Jacobian matrix is

$$\begin{pmatrix} A - \lambda_k I & -x_k \\ -x_k^\top & 0 \end{pmatrix} = \begin{pmatrix} D - \lambda_k \mathbf{I} & -x_k \\ -x_k^\top & \mu \end{pmatrix} - \begin{pmatrix} L + U & 0 \\ 0^\top & \mu \end{pmatrix} \tag{3.5}$$

The first matrix includes mainly the non-constant factors of the Jacobian matrix aside from the diagonal elements of the system matrix A. Furthermore, the aim of constructing a matrix, which can easily be inverted, has also been achieved. The second matrix consists only of the strict upper and lower matrices of A and the factor μ.

The first splitting in Equation 3.4 requires $2/3n^3$ operations for a LU decomposition of the system matrix A before entering the iterative process. In each iteration two systems have to be solved with forward and back substitution each costing n^2 operations per iteration. The matrix-vector product per iteration is very cheap due to the extremely sparse matrix C and only requires $3n$ operations. To improve all eigenpairs the total number of operations is therefore

$$\frac{2}{3}n^3 + k_s(2n^2 + 3n)n \quad \text{operations}$$

k_s denotes the sum of all inner iterations required by the iterative method using the split matrices for all iterative refinement iterations. The second approach using a Jacobi splitting in Equation 3.5 does not require a precomputed matrix decomposition. In each iteration the matrix B is decomposed using the following special form of the LU decomposition:

$$\begin{pmatrix} D-\lambda_k I & -x_k \\ -x_k^\top & \mu \end{pmatrix} = \begin{pmatrix} I & 0 \\ \frac{-x_{k,j}}{d_{jj}-\lambda_k} & 1 \end{pmatrix} \begin{pmatrix} D-\lambda_k I & -x_k \\ 0^\top & \mu - \sum_{j=1..n} \frac{x_{k,j}^2}{d_{jj}-\lambda_k} \end{pmatrix}$$

The decomposition requires only $2n$ operations for the last row of L and the last element of U, the forward substitution using L requires n operations and $2n$ operations are used computing the back substitution with U. The total number of operations for solving a system is therefore $5n$ operations. Each iteration also computes a matrix-vector product using C requiring n^2 operations. The improvement of all eigenpairs using this approach results it the total number of operations being

$$k_s(5n + n^2)n \quad \text{operations}$$

This is a significant reduction compared to the first splitting approach, eliminating the need for a precomputed decomposition and additionally reducing the n^2 factor for each iteration.

The main problem with the splitting methods is the convergence due to the spectral radius in almost all cases not being less than 1 and therefore not converging to the eigenvalues and eigenvectors. The Jacobi splitting always converges to the maximum absolute eigenvalue. Exploiting this behaviour by removing the improved eigenpair from the matrix A using a rank 1 update results in the iterative process converging to the next absolute largest eigenvalue.

$$A_{k+1} = A_k - \lambda_i x_i x_i^\top$$

A disadvantage of this method is the limitation of improving the eigenvalues in the order of their absolute value.

3.5 Approach 2: Saddle Point Problems

Saddle point systems [Vav94, BGL05, GSU03], also called equilibrium systems, occur in many different fields, for example in electrical networks or finite element methods. For reasons of simplicity, the notation used to describe the saddle point problems does not correspond with the previous equations. Saddle point matrices have the following special structure:

$$\begin{pmatrix} A & B_1 \\ B_2^\top & C \end{pmatrix}$$

These properties can be exploited when solving such systems. There are direct and iterative methods to solve systems of equations which operate on equilibrium problems of the form

$$\begin{pmatrix} A & B \\ B^\top & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}$$

A direct method is the Range Space Method [GSU03], which assumes A being symmetric, $B_1 = B_2$ and $C = 0$

$$\begin{aligned} x &= A^{-1}(b - By) \\ B^\top A^{-1} By &= B^\top A^{-1} b - c \end{aligned}$$

The Jacobian matrix in section 3.3 can be viewed as a saddle point matrix and the Jacobian in Equation 3.2 is an example for a symmetric saddle point matrix with $B_1 = B_2$ being the negative eigenvector. Applying the range space method to the iterative refinement solves the eigenvalue problem. This does not yet reduce the complexity of the iterative refinement method because multiple linear systems have to be solved for the range space method. Compared to the matrix splitting techniques, the range space method only requires the solution to the constantly changing linear systems of $A - \lambda I$ and no longer to the system expanded by the eigenvectors. A solver for the shifted linear system $A - \lambda I$ is required.

A possible approach to solving the resulting linear system of equations would be the use of iterative solvers in combination with a suitable and preferably cheap preconditioner to improve the convergence rate.

3.5.1 Schur factorisation

The initial approximation for the eigenvalues can be computed using the Schur factorisation $A = QUQ^*$. The Schur factors can then be used to solve the linear systems by applying the shift to U and inverting $U - \lambda I$. This removes the necessity of a decomposition in each iteration and reduces the complexity for improving an eigenpair to $O(n^2)$. In the case of symmetric system matrices, as required by the range space method, the Schur factor U would be a diagonal matrix and the shifted system could be inverted at a very low cost of n operations.

One problem still remains: when computing $U - \lambda I$, the eigenvalue is subtracted from the diagonal of U. U and the eigenvalue λ have the same precision and in the first refinement step both values are identical as the approximation for the eigenvalue originates from the diagonal elements of U. This results in one element of the diagonal of U becoming 0 and therefore U becoming singular, causing the inversion to fail. This also occurs in subsequent iterations due to the improved eigenvalue being used in the same lower working precision as the Schur factors and the eigenvalues on the diagonal of U already being accurate to the lower working precision. To overcome this singularity problem, a small correction δ has to be introduced when subtracting the eigenvalue from the diagonal of U to ensure non-singularity. The choice of the magnitude of the correction is analysed in section 3.6.

Acquiring the initial approximate eigenvalues through the Schur decomposition requires $9n^3$ operations [GVL13]. For each eigenpair, computing the residual costs n^2 operations. The range space method requires the solution of three linear systems using the already available Schur factors, each solution therefore consisting of two matrix-vector operations $(2n^2)$ and a back-substitution $(n^2/2)$ to invert $U - \lambda I$. The total number of operations is $8.5n^2$ in each iteration for each eigenpair.

$$9n^3 + 8.5kn^3 \quad \text{operations}$$

Taking into account the mixed precision computation with a factor of 2 for single precision, only n^2 operations per iteration would be computed in double precision and the total number of operations would be further reduced to

$$(9n^3 + 7.5kn^3)/2 + kn^3 \approx (4.5 + 4.75k)n^3 \quad \text{double precision operations.}$$

In the case of A being symmetric, the number of operations is reduced because the Schur factor U is a diagonal matrix and only requires n multiplications, leading to $9n^3 + 7kn^3$ operations and using single precision to $9n^3 + 4kn^3$ operations. The new method has a lower complexity than the algorithm in [Don82] with $(5 + 7k)n^3$ operations as shown in section 3.1.

The unitary matrix Q from the Schur factorisation can be used as the initial approximation of the eigenvectors, but random values can also be used instead although it will increase the number of iterations required until convergence. This behaviour will be shown in section 3.6.

3.5.2 Householder tridiagonalisation

Another factorisation of a symmetric matrix A is the Householder tridiagonalisation $A = QTQ^\top$, with T being tridiagonal and Q the product of the Householder transformations. As described previously for the Schur decomposition, the shifted linear systems can be solved analogously by applying the shift to T, again resulting in the reduced complexity of $O(n^2)$ for each improved eigenpair.

The approximate eigenvalues are obtained computing the Householder tridiagonalisation in $4n^3/3$ operations followed by the Pan-Walker-Kahan QR algorithm with a complexity of $O(n^2)$ [GVL13]. The product Q of the Householder transformations are needed explicitly for solving the shifted linear systems, which requires $4n^3/3$ operations. The three linear systems solved for the range space method, consist of two matrix-vector operations $(2n^2)$ and a bidiagonalisation to solve the tridiagonal shifted system $T - \lambda I$ with a complexity of $O(n)$.

$$\frac{8}{3}n^3 + 7kn^3 \quad \text{operations}$$

Applying mixed precision, only the computation of the residual requires double precision with n^2 operations, reducing the total number of operations to

$$(\frac{8}{3}n^3 + 6kn^3)/2 + kn^3 \approx (\frac{4}{3} + 4k)n^3 \quad \text{double precision operations.}$$

3.6 Experimental Evaluation

In this section, we compare the eigenvalue iterative refinement by Dongarra, Moler and Wilkinson (SICEDR [Don82]), the Jacobi splitting Equation 3.5 for sorted eigenvalues (JSSIR) and the saddle point problems with Schur factorisation (SPSIR) (subsection 3.5.1) and Householder tridiagonalisation (SPHIR) (subsection 3.5.2). Almost all experiments were conducted using Matlab 2010a with the exception of SICEDR which was implemented in C. The experiments summarised in this section focus on the number of iterations required for convergence and the accuracy of the results, which is compared based on the relative residual

$$r_{rel} = \frac{\|Ax - \lambda x\|_1}{\|A\|_1 \, \|A\|_1} \quad . \tag{3.6}$$

The iterative process terminates if the correction term is less than a defined threshold ϵ

$$\|\Delta(x, \lambda)\|_\infty < \epsilon$$

and a predefined maximum number of iterations is set as an additional termination criterion. For our experiments random, symmetric matrices are used, $\epsilon = 10^{-12}$ and the maximum number of iterations is set to 20.

3.6.1 Number of iterations and convergence rate

Figure 3.1 shows the average number of iterations per eigenpair for different methods. The number of iterations increases with the system size n. The comparison cannot be limited to the number of iterations and also has to include the total floating-point operations. Table 3.1 shows the double precision operations for the mixed precision methods for $n = 500$ based on the operations count described in the previous sections. The average iteration count k for SPSIR initialised with Q is slightly higher compared to SICEDR, on average about one more iteration is needed, but the operations count is lower.

In Figure 3.2, the convergence history for the first eigenvalue λ_0 of a matrix with $n = 1000$ is shown for all methods using the relative residual Equation 3.6. The Jacobi splitting achieves the best result, but uses an iterative solver in each iteration, in this case using $\{24, 3, 1\}$ inner iterations for the corresponding outer iteration number. SPSIR initialised with the Schur factor Q also only requires 3 iterations, achieving almost the same accuracy as the Jacobi

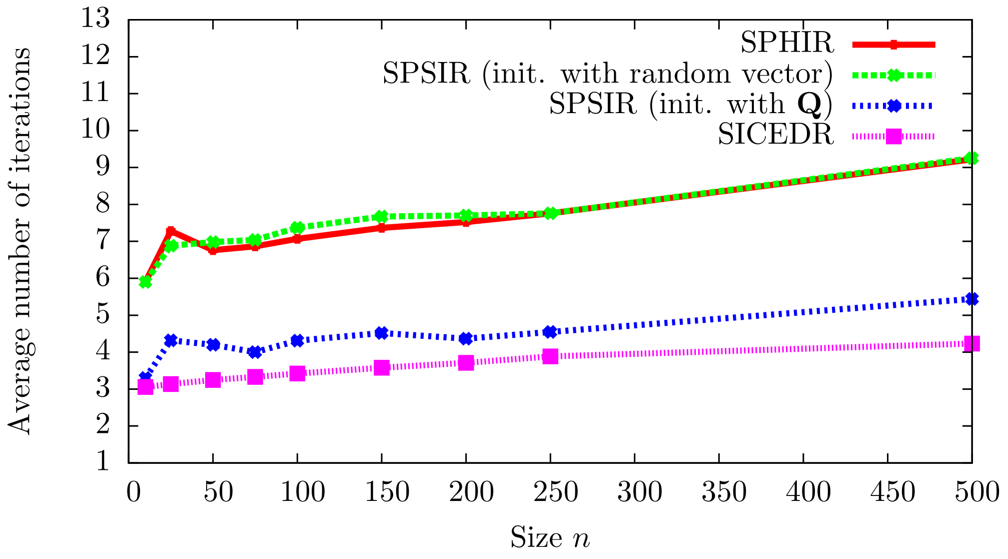


Figure 3.1: Average number of iterations for different system sizes n

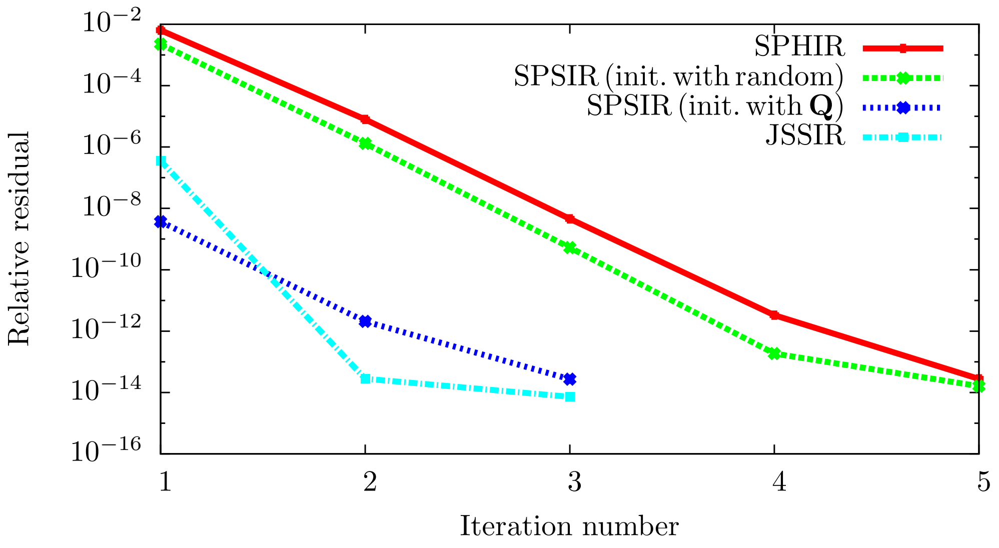


Figure 3.2: Comparison of the convergence history for the eigenpair of λ_0 of a symmetric matrix with $n = 1000$

Table 3.1: Comparison of the double precision operations for $n = 500$ and the average number of iterations required to reach the termination criteria.

Method	**FLOPs**	**Average k**	**FLOPs(k)**
SICEDR	$(5 + 7k)n^3$	4.23	$34.61n^3$
SPSIR	$(4.5 + 4.75k)n^3$	5.44	$30.34n^3$
SPHIR	$(\frac{4}{3} + 4k)n^3$	9.22	$38.21n^3$

splitting. Using a random vector as the first approximation, leads to a higher iteration count, achieving the target precision in 5 iterations. SPHIR does not have an approximation for the eigenvectors and therefore starts with a random vector, also converging in 5 iterations.

3.6.2 Correction for shifted linear systems

As described in section 3.5, a correction δ has to be introduced when solving the linear systems shifted by the eigenvalue to avoid the inversion of a singular system. δ has to be chosen relative to the magnitude of the current eigenvalue. The magnitude is defined as a constant factor γ.

$$\delta = 10^{\lfloor \log_{10} |\lambda_i| \rfloor + \gamma}$$

The correction determines which digit of the eigenvalue is changed to ensure the non-singularity of the shifted linear system and the convergence of the iterative refinement.

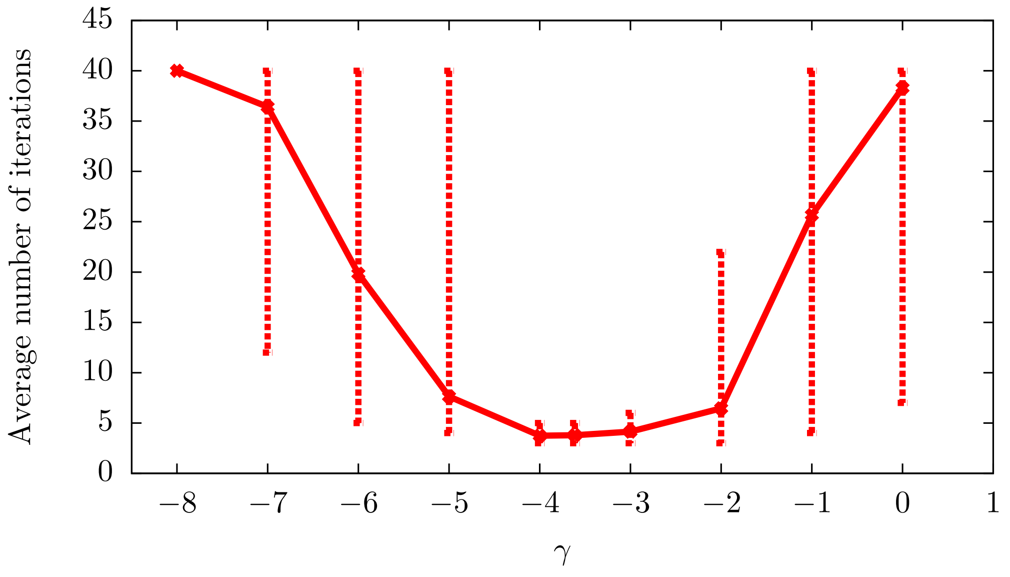


Figure 3.3: Analysis of the magnitude of the correction added to the shifted linear system for SPSIR initialised with Q and its influence on the convergence and the average number of iterations.

Figure 3.3 shows the influence of γ on the convergence of SPSIR initialised with Q for a symmetric matrix of size $n = 50$. γ is plotted on the x-axis and the average number

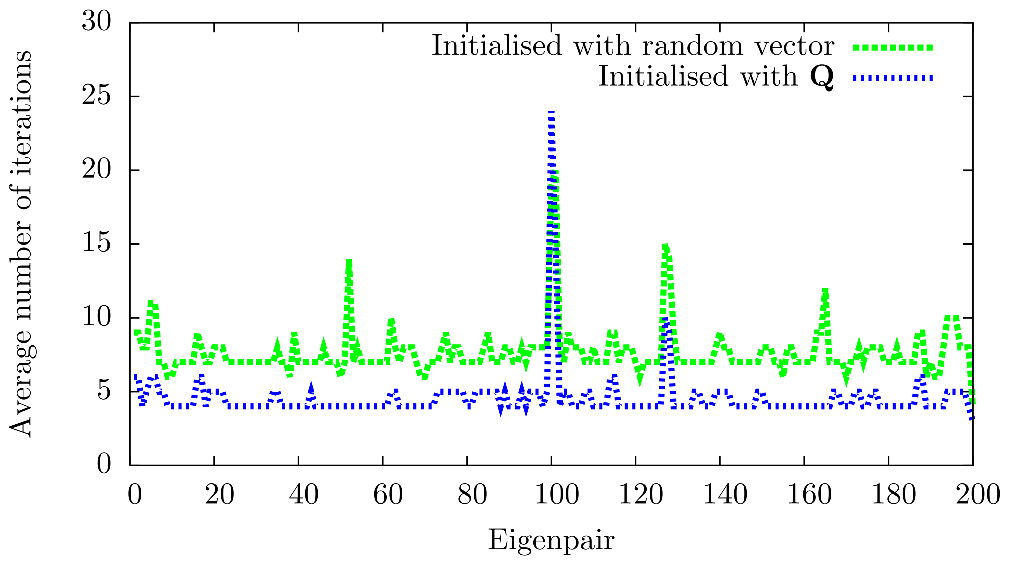


Figure 3.4: Comparison of different initial eigenvector approximations for SPSIR

of iterations on the y-axis. The vertical lines show the minimum and maximum number of iterations required by the algorithm for each γ. The range of γ is chosen to cover the range of single precision, the working precision of the algorithm. The method converges for all eigenpairs if $\gamma = \{-2, -3, -4\}$ and as expected does not converge for most eigenvalues if the correction is close to single precision. The lowest average number of iterations is achieved if $\gamma = -4$ and if γ is half of the representable number of digits of single precision ($\gamma = -7.23/2 \approx -3.61$). Therefore, the latter is being used in all experiments presented in this section.

3.6.3 Initial eigenvectors

As mentioned at the end of subsection 3.5.1, the Schur factor Q can be used as the initial approximation of the eigenvectors. Figure 3.4 shows the average number of iterations required to converge each eigenpair due to the initial eigenvector approximations, either using the vectors from the Schur factor Q or random vectors. As already seen in Figure 3.1 and Figure 3.2, random vectors require more refinement steps to reach convergence, but the acquired accuracy is almost the same regardless of the input data for the initial eigenvectors. In this example, for a symmetric matrix $n = 200$ an average of 4.57 iterations are performed for Q as the initial data. Using random input data, the average number of iterations increases to 7.69.

3.7 Conclusion

Iterative refinement for eigensolvers was derived from Newton's method and solutions based on matrix splitting methods and equilibrium problems have been investigated. The Jacobi splitting can be used to retrieve the eigenvalues and eigenvectors in the order of their absolute value by removing the already found eigenpairs from the original system.

New approaches for mixed precision eigenvalue iterative refinement have been presented based on the solution of saddle point problems. The range space method is used in combination with the Schur factorisation and Householder tridiagonalisation to continuously solve the resulting non-constant linear systems. It has been shown that the number of floating-point operations is lower than the previously described iterative improvement method by Dongarra, Moler and Wilkinson [Don82], even though the number of iterations is higher. Another advantage of the eigenvalue iterative refinement methods is the freedom of being able to refine a single eigenpair independently from the other eigenpairs.

The convergence and the numerical error analysis of the presented methods is left as a task for future research. Another future topic of interest would be the behaviour for non-symmetric system matrices with imaginary eigenvalues. Saddle point solvers other than the range space method should also be investigated.

As shown in this chapter, the eigenvalue problem was reformulated to solve a linear system, the original problem considered by iterative refinement. In the following chapter, we will therefore focus on the possibilities that arise when using arbitrary precision in the iterative refinement process based on linear systems. Later, we will use our newly gained knowledge in Chapters 6 and 7 and apply it to least squares problems, which can also be described as the solution of a linear system.

Chapter 4

Arbitrary Precision Iterative Refinement

As discussed in Chapter 2, mixed precision iterative refinement (MPIR) is a special case of IR, where the majority of operations are performed in a lower precision, usually IEEE single precision (SP), and a higher precision, usually IEEE double precision (DP), is only used for a few low complexity operations critical to the accuracy of the solution.

As long as the linear system is not too ill-conditioned, MPIR reaches the same or higher accuracy than a DP direct solver while achieving a performance benefit by predominantly operating at the lower precision. This is particularly attractive on chip architectures where lower precision operations exhibit significant performance benefits over higher precision operations. On CPUs, the peak performance of SP operations is about twice as high as for DP operations. However, on general-purpose GPU architectures, the speedup of SP over DP operations is normally much higher. For example, the NVidia GeForce GTX 1080 Ti and RTX 2080 Ti have a DP:SP ratio of 1:32 [NVI19], having 1 FP64 CUDA core for every 32 FP32 CUDA cores. Server-grade GPUs like the NVidia Tesla V100 [NVI18, NVI19] exist which have the same DP:SP ratio as CPUs (1:2), but they come at a significantly higher cost than their general-purpose GPU counterparts.

In this chapter, we focus on a generalisation of MPIR, *arbitrary precision iterative refinement* (*APIR*), for solving dense linear systems based on LU factorisation. In APIR, the two precision levels involved are not restricted to SP and DP, but flexible. More specifically, we distinguish the *target precision* α and the *working precision* β. Their exponent length in the floating-point representation will be referred to by e_α or e_β, and their mantissa length by p_α or p_β. We investigate the potential performance benefits of APIR over standard SP/DP MPIR. For this purpose, the effects of the parameters α and β on convergence behaviour and performance of IR as well as the relationship between them are studied.

Reconfigurable hardware provides the only platform which supports hardware implementation of floating-point units (FPUs) for arbitrary precision floating-point (FP) formats without a restriction on the system word width. We model the benefits of APIR on field programmable gate arrays (FPGAs). High-end FPGAs can achieve the theoretical DP peak performance of CPUs [Alt10, SSW10, LV08, Alt13, Int17b, FL16, NDI18]. FPUs for smaller number formats (smaller bit-widths) typically require fewer hardware resources and can run at higher clock frequencies [SPS08, MLG11]. We derive a performance model of APIR taking into account both arithmetic precisions in the floating-point representation and the system's condition number and size and argue that FPGAs are a suitable and efficient platform to implement APIR. We will show that no prior work presented a coherent performance model of APIR on FPGAs.

After reviewing the related work on FPGA implementations of IR in section 4.1, we discuss the algorithmic foundation of APIR in section 4.2. We use the model described in section 2.4 for the number of iterations required by IR methods and expand it to two precision levels. Further, we discuss the convergence and error analysis for APIR. section 4.3 focuses on our performance model to predict the performance benefits of using lower working precisions in APIR on FPGAs. Finally, we show experimental results in section 4.4 using a software implementation of APIR based on the arbitrary precision GNU MPFR library [FHL*07].

4.1 Related Work

In this section, we summarise the existing literature on FPGA implementations for IR. The related work about IR for linear systems (and other solvers) has already been discussed in Chapter 2. Here, we will give a short recap of the most relevant iterative refinement methods and categorise them into accuracy- and performance-oriented algorithms. We also consider the precisions used by each method.

4.1.1 Iterative Refinement

In this chapter, we focus on LU-based solvers for dense linear systems. Existing IR methods can be categorised into (i) approaches which focus on achieving high accuracy solutions, and (ii) approaches which gain a performance benefit by using working precisions β below the target precision α.

(i) Accuracy-oriented: The standard approach improves the accuracy of the initial solution using the *same* precision level ($\alpha = \beta$), for example, in [OOR09] for ill-conditioned linear systems. We call this method *standard iterative refinement* (*SIR*). Moler [Mol67] expands the initial IR analysis performed by Wilkinson [Wil63], which used fixed-point arithmetic, to cover floating-point (FP) arithmetic. The use of FP arithmetic especially affects the calculation of the residual, which is required to improve the initial solution. The analysis in [Mol67] is not limited to specific FP precisions but provides results for an arbitrary number of bits in the mantissa. To improve the accuracy for ill-conditioned systems, *extra precise IR* (*EPIR*) [DHK*06] uses a higher working precision ($\beta = 2\alpha$) to calculate the residual and the correction term of the solution. In [DHK*06], reliable error bounds for the solution of IR are

derived which can be evaluated with low overhead. The use of extra precision is triggered by the rate of convergence of the IR process. *Binary cascade IR* (BCIR) [Kie81] uses *multiple* working precisions without any restrictions on the FP format (mostly larger than DP). This sequence of working precisions is determined a priori based on the input parameters (problem size n, condition number κ and target precision p_α).

(ii) Performance-oriented: The prime example for performance-oriented IR methods is *mixed precision IR* (*MPIR*), which uses β = SP and $\alpha = 2\beta$ = DP in order to exploit performance benefits of SP over DP on existing chip architectures for both dense and sparse systems [LLL*06, BDL*07, KD07, BDK*08, BBD*09, HS10]. In [BDL*07], a generic forward and backward error analysis is given for MPIR (independent of the specific number formats used), complementing bounds given by Higham [Hig02] for IR using a single precision level. The system's condition number $\kappa(A)$ is identified as a limiting factor for IR, as we will discuss in subsection 4.2.1. Implementations of MPIR are available in LAPACK for the LU and Cholesky factorisation.

In [HWTD17, HAZ*18, HTDH18], the authors investigate IR in combination with GMRES using β = HP (half precision) on an NVIDIA V100 for the initial factorisation and demonstrate the power efficiency of using hardware supported reduced precision arithmetic and tensor cores. In [HPZ19], an algorithm was developed to deal with the limited range of half-precision arithmetic. A higher precision matrix is diagonally scaled and multiplied by a scalar before being rounded to half-precision to maximise the use of the reduced number format. The authors demonstrate an improved convergence of GMRES compared to directly rounding the input matrix to half-precision.

4.1.2 FPGA Implementations

There has been growing interest in investigating concrete implementations of arbitrary precision algorithms since FPGAs are becoming more competitive in terms of FP performance. In particular, FPGA architectures have been suggested for LU decomposition [GCP*04, ZP06, SJZW09] and for complete LU-based linear solvers (including pivoting, forward and back substitution) [WDL*09]. However, most of these papers investigate only one precision level on the FPGA, usually SP or DP.

Sun et al. [SPS08] implemented MPIR on a hybrid system where the higher precision (DP) operations are performed on a CPU while lower precision operations (LU decomposition) are performed on an FPGA. In this paper, simulations with six different number formats were conducted. The authors report empirical observations on the number of iterations required to converge for various configurations and system sizes, but do not investigate this relationship further. We note that the number formats' exponent fields were kept constant ($e_\alpha = e_\beta = 11$ bits) when investigating the required number of iterations, while the number formats implemented in hardware use non-uniform exponent fields (11, 8, 7 bits).

eXtended MPIR (XMIR) [LP11] is an IR implementation on FPGAs and is targeted at achieving arbitrarily high accuracies with $\alpha \geq DP$. For the working precision, only a few mantissa sizes equal or above SP were investigated by the authors and working precisions lower than SP were not considered.

4.2 Arbitrary Precision Iterative Refinement

In this section, we review the algorithmic basics of IR for dense linear systems, summarise its properties and the error analysis available in the literature. We then consider the generalisation of the MPIR concept to arbitrary precision levels, resulting in the *arbitrary precision IR (APIR)*.

4.2.1 The Evolution of APIR from SIR and MPIR

The steps for all three algorithms (SIR, MPIR and APIR) are identical, the only difference being the precisions they use. Given a linear system $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ and $b, x \in \mathbb{R}^n$, an approximate initial solution is computed using an LU decomposition of A. Subsequently the IR algorithm increases the accuracy of the solution by using the residual r as the right-hand side to solve the linear system for the correction term Δx. Finally, the correction term is added to the solution vector x to improve the result of the linear system. The residual r is recomputed using the updated x and the termination criterion is checked. The refinement continues until the norm of the residual $\|r\|_2$ reaches the threshold $\tau_r := n\,\kappa(A)\,2^{-p_\alpha}$ [Wil63], where $\kappa(A)$ is the condition number of A. If the process converges too slowly or not at all, the algorithm will terminate after reaching a maximum number of iterations i_{max}. The number of iterations required for convergence directly relates to $\kappa(A)$, as we will see in subsection 4.2.2. The steps of the algorithm are shown in Algorithm 1.

The cost of IR is very low because the additional operations required by IR have a complexity of $O(n^2)$ whereas the LU factorisation has $O(n^3)$. The process also uses the already computed factors L and U to solve the second system for Δx in the iterative process (line 5 in Algorithm 1).

SIR uses the same precision for all computing steps ($\beta = \alpha = \mathrm{DP}$). MPIR computes the majority of the operations, i. e. the LU decomposition (line 1 in Algorithm 1) and solving the linear systems (lines 2 and 5), using the lower working precision $\beta = \mathrm{SP}$. Only the critical operations, computing the residual (lines 3 and 7) and updating the solution (line 6), are performed in the higher target accuracy $\alpha = \mathrm{DP}$.

APIR is a generalisation of the standard SP/DP MPIR approach and can use any working precision below (or above) the standard IEEE 754-2008 [IEE08] FP formats to reach the target accuracy α. Naturally, α is also not restricted by the standard FP formats, but in this chapter we will focus on $\alpha = \mathrm{DP}$ and the performance benefits that can be gained with $\beta < SP$[1].

[1]We define the relation "<" of two ordered pairs (e_i, p_i) as follows:

$$(e_1, p_1) < (e_2, p_2) \text{ iff } \forall q_{i=\{1,2\}} \in \{e_i, p_i\} : q_1 \leq q_2 \text{ and } \exists\, q_1 < q_2$$

As stated in [BDK*08], MPIR still achieves the same and often higher accuracy than a DP direct solver (without IR), as long as the system is not too ill-conditioned. The lower working precision β, more precisely e_β, imposes a natural limit on $\kappa(A)$. As described by Higham [Hig02], IR will converge if $\psi(n)\kappa(A)\varepsilon_\beta < 1$, where $\psi(n)$ is a small function of n and ε_β the machine precision for β. This inequality defines a lower limit on the choice of p_β based on the input parameters.

Algorithm 1 Arbitrary Precision Iterative Refinement (APIR)

```
Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$
 1: $[L, U] \leftarrow \texttt{lu}(A)$                  ▷ factorise in $\beta$
 2: Solve $LUx = b$                                      ▷ solve in $\beta$
 3: $r \leftarrow Ax - b$                                ▷ compute residual in $\alpha$
 4: for $i = 0 : i_{max}$ do
 5:     $\Delta x \leftarrow$ solve $LU\Delta x = r$     ▷ solve in $\beta$
 6:     $x \leftarrow x + \Delta x$                      ▷ update $x$ in $\alpha$
 7:     $r \leftarrow Ax - b$                            ▷ compute residual in $\alpha$
 8:     if $\|r\|_2 < \tau_r$ then
 9:         break $\rightarrow$ converged
10:     end if
11: end for
```

4.2.2 Extending the Number of Iterations Model to Arbitrary Precision

In order to model the influence of the mantissa widths on the convergence of IR we need the number of iterations until convergence as a function of the mantissa widths p_α and p_β.

The model to estimate the number of iterations required by IR from section 2.4 can be expanded to cover arbitrary precision by setting p in the numerator to the base-b target precision and in the denominator to the base-b working precision:

$$i_{\text{conv}}(p_\alpha, p_\beta, \kappa) \approx \frac{p_\alpha}{p_\beta - log_b(\kappa)} \tag{4.1}$$

In base-2 arithmetic, this model exhibits two drawbacks: the size of the system is not accounted for and the influence of the condition number is too large. Therefore, we heuristically improved the model specifically for base-2 arithmetic based on experimental data. The resulting model may not be applicable to other bases. The influence of the term $log_2(\kappa)$ is dampened by applying the square root. To account for the system size, κ is multiplied by n. Further heuristic experiments lead to the conclusion that an additional factor of 2 had to be added to increase the reliability of predicting the required number of iterations, which results in the final model:

$$i_{\text{conv}}(p_\alpha, p_\beta, n, \kappa) \approx \frac{p_\alpha}{p_\beta - 2\sqrt{\log_2(n\kappa)}} \quad . \tag{4.2}$$

This model provides good estimates for the number of iterations required by IR for different system sizes and condition numbers, as we will see in subsection 4.4.2. The difference between the experimental data and the model is very low.

4.2.3 Convergence and Error Analysis

The correction term in iteration i, Δx_i, is found by solving

$$A\Delta x_i = A(x_{i+1} - x_i) = b - (b - r_i) = r_i \quad .$$

Further, as shown in [Dat10], IR can produce a better approximation than the previous intermediate solution x_i, since

$$Ax_{i+1} = A\left(x_i + \Delta x_i\right) = Ax_i + A\Delta x_i = (b - r_i) + r_i = b \quad .$$

The convergence of SIR is described in [Wil65] for Gaussian elimination with partial pivoting based on the following factor:

$$\Delta = n\,2^{-p}\left\|A^{-1}\right\|_\infty \quad .$$

If $\Delta < 2^{-p}$ then the number of correct binary digits of the solution will increase by at least p digits per iteration and the residual will decrease by a factor of 2^p or more. The method will normally not converge if $\Delta > 1/2$.

Error analysis for SP/DP MPIR is discussed by Higham [Hig02] and Buttari [BDL*07]. As shown by Sun [SPS08] the error analysis can easily be extended to arbitrary precision. The backward stability of the two steps performed in the higher target precision α, computing the residual and adding the correction term, can be described by the classical error bounds:

$$r_k = fl(b - Ax) \equiv b - Ax_k + e_k$$
$$\text{where } \|e_k\|_2 \leq \varphi_1(n)\varepsilon_\alpha(\|A\|_2\,\|x_k\|_2 + \|b\|_2)$$

$$x_{k+1} = fl(x_k + d_k) \equiv x_k + d_k + f_k$$
$$\text{where } \|f_k\|_2 \leq \varphi_2(n)\varepsilon_\alpha(\|x_k\|_2 + \|d_k\|_2)$$

with ϕ_1 and ϕ_2 being small functions of n. Regarding the backward error analysis of IR, the following relation holds:

$$\frac{\|b - Ax_{k+1}\|_2}{\|A\|_2\,\|x_{k+1}\|_2} \leq \eta\frac{\|b - Ax_k\|_2}{\|A\|_2\,\|x_k\|_2} + \theta$$

with η and θ being defined as

$$\eta = \psi(n)\kappa(A)\varepsilon_\beta \text{ and } \theta = \rho(n)\varepsilon_\alpha$$

where $\psi(n)$ and $\rho(n)$ are small functions of n. η describes the convergence rate and θ the limiting accuracy of the IR method. This leads to the following condition at convergence

$$\lim_{k\to\infty} \frac{\|b - Ax_k\|_2}{\|A\|_2 \, \|x_k\|_2} = \theta(1-\eta)^{-1} = \frac{\rho(n)}{1 - \psi(n)\kappa(A)\varepsilon_\beta}\varepsilon_\alpha$$

showing the norm-wise backward stability of the method as long as the matrix A is not too ill-conditioned, satisfying $\psi(n)\kappa(A)\varepsilon_\beta < 1$.

4.3 A Performance Model for APIR

In this section, we investigate the sources of improved performance on CPUs and FPGAs when using FP number formats with lower precision than DP.

The IEEE standard for FP arithmetic [IEE08] defines a generic binary representation for FP numbers comprising sign (1 bit), exponent (e bits), and mantissa ($p-1$ bits). The standard explicitly lists four binary representations of FP numbers: binary16 or "half precision" (HP, $p = 11$, $e = 5$), binary32 or "single precision" (SP, $p = 24$, $e = 8$), binary64 or "double precision" (DP, $p = 53$, $e = 11$) and binary128 or "quad precision" ($p = 113$, $e = 15$). In addition to these predefined format implementations, the standard defines ([IEE08] section 3.7, p.14) the *extendable precision format* as "a format with a precision and range that are defined under user control", giving full control over the precision and exponent while enforcing the same rules on special values and operations as for all other formats. The FP formats discussed in this chapter are fully compliant with the IEEE standard.

4.3.1 Performance Metrics for Floating-Point Computations

In order to quantify the performance benefits of arbitrary precision arithmetic, we first review the relevant metrics for the performance of FP computations on CPUs and FPGAs.

4.3.1.1 (Theoretical) Peak Performance

The *theoretical peak performance* $P(F)$ is the theoretical maximum (upper bound) of the achievable number of FP operations per unit time on a given *fixed* computing architecture. In this chapter, we investigate the peak performance as a function of the FP number format F. It is calculated from $N_{Op}(F)$, the maximum number of (FP) operations which can be issued in parallel per clock cycle for the given number format F, and the clock frequency f:

$$P(F) = N_{Op}(F) \cdot f \quad \text{[flop/s]} \quad .$$

4.3.1.2 Area-Time Product

The *area-time product* $C_{<Op>}(F)$ is a measure for the complexity of some functionality Op implemented in digital logic. It is computed by multiplying the chip area A in mm^2 required to implement some fixed operation Op with the time T in seconds it takes to perform the operation.

$$C_{<Op>}(F) = A(F) \cdot T(F) \quad [\mathrm{mm}^2\mathrm{s}]. \tag{4.3}$$

This measure can only be applied to integrated circuits where the used chip area is a direct (and known) function of the implemented design. Note that C, A and T all depend on the number format F and on the operation Op. On FPGAs, the chip area $A(F)$ used can be approximated by counting the number of hardware resources used, e. g. look-up tables (LUTs), arithmetic units (DSP blocks), etc. As the required interconnect is typically not included in the reported hardware resources, this is only a rough approximation. Care has to be taken in comparing and combining counts of different functional blocks as the functional density of reconfigurable LUTs and non-reconfigurable DSP blocks varies significantly.

4.3.1.3 (Theoretical) Peak Area Throughput

The (theoretical) *peak area throughput* $P_{<Op>,R}(F)$ is a measure for the theoretical maximum number of FP operations achievable on a *configurable* computing platform, given a specific implementation of the functionality $< Op >$, a certain amount of available hardware resources R and a certain number format F. The theoretical peak area throughput describes an upper bound on the performance similar to the theoretical peak performance on static computing platforms. Because of this conceptual correspondence we use the same variable P in our notation. While both metrics provide the same information and depend on a physical device (either a given CPU or FPGA), a reconfigurable architecture's peak area throughput additionally depends on the architectural description of the implemented functional blocks. The core advantage of reconfigurable hardware over static architectures is the fact that freed resources can be reused to implement additional functional units resulting in increased parallelism. The fewer resources are required for a single functional unit, the more units are possible and thus the higher the respective peak area throughput.

The peak area throughput $P_{<Op>,R}(F)$ is defined as the number of FP units N_u operating on input data of number format F which can be realised with a given amount of hardware resources R multiplied with the number of parallel (FP) operations N_{op} per functional unit divided by the time $T(F)$ needed to complete the operation:

$$P_{<Op>,R}(F) = \frac{N_u(F)[\mathrm{unit}] \cdot N_{Op}(F)[\mathrm{Flop/unit}]}{T(F)[s]}$$

The maximum number of functional units can be obtained by dividing the available hardware resources R by the resources $A(F)$ required for a single functional unit in the required number format F. Applying Equation 4.3 yields an alternative formulation of the peak area throughput $P_{<Op>}$ in terms of total resources R, parallel operations per functional unit N_{Op} and area-time product $C(F)$.

$$P_{<Op>,R}(F) = \frac{N_u(F) \cdot N_{Op}(F)}{T(F)} = \frac{R}{A(F)} \frac{N_{Op}(F)}{T(F)} = \frac{R \cdot N_{Op}(F)}{C(F)}$$

4.3.2 Performance Benefits of Reduced Precision

On CPUs, the system word size determines the optimal FP representation. CPU instruction sets like SSE [Int07] or AVX [Int16] allow the processing of multiple operands in parallel (short vector SIMD). Consequently, the peak performance for SP operations is about double the peak performance of DP operations [LLL*06] ($P(SP) \approx 2 \cdot P(DP)$).

On FPGAs, we denote the achievable improvement (speedup) in P of some functionality Op due to using a smaller number format $\beta < \alpha$ by

$$s_{<Op>}(\beta, \alpha) = \frac{P_{<Op>,R}(\beta)}{P_{<Op>,R}(\alpha)} = \frac{A(\alpha) \cdot T(\alpha)}{A(\beta) \cdot T(\beta)} = \frac{C(\alpha)}{C(\beta)} \quad . \tag{4.4}$$

The speedup $s_{<Op>}(\beta, \alpha)$ strongly depends on the type of FP operation. In the following, we derive models for the peak area throughput as a function of the precision p for elementary FP operations. For a more detailed analysis of a few basic arbitrary precision FP operators, we refer the interested reader to [MLG11]. The derived models will be used in section 4.4 to explore the trade-off between performance and accuracy in IR as a function of the FP formats used.

Multiplication The dominant part of FP multiplication (with respect to hardware resources consumed) is the (integer) multiplication of the two mantissas (see [MBdD*10] for full details). FPGAs provide small embedded multiplier blocks which favour decomposition of multiplication into a sum of partial products [Par00, Kor01] leading to a quadratic complexity of the area-time product: $A(n) \cdot T(n) = O(n^2)$.

While the concrete performance of FP multiplication depends on many details, a quadratic relationship between mantissa bit-width and area-time product can be observed for FPGA implementations of FP multiplication [GST07, SPS08, MLG11]. We therefore model the area-time product $C_*(\alpha)$ of an FP multiplication as a function of the number format α by a second-order polynomial in the number format's precision p_α:

$$C_*(\alpha) \approx c_1 + c_2 \cdot p_\alpha + c_2 \cdot p_\alpha^2 \tag{4.5}$$

Addition Addition is a complex operation in FP arithmetic and many different designs exploiting different properties exist [MBdD*10]. The dominant operations of FP addition are wide integer addition of the mantissas and the leading-one detection necessary for normalisation of the sum. Experiments [MLG11] show that the complexity of FP addition on FPGAs remains linear in the precision p. We therefore model the area-time product

$C_+(\alpha)$ of FP addition on FPGAs as a function of the number format α with precision p_α as $C_+(\alpha) = c_4 + c_5 \cdot p_\alpha$.

Fused Multiply-Accumulate (FMA) FMA implements the three-operand operation $a * b + c$. By omitting one rounding operation, FP FMA yields a more compact implementation and a more accurate numerical result compared to implementations performing FP multiplication and addition in sequence [MBdD*10]. Based on the discussions in the preceding paragraphs, we can model the area-time product $C_{\text{FMA}}(F)$ of FP FMA as

$$C_{\text{FMA}}(\alpha) = c_6 + c_7 \cdot p_\alpha + c_8 \cdot p_\alpha^2 \quad . \tag{4.6}$$

Most embedded multiplier blocks available in contemporary FPGAs do implement (short-operand) FMA [Alt11, Int17a] which, if exploited properly, allows for effectively hiding of the complexity of addition in FP FMA. While the resulting coefficients c_6, c_7, c_8 will in general differ from c_1, c_2, c_3 used in Equation 4.5, the functionality of embedded multiplier blocks justifies the assumption that $c_6 \ll c_8$ and $c_7 \ll c_8$. Inserting Equation 4.6 into Equation 4.4 yields the following speedup:

$$s_{\text{FMA}}(\beta, \alpha) = \frac{C_{\text{FMA}}(\alpha)}{C_{\text{FMA}}(\beta)} = \frac{c_6 + c_7 \cdot p_\alpha + c_8 \cdot p_\alpha^2}{c_6 + c_7 \cdot p_\beta + c_8 \cdot p_\beta^2} \approx \frac{p_\alpha^2}{p_\beta^2} \quad . \tag{4.7}$$

For real implementations, the quadratic term dominates both C_{FMA} and P_{FMA} [MLG11]. Based on experimental data, we therefore neglect the constant and linear term, yielding an approximation of the improvement in peak area throughput by the relation of the squares of the respective precisions.

4.3.3 Performance Model for APIR

As a simplification, we assume that each FP multiplication can be combined with an FP addition/subtraction to form an FMA operation, which is a valid assumption for all components of the APIR algorithm (the implementation of an LU decomposition, forward and back substitution and computing the residual). We neglect the resources required for division operations, since the number of divisions is of lower order. The number of FMA flops $N_{<Op>}$ required by the LU factorisation [GVL13] and the different parts of IR are shown in Table 4.1.

The speedup for all IR methods can be determined using Equation 4.4. For example, the speedup $s_{\text{LU}}(\beta, \alpha)$ for an LU-based linear system solver with $P_{\text{LU}}(n, F) = P_{\text{FMA}}(F)/N_{\text{LU}}(n)$ is the ratio of the two peak area throughputs for β and α:

$$s_{\text{LU}}(\beta, \alpha) = \frac{P_{\text{LU}}(n, \beta)}{P_{\text{LU}}(n, \alpha)} = \frac{P_{\text{FMA}}(\beta)}{P_{\text{FMA}}(\alpha)} = \frac{C_{\text{FMA}}(\alpha)}{C_{\text{FMA}}(\beta)} \approx \frac{p_\alpha^2}{p_\beta^2} \quad . \tag{4.8}$$

Notably, the speedup does not depend on n but only on the two precision levels p_α and p_β.

Table 4.1: Number of FMA flops required by different parts of IR

Operation	**FMA flops**
LU factorisation	$N_{\text{LU}} = n^3/3$
Forward and back substitution	$N_{\text{Solve}} = n^2$
Computing the residual	$N_{\text{Res}} = n^2$
IR for $\beta = \alpha$	$N_{\text{IR}} = N_{\text{Solve}} + N_{\text{Res}} = 2n^2$
IR for $\beta \neq \alpha$	$N_{\text{IR},\alpha} = N_{\text{Res},\alpha} = n^2$
	$N_{\text{IR},\beta} = N_{\text{Solve},\beta} = n^2$

The execution time of APIR $t_{\text{APIR}}(\alpha, \beta)$ can be approximated by the sum of the execution time of the initial solver $t_{LU}(\beta) + t_{\text{Solve}}(\beta)$ and of a single refinement step $t_{\text{IR}}(\alpha, \beta)$ multiplied by the number of iterations $i_{\text{conv}}(\alpha, \beta, n, \kappa(A))$ required until convergence:

$$t_{\text{APIR}}(\alpha, \beta) \approx t_{\text{LU}}(\beta) + t_{\text{Solve}}(\beta) + i_{\text{conv}}(\alpha, \beta, n, \kappa(A))\, t_{\text{IR}}(\alpha, \beta) \quad . \tag{4.9}$$

This execution time model assumes no overlap of computations in the two precisions considered and neglects communication overhead. The effects of these two assumptions on estimated execution time partially cancel each other, potentially rendering the naive model more realistic than expected at first sight. The degree of cancellation depends on a multitude of design choices on the algorithm (e. g. blocked matrix operations) and on the hardware platform. Exploring this design space is an interesting avenue for future work.

4.3.4 A Note on Finding the Optimal Working Precision

The performance model presented in this section can be used to determine the optimal working precision β^* for a given problem before the computation. However, the model depends on a critical component required to predict the number of IR iterations: $\kappa(A)$.

Computing the condition number of a general matrix has a complexity of $O(n^3)$, the same order as the LU factorisation. An alternative to computing $\kappa(A)$ directly would be a condition number estimator, but in many cases, they require the LU factors of A to approximate $\left\|A^{-1}\right\|_2$ by solving a linear system. One of these approaches described by Higham [Hig02] uses the power iteration to estimate the largest singular value of A and in turn estimate the 2-norm condition number. The matrix A in the power iteration is subsequently replaced by A^{-1} and the resulting linear system then has to be solved using the LU factors of A. Even though $\|A\|_2$ can be computed at very little cost, $\left\|A^{-1}\right\|_2$ is computationally much too expensive. Demmel, Diament and Malajovich [DDM00] proved that an estimate of $\left\|A^{-1}\right\|_2$ of guaranteed quality is as computationally expensive as testing if the product of two matrices equals zero, which is assumed to cost the same as a matrix multiplication.

If a good estimate of $\kappa(A)$ is already known, due to the linear system resulting from a special application or a special matrix type, our models Equation 4.2 and Equation 4.9 can be used to find the optimal working precision β^* to achieve a high performance. Even only knowing the order of magnitude of $\kappa(A)$ would already be sufficient to have a very good approximation of the optimal β^*. Initial observations compared to experimental data have also shown that better results can be achieved using the cheaper 1-norm instead of the 2-norm condition number.

4.4 Experimental Evaluation

In this section, we combine the models and observations from section 4.3 with experimental results for the number of iterations based on our software implementation of APIR to quantitatively estimate the potential performance gains over a direct LU-based linear solver.

Without loss of generality, we focus on the case $\alpha = DP$ and the effects of $\beta < \alpha$ on the performance of APIR. We only consider number formats with reduced mantissa while keeping the exponent range constant ($e_\beta = e_\alpha$). Not reducing the exponent range keeps the numerical range identical to DP, thereby preventing any FP overflow behaviour. However, depending on the problem setting, the bit-width of the exponent e_β could be chosen lower than e_α. Smaller exponents would increase the area available for additional functional units and would further increase the performance benefits of APIR on FPGAs.

We implemented APIR in C using the GNU MPFR library [FHL*07], a fully IEEE 754 compliant arbitrary precision library. Unlike many other arbitrary precision packages, MPFR allows the precision to be set individually for each variable. Furthermore, the precision of a variable can be set to exactly the number of bits in the mantissa and does not have to be a multiple of the system word size. This allows for a correct emulation of any FP representation.

In subsection 4.4.2, we first analyse the model for the number of iterations of APIR Equation 4.2 for different working precisions and different condition numbers and compare the model predictions with measurements from our MPFR implementation of APIR. In subsection 4.4.3, we use the performance model Equation 4.9 from section 4.3 and show the modelled speedup for different system sizes and condition numbers for APIR and other existing IR methods. Last but not least, we analyse the numerical accuracy of different IR methods and especially of APIR at working precisions below SP in subsection 4.4.4.

4.4.1 Generating Test Matrices

For the experiments, we require test matrices with specific condition numbers κ to analyse the accuracy of the algorithms. We consider the condition number with respect to the 2-norm, $\kappa(A) = \sigma_{\max}/\sigma_{\min}$, where σ_i are singular values of A. In this section we describe our procedure for generating our test matrices.

The idea of our approach is to modify the singular values of A to receive the desired condition number for the matrix. The algorithm for generating the test matrices is shown in Algorithm 2. The method requires a matrix A and the targeted condition number ϑ as its input and returns a modified matrix $\hat{A}$ where $\kappa(\hat{A}) = \vartheta$. First, a singular value decomposition (SVD) of A is computed in line 1, where Σ holds the singular values σ_i which are sorted in descending order. Depending on the requested ϑ, the singular values have to be modified. We distinguish between the following cases:

1. The simplest case is $\vartheta = 1$, which can only be reached by setting all singular values to 1 (lines 3-4).
2. If $\kappa(A) > \vartheta$, a pair of singular values is sought for which satisfies $\sigma_i/\sigma_{m-i} \leq \vartheta$, where $i \in [1, m/2]$ (lines 6-10).
    (a) If no pair of singular values matches this criterion, then we fall back to the simplest case of setting all singular values to 1 (lines 11-12). The first singular value σ_1 will then be set to the desired condition number ϑ in line 19.
    (b) Otherwise, the singular values larger than σ_i are set to σ_i and the ones smaller than σ_{m-i} are set to σ_{m-i} (lines 13-15).
3. In the case $\kappa(A) \leq \vartheta$, no specific changes are necessary before the scaling in line 19.

In all cases, the first singular value is then set to the last singular valued scaled by ϑ (line 19). Finally, the new matrix $\hat{A}$ is computed using the factors U and V from the SVD and the modified singular values stored in Σ (line 20), leading to $\kappa(\hat{A}) = \vartheta$.

4.4.2 Model for the Number of Iterations

In this section, we demonstrate the reliability of the analytical model Equation 4.2 for the number of iterations with experimental data.

In these experiments, the maximum number of iterations was set to 30. The number of iterations for different working precisions p_β are shown in Figure 4.1 for $n = 3000$, $p_\alpha = 53$ and different condition numbers κ on the z-axis. For the majority of cases, APIR requires $p_\beta \geq 14$ bits to converge. Only the better conditioned systems require slightly less precision, a perfectly conditioned system converging for $p_\beta = 11$ bits.

Figure 4.2 shows the prediction deviation between the analytical model Equation 4.2 to achieve p_α and the observations from the experiments. The prediction deviation is calculated by subtracting the number of iterations used in the experiments from the value predicted in the analytical model Equation 4.2:

$$i_{conv}(\alpha, \beta, n, \kappa) - i_{experiment} \ .$$

The graphs for three different system sizes are plotted for p_β between 11 and 52 bits. In addition to the termination criterion $\|r\|_2 < \tau_r$ (see subsection 4.2.1), the experiments were also terminated after reaching the maximum number of iterations $i_{max} = 30$, whereas the model is not limited to an upper bound on the iterations. As one can see, the model comes very close to the measured number of iterations for all three test cases. If the model does not match the experimental data, it predominantly predicts a slightly higher number of iterations.

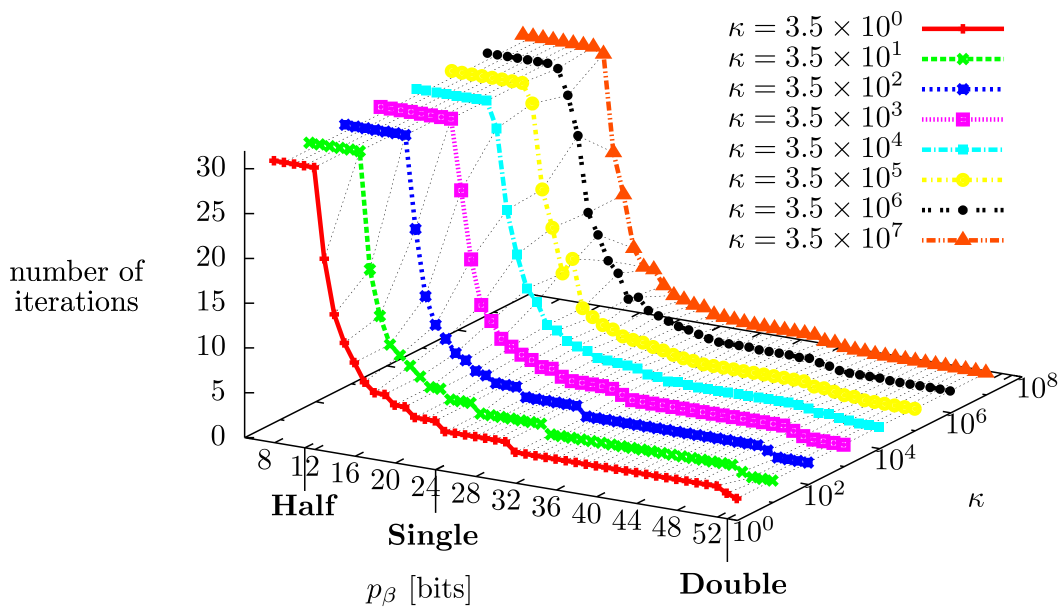


Figure 4.1: Number of iterations used by APIR to reach convergence for different working precisions p_β and varying κ with $p_\alpha = 53$ and $n = 3000$. In these experiments, the maximum number of iterations was limited to 30.

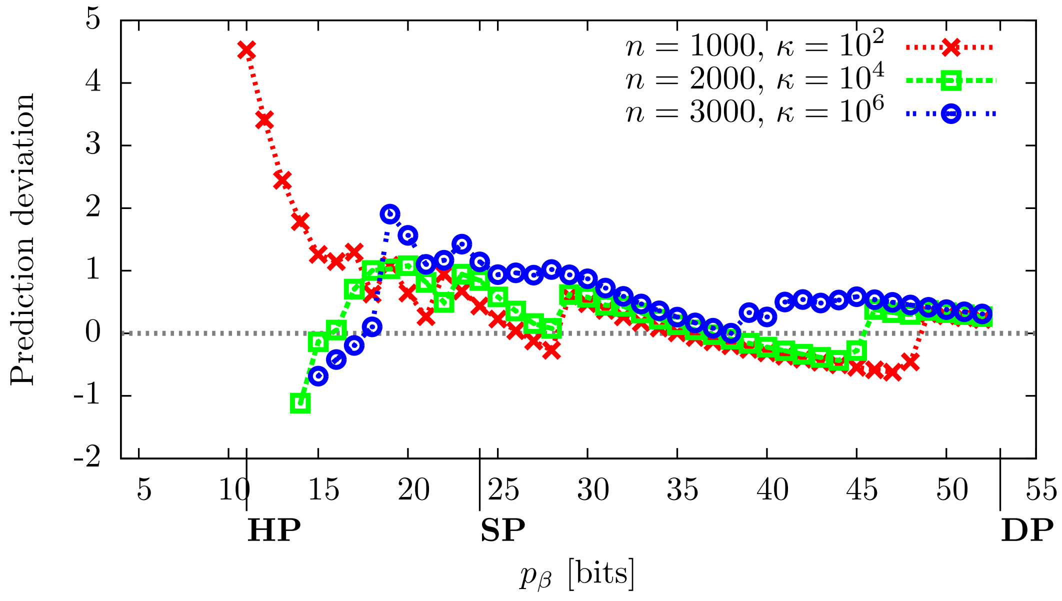


Figure 4.2: Prediction deviation of the analytical model Equation 4.2 for the number of iterations compared to the experimental data $(i_{\text{conv}}(p_\alpha, p_\beta, n, \kappa) - i_{\text{experiment}})$.

Algorithm 2 Generating test matrices with prescribed condition number κ

```
Input: $A \in \mathbb{R}^{n \times m}$ where $n \geq m$, targeted condition number $\vartheta$
Output: $\hat{A} \in \mathbb{R}^{n \times m}$ where $\kappa(\hat{A}) = \vartheta$
1: $U, \Sigma, V \leftarrow \texttt{svd}(A)$    ▷ Singular value decomposition of $A$ where $\forall i : \sigma_i \geq \sigma_{i+1}$
2: $\kappa(A) = \sigma_1 / \sigma_m$
3: if $\vartheta == 1$ then
4:     $\sigma_i = 1 \quad \forall i \in [1, m]$    ▷ Set all singular values (sv) to 1
5: else
6:     if $\kappa(A) > \vartheta$ then
7:         $i = 1$
8:         while $i \leq m/2$ and $\sigma_i / \sigma_{m-i} > \vartheta$ do    ▷ Find a pair of sv with ratio $\leq \vartheta$
9:             $i = i + 1$
10:        end while
11:        if $i > m/2$ then    ▷ No pair of sv found with ratio $\leq \vartheta$
12:            $\sigma_i = 1 \quad \forall i \in [1, m]$
13:        else    ▷ Pair of sv found with ratio $\leq \vartheta$
14:            $\sigma_j = \sigma_i \quad \forall j \in [1, i-1]$
15:            $\sigma_j = \sigma_{m-i} \quad \forall j \in [m-i+1, m]$
16:        end if
17:    end if
18: end if
19: $\sigma_1 = \vartheta \sigma_m$
20: $\hat{A} \leftarrow U \Sigma V^\top$
```

4.4.3 Performance Benefits of APIR

Software emulated arbitrary precision exhibits hardly any performance difference for the small range of mantissa widths investigated in the context of APIR. Time measurements are hence not conclusive when trying to identify performance gains based on the use of different precisions. Therefore, our estimate of the achievable speedup is based on the number of FMA flops (shown in Table 4.1) instead of the execution time. The performance comparison is based on the performance model Equation 4.9, which accounts for the performance gains on FPGAs due to the different working precisions. The following experimental results compare the speedup of different IR methods discussed or introduced in this chapter over a direct LU solver in precision α. The FP representations used by the methods are: $\beta = \alpha$ (SIR), $\beta = 2\alpha$ (EPIR), $\beta = 0.5\alpha$ (MPIR) and $\beta < \alpha$ (APIR).

In Figure 4.3, the modelled speedup shows that SIR and EPIR are almost as fast as the direct solver, demonstrating the low complexity of the additional work caused by IR compared to the LU decomposition. In SIR and EPIR, the LU decomposition is calculated using the target precision α and EPIR uses the higher working precision $\beta = 2\alpha$ only to calculate the residual and the correction term of the solution. MPIR is the first IR method to have a speedup larger than 1 and is 4.71 times faster than the DP direct solver. On CPUs, the maximum theoretical speedup achievable by MPIR would be limited to 2. However, due to the quadratic relationship of FP FMA operations on FPGAs described in Equation 4.7,

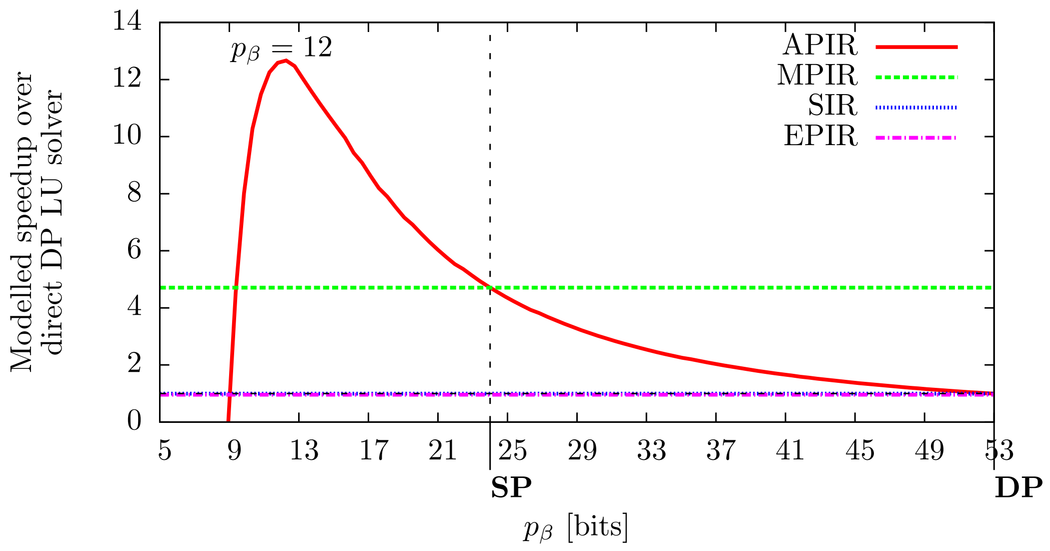


Figure 4.3: Comparison of the modelled speedup for different IR methods and varying working precision p_β for $n = 1000$, $\kappa = 10^3$ and $p_\alpha = 53$.

the maximum theoretical speedup on an FPGA would be $p_\alpha^2/p_\beta^2 = 53^2/24^2 \approx 4.88$. Based on the model, APIR achieves the highest speedup of 12.55 with $p_\beta = 12$ bits, being almost 2.7 times faster than MPIR. The maximum speedup of APIR continues to increase with the dimension of the system, as shown in Figure 4.4. At $n = 10000$, APIR uses the low working precision of $p_\beta = 11$ and achieves a speedup of 20.27. The speedup of MPIR saturates at an early stage and reaches 4.86 for $n = 10000$.

It is not possible to directly compare the number of iterations of different IR methods because the amount of work per iteration and the working precisions p_β differ. Figure 4.5 shows the number of iterations for different condition numbers for a system with $n = 2000$. SIR requires the lowest number of iterations and in most cases only requires one iteration using the same precision for all operations. MPIR uses more iterations than SIR, but the majority of operations are performed in p_β = SP, which results in a higher performance compared to SIR despite using a higher number of iterations. EPIR not only requires a high number of iterations, but also performs some of them at working precisions higher than the target precision, requiring on average 2 iterations in each of the used precisions. APIR uses the highest number of iterations, but these are executed at extremely low working precisions p_β which gives APIR a significant performance advantage over the other IR methods despite the higher number of iterations.

The modelled speedup due to the usage of different working precisions can be seen in Figure 4.6. Even though APIR uses a high number of iterations, it achieves a significant speedup compared to a direct LU solver and the other IR methods. For a perfectly conditioned system, it is almost 16 times faster than the direct solver and for a system with $\kappa = 10^7$ it is more than 9 times faster. The increasing speedup for $\kappa = 10^5$ and $\kappa = 10^6$ compared to $\kappa = 10^4$ is due to APIR being able to use lower working precisions than for $\kappa = 10^4$ for the

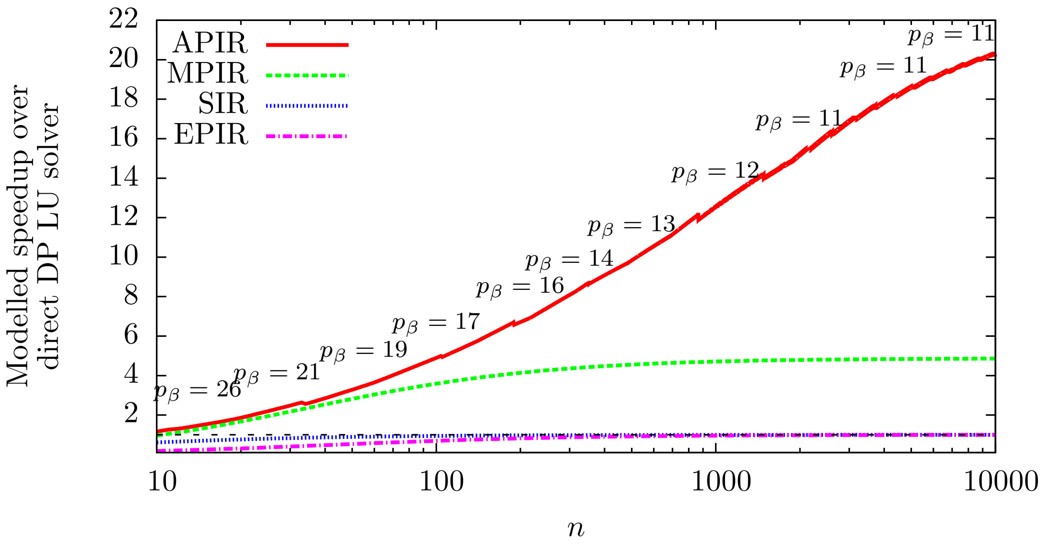


Figure 4.4: Comparison of the maximum modelled speedup for different IR methods for increasing system size n. $p_\alpha = 53$, $\kappa = 10^3$.

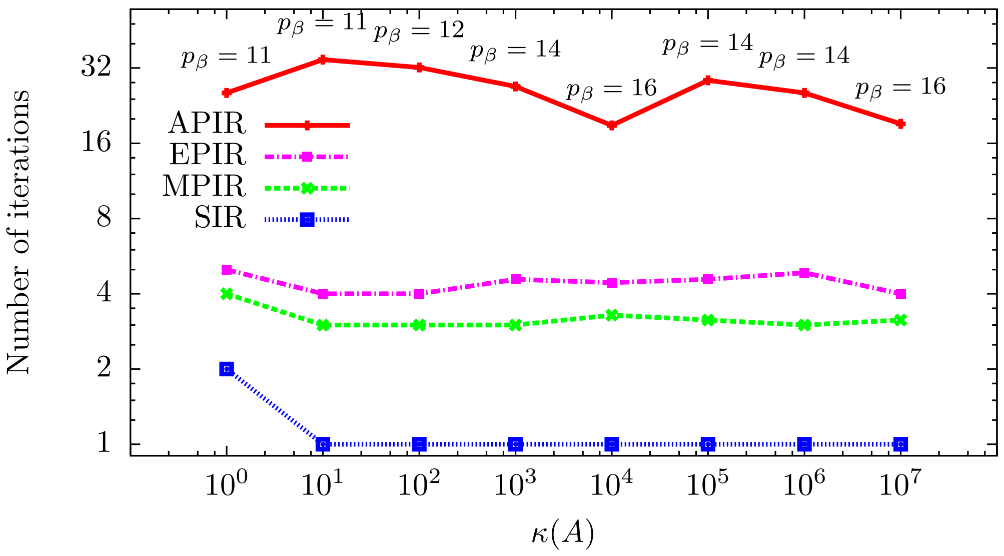


Figure 4.5: Number of iterations for different IR methods for $n = 2000$ and varying κ.

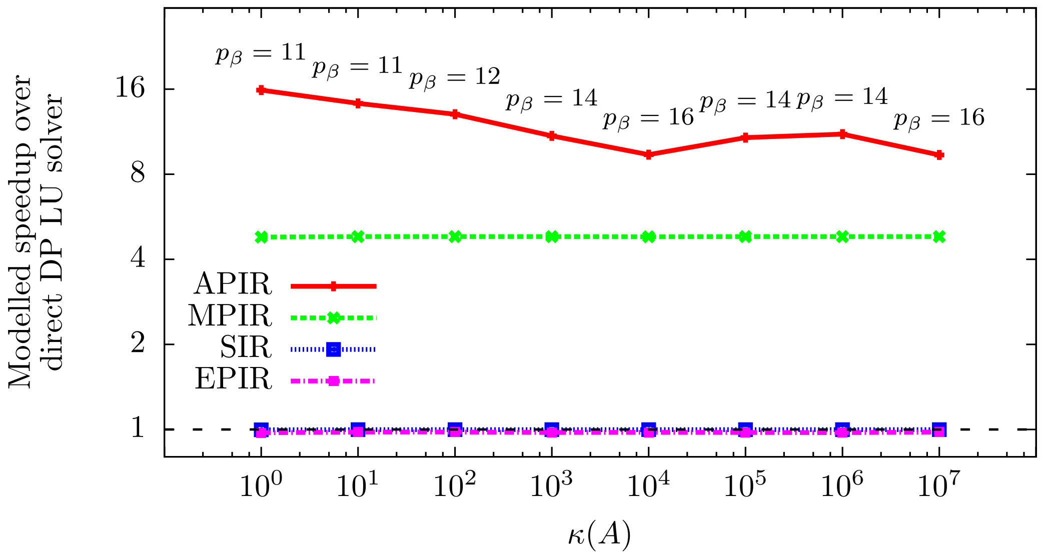


Figure 4.6: Speedup for different IR methods for $n = 2000$ and varying condition numbers κ.

matrices used in these experiments while still achieving an accurate result. MPIR reaches a speedup of 4.5 for all matrices with $n = 2000$. SIR and EPIR are both only slightly slower than a direct LU solver with a slow-down of 0.97 and 0.9, respectively.

4.4.4 Numerical Accuracy

The next aspect to be analysed is the accuracy achieved by the different IR methods by comparing the relative residual of the computed solution $\tilde{x}$:

$$r_{\text{rel}} = \frac{\|A\tilde{x} - b\|_1}{\|A\|_1 \|\tilde{x}\|_1}$$

In Figure 4.7, the relative residual is plotted for a linear system with $n = 2000$ for different condition numbers κ shown on the x-axis. The first (turquoise) line is the LU solver without IR. Using IR, r_{rel} can be improved by almost 2 orders of magnitude. The very ill-conditioned systems do not profit as much from IR, but still improve the result by almost 1 order of magnitude, with the exception of MPIR and APIR, which do not find a better result for these systems, and EPIR, which is hardly any better than the direct LU solver. In all other cases the IR methods have significantly improved the result, all achieving almost the same accuracy for systems with $\kappa \leq 10^4$. Most of the time, APIR produces very similar relative residuals compared to the other IR methods, while using much lower working precisions and benefiting in terms of performance.

For growing n, as shown in Figure 4.8, the relative residual of the direct LU solver increases, but the refined solutions remain at the same level of accuracy for all larger systems. All IR methods achieve again approximately the same improvement. Even though APIR uses

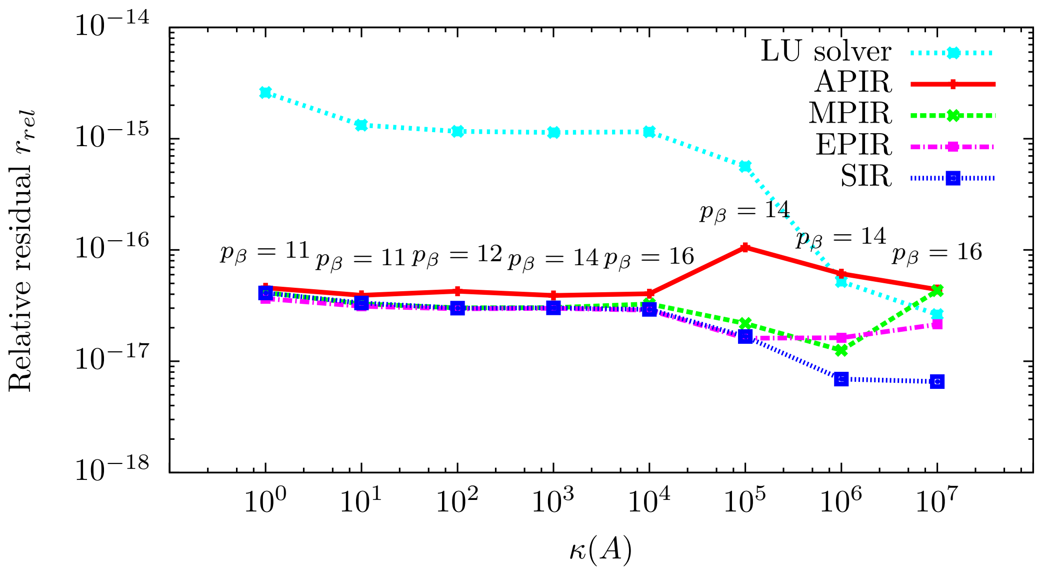


Figure 4.7: Relative residual r_{rel} for different IR methods for $n = 2000$ and varying condition numbers κ.

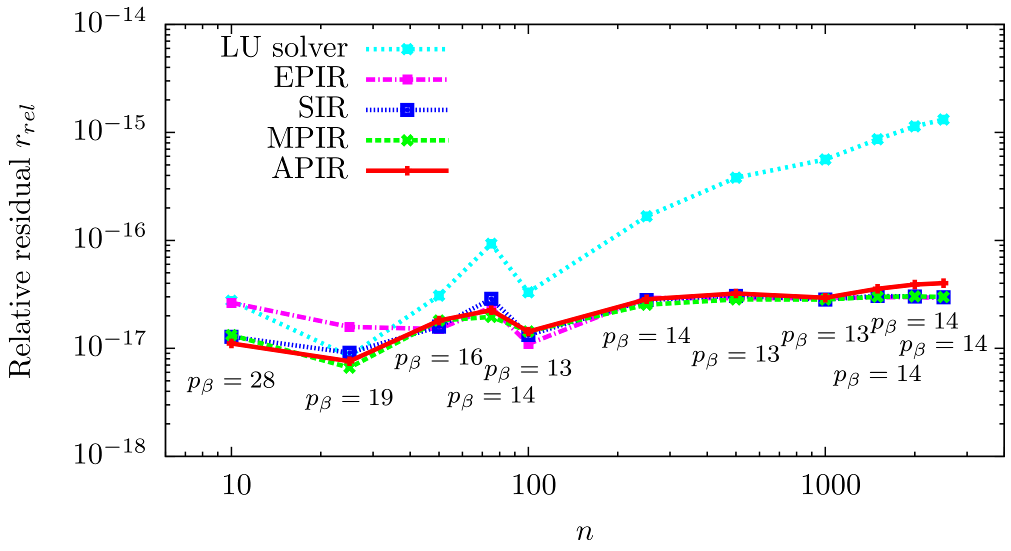


Figure 4.8: Relative residual for different IR methods for $\kappa = 10^3$ and varying system size n.

lower working precisions, the achieved accuracy is very close to the other IR methods, with the difference being negligible.

4.5 Conclusion

In this chapter, we considered the LU-based linear system solver APIR as a generalisation of MPIR. APIR is not restricted to the standard IEEE 754 FP representations but can choose any precision as its working and target precisions. On CPUs, the speedup of SP over DP scales approximately linearly, in general being limited to two. On other architectures, like GPUs and FPGAs, the performance advantage of SP operations can be significantly higher. The peak area throughput achievable on FPGAs scales approximately with the inverse of the square of the precision, which results in a much higher relative performance gain from the use of lower precisions, especially below SP. Furthermore, the cost of implementing FP operations on FPGAs decreases quadratically with the reduction of the precision.

We showed that APIR can outperform direct LU solvers and other IR algorithms (SIR, MPIR, EPIR). We developed a model to determine the number of iterations required by IR using two different precision levels (α, β) based on the input data (n, κ). Using a software implementation of APIR in GNU MPFR, we verified the good fit of the model with the experimental data. Even though APIR requires more iterations than other IR methods, the majority of the operations (the LU factorisation) are performed at the lower working precision. In comparison, the additional cost for an increased number of iterations is very low. The performance model developed in this chapter can be used to predict the performance of APIR on FPGAs and takes into consideration the design and implementation choices of digital logic on FPGAs. We illustrate that APIR can achieve very high speedups by exploiting the performance benefits of low working precisions. For a linear system with $n = 10\,000$ APIR is projected to be more than 20 times faster than a direct LU solver. Furthermore, we investigated the numerical accuracy achieved by different IR methods for different condition numbers $\kappa(A)$ and the same target precision $\alpha = DP$. We showed that APIR achieves a very similar accuracy compared to the other algorithms, even if it uses working precisions far below SP.

The insights we have gained from our research of APIR will be used in the context of our truly distributed linear least squares solvers in Chapters 6 and 7. The use of lower precisions than the standardised IEEE floating-point representations will increase the performance of our algorithms and reduce the amount of communication required within a low-powered network of sensor nodes, while still achieving a result with high accuracy.

Part II

Truly Distributed Linear Least Squares Solvers

Chapter 5

Towards a Truly Distributed Linear Least Squares Solver

The solution of linear least squares (LLS) problems across large loosely connected distributed networks (such as wireless sensor networks) requires distributed algorithms which ideally need very little or no coordination between the nodes. In this chapter, we first provide an extensive overview of distributed least squares solvers appearing in the literature and classify them according to their communication patterns. We are particularly interested in *truly distributed* algorithms which do not require a fusion centre, cluster heads or any multi-hop communication.

Beyond existing methods, we propose the novel least squares solver PSDLS, which utilises a distributed QR factorisation algorithm [SGZ12, SG13]. All communication between nodes is exclusively performed within the push-sum algorithm for distributed aggregation. The PSDLS algorithm will form the basis for some of the algorithms presented in the following chapters, where additional fault tolerance techniques will be explored and performance improvements will not only be implemented, but designed, developed and analysed.

We analytically compare the communication cost of PSDLS and the truly distributed algorithms existing in the literature. In all these algorithms, the communication cost of reaching a predefined accuracy depends on many factors, including network topology, problem size, and settings of algorithm-specific parameters. We illustrate with simulation experiments that our novel PSDLS solver requires significantly fewer messages per node than the previously existing methods to reach a predefined solution accuracy.

5.1 Distributed LLS Solvers on WSNs

We consider the problem of solving the linear least squares problem

$$\min_x \|b - Ax\|_2 \tag{5.1}$$

for $x \in \mathbb{R}^m$ in a truly distributed way, where $A \in \mathbb{R}^{n\times m}$ with $n \geq m$ and $b \in \mathbb{R}^n$. We are interested in solving such problems over a loosely connected, decentralised network, e. g. a wireless sensor network (WSN), where each node holds part of the input data. In particular, we assume that A is distributed row-wise over the N nodes of the network and that the element $b(i)$ resides on the same node as the i^{th} row of A. For $n > N$, each node contains a block of consecutive rows of A.

Many applications in WSNs require the distributed solution of a linear least squares problem, e. g. the reconstruction of physical fields [RMG12], target tracking [Say14], the solution of the seismic tomography inversion problem [SSX*13] when monitoring volcanic activity or localisation [RBTB06]. WSNs typically consist of a large number of inexpensive sensor nodes which act autonomously but cooperate with each other to achieve a common goal. Working in a fully decentralised manner allows for decisions to be made on any node. In combination with actuators, the nodes can take autonomous actions in the physical world. Asynchronous communication is an important challenge to be considered in the design of a truly distributed algorithm. The sensor nodes are normally constrained in terms of their resources, primarily their energy supply and computation capabilities. One of the sources of high power consumption is communication. The energy required by the nodes to communicate with other nodes is directly proportional to the communication range. This implies that communicating with the immediate neighbourhood of a node is significantly cheaper than communicating with very distant nodes. Preserving energy also increases the lifespan of the nodes and in turn of the entire network.

As we will summarise in the following section, many distributed least squares solvers can be found in the literature, but most of them do not operate in a truly distributed manner without the need for centralised fusion centres, cluster heads or multi-hop communication. Multi-hop communication requires routing tables and setting those up requires additional communication. The overhead is particularly large if the routing tables have to be updated frequently.

Dynamic changes and distributed fault tolerance are also important factors in the design of a distributed algorithm for WSNs. These difficult scenarios can be considered implicitly by the use of gossip algorithms for aggregation. The push-sum algorithm [KDG03] used by the PSDLS algorithm proposed in this chapter can be directly replaced by fault-tolerant alternatives which are able to recover from silent message loss and temporary or permanent link failures, e. g. push-flow [GNSSG13, NSG12]. The fault tolerance properties of push-flow will be shown in the experiments in Chapter 7 in the context of our truly distributed linear least squares solvers GLS-IR-SNE and GLS-IR-NE.

In the following section, section 5.2, we provide motivational examples of the possible applications for WSNs. In section 5.3, we provide an extensive review of the existing literature about distributed least squares solvers and classify them based on their communication patterns. In section 5.4, we introduce the new push-sum-based distributed least squares solver PSDLS. section 5.5 provides an analytical comparison of the communication cost of PSDLS and the truly distributed algorithms appearing in the literature up until now. Simulation results are presented in section 5.6, and the conclusions of this chapter and an outlook of the following chapters are summarised in section 5.7.

5.2 Applications of WSNs

The range of applications of wireless sensor networks (WSN) is vast. They are ideal for deployment in remote or hostile areas or places where it would be difficult to deploy a wired network of sensors, e. g. in a city. The relatively inexpensive components allow for monitoring wide areas of interest at low costs.

In London, a WSN was set up to monitor air pollution [MRG*08]. The network consisted of a static set of nodes and additional mobile nodes, all of which could preprocess the data at the nodes. Such networks can also be used to detect toxic levels of chemicals and issue alerts based on cooperatively computed results. A similar area of application is monitoring the quality of water [DHS*07]. Another application can be found in agriculture, where irrigation management can be handled by WSNs [ASY*10]. This allows for water resources to be applied more efficiently based on temperature level and soil moisture and can even be completely automated when combining the network with an irrigation system. WSNs can also be used to protect the infrastructure of the power grid in the “smart grid” [MRAMBJ12], providing sustained on-demand delivery of electricity and improving the efficiency and reliability of the power grid.

A WSN deployed in a forest could detect when and where a fire broke out based on sensor measurements of temperatures and gases. It can further help the firefighters to predict the direction the fire is spreading. An early warning system could greatly decrease the response time of the fire brigade and reduce the devastation forest fires cause every year.

A distributed sensor network can also be used to detect traffic anomalies, e. g. accidents, based on the behaviour of each car [TGTB13]. In [TSS07], multiple layers of WSNs are used to control traffic flow. The sensors measure the number of vehicles and their speed and are responsible for adapting the dynamically changing traffic volume. Efficient parking management, smart parking [SPD*09], can help reduce traffic congestion by informing drivers accurately about available parking spaces.

Deployment of WSNs is also considered for planetary explorations, e. g. on Mars [HGB01, UYA03]. They could be used to measure different physical properties, e. g. gases, chemicals, temperature, etc. This is a prime example for a harsh environment, where the nodes should be able to operate autonomously primarily interacting with each other. Both papers mentioned here consider the use of a central base station where the sensor nodes relay their measurements for further processing, but it would be interesting to remove the need of a single point of failure and perform computations on the nodes directly.

Monitoring volcanic activity to predict volcanic eruptions is a highly anticipated research area. Many research projects have investigated the use of WSNs in this context and have also already deployed WSNs at active volcanoes. One example is Werner-Allen et al. [WAJR*05] from Harvard Sensor Network Lab. In 2005, the group deployed a 16-node network at a volcano in Ecuador [WALW*06]. Due to the limited bandwidth of the sensor nodes, they avoided the continuous collection of the measured data and instead used a "triggered event collection". Georgia State University has an ongoing research project[2] focusing on the use of large wireless sensor networks to create a four-dimensional tomography of an active volcano. The goal of the research is to use the model to learn and understand the volcanic processes taking place during an eruption, using the seismic measurements of the sensor nodes, and, in the future, being able to predict and detect volcanic threats. The deployment of a large wireless sensor network employing more than 500 sensor nodes is an essential focus of their project.

5.3 Classification of Existing Distributed LLS Solvers

In this section, we summarise the efforts presented in the literature for solving the linear least squares problem Equation 5.1 in a distributed setting. We categorise existing algorithms into three groups: (*i*) *Centralised approaches* using a fusion centre or approaches which require *global communication*, (*ii*) *clustered approaches* where the communication of each node is limited to a subset of the network (cluster) with a cluster head, and (*iii*) *truly distributed approaches* where the communication of each node is limited to its immediate neighbourhood *without* using any multi-hop communication.

5.3.1 Centralised Approaches or Global Communication

A strategy that has been studied extensively is the use of a central unit (*fusion centre*) which performs the computation for the entire network. The fusion centre approach first collects the data from all nodes in the network (global communication), then solves problem Equation 5.1 at the fusion centre and finally distributes the result to all nodes (global communication). The positioning of the fusion centre is crucial for communication cost and scalability (cf. [SSX*13]). There are several drawbacks to this approach: Potential congestion effects (particularly around the fusion centre [KGH13]) can lead to delays and in the worst case to data loss. Multi-hop communication and setting up routing tables incur additional overhead. Last, but not least, the fusion centre becomes a single point of failure. Research on fusion centre approaches often focuses on the efficient accumulation of the data at the fusion centre (see, e. g. [LBNAS10]). Other efforts perform only parts of the computation at the fusion centre and offload other parts onto the individual nodes (see, e. g. [RBTB06]). However, these approaches still require global (multi-hop) communication of each node with the fusion centre.

[2] "*VolcanoSRI: 4D Volcano Tomography in a Large-Scale Sensor Network*" `https://sensorweb.engr.uga.edu/index.php/volcanosri/`

Reichenbach et al. [RBTB06] consider the problem that each node needs to determine its location and analyse three methods for solving the least squares problem arising in this context: normal equations, QR factorisation and singular value decomposition. For all three methods, they split the computation into two parts in order to distribute them between a high-performance base station and wireless sensor nodes. The base station computes the computationally intensive tasks and then sends the result to the nodes, which only have to perform low complexity computations to determine their location. This approach significantly reduces the amount of computation performed on the sensor nodes, saving more than 47% of floating-point operations for normal equations and more than 99% for the QR factorisation and the SVD. The disadvantage is the communication cost incurred by the nodes having to send their measurements to the fusion centre either over long distances or with multi-hop communication and non-static routing.

Borgne et al. [LBNAS10] presented one example for exploiting a specific routing structure, where the measured data is aggregated at each node towards the fusion centre along a routing tree. The authors extend the basic set of available aggregation functions (minimum, maximum, sum, count and average) to a regression operator which uses the sensor node measurements as input, reducing the amount of data based on the regression model. The advantage of this approach is the reduction of the communication range of the nodes to a localised neighbourhood. However, the final result is only available at the fusion centre, which in the event of a failure leads to the breakdown of the entire computation.

The distributed multisplitting method [SSK*13], based on the parallel multisplitting method by Renaut [Ren98], applies the well-known fixed-point iteration methods Jacobi, Gauss-Seidel and successive over-relaxation to the normal equations. The matrix A is distributed column-wise over the nodes and weighting matrices are used to recombine the solutions of the local problems, which are independent problems resulting from the linear multisplitting of A. Note that in this method, the solution x is not replicated, but distributed across the nodes. In each iteration a vector of size n has to be broadcast to all other nodes (global communication).

The distributed modified conjugate gradient least squares (D-MCGLS) algorithm [SSK*13] exploits the fact that the conjugate gradient method can be applied to the symmetric and positive definite normal equations. It is also based on a parallel method, MCGLS by Yang and Brent [YB04], which is targeted at distributed memory architectures. Yang and Brent have improved the parallel performance of the standard CGLS method by reducing the global synchronisation points for the inner products. D-MCGLS requires A to be distributed row-wise. If A is not symmetric, for each local row of A, the node also needs to have the corresponding column locally. Each node has to use the same initialisation for x. In each iteration, a vector of length m and a scalar value have to be broadcast to all other nodes in the network (global communication).

5.3.2 Clustered Approaches

A first step towards a more decentralised setting than the fusion centre approaches summarised in subsection 5.3.1 is based on clustering. The network is divided into clusters. In each cluster, one node acts as the cluster head, which often is more powerful than the other nodes in the cluster. The division is based on a certain criterion, e. g. on the geographical location of the nodes or on the predefined communication radius of the cluster head. The cluster heads act as intermediate fusion centres for the clusters. The nodes of a cluster only communicate with their cluster head and with nodes within the same cluster. Compared to the fusion centre approaches, a multi-tier model is used where only the cluster heads communicate with the fusion centre, reducing the communication cost and also the risk of congestion.

Behnke et al. [BSLT09] address issues arising with the clustered version of the distributed least squares algorithm presented in [RBTB06]. They report that the algorithm does not scale well with an increasing number of nodes and on large networks does not work at all due to the assumption that each node can communicate with all cluster heads which distribute the precomputed parts of the solution. They develop the scalable distributed least squares (sDLS) algorithm to overcome these drawbacks by limiting the communication of each node to its cluster head. To achieve this, each node is provided with individual precomputed data, in turn reducing the size of the data transferred to each node and also the computations to be performed by each node. Communication and computation costs are therefore independent of the network size and enable scalability of the algorithm also in large networks.

Shakibian and Charkari [SC10] propose a clustered, multi-swarm version of the particle swarm optimisation algorithm (MMS-PSO) for solving a least squares problem as a minimisation problem. Each cluster head manages the member nodes acting as a sub-swarm of the process. They also use a fusion centre to get the final global result from all cluster heads through weighted averaging. The authors claim that their method decreases the latency through clustering and converges faster than a fusion centre approach.

Summarising, clustering reduces but does not eliminate the risk of a single point of failure affecting the entire network. The cluster heads usually have to be more powerful than the other nodes to be able to handle the higher volume of messages received. If a cluster head fails, the complete area covered by the cluster and its data are lost until a new cluster head takes over.

5.3.3 Truly Distributed Approaches

The most decentralised approach is to limit the communication of the nodes to their immediate neighbourhood (defined by the communication range). Each communication partner has to be reachable in a single hop as multi-hop communication would incur additional overhead through routing and thus increase the energy consumption of the resource restricted nodes.

Zhou et al. [ZKHP11] propose a distributed least squares solver which they claim is robust against reported node failures. The algorithm is designed for $m = 1$ and higher dimensions are not considered in [ZKHP11]. The distributed iterative algorithm exchanges the values of A and b with the neighbours and updates them using a Metropolis weight based on the degree of the node's neighbours, which are determined before the iterative algorithm initialises. In the event of a node failure, convergence is still guaranteed, but the result will no longer be correct. Therefore, the authors extend their algorithm, trying to reduce the magnitude of the occurring error. A disadvantage is that node failures have to be detectable. Once detected, the weights used in the computation have to be updated throughout the network, which poses a global updating problem requiring communication across the entire network. In the event of a node failure, the magnitude of the error depends on the network topology. Although the algorithm presented in [ZKHP11] is truly distributed, we do not consider it in our analysis and in our simulations because it is restricted to the special case $m = 1$.

Sayed et al. [Say14, CS10, TS12] propose a diffusion-based least mean square estimator (diffLMS) using steepest-descent iterations for solving the normal equations. Diffusion strategies are seen as an alternative to consensus strategies for distributed optimisation problems, both limiting the communication to the neighbourhood. The data A and b are both distributed row-wise. In each iteration, diffLMS consists of two main steps, an adaption step and a combination step, and delivers an estimate of the solution x on each node. The authors provide two variants of their algorithm, adapt-then-combine (ATC) and combine-then-adapt (CTA), which differ in the order of these computation steps (for details, see section 5.5).

Another fully distributed approach is the distributed least mean squares method (D-LMS) by Schizas, Mateos and Giannakis [SGRR08, MSG09, SMG09]. D-LMS is based on Lagrange multipliers and uses the least squares residual and the difference between the estimates of x from the neighbourhood in a correction step to compute the least squares solution iteratively. The data distribution of A and b is again row-wise. At each step, an estimate for the solution x is available in each node. D-LMS communicates twice in each iteration, once to broadcast the current estimate to all neighbours and a second time to send individual correction vectors to each neighbour (single-hop unicast – for details, see section 5.5).

Linear least squares problems are convex optimisation problems. Algorithmic ideas which are very similar to D-LMS and diffLMS also appear in the distributed optimisation literature [SNV10, TBA86, NO10]. In [NO10], the authors provide a general framework how to solve convex optimisation problems in a distributed environment. The goal of the research is to cooperatively optimise a global objective function while the local objective functions are only known to the nodes themselves. The research builds on the work by Tsitsiklis et al. [TBA86], who developed a framework for the analysis of asynchronous distributed iterative optimisation algorithms. Tsitsiklis et al. considered algorithms that are gradient-like and each update minimises a cost function in a descent direction. Nedic and Ozdaglar [NO10] combine first-order methods, in this case the subgradient method, with the consensus algorithm to achieve distributed optimisation methods. The local objective function is minimised using the subgradient method, while the consensus step aligns its decision with the decisions of its neighbours, leading to a decentralised solver.

5.4 A Novel Distributed Linear Least Squares Solver

In this section, we introduce the Push-Sum Distributed Least Squares Solver (PSDLS), shown in Algorithm 3, for problem Equation 5.1. The matrix A and the vector b are distributed row-wise across the participating nodes. The parts of A and b available locally at node u will therefore be denoted by $A^{(u)}$ and $b^{(u)}$, respectively. The solution x is approximated at each node. The local instance of a vector v which occurs at every node u will be referred to as v_u, and $v_u(i)$ refers to the i^{th} element of v_u. In particular, x_u refers to the approximation of the entire solution vector x at node u. The algorithm does not require any knowledge about the global topology of the network and it does not assume any specific connections between the nodes. Each node only needs to know its neighbours. In such a setup, the push-sum algorithm [KDG03] provides a truly distributed way for summing or averaging values across the nodes of the network. If each node knows the total number of nodes N in the network, then the sum of the values over all nodes can be computed using distributed averaging. Note that N can also be estimated in a truly distributed way [SR13]. Alternatively, the push-sum algorithm can be used to compute the sums directly without the need to know N at every node. However, based on our experience, this variant leads to slightly slower convergence.

Algorithm 3 Push-Sum Distributed Least Squares Solver (PSDLS)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ nodes
Output: $x_u \in \mathbb{R}^m$ on every node
1: in each node $u$ do
2:     $[Q^{(u)}, R_u] \leftarrow \text{vdmGS}(A^{(u)})$
3:     $z_u \leftarrow \text{dmmv}({Q^{(u)}}^\top, b^{(u)})$
4:     $x_u \leftarrow$ solve $R_u x_u = z_u$                    ▷ local
```

PSDLS is a direct least squares solver first computing a distributed QR factorisation of A (line 3.2[3]) and subsequently solving *locally* a linear system with the triangular matrix R_u at every node (line 3.4). For the distributed QR factorisation we use the gossip-based distributed modified Gram-Schmidt orthogonalisation method *vdmGS* introduced in [SGZ12, SG13]. vdmGS returns the orthonormal matrix $Q \in \mathbb{R}^{n\times m}$ distributed row-wise (denoted by $Q^{(u)}$) and the complete upper-triangular matrix $R \in \mathbb{R}^{m\times m}$ in every node (denoted by R_u). Consequently, $Q^\top$ is distributed column-wise across the nodes. To compute the right-hand side of the linear system (line 3.3), the distributed matrix-vector multiplication *dmmv* described in [SG13] is used, which accepts the matrix argument distributed column-wise and the vector argument distributed row-wise. The solution of the linear system (back substitution) can be computed locally and does not need any further communication with the other nodes because

[3]Line x.y refers to line y in Algorithm x

every node has its local estimate of R. At the end of the algorithm, each node u has its own local approximation x_u of the solution of the least squares problem Equation 5.1.

5.5 Communication Cost of Distributed LS Solvers

We compare the communication cost of the novel PSDLS method, both variants of diffLMS described in [CS10] and D-LMS described in [SMG09] in terms of number of messages and amount of data sent per node.

5.5.1 Review and Analysis of Existing LLS Solvers

In this section, we will review the existing algorithms diffLMS and D-LMS and identify their communication steps.

5.5.1.1 diffLMS

There are different versions of the diffLMS algorithm aside from the order of execution in ATC and CTA mentioned previously. diffLMS can also exchange the observations $b^{(u)}$ and matrix rows $A^{(u)}$ with the neighbouring nodes to improve the estimate of the solution. This requires an additional step for exchanging the information which increases the communication cost. For better comparison with [CS10], we will limit the analysis to the versions without the additional information exchange.

In the ATC version of the diffLMS method, shown in Algorithm 4, each node u first computes an intermediate value $\psi_u \in \mathbb{R}^m$, which adds a step-size μ of the least squares residual to the current estimation of x_u, where $A^{(u)}$ and $b^{(u)}$ correspond to the rows of A and b available locally on node u. The intermediate value ψ_u is subsequently broadcast to the local neighbourhood D_u. Each node then updates its estimate of x_u with a weighted sum of all received ψ_i ($i \in D_u$), and its own ψ_u, the weights being denoted as $\omega_u(i)$. A proof of convergence and several possible weighting matrices are given in [CS10].

The CTA variant of diffLMS performs exactly the same operations but in a different order. The intermediate values ψ_u are first broadcast to the neighbourhood, then each node computes its estimate of x_u and in the last step the new intermediate value ψ_u. According to [Say14, p.31], *"...the difference between the implementations lies in which variable we choose to correspond to the updated weight estimate."*. In ATC, x_u is the result of the combination step (line 4.5 of ATC), in CTA it is the result of the adaption step (line 4.5 of CTA). However, mathematically and numerically this does not result in the same solution.

5.5.1.2 D-LMS

The D-LMS method is shown in Algorithm 5. A node u first broadcasts its current estimate x_u to its neighbourhood D_u (line 5.3). Then an individual correction vector v_u^i is computed for each neighbour $i \in D_u$ using the received estimation x_i and its own estimation x_u (line 5.5). These values are then sent to each corresponding node i. In the last step of each iteration (line 5.7), the new estimate x_u is computed using a least squares residual from $A^{(u)}$ and $b^{(u)}$, the locally available parts of A and b, and the correction terms v_u^i and v_i^u received from the neighbourhood. This term is added to the current x_u and weighted with a step-size parameter μ resulting in an estimate x_u of the solution x in each node. Proof of convergence is given in [SGRR08].

5.5.2 Comparison of Communication Cost

The cost of a broadcast to all neighbours ("local broadcast") depends on the topology and on the type of connection. Therefore, we introduce the broadcasting parameter $B(d)$ for denoting the number of messages required for broadcasting to d neighbours. In a wireless setting, a single message is required to perform a broadcast to all neighbours, thus $B(d) = 1$. However, in a setting with point-to-point communication (e. g. wired connections), d messages are required for sending a message to d neighbours, thus $B(d) = d$. For a global broadcast beyond the neighbourhood in any network other than a fully connected one, additional messages are needed for multi-hop message relaying over intermediate nodes.

The communication patterns and costs for ATC and CTA are identical. In each iteration, each node u broadcasts a vector of size m to its neighbourhood D_u. In k_1 iterations, node u sends $k_1 B(|D_u|)$ messages. D-LMS requires communication in two of its steps. In line 5.3, a local broadcast is required to distribute the vector x_u of size m to the neighbours. Line 5.6 sends $|D_u|$ individual messages of size m to distribute the correction term. This results in $k_2(B(|D_u|) + |D_u|)$ messages and $k_2(B(|D_u|) + |D_u|)m$ data values sent per node.

Although PSDLS is not an iterative method, we have to consider the number of rounds R required by each push-sum algorithm. Note that in practice R may vary slightly for different push-sum calls due to the randomisation. In the distributed QR decomposition, for the first $m-1$ columns of the matrix A two push-sum calls have to be executed, the first one summing

Algorithm 4 Diffusion Least Mean Square (diffLMS) - ATC and CTA

Input: $A \in \mathbb{R}^{n\times m}$ with $n > m, b \in \mathbb{R}^n$, both distributed row-wise over N nodes
For all nodes u: x_u and ψ_u initialised with zero
Output: $x_u \in \mathbb{R}^m$ on every node

Adapt-then-Combine (ATC)

```
1: in each node $u$ do
2:     while not converged do
3:         $\psi_u \leftarrow x_u + \mu A^{(u)\top}(b^{(u)} - A^{(u)}x_u)$
4:         Broadcast $\psi_u$ to $D_u$
5:         $x_u \leftarrow \omega_u(u)\psi_u + \Sigma_{i\in D_u}\omega_u(i)\psi_i$
6:     end while
```

Combine-then-Adapt (CTA)

```
1: in each node $u$ do
2:     while not converged do
3:         Broadcast $\psi_u$ to $D_u$
4:         $\psi_u \leftarrow \omega_u(u)x_u + \Sigma_{i\in D_u}\omega_u(i)x_i$
5:         $x_u \leftarrow \psi_u + \mu A^{(u)\top}(b^{(u)} - A^{(u)}\psi_u)$
6:     end while
```

Algorithm 5 Distributed Least-Mean Squares Solver (D-LMS)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m, b \in \mathbb{R}^n$, both distributed row-wise over $N$ nodes
  For all $u$ and $\forall i \in D_u$: $x_u$ and $v_u^i$ initialised with zero
Output: $x_u \in \mathbb{R}^m$ on every node
1: in each node $u$ do
2:     while not converged do
3:         Broadcast $x_u$ to $D_u$
4:         for each node $i \in D_u$ do
5:             $v_u^i = v_u^i + \frac{c}{2}(x_u - x_i)$
6:         Send $v_u^i$ to each corresponding node $i \in D_u$
7:         $x_u = x_u + \mu[2{A^{(u)}}^\top(b^{(u)} - A^{(u)}x_u) - \Sigma_{i\in N_u}(v_u^i - v_i^u) - c\Sigma_{i\in N_u}(x_u - x_i)]$
8:     end while
```

Table 5.1: Comparison of the communication cost for diffLMS, D-LMS and PSDLS.

Algorithm	**Number of messages sent per node**	**Total amount of data sent per node**
diffLMS	$k_1\, B(\|D_u\|)$	$k_1\, B(\|D_u\|)\, m$
D-LMS	$k_2\,(\,B(\|D_u\|) + \|D_u\|\,)$	$k_2\,(\,B(\|D_u\|) + \|D_u\|\,)\, m$
PSDLS	$2mR$	$\frac{1}{2}\left(m^2 + 7m\right) R$

scalars and the second one summing vectors. In column l of A, the length of these vectors is $m - l$. For column m only one scalar push-sum call has to be executed. The matrix-vector product $Q^\top b$ requires one more push-sum call on vectors of length m. Consequently, the number of messages sent per node is $2mR$. In each push-sum call, the values *and* a weight have to be transmitted [KDG03].

Table 5.1 summarises the analytical results of this section. We conclude that independently of the number of iterations k_1 and k_2, D-LMS sends $|D_u|$ more messages and more data per iteration than diffLMS. For comparing the communication cost, information about the number of iterations k_1 and k_2 required by diffLMS and D-LMS, respectively, and the number of push-sum rounds R required by PSDLS is necessary. As our simulation results in section 5.6 illustrate, these quantities differ significantly across the three methods.

5.6 Experiments

The simulation results presented in this section demonstrate the different convergence speeds in terms of average number of messages sent per node and therefore provide some qualitative insight into typical values of k_1, k_2 and R for the algorithms compared in this chapter. Our simulations are based on Matlab implementations of the algorithms. The implementation of the push-sum algorithm is round-based and synchronised. The neighbours are selected at random from a uniform distribution. For all methods, A and b are distributed row-wise over all N nodes. Without loss of generality, we consider the special case $n = N$, i. e. each node holds one row of A and one element of b. Like in [CS10], the relative degree weight matrix was used for both diffLMS and D-LMS.

In order to evaluate the accuracy of the approximate solution x_u computed by the algorithms, we evaluated the relative error

$$\max_{u=1,..,N} \|x_u - x^*\|_\infty / \|x^*\|_\infty \quad , \tag{5.2}$$

where x^* is the solution computed sequentially in Matlab.

diffLMS and D-LMS are both iterative methods, whereas PSDLS is a direct method with an iterative building block (the push-sum algorithm) in each step. For a fair comparison of the methods, the instances of the push-sum algorithm in PSDLS were not terminated based on reaching a predefined accuracy, but based on a predefined maximum number of rounds.

The behaviour of diffLMS and D-LMS strongly depends on the choice of the step-size parameter μ. Based on our experience, in particular the convergence speed of diffLMS is very sensitive to the choice of μ, and for bad choices of μ the methods even diverge. The best choice for μ in terms of convergence speed seems to vary greatly with m, the topology and the average node degree. Unfortunately, the literature does not give any guidance on how to choose μ. Thus, we performed extensive simulations across a wide range of values for μ and chose the values at which the respective algorithm eventually achieves the highest accuracy.

Figure 5.1 shows the convergence behaviour of the different algorithms for $N = 64$ nodes arranged in a hypercube and as a random geometric graph on the unit square with a communication radius 0.2. The x-axis shows the average number of messages sent per node and the relative error Equation 5.2 achieved for this number of messages sent per node is plotted on the y-axis. The experiments show that the diffLMS methods do not reach the targeted accuracy of 10^{-8} and after 12000 messages only achieve an accuracy of 10^{-2} on a hypercube. On a random geometric graph diffLMS diverges at around 3100 messages and does not even reach 10^{-1}. On a hypercube network, the D-LMS algorithm achieves an accuracy of 10^{-8}, but requires around 32600 messages to be sent per node. The PSDLS method converges significantly faster than the other algorithms requiring only about 1950 messages per node to reach an accuracy of 10^{-8}, which is a factor of 16 less than D-LMS. The amount of data sent per node is also significantly lower for PSDLS, sending only 5400 values compared to 261000 values sent by D-LMS. Similar behaviour can be observed for the random geometric graph. PSDLS converges more than 7 times faster than D-LMS and sends only 0.05% of the data sent by D-LMS.

5.7 Conclusion

In this chapter, we surveyed existing distributed least squares solvers and classified them based on their communication pattern. We introduced a novel truly distributed least squares

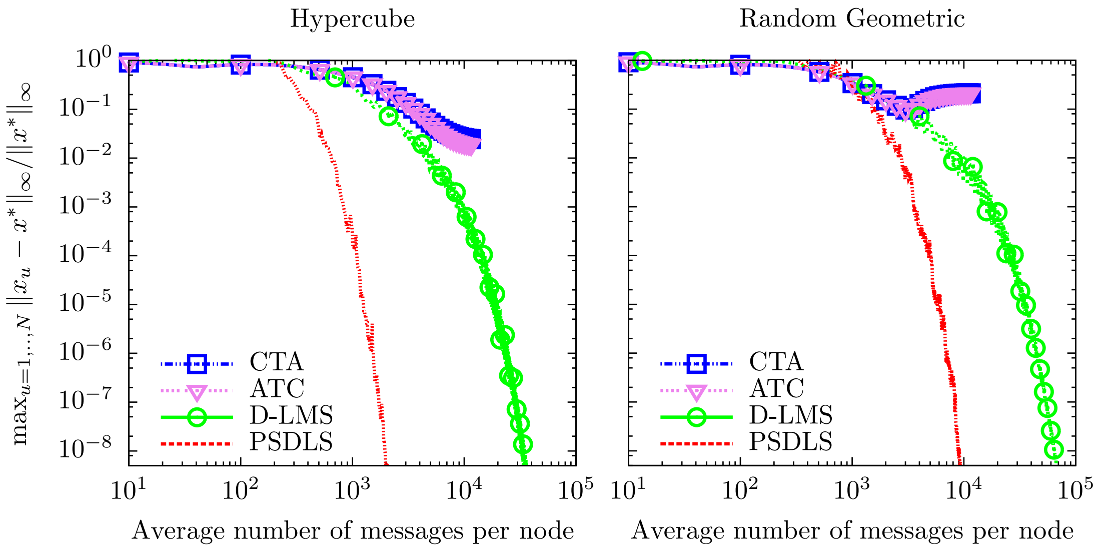


Figure 5.1: Comparison for $N = n = 64, m = 8$ on different topologies: hypercube (left) and random geometric (right). The step-sizes are $\mu_{\text{ATC}} = \mu_{\text{CTA}} = 0.01$ and $\mu_{\text{D-LMS}} = 0.2$.

solver PSDLS based on the push-sum algorithm, which limits the communication to the immediate neighbourhood of each node and does not require a fusion centre or clustering.

We analysed and compared the communication cost of all existing *truly* distributed methods in terms of the number of messages and the amount of data sent per node. Numerical simulations showed that the number of messages per node required for a solution accuracy of 10^{-8} is more than a factor of seven lower for the novel PSDLS algorithm than for the other truly distributed methods.

In the following chapters, we will expand on the PSDLS method by combining the algorithm with iterative refinement. Chapter 6 will focus on the numerical properties of our new algorithm ARPLS-MPIR and present performance evaluations on high-performance systems in terms of execution time, but naturally also in terms of communication cost. In Chapter 7, we will return to the loosely coupled wireless sensor network environment and analyse the potential of our algorithmic changes in a distributed setting. Through the use of iterative refinement, we already introduce one component which improves fault tolerance at the algorithmic level. Using the push-flow algorithm in the distributed environment provides fault tolerance at the communication level. Both fault tolerance aspects will be demonstrated through experiments in Chapter 7.

Chapter 6

Parallel Iterative Refinement Linear Least Squares Solvers based on All-Reduce Operations

We consider the problem of solving the dense linear least squares (LLS) problem

$$\min_x \|b - Ax\|_2 \tag{6.1}$$

in *parallel*, where $A \in \mathbb{R}^{n \times m}$ with $n \geq m$ and $b \in \mathbb{R}^n$. A typical problem in scientific applications is fitting the parameters of a mathematical model to observations which are subject to errors. Performing linear regression analysis on these observations requires efficient LLS solvers. Of special interest are strongly overdetermined LLS problems where the matrix A has many more rows than columns ($n \gg m$), also known as *tall and skinny* LLS problems. Many big data applications naturally exhibit such a strongly rectangular structure, having billions of data points with only a few hundred descriptors. For example, monitoring seismological activity generates massive amount of data. In [WAJR*05], a wireless sensor network with only three nodes was deployed around a volcano and returned millions of rows of sensed data. Tall and skinny problems also arise in partial differential equations [CMM97]. Another field of application where high dimensional regression is required is genetics [LLL*13]. Single nucleotide polymorphisms (SNPs) can help exhibit a human's susceptibility to different diseases. Millions of SNPs are known today, but the number of subjects for a study of a certain disease is often very low, often limited to a few thousand due to high costs.

We present and evaluate the novel *all-reduce parallel least squares* solvers ARPLS-IR and ARPLS-MPIR for solving problem Equation 6.1 which are based on the method of seminormal (SNE) or normal equations (NE) with (mixed precision) iterative refinement (IR). A and b are distributed row-wise across all N processes and the solution $x \in \mathbb{R}^m$ is replicated across the processes. All internode communication in the ARPLS algorithms is contained in all-to-all reduction operations across the participating processes. We consider different variants of the ARPLS algorithm depending on the conditioning of the problem. We show that the application of *mixed precision* iterative refinement (MPIR) in the context of parallel LLS solvers not only reduces the amount of computation but also the communication costs. To the best of our knowledge, the combination of MPIR with LLS solvers has not been studied so far. The mixed precision SNE approach is limited to systems with a condition number up to $\kappa(A) \approx 10^7$ due to single precision being used throughout the majority of the algorithm. In the case of NE, the mixed precision approach is further limited because the normal equations square the condition number of A. Therefore, mixed precision NE does not work for ill-conditioned systems. However, we demonstrate that the accuracy of the standard precision IR method is comparable to existing methods also for higher condition numbers of A. For ill-conditioned systems, unlike some other methods, the approach using SNE with IR (ARPLS-SNE-IR) can still solve the LLS problem and in all cases returns the highest accuracy among the compared algorithms. Moreover, we provide an analysis and comparison of the communication cost of different parallel LLS solvers. Like many standard parallel solvers for dense LLS problems, many ARPLS variants require a parallel QR factorisation algorithm. We thoroughly compare an all-reduce-based parallel version of modified Gram-Schmidt with Tall Skinny QR [DGHL12] in terms of computation and communication cost and show how to optimise the communication cost of the all-reduce-based QR factorisation.

This chapter is organised as follows. section 6.1 summarises related work on parallel LLS solvers. section 6.2 describes the mathematical basis for our approach. section 6.3 introduces and discusses different variants of the ARPLS method and a parallel QR factorisation method based on modified Gram-Schmidt. section 6.4 provides an analysis of the communication cost and a comparison with an LLS solver based on the communication-avoiding QR (CAQR) algorithm [DGHL12] which achieves the theoretical minimum in terms of communication cost. In section 6.5 we summarise numerical experiments conducted on a large scale cluster for comparing the performance of ARPLS methods and the LLS solvers from DPLASMA and ScaLAPACK, followed by a conclusion of this chapter in section 6.6.

6.1 Related Work on Parallel LLS Solvers

Many parallel algorithms for solving LLS problems have been studied in the literature and are available in high-performance libraries like ScaLAPACK [BCC*97], aimed at distributed memory parallel computers, PLASMA [ADD*09], designed for shared-memory multi-core machines, MAGMA [ADD*09], considering heterogeneous and hybrid architectures with multi-core and GPU systems, or DPLASMA [BBD*11], which extends PLASMA to distributed heterogeneous systems. PLASMA and DPLASMA use specialised dynamic scheduling systems (QUARK and PaRSEC, respectively) based on building a direct acyclic graph of parallel tasks and considering the data dependencies of these tasks.

The basic building block of many LLS solvers is a QR factorisation. In PLASMA this is implemented using the tiled QR factorisation algorithm [BLKD08], which divides the matrix into small square tiles instead of using rectangular panels seen in block algorithms. The finer granularity achieved by the square tiles is better suited for multicore architectures [HLAD10]. Demmel et al. [DGHL12] introduced communication-avoiding QR factorisation (CAQR) and proved that its communication cost is optimal up to polylogarithmic factors. The CAQR algorithm factorises block-columns of A, called panels, in parallel using the TSQR algorithm, which is designed for tall and skinny matrices. The panels are divided into block-rows, called domains, which are factorised independently and then merged using a binary tree strategy. In [AFR*14], a systolic QR factorisation algorithm is implemented for a distributed memory machine using the PaRSEC parallel scheduler. The authors target a 3D torus topology and limit the communication of the algorithm to neighbouring nodes, aiming to minimise the amount of communication in the reduction trees.

An example for a parallel LLS solver is the parallel multisplitting method by Renaut [Ren98], which uses the well-known fixed-point iteration methods Jacobi, Gauss-Seidel and successive over-relaxation to solve the LLS problem by forming the normal equations. The matrix A is distributed column-wise over the network nodes and weighting matrices are used to recombine the local problems, which are independent problems resulting from the linear multisplitting of A. In each iteration a vector of size n has to be broadcast to all other nodes in the network.

6.2 Iterative Refinement for Linear Least Squares Problems

Mathematically, our approach for solving problem Equation 6.1 in parallel is either based on the semi-normal equations (SNE) or the normal equations (NE) in combination with iterative refinement (IR).

6.2.1 Normal Equations

The method of normal equations (NE) solves problem Equation 6.1 by forming and solving the normal equations:

$$A^\top Ax = A^\top b \tag{6.2}$$

Assuming A has full rank, the LLS problem has a unique solution. The normal matrix $C := A^\top A \in \mathbb{R}^{m\times m}$ is symmetric and positive definite. Therefore, Equation 6.2 can be solved using the Cholesky factorisation $C = LL^\top$:

$$LL^\top x = A^\top b$$

The main drawback of NE is that the method is not necessarily backward stable [Hig02]. Forming the cross product squares the condition number: $\kappa(A^\top A) = \kappa(A)^2$. The best forward error bound that can be expected is

$$\frac{\|x^* - x\|_2}{\|x^*\|_2} \lesssim nm\kappa(A)^2\varepsilon$$

with x^* being the exact solution and $\varepsilon = b^{1-t}$ being the machine epsilon with b defining the base of the floating-point representation and t the precision. If $\kappa(A) \geq \varepsilon^{-1/2}$, C can be singular or indefinite and the Cholesky factorisation of C will break down. Only if A is well-conditioned, the approach based on the NE is guaranteed to be backward stable.

6.2.2 Semi-Normal Equations

The method of semi-normal equations (SNE) for solving LLS problems is derived from the normal equations using a QR factorisation $A = QR$:

$$A^\top Ax = A^\top b \quad \Leftrightarrow \quad R^\top Rx = A^\top b \quad .$$

Note that the factor Q is not needed to compute the solution, due to $Q^\top Q = I$. For a large and sparse matrix A, storing and accessing Q is often uneconomical and therefore Q is often discarded. Having the original matrix A, it is still possible to solve multiple right-hand sides b with SNE without Q.

The stability of the SNE method for the LLS problem is analysed extensively in [Bjo87]. It has been shown that SNE are not backward stable and that the error in x is similar to the error for the method of normal equations. They have the same numerical properties as the normal equations, even though the factor R from the QR factorisation is of better quality than a Cholesky factorisation of $A^\top A$. The dominating error arises from the rounding errors in the computation of the right hand-side $A^\top b$. However, adding an iterative refinement correction step, as shown in [Gol65] and [Bjo67], leads to much more satisfactory results. As long as A does not have widely differing row norms, the SNE with IR become backward stable under certain conditions which will be discussed in subsubsection 6.2.3.1.

6.2.3 Iterative Refinement

As described in detail in Chapters 2-4, a wide range of variations of IR exist which mainly differ in the precisions used for computing the different steps in the process. The terms *target precision* p_α and *working precision* p_β will be used here to distinguish between the targeted precision of the solution and the precision used for the majority of the computation steps, respectively.

6.2.3.1 Corrected Semi-Normal Equations

For LLS problems, the iterative refinement approach [Gol65], which is also referred to in the literature as *corrected SNE* (CSNE), is defined by

$$\|b - Ax\|_2 = \|r - A\Delta x\|_2$$

where $x = \hat{x} + \Delta x$ with $\hat{x}$ being the initial solution, Δx the correction term and $r = b - A\hat{x}$ is the residual vector of the LLS problem. The least squares residual satisfies $A^\top r = 0$ [Bjo96].

The correction term Δx is itself the solution to a linear least squares problem. The steps of IR for the SNE method are as follows:

1. Factorise $A = QR$
2. Solve $R^\top R\hat{x} = A^\top b$ for $\hat{x}$
3. $x_0 := \hat{x}$
4. For $i = 0, 1, 2, \ldots$
   (a) Compute $r_i = b - Ax_i$
   (b) Solve $R^\top R \Delta x_i = A^\top r_i$ for Δx_i
   (c) Update $x_{i+1} = x_i + \Delta x_i$

In Step 1, a QR factorisation of A is computed and then used in Step 2 to compute an initial solution $\hat{x}$. Subsequently the iterative refinement algorithm tries to increase the accuracy of the solution by computing the residual r_i of the result in Step 4a and using $A^\top r_i$ as the right-hand side to solve the system for the correction term Δx_i in Step 4b using the already available factor R. Finally, the correction term is added to the result to improve the solution of the LLS problem in Step 4c. This process is repeated until the accuracy of the solution is satisfactory. The rate of convergence is shown in [Bjo67] to be roughly linearly dependent on the condition number $\kappa(A)$.

Summarising the analysis in [Bjo87], assuming that R is non-singular, then the bound of the estimate of the absolute error for SNE with a single step of IR is

$$\|x^* - x_i\|_2 \leq \sigma\kappa\varepsilon \left(c_2 \|x^*\|_2 + m^{1/2} n \frac{\|b\|_2}{\|A\|_2} \right) + m^{1/2}\kappa\varepsilon \left(m \|x^*\|_2 + n\kappa \frac{\|r\|_2}{\|A\|_2} \right) + m^{1/2}\varepsilon \|x\|_2$$

with $\kappa := \kappa(A)$, x^* being the exact solution and x_i the solution after the i^{th} refinement step. $\varepsilon = b^{1-t}$ is the machine epsilon with b defining the base of the floating-point representation and t the precision. In this bound, $c_2 = 2m^{1/2}(c_1 + m)$ and $c_1 = c_1(n, m)$ is a polynomial in n and m and depends on the method used to compute the QR factorisation. Moreover,

$$\sigma = c_3\kappa^2\varepsilon, \text{ with } c_3 \leq 2m^{1/2} \left(c_1 + 2m + \frac{n}{2} \right) \quad .$$

The combination of SNE with IR is not in general backward stable, but for $\sigma < 1$ it is more accurate than the QR method (a QR factorisation followed by triangular solve) and less accurate if $\sigma > 1$.

For multiple iterative refinement steps, the error bound of a single step is given in [Bjo87] by

$$\|x^* - x_i\|_2 \leq m^{3/2}\kappa\varepsilon \left(\|x^*\|_2 + \frac{n}{m}\kappa \frac{\|r\|_2}{\|A\|_2} \right) [1 + O(\kappa\varepsilon)]$$

As long as $O(\kappa\varepsilon)$ is negligible compared to 1, this error estimate also holds for further steps of IR. If the refinement converges initially, the limiting accuracy does not depend strongly on the starting vector x_0. Therefore, SNE with IR is backward stable if

$$2c_1 m^{1/2}\kappa\varepsilon < 1 \quad .$$

In [Bjo87], the author also provides an example for the very ill-conditioned Hilbert matrices, where SNE with IR still produces useful results whereas normal equations (without IR) fail completely and SNE without IR is unstable. The results from SNE with IR are shown to be comparable to the QR method. In subsection 6.5.3, we experimentally investigate the numerical stability of SNE with IR.

6.2.3.2 Other Iterative Refinement Approaches for LLS Problems

The method of normal equations (NE) can also be used instead of a QR factorisation. Step 1 is then replaced by forming the normal equations $C := A^\top A$ followed by a Cholesky factorisation $C = LL^\top$. To solve the systems in Steps 2 and 4b, the factor L is used to compute $LL^\top y = A^\top b$, where y is either $\hat{x}$ or Δx_i, respectively. As stated in [Hig02], for NE the rate of convergence depends on $\kappa(A)^2$ instead of $\kappa(A)$.

In [DHRL09], an extra precise IR (EPIR) method for LLS problems is proposed, where the critical parts of computing the residual and updating the solution are performed in extended precision, which refers to using double-double precision with a 106 bit significand for intermediate results. This leads to a reduction of the forward norm-wise and component-wise errors to $O(\varepsilon)$ for the solution x and the residual r. The LLS problem can also be formulated as an augmented linear system of dimension $n + m$, as shown in [Bjo67]:

$$\begin{pmatrix} I_n & A \\ A^\top & 0 \end{pmatrix} \begin{pmatrix} r \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix} \tag{6.3}$$

According to [DHRL09], the analysis for EPIR for linear systems [DHK*06] can be applied directly to the augmented system Equation 6.3 for the LLS problem.

To the best of our knowledge, the application of *mixed precision* IR in the context of LLS solvers has not been studied so far. However, formulating the LLS problem as an augmented linear system Equation 6.3 makes all IR methods developed for linear systems applicable to LLS problems. Therefore, the same proofs of convergence and numerical improvement can be applied to MPIR for LLS problems.

6.3 All-Reduce Parallel LLS Solvers – ARPLS

In this section, we discuss a parallelisation strategy for an LLS solver based on SNE or NE and IR (as summarised in section 6.2). All communication of the resulting algorithms is contained in all-reduce operations of the participating processes, and we therefore call our algorithms *all-reduce parallel least squares* (*ARPLS*) solver. These algorithms comprise three main components: (*i*) a parallel QR factorisation or a parallel matrix multiplication, (*ii*) a parallel matrix-vector multiplication followed by one (without SNE or NE) or two (with SNE or NE) local triangular solves to compute the solution to the LS problem, and (*iii*) IR to stabilise and improve the solution computed in the previous step (this will only be applied to algorithms based on SNE or NE), also requiring a parallel matrix-vector multiplication.

6.3.1 Parallel QR Factorisation

The first component required by most variants of the ARPLS algorithm is a parallel QR factorisation. In this chapter, we will consider two parallel methods, a variation of the distributed modified Gram-Schmidt orthogonalisation (dmGS) [SGZ12] and the Tall Skinny QR (TSQR) [DGHL12].

6.3.1.1 All-reduce Modified Gram-Schmidt

A distributed modified Gram-Schmidt orthogonalisation (dmGS) was presented in [SGZ12] using a gossip-based reduction algorithm. The parallel variant, the *all-reduce modified Gram-Schmidt* (armGS) algorithm, is outlined in Algorithm 6. armGS assumes that the matrix A is distributed row-wise across the processes. The part of A available locally at process u is therefore denoted by $A^{(u)}$. armGS returns the factor Q distributed row-wise (the same distribution as A) and the upper-triangular factor R which is fully available on every process. The armGS algorithm only differs from sequential mGS in the parallel computation of two sums using two reduction operations. The first one, `arsum` in line 3, is a reduction of the local sums to compute the 2-norm of column j and the second one is a parallel matrix-vector multiplication `argemv` of the transpose of column j of Q with A. `argemv` first computes the product using the locally available factors and then forms the sum of the local results using a parallel reduction operation. No additional communication is necessary and the rest of the computations are performed locally.

Algorithm 6 All-reduce Modified Gram-Schmidt (armGS)

```
Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$ distributed row-wise over $N$ processes
Output: $Q \in \mathbb{R}^{n \times m}$ distributed row-wise over $N$ processes, $R \in \mathbb{R}^{m \times m}$ on every process
1: in each process $u$ do
2:     for each column $j = 1..m$ do
3:         $v \leftarrow \texttt{arsum}(\langle {A^{(u)}}^\top(:,j), A^{(u)}(:,j)\rangle)$
4:         $R(j,j) \leftarrow \sqrt{v}$                                  ▷ norm of column $j$
5:         $Q^{(u)}(:,j) \leftarrow A^{(u)}(:,j)/R(j,j)$
6:         $R(j,j+1:m) \leftarrow \texttt{argemv}({Q^{(u)}}^\top(:,j), A^{(u)}(:,j+1:m))$
7:         $A^{(u)}(:,j+1:m) \leftarrow A^{(u)}(:,j+1:m) + Q^{(u)}(:,j)R^\top(j,j+1:m)$
                                                               ▷ rank-1 update on $A^{(u)}$
8:     end for
```

For the SNE, the factor Q is not required. The Q-less mGS approach (denoted as *armGSR* in the following) therefore only returns the full factor R of the QR factorisation and discards the computed columns of Q. This reduces the memory requirements compared to the armGS algorithm by $n(m-1)$ scalars as only one vector of Q of length n is needed for the computation.

Both methods can be further improved in terms of communication cost by postponing the scaling of the column of A by the diagonal element of R after the computation of the second parallel summation in line 6. The first summation of a scalar in line 3 can be combined with the second parallel reduction operation by appending a single value to the vector. This reduces the communication cost from $2m - 1$ to m messages and eliminates the overhead caused by the communication of a scalar value. In the following, we will refer to this method as armGSR-Opt.

6.3.1.2 Tall Skinny QR (TSQR)

The parallel Tall Skinny QR (TSQR) method [DGHL12] is aimed at narrow matrices with $n \gg m$ which are distributed row-wise over all processes. As mentioned before, in the context of semi-normal equations, only the factor R is required. Therefore, we only outline the algorithm for this case and do not consider the computation of Q. However, the Q factor is implicitly represented by the intermediate parts of Q computed along the reduction path and can be reconstructed if needed.

TSQR first computes a local QR factorisation of the locally available rows and then performs a reduction operation of the local R factors (denoted by $R^{(u)}$ at process u) to compute the full R factor of A. For example, if the input matrix A is split into two block-rows $A^{(0)}$ and $A^{(1)}$, the local triangular factors $R^{(u)}$ would be $R^{(0)} = \text{qr}(A^{(0)})$ and $R^{(1)} = \text{qr}(A^{(1)})$. Performing a QR factorisation of the local factors $R^{(u)}$ stacked on top of one another leads to the R factor of the entire matrix A:

$$R = \text{qr}\begin{pmatrix} R^{(0)} \\ R^{(1)} \end{pmatrix}$$

For more than two processes, this approach can be applied recursively to all $R^{(u)}$ to compute the QR factorisation in parallel along a reduction tree. The method is associative and can also be made commutative by ensuring that the diagonal of each computed R factor only contains positive entries and is therefore unique (provided A is non-singular). This can be achieved by multiplying the rows of R having a negative diagonal element with -1. The algorithm is outlined in Algorithm 7.

Algorithm 7 Tall Skinny QR (TSQR)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n \gg m$ distributed row-wise over $N$ processes
Output: $R \in \mathbb{R}^{m\times m}$ on every process
1: in each process $u$ do
2:     $R^{(u)} \leftarrow \mathrm{qr}(A^{(u)})$                    ▷ local QR factorisation
3:     $R \leftarrow \text{all-reduce}(R^{(u)})$
```

TSQR, like armGS, can be implemented using only all-reduce operations, which also replicates the final R factor over all processes. In MPI, the collective operations can be used directly with a user-defined reduction operation, benefiting from the reduction trees available in the MPI implementations, which are usually optimised for the targeted architecture.

TSQR has been shown to be communication optimal [DGHL08], only requiring $O(\log N)$ messages for the reduction operation of R. However, even though the algorithm is communication optimal, in each step of the reduction tree, a QR factorisation has to be computed, which can have a significant performance impact (depending on m). The number of flops is $O(m^3 \log(N))$ along the critical path [ACD*10]. The performance can be improved by exploiting the triangular structure of the stacked matrices and using a custom local QR factorisation. In [DGHL08], the recursive approach of Elmroth and Gustavson [EG00] has been used to achieve a reduction of the number of flops by a factor of 5 compared to a standard QR factorisation with $\frac{10}{3}m^3$ flops.

6.3.2 Parallel LLS Solver

One of the standard methods for solving the LLS problem Equation 6.1 is the use of the QR factorisation to solve the system $Rx = Q^\top b$. The ARPLS-QR algorithm (shown in Algorithm 8) computes the QR factorisation of A in parallel, either using armGS or TSQR, followed by a parallel matrix-vector multiplication `argemv` of ${Q^{(u)}}^\top$ and $b^{(u)}$ in line 3 in Algorithm 8. The final step in ARPLS-QR is the local back substitution using the locally available factor R and the result z of `argemv`. Each process then holds the solution x.

Algorithm 8 All-Reduce Parallel Least Squares Solver based on QR Factorisation (ARPLS-QR)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ processes
Output: $x \in \mathbb{R}^m$ on every process
1: in each process $u$ do
2:     $[Q^{(u)}, R] \leftarrow \mathtt{qr}(A^{(u)})$      ▷ parallel QR factorisation (armGS or TSQR)
3:     $z \leftarrow \mathtt{argemv}({Q^{(u)}}^\top, b^{(u)})$
4:     $x \leftarrow$ solve $Rx = z$                  ▷ local linear system solve
```

For ARPLS-SNE (see Algorithm 9), whose mathematical basis has been reviewed in section 6.2, the process is identical in terms of communication. The Q-less mGS method, armGSR, requires the same amount of computation and communication as armGS, but only returns the factor R. `argemv` is used to compute ${A^{(u)}}^\top b^{(u)}$ in parallel, with ${A^{(u)}}^\top$ having the same row-wise distribution as ${Q^{(u)}}^\top$ in ARPLS-QR. Finally, the system is solved using a forward and back substitution using R and $R^\top$.

Algorithm 9 ARPLS based on Semi-Normal Equations (ARPLS-SNE)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ processes
Output: $x \in \mathbb{R}^m$ on every process
1: in each process $u$ do
2:     $R \leftarrow$ qr($A^{(u)}$)                    ▷ parallel QR factorisation (armGSR or TSQR)
3:     $z \leftarrow$ argemv(${A^{(u)}}^\top$, $b^{(u)}$)
4:     $x \leftarrow$ solve $R^\top Rx = z$                 ▷ local linear system solve
```

As described in subsection 6.2.1, an alternative to solving the LLS problem using the QR factorisation is the computation of the normal equations (NE). ARPLS-NE (shown in Algorithm 10) first computes $C = A^\top A$ in parallel, which only requires a single reduction operation to compute the sum of $O(m^2)$ local values. Therefore, this approach is as communication optimal as TSQR, only requiring $O(\log N)$ messages for the reduction operation of C but performing a much simpler operation than TSQR along the critical path. The following Cholesky factorisation of C is performed locally. The subsequent steps are the same as in ARPLS-SNE. `argemv` is used to compute ${A^{(u)}}^\top b^{(u)}$ in parallel and the system is solved using a local forward and back substitution using the local factor L.

Algorithm 10 ARPLS based on Normal Equations and Cholesky Factorisation (ARPLS-NE)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ processes
Output: $x \in \mathbb{R}^m$ on every process
1: in each process $u$ do
2:     $C \leftarrow$ arsum(${A^{(u)}}^\top \cdot A^{(u)}$)        ▷ parallel computation of the normal equations
3:     $L \leftarrow$ cholesky($C$)                         ▷ local Cholesky factorisation
4:     $z \leftarrow$ argemv(${A^{(u)}}^\top$, $b^{(u)}$)
5:     $x \leftarrow$ solve $LL^\top x = z$                 ▷ local linear system solve
```

6.3.3 Iterative Refinement

We denote our ARPLS solvers using iterative refinement as ARPLS-IR and ARPLS-MPIR (see Algorithm 11). They first compute an initial solution by using ARPLS-SNE or ARPLS-NE and then improve the solution by IR. They compute the residual locally (line 5 in Algorithm 11) and then in parallel apply ${A^{(u)}}^\top$ using `argemv`. Subsequently, a correction term Δx is computed by solving the system using the already computed factor R. Finally, the approximate solution is updated by the correction term. The process continues until a convergence criterion is met.

ARPLS-IR uses the same precision throughout the process ($p_\alpha = p_\beta$). The mixed precision approach ARPLS-MPIR computes the initial solution completely in the lower working precision p_β. Computing the residual and applying ${A^{(u)}}^\top$ to r (lines 5 and 6 in Algorithm 11) have to be computed in p_α. The correction term is computed in the lower precision using the factor R or L, which is only available in p_β. The final step, updating the solution, is again performed in the higher target precision p_α. The majority of the computations, i. e. the factorisation or forming the normal equations, and of the communication are both performed in the lower working precision, leading to improved performance for the local computations and smaller message sizes during the communication.

Algorithm 11 Mixed Precision ARPLS-IR (ARPLS-MPIR)

```
Input: $A \in \mathbb{R}^{n\times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ processes
Output: $x \in \mathbb{R}^m$ on every process
 1: in each process $u$ do
 2:     $[R, x] \leftarrow$ ARPLS-SNE($A^{(u)}, b^{(u)}$)   or   $[L, x] \leftarrow$ ARPLS-NE($A^{(u)}, b^{(u)}$)
                                                              ▷ parallel and local, $p_\beta$
 3:     $i = 0$
 4:     while $i <$ maxiter do
 5:         $r \leftarrow b^{(u)} - A^{(u)}x$                                  ▷ local, $p_\alpha$
 6:         $s \leftarrow$ argemv(${A^{(u)}}^\top, r$)                               ▷ $p_\alpha$
 7:         if $\|s\|_2 <$ tolerance then
 8:             break → converged
 9:         end if
10:         $\Delta x \leftarrow$ solve $R^\top R\Delta x = s$ (ARPLS-SNE)   or   solve $LL^\top \Delta x = s$ (ARPLS-NE)
                                                              ▷ local, $p_\beta$
11:         $x \leftarrow x + \Delta x$                                       ▷ local, $p_\alpha$
12:         $i = i + 1$
13:     end while
```

6.3.4 Combining Iterative Refinement with Other LLS Solvers

One could consider the use of iterative refinement (IR or MPIR) with other LLS solvers. Using the QR factorisation to solve the initial LLS system with the factor Q does not make any difference in the amount of communication compared to SNE and NE. All these approaches require a parallel matrix-vector product for forming $z = {Q^{(u)}}^\top b^{(u)}$ (line 3 in Algorithm 8) or $z = {A^{(u)}}^\top b^{(u)}$ (line 3 in Algorithm 9 and line 4 in Algorithm 10). The local linear system solve is more expensive for ARPLS-IR since it requires a backward *and* forward substitution instead of only one backward substitution in ARPLS-QR. However, those operations are only of order m^2 and therefore have a very low impact on the overall computation time, which is dominated by the QR factorisation or by forming the normal equations, both being of order $O(nm^2)$.

The main advantage of using SNE or NE appears in the iterative refinement. To compute $s = A^\top(b - Ax) = A^\top r$ (line 6 in Algorithm 11), a parallel matrix-vector operation is needed in each iteration of IR. This parallel computation is required by all variants of the LLS solver but at different steps of the algorithm. For SNE or NE, s is required to compute the correction term and at the same time provides the accuracy of the current result to determine if the method has already converged. Not using SNE or NE, this step is only required to check for convergence after the computation of the correction term. To solve the system for Δx, ARPLS-QR would multiply r by $Q^\top$, which would incur an additional parallel matrix-vector operation. This step is not required by the SNE or NE approach in ARPLS-IR because all information needed to solve the system (the R factor from the QR factorisation or the L factor from the Cholesky factorisation) is already available locally and no further communication is necessary. Combining Algorithm 8 with IR would also require more memory for storing $Q \in \mathbb{R}^{n\times m}$ than SNE or NE with IR.

Another disadvantage arises when mixed precision IR is being considered for ARPLS-QR (Algorithm 8). When applying $Q^\top$ to r, the factor Q has to be available in the higher target precision p_α. However, to exploit the potential performance benefits of MPIR, the initial QR factorisation has to be computed in the lower working precision p_β. In this case, the factor Q is not available in p_α and without the SNE or NE, MPIR will not be able to improve the result beyond the lower precision p_β.

Overall, we can conclude that combining SNE or NE with IR leads to the best results in terms of communication and computation compared to other dense LLS solvers.

6.3.5 Extensions of ARPLS-IR and ARPLS-MPIR

All communication in our ARPLS solvers is concentrated on reduction operations. Through the use of fault tolerant reduction operations, the algorithms have the potential to become fault tolerant against silent communication failures, such as message loss. Fault tolerant reduction operations include gossip-based, randomised algorithms like push-flow [GNSSG13], which limit their communication to the immediate neighbourhood of a computing node. This approach is well suited for extreme scale systems and loosely coupled systems (e. g. wireless sensor networks).

In order for parallel iterative refinement to work, it is important that all nodes use the same x to compute the residual in the first step of each IR iteration (line 5 in Algorithm 11). In the case of `MPI_Allreduce`, this is already guaranteed by the MPI standard which requires all processes to receive identical results [Mes12]. For other types of reduction algorithms, for example, gossip-based algorithms, which compute the sum iteratively and therefore produce different approximations of the sum at each node, it is necessary to ensure that each node u has the same approximation x_u, at least to the accuracy targeted for the solution. Otherwise the computation of the correction term Δx will fail. This can either be achieved by averaging the approximate solution vectors over all nodes, accurate to the targeted accuracy, or by selecting one node to distribute its result to all other nodes in the network. For ARPLS-MPIR, if x_u has to be averaged over all nodes, this operation has to be computed in the higher target precision $p_\alpha = DP$.

6.4 Analysis of Communication Cost

In this section, we first analyse the communication cost for the different variants of the ARPLS method (see Table 6.1) and then compare the costs to a parallel LLS solver based on CAQR. Denoting N as the number of processes, a single reduction operation requires about $2 \log N$ messages [TRG05]. For reasons of simplicity, we assume that the number of messages is independent of the size of the data being reduced.

armGS or armGSR can be used by the ARPLS variants having to compute a QR factorisation. Both methods require $2m - 1$ sum reduction operations and send the same amount of data $\frac{m(m+1)-2}{2} + 2m = O(m^2)$ per process. In armGSR-Opt the number of reduction operations is further reduced to m, as described in subsubsection 6.3.1.1. We will append $OGSR$ to the name of the methods using this optimisation. Solving the LLS problem requires one additional reduction operation for the matrix-vector product, resulting in a total of $2m$ reduction operations for armGS and armGSR or $m+1$ reduction operations for armGSR-Opt.

The TSQR method is communication optimal, as shown in [DGHL12], requiring only one reduction operation and therefore being of order $O(\log N)$. It has to be noted, that the reduction operation is not a simple summation, but a more complex and computationally intensive task of a QR factorisation of two $R^{(u)}$ factors at every step of the reduction. Depending on the width of the matrix m, this can have a significant impact on the performance of the algorithm compared to simple sum reduction operations. TSQR reduces the amount of communication by increasing the number of flops by an additional $O(m^3 \log(N))$ along the critical path [ACD*10]. In terms of amount of data, each process sends an upper triangular matrix of size $m \times m$ per process, leading to $m(m + 1)/2$ values.

Forming the normal equations in parallel is also communication optimal, again only requiring one reduction operation and therefore also being of order $O(\log N)$. However, in contrast to TSQR, the reduction operation is much simpler and only has to compute the sum of symmetric matrices. Therefore, only $m(m + 1)/2$ elements have to be sent per process.

Adding iterative refinement slightly increases the communication cost due to the additional sum reduction operation for each iteration in line 6 of Algorithm 11. Compared to an armGS QR factorisation, this increase is negligible, since the number of iterations k is very small for well-conditioned matrices (usually, 2-3 iterations suffice). Each reduction operation sums vectors of length m. MPIR requires the same number of reduction operations as IR, but sends less data and therefore smaller messages for the bulk of the communication performed in the QR factorisation or in the computation of the normal equations because of its use of single precision. This halves the amount of data sent per process in all parallel QR factorisation methods and in the summation of the local symmetric matrices to compute the NE.

The main part of the communication cost originates in the parallel QR factorisation, especially if the modified Gram-Schmidt method is used. The TSQR method, which is only intended for tall and skinny matrices, has been shown to be optimal with $2 \log N$ messages [DGHL12]. Therefore, the communication cost of the communication-avoiding QR

Table 6.1: Theoretical communication cost per process for the different ARPLS methods. m denotes the number of columns in A and k the number of iterations required by iterative refinement.

ARPLS method	**Factorisation**	**No. of reduction operations**	**Total amount of data sent per process**
QR (GS)	armGS	$2m$	$\frac{m(m+1)-2}{2} + 2m$
SNE (GSR)	armGSR	$2m$	$\frac{m(m+1)-2}{2} + 2m$
SNE-IR (GSR)	armGSR	$2m + k$	$\frac{m(m+1)-2}{2} + m(2+k)$
SNE-MPIR (GSR)	armGSR	$2m + k$	$\frac{m(m+1)-2}{4} + m(1+k)$
SNE (OGSR)	armGSR-Opt	$m + 1$	$\frac{m(m+1)-2}{2} + m$
SNE-IR (OGSR)	armGSR-Opt	$m + 1 + k$	$\frac{m(m+1)-2}{2} + m(1+k)$
SNE-MPIR (OGSR)	armGSR-Opt	$m + 1 + k$	$\frac{m(m+1)-2}{4} + m(\frac{1}{2}+k)$
SNE (TSQR)	TSQR	2	$\frac{m(m+1)}{2} + m$
SNE-IR (TSQR)	TSQR	$2 + k$	$\frac{m(m+1)}{2} + m(1+k)$
SNE-MPIR (TSQR)	TSQR	$2 + k$	$\frac{m(m+1)}{4} + m(\frac{1}{2}+k)$
NE	Cholesky	2	$\frac{m(m+1)}{2} + m$
NE-IR	Cholesky	$2 + k$	$\frac{m(m+1)}{2} + m(1+k)$
NE-MPIR	Cholesky	$2 + k$	$\frac{m(m+1)}{4} + m(\frac{1}{2}+k)$

(CAQR) algorithm, which has been shown to be optimal up to polylogarithmic factors, is a reasonable lower bound for the communication cost of a general parallel LLS solver. CAQR sends

$$msg_{\mathrm{CAQR}}(n,m,N) = \frac{1}{4}\sqrt{\frac{mN}{n}}\log^2\frac{nN}{m}\log\left(N\sqrt{\frac{nN}{m}}\right) \tag{6.4}$$

messages and

$$data_{\mathrm{CAQR}}(n,m,N) = \sqrt{\frac{nm^3}{N}}\log N - \frac{1}{4}\sqrt{\frac{m^5}{nN}}\log\frac{mN}{n} \tag{6.5}$$

data per process. For the special case of almost square matrices $n \approx m$, this simplifies to

$$msg_{\mathrm{CAQR}}(n,n,N) = O\left(\sqrt{N}\log^3 N\right) \quad \text{and} \quad data_{\mathrm{CAQR}}(n,n,N) = O\left(\frac{1}{\sqrt{N}}\log(N)\right)$$

Comparing this to the number of messages required by the parallel mGS QR factorisation

$$msg_{\mathrm{armGS}}(n,n,N) = O\left(n\log N\right)$$

reveals that armGS requires a factor $n/\left(\sqrt{N}\log^2(N)\right)$ messages more than CAQR. Considering the amount of data, armGS requires

$$data_{\mathrm{armGS}}(n,n,N) = O(n^2\log N)$$

which is $\sqrt{N}$ more than the communication optimal CAQR method.

The communication cost of CAQR in Equation 6.4 and Equation 6.5 assumes that n and m are sufficiently large in comparison with the block size. For the case $n \gg m$, CAQR is reduced to performing TSQR on a single panel and therefore only requires $2 \log N$ messages. For armGS, the number of messages depends on m and for the optimised version of armGSR requires m reduction operations, which results in $2m \log N$ messages. Compared to TSQR, armGSR sends $m - 1$ more messages. The amount of data sent per message is lower for armGSR, which only sends vectors with a length of up to m scalars per reduction operation. TSQR has to send a triangular matrix which has $m(m+1)/2$ values. However, the total amount of data sent by both methods is identical.

The communication cost is not the only factor that has to be considered. A low number of messages will not guarantee a low runtime, especially if the computation during the reduction operation costs more than the communication itself. The modified Gram-Schmidt orthogonalisation armGSR requires $\frac{2nm^2}{N}$ flops to compute the QR factorisation. During this operation, m reductions are executed computing a sum of up to m elements, an operation costing $m \log N$ flops, resulting in the total computation costs for the reduction of $O(m^2 \log N)$. The TSQR algorithm also requires $\frac{2nm^2}{N}$ flops to compute the initial QR factorisation of the locally available data. The reduction operation then computes QR factorisations at every reduction step, leading to a computation cost of $O(m^3 \log N)$. The computation costs of this reduction are one order of magnitude higher than the total computation costs for the reduction in armGSR. CAQR uses TSQR for the panel factorisations in CAQR. Therefore, CAQR also computes a local QR factorisation with $\frac{2nm^2}{N}$ flops and additionally uses $O(m^3 \log N)$ operations to compute the solution of the QR factorisation, again a factor m more operations than armGSR.

An analysis of the communication and computation costs is given in Figure 6.1, which shows a trace of armGS and TSQR using the VampirTrace library [STIH11]. The green fields represent computational tasks and the red fields display the MPI communication. To illustrate the effect, a wider matrix A was used with $n = 16384$ and $m = 2048$. Both methods are displayed on the same time scale and in this example armGS is about 30% faster. Naturally, for wide matrices one would apply TSQR to panels of A (using CAQR) and achieve a much better performance. Furthermore, TSQR also has the advantage of exploiting the performance of level 3 BLAS operations, whereas armGS is limited to level 2 BLAS. However, this toy example is intended to demonstrate the influence of the synchronisation on the communication time. TSQR only requires a single `MPI_Allreduce`, but during this operation most processes are idle, waiting for the MPI call to complete before they can continue with the next panel or with the solution of the LLS problem. With every merge of two processes to compute the QR factorisation of a stacked matrix, fewer processes are involved in the computation. At the end, only a single process is computing the final result, which is then distributed to all other processes.

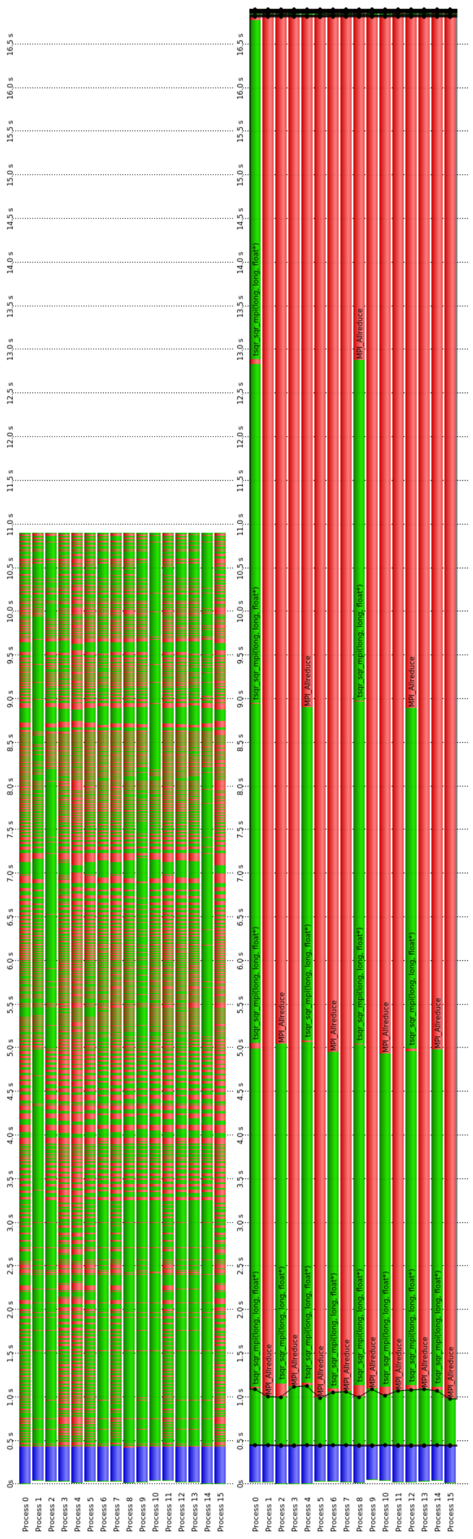


Figure 6.1: Parallel execution of armGS (left) and TSQR (right) on the same time scale for a wider matrix with $n = 16384$ and $m = 2048$. The green blocks represent the computation time, whereas the red blocks show the communication time. The blue blocks are comprised of the initialisation steps (data allocation, matrix generation, MPI initialisation, ...).

6.5 Experiments

In this section, we present performance results for the ARPLS solvers and compare them to state-of-the-art parallel dense LLS solvers.

6.5.1 Experimental Setup

All experiments were run on the Vienna Scientific Cluster VSC-2[4] consisting of 1314 nodes. Each node holds two AMD Opteron 6132HE processors with eight cores each and has 32 GB of main memory. The nodes are connected through Infiniband QDR using a fat tree topology.

The ARPLS variants only use the `MPI_Allreduce` subroutine to perform the parallel summations, which are the only operations which are computed in parallel. For DPLASMA [BBD*11] (version 1.2.1), ScaLAPACK [BCC*97] and the ARPLS variants the MPI library achieving the best performance on the VSC-2 was chosen. DPLASMA achieved the highest performance using OpenMPI, whereas the best performance of `MPI_Allreduce` and therefore of ARPLS was achieved with MVAPICH2 or Intel MPI. In our setup, the `MPI_Allreduce` subroutine in OpenMPI was on average 100 times slower than the one in the other MPI libraries available on the cluster. As ARPLS strongly depends on an efficient implementation of `MPI_Allreduce`, MVAPICH2 was used for the ARPLS algorithms and OpenMPI for DPLASMA. ScaLAPACK performed best using MVAPICH2.

We compare the performance of our approaches with the routine `pdgels` from ScaLAPACK and with the routine `dplasma_dgels` from DPLASMA, since these two routines represent the state-of-the-art in available high-performance implementations of parallel dense LLS solvers. DPLASMA is executed using one process per node, with each node on the VSC-2 having 16 cores available. DPLASMA provides many different parameters to tune its routines for high performance, including various block sizes (tile, supertile and inner blocking), parameters for defining the process grid and the type and size of the high and low-level reduction trees. The high-level trees are specific to the reduction between nodes and the low-level trees take care of the reduction within the nodes. Four types of reduction trees are currently implemented for both levels: a flat tree, a binomial tree, a Fibonacci tree and a greedy tree, which is also the default tree used by DPLASMA. We tested various tile sizes for the different problem sizes and selected the ones delivering the best performance on our test machine. All variants of the reduction trees were also tested using various tree sizes, but a performance increase compared to the default greedy tree was only observed for a single problem size (about 20% performance increase for $n = 2^{22}, m = 16$). In all other cases, any combination of the possible parameters described above had none or a negative effect on the performance. In the following, for DPLASMA we always report the best performance results achieved over many different parameter combinations.

[4] http://vsc.ac.at/systems/vsc-2/

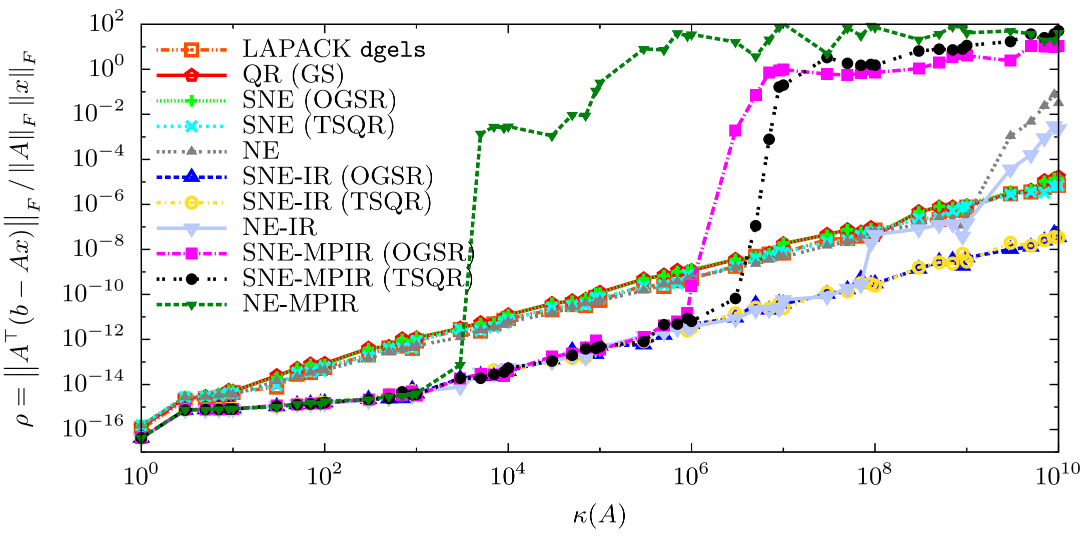


Figure 6.2: Comparison of the achieved accuracy ρ of all ARPLS methods and LAPACK `dgels` for $n = 1024, m = 64$ and different condition numbers κ. The prefix “ARPLS” is omitted from the legend.

6.5.2 Generating Test Matrices

For the experiments, due to the dependence of the NE and mixed precision on the conditioning of the input matrices, we require test matrices with specific condition numbers κ to analyse the accuracy of the algorithms. We consider the condition number with respect to the 2-norm, $\kappa(A) = \sigma_{\max}/\sigma_{\min}$, where σ_i are singular values of A. The procedure for generating our test matrices has already been described in subsection 4.4.1. We use the same algorithm for the experiments in this chapter.

6.5.3 Numerical Accuracy

The accuracy of the result is determined by considering the relative residual

$$\rho = \frac{\|A^\top r\|_F}{\|A\|_F \, \|x\|_F}$$

Figure 6.2 shows the accuracy achieved by the different methods for a set of tall and skinny test matrices generated according to Algorithm 2 with $n = 1024$, $m = 64$ and varying condition numbers $\kappa(A)$. Starting with Figure 6.2, we omit the prefix “ARPLS” from the legend. The LAPACK subroutine `dgels`, which uses a QR factorisation to solve the LLS problem, is also included. ARPLS-QR and ARPLS-SNE result in the same accuracy, showing that the SNE method has no adverse effect on the numerical accuracy of the result. `dgels` achieves the same accuracy as ARPLS-SNE (TSQR), which is just slightly better than ARPLS-QR (GS) and ARPLS-SNE (OGSR). In general, TSQR achieves the same or slightly better results than armGS. Up until $\kappa = 10^9$, ARPLS-NE also achieves the same accuracy as ARPLS-QR (GS) and ARPLS-SNE. For $\kappa > 10^9$, ARPLS-NE can no longer compute an acceptable result due to the loss of accuracy when forming the normal equations.

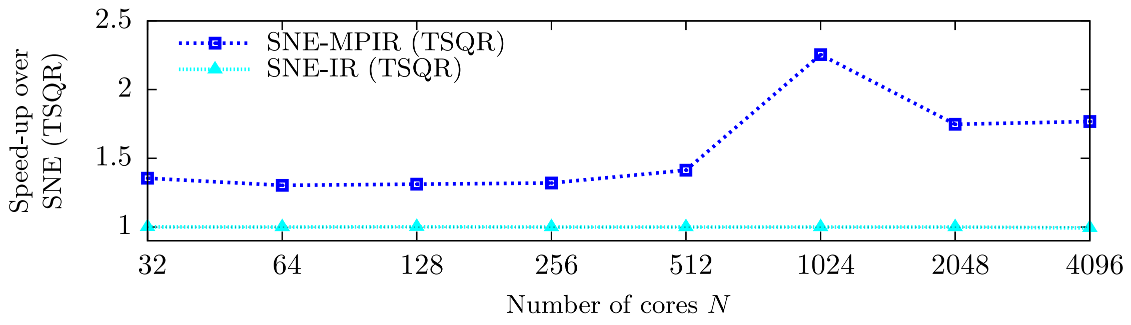
Using IR improves the accuracy in almost all cases and reaches relative residuals which can be almost two orders of magnitude lower compared to the methods without IR. The only exception is ARPLS-NE-IR, which only benefits from IR up until $\kappa = 10^8$. For worse conditioned matrices ($\kappa > 10^8$), ARPLS-NE-IR returns residuals close to ARPLS-NE up until $\kappa = 10^9$ and then fails to compute a correct result.

The improvement of the MPIR variants is limited by the accuracy of the working precision $p_\beta = \text{SP}$. If the QR factorisation computed in p_β does not contain any correct digits then MPIR is not able to improve the result. However, for mildly ill-conditioned systems up until $\kappa \approx 10^6$, ARPLS-SNE-MPIR achieves the same accuracy as ARPLS-SNE-IR. As shown in Equation 4.1, the number of iterations k increases with κ to reach the displayed accuracies. For $\kappa \leq 10^2$ all MPIR variants normally converged after only 2 iterations and for $\kappa \approx 10^5$ ARPLS-SNE-MPIR needed 8 iterations. However, due to the low additional computational complexity of $O(m^2)$ per iteration, the performance of the algorithms is not significantly influenced by the number of iterations. For example, for ARPLS-SNE-MPIR (OGSR) with $n = 2^{22}$ and $m = 256$, an iteration on average only made up 0.005% of the total execution time. Even multiple iterations could be performed at very low cost. When the limiting condition number $\kappa = 2^{23} \approx 8.4 \cdot 10^6$ is reached, the number of iterations grows very fast. As discussed in subsection 4.2.2, the relation p_β / k denotes the number of digits (in bit) which can be improved per iteration. If $k > p_\beta$, the result can no longer be corrected because the improvement per iteration would be less than a bit. In the case of ARPLS-NE-MPIR, forming the normal equations squares the condition number of A and therefore the MPIR method already ceases to achieve an accurate result after $\kappa \approx 10^3$, roughly the square root of the condition number ARPLS-SNE-MPIR is able to handle. Nevertheless, for mildly ill-conditioned systems ARPLS-SNE-MPIR achieves the same accuracy as the double precision IR solvers and additionally benefits from the performance gain due to the use of the lower working precision.

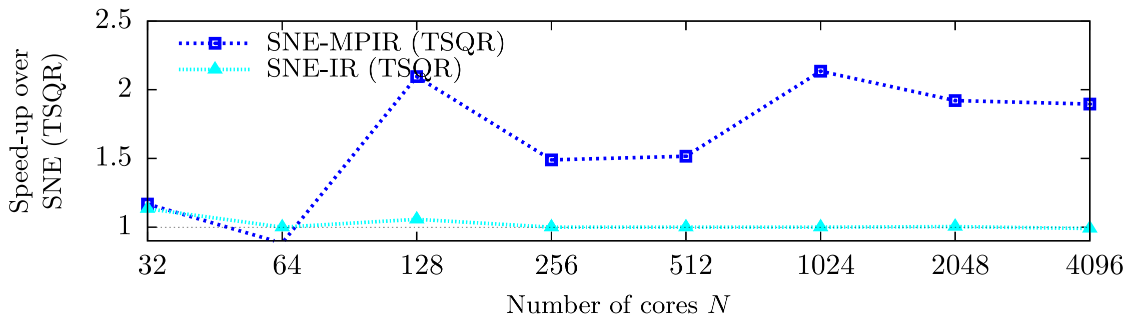
6.5.4 Runtime Performance and Discussion

In this section, we will consider two different test cases: well-conditioned and ill-conditioned matrices. In the former case, we are interested in the performance benefits achieved through the use of mixed precision in the ARPLS-MPIR variants. In the second test case the performance of the ARPLS-IR solvers is investigated for LLS problems with $\kappa(A) = 10^{10}$. As seen in Figure 6.2, all ARPLS-NE and ARPLS-MPIR variants are no longer able to achieve an acceptable accuracy for such ill-conditioned systems and are therefore excluded from these experiments.

Our experiments have shown that the execution times of the ARPLS methods using armGS and armGSR-Opt to compute the QR factorisation are almost always the same for thin matrices. The optimised version achieves more significant speed-ups for wider matrices due to fewer large messages being sent during the reduction operations. Over all our experiments,



(a) Speed-up of the total *execution time*



(b) Speed-up of the *communication time*

Figure 6.3: Speed-up achieved due to mixed precision iterative refinement for $n = 2^{22}, m = 256$, well-conditioned

ARPLS-SNE-MPIR (OGSR) was faster in 73% of the cases. If we account for fluctuations in the measurements and also integrate almost negligible slow-downs at or above 0.97, then ARPLS-SNE-MPIR (OGSR) was faster in over 87% of the cases. We therefore only show results for the methods using armGSR-Opt.

6.5.4.1 Well-conditioned LLS Problems

In the following experiments, A and b are initialised with random values between -1 and 1. The IR and MPIR based algorithms are terminated after reaching $\rho \leq 10^{-15}$ which takes place after 2-3 iterations due to the matrices being well-conditioned when generated this way.

ARPLS-SNE using TSQR to compute the QR factorisation benefits from the usage of mixed precision iterative refinement in ARPLS-SNE-MPIR, as shown in Figure 6.3a. As expected, ARPLS-SNE-IR is slightly slower than ARPLS-SNE due to the additional computations of order $O(n^2)$ required by IR. ARPLS-SNE-MPIR always outperforms ARPLS-SNE and for $n = 2^{22}$ and $m = 256$ achieves speed-ups between 1.3 and 2.3. However, for wider matrices (larger m) the execution time of TSQR increases significantly due to the computation of a QR factorisation at every step of the reduction. Therefore, figures for wide matrices will not include results for any ARPLS methods using TSQR. Focusing only on the communication time, as shown in Figure 6.3b for the same experiments as in Figure 6.3a, ARPLS-SNE-MPIR also benefits from the mixed precision approach and achieves speed-ups between 1.5 and 2.1 for $N \geq 128$ due to the data being sent in the lower working precision (SP).

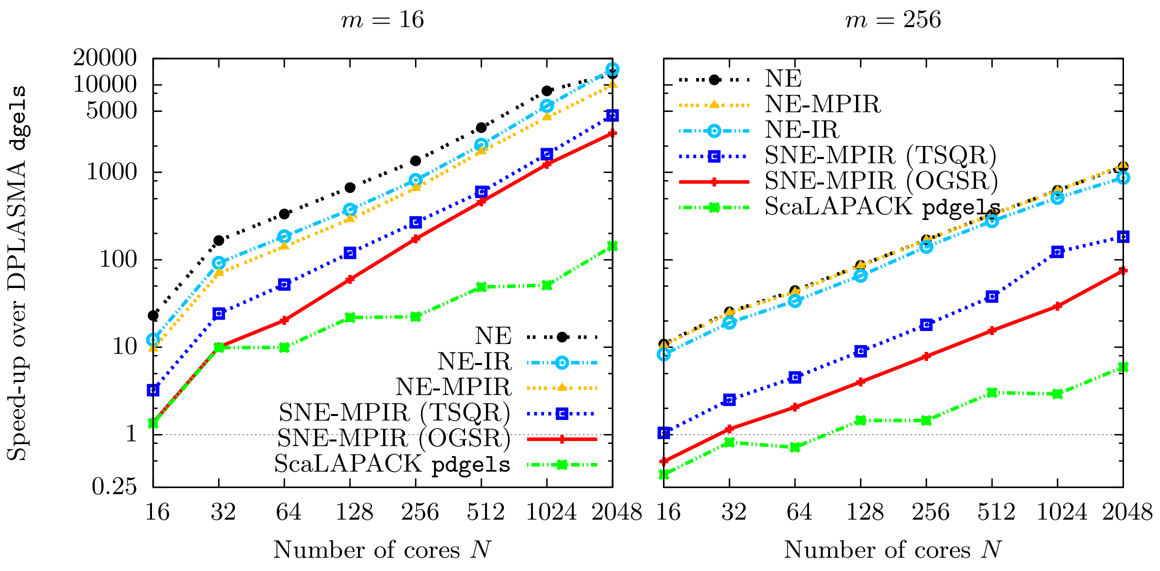


Figure 6.4: Speed-up over DPLASMA for tall and skinny matrices ($n = 2^{22}$, large n/m, well-conditioned)

Figure 6.4 shows the speed-up achieved by different ARPLS algorithms and ScaLAPACK over DPLASMA for very tall and skinny matrices with $n = 2^{22}$ and $m = \{16, 256\}$ for different numbers of cores along the x-axis. For very thin matrices with $m = 16$, ARPLS-SNE-MPIR (OGSR) achieves a speed-up of 2802 and ARPLS-SNE-MPIR (TSQR) of 4477 for the maximum number of cores. For $m = 256$, the speed-ups grow up to 75 for ARPLS-SNE-MPIR (OGSR) and up to 183 ARPLS-SNE-MPIR (TSQR) on 2048 cores. The ARPLS-NE methods outperform the other methods for these well-conditioned matrices, being up to 3.4 times faster than ARPLS-SNE-MPIR (TSQR). For the smaller problem size with $m = 16$, ARPLS-NE-IR and ARPLS-NE-MPIR are slower than ARPLS-NE because the communication dominates their execution time. ARPLS-NE only requires a single all-reduce operation to compute the normal equations, whereas the corresponding IR and MPIR methods additionally require 2-3 iterations to improve the result and have to perform an all-reduce operation in each of those iterations. However, for $m = 256$ ARPLS-NE-MPIR benefits from the use of single precision and achieves the same speed-up as ARPLS-NE (up to 1204 for 2048 cores). ARPLS-NE-IR and ARPLS-NE-MPIR achieve a higher accuracy than ARPLS-NE for these well-conditioned matrices (as seen in Figure 6.2), but for $m = 256$ ARPLS-NE-MPIR is faster than the standard IR method and requires the same runtime as ARPLS-NE. The benefit of ARPLS-NE-MPIR continues to increase with growing m and outperforms ARPLS-NE.

6.5.4.2 Ill-conditioned LLS Problems

The following experiments use worse conditioned matrices by initialising A and b with random values between -1 and 1 and then modifying A to have $\kappa = 10^{10}$ by computing the SVD and scaling the singular values appropriately (as explained in subsection 4.4.1). These experiments were terminated after reaching $\rho \leq 10^{-8}$, which occurred after 3 iterations of

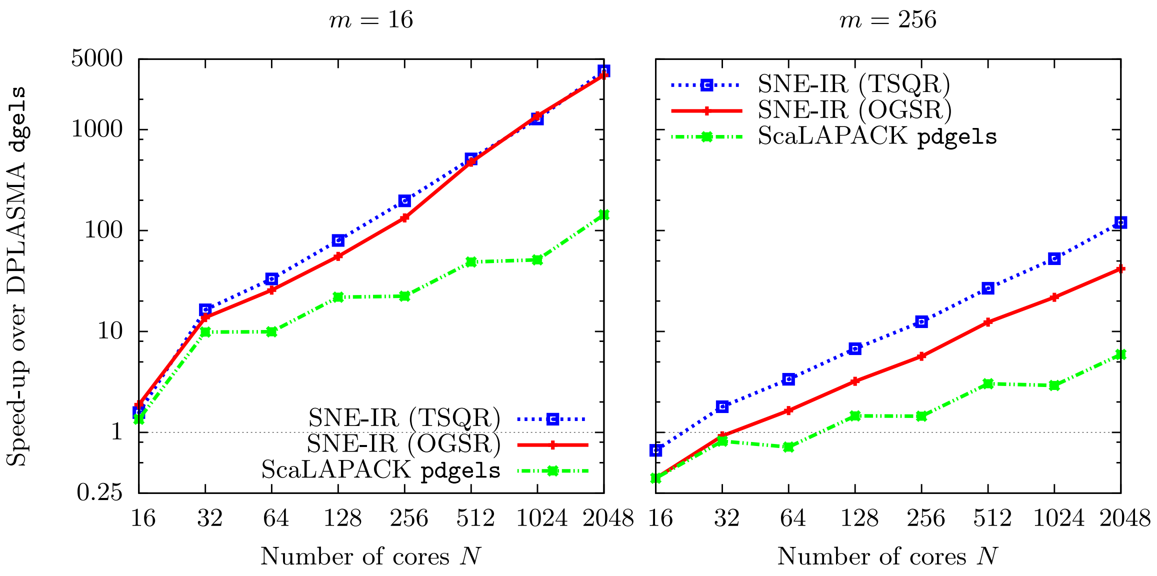


Figure 6.5: Speed-up over DPLASMA for tall and skinny matrices ($n = 2^{22}$, large n/m, $\kappa = 10^{10}$)

standard precision IR. Since these matrices were not well-conditioned, the ARPLS-NE and ARPLS-SNE-MPIR variants could not be used anymore. We therefore focus our attention on the ARPLS-SNE-IR methods.

In Figure 6.5 the speed-ups achieved with ARPLS-SNE-IR (OGSR) and ARPLS-SNE-IR (TSQR) over DPLASMA are shown for very tall and skinny matrices with $n = 2^{22}$ and $m = \{16, 256\}$ for different numbers of cores along the x-axis. For $m = 256$ the speed-up grows steadily reaching 42 for 2048 cores for ARPLS-SNE-IR (OGSR) and 120 for ARPLS-SNE-IR (TSQR). For even thinner matrices with $m = 16$, the speed-up reaches 3468 for ARPLS-SNE-IR (OGSR) and 3820 for ARPLS-SNE-IR (TSQR) for the maximum number of cores tested on the cluster. ScaLAPACK also achieves speed-ups compared to DPLASMA (up to 143 times faster for $m = 16$) but is generally much slower than both ARPLS-SNE-IR variants (up to 26 times for 2048 cores and $m = 16$).

As also stated in [HLAD10], PLASMA or PLASMA-like tiled QR algorithms are more efficient for wide matrices (large m) and are the most efficient for square matrices ($n = m$). These algorithms exploit parallelism to achieve good cache usage and do not perform well on very tall and skinny matrices.

Figures 6.6 and 6.7 show the scaling behaviour of the various algorithms with n (for fixed m). For thin matrices with $m = 256$, ARPLS-SNE-IR (OGSR) again displays faster execution times than DPLASMA, for all n being about 5, 12 and 50 times faster for 64, 256 and 1024 cores, respectively. ARPLS-SNE-IR (TSQR) is slower than ARPLS-SNE-IR (OGSR) for smaller n and but does perform better than ARPLS-SNE-IR (OGSR) for $n \geq 2^{16}$. ScaLAPACK is in general slower than DPLASMA and always slower than ARPLS-SNE-IR (OGSR). Analysing the results for wider matrices, as shown for $m = 4096$ in Figure 6.7, DPLASMA is faster than ARPLS-SNE-IR (OGSR) for 64 cores, but again, due to DPLASMA

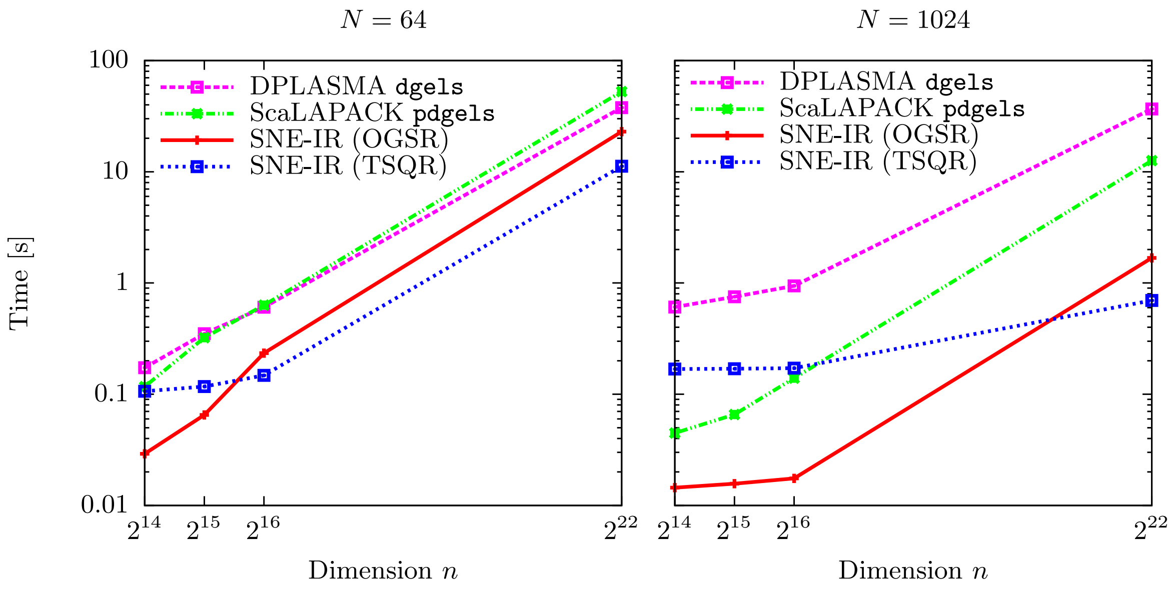


Figure 6.6: Scaling behaviour for $m = 256$ and growing n ($\kappa = 10^{10}$)

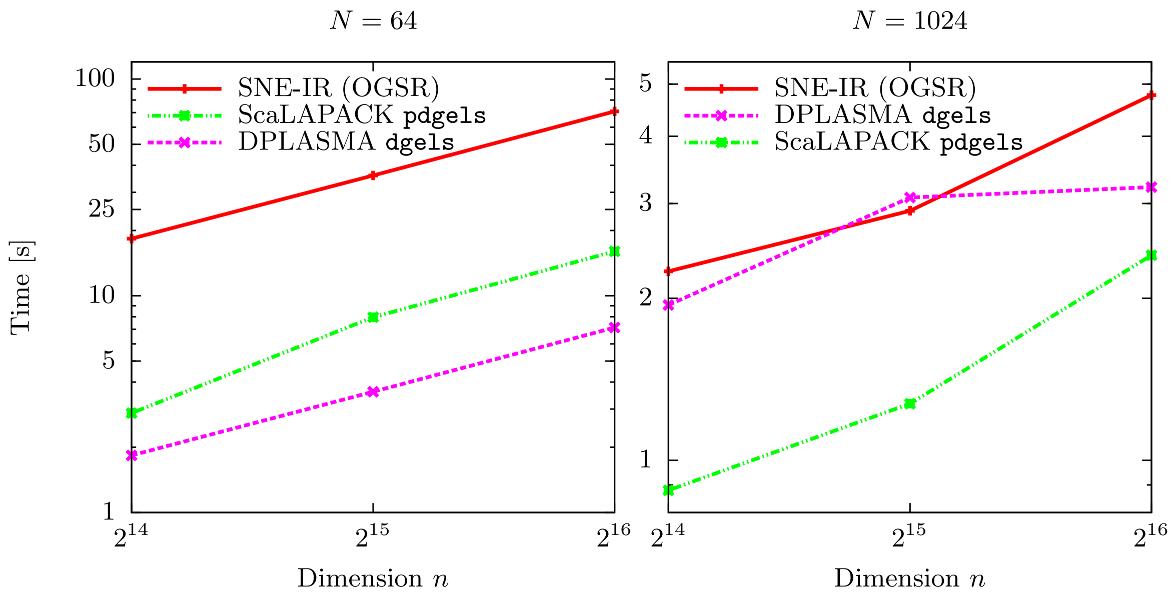


Figure 6.7: Scaling behaviour for $m = 4096$ and growing n ($\kappa = 10^{10}$)

not scaling further than 256 cores for the tested problem sizes, ARPLS-SNE-IR (OGSR) can achieve a higher performance for 1024 cores most of the time. ScaLAPACK is slower than DPLASMA on 64 cores but performs between 1.3 and 2.4 times better on 1024 cores. Compared to ARPLS-SNE-IR (OGSR), ScaLAPACK is faster on 64 and 1024 cores for these wider matrices.

The performance of the tested algorithms for varying number of columns m is shown in Figure 6.8. With increasing m, DPLASMA comes closer to and also overtakes ARPLS-SNE-IR (OGSR). However, for small values of m ARPLS-SNE-IR (OGSR) performs significantly better than DPLASMA and also exploits the available computing cores more efficiently for

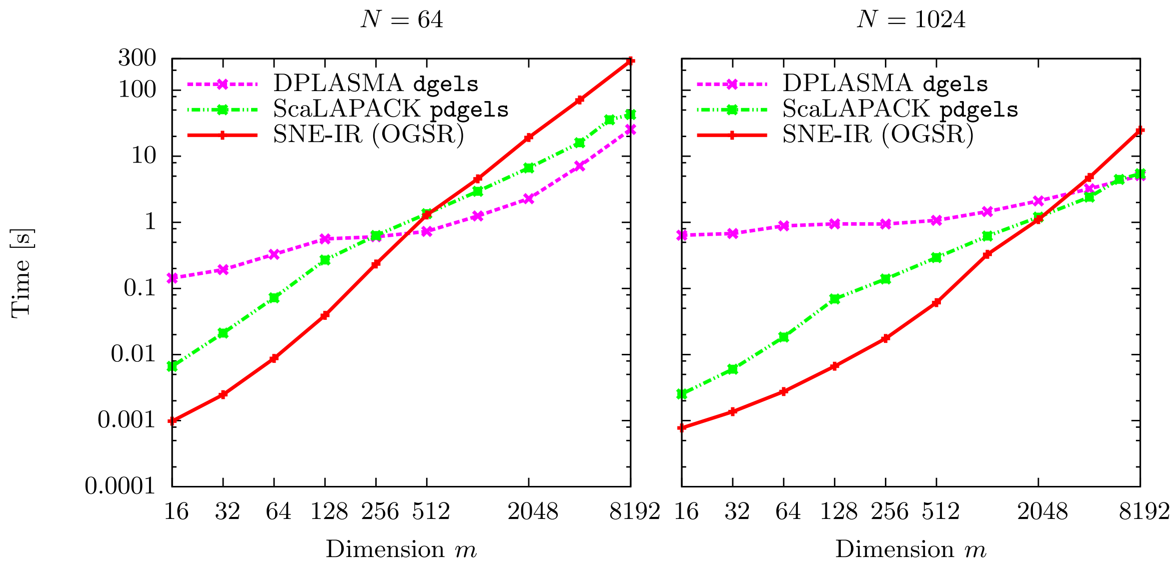


Figure 6.8: Scaling behaviour for $n = 65536$ and growing m ($\kappa = 10^{10}$)

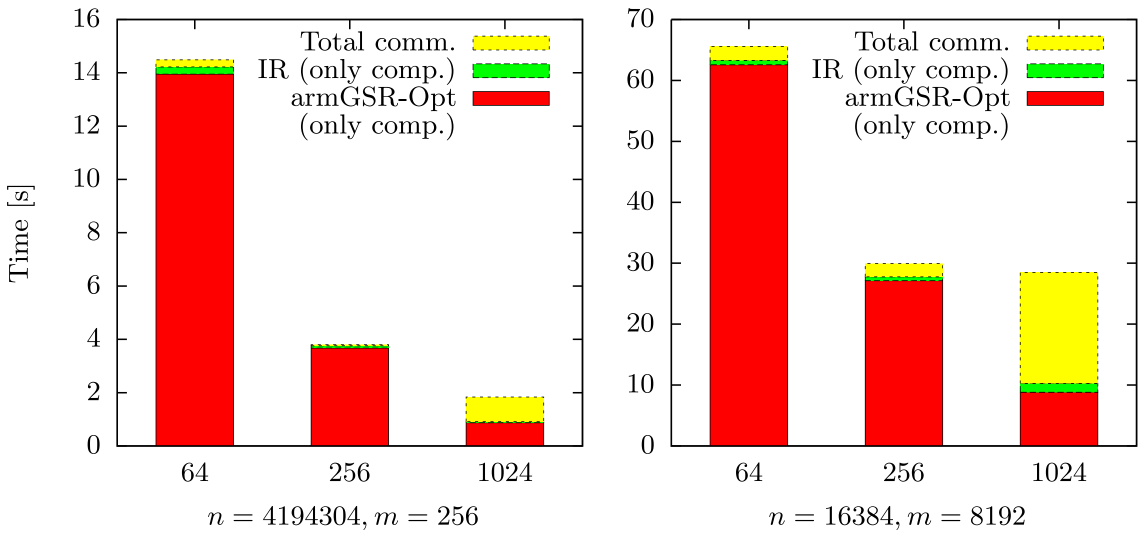


Figure 6.9: Communication and computation time of ARPLS-SNE-MPIR (OGSR) for different numbers of cores. The matrices are well-conditioned and therefore IR only requires 2-3 iterations to achieve the targeted double precision accuracy.

skinny matrices. ARPLS-SNE-IR (OGSR) is also faster than ScaLAPACK up until $m = 2048$ on 1024 cores. Considering wider matrices (large m), ARPLS-SNE-IR (OGSR) is in general slower, but with increasing number of cores catches up with and even overtakes DPLASMA. ARPLS-SNE-IR (OGSR) profits from the increased number of cores and scales well, reaching a speed-up of up to 1.6 over DPLASMA for $n = 65536$ and $m = 2048$ on $N \geq 512$ cores, as can be seen in Figure 6.8 for $N = 1024$.

6.5.5 Communication Cost Analysis

In Figure 6.9, the execution time is shown for the different parts of the ARPLS-SNE-MPIR (OGSR) algorithm. As one can see, the iterative refinement shown as a green bar always only accounts for a very small percentage of the computation time due to its low complexity compared to the QR factorisation. In most cases, armGSR-Opt shown in red makes up for the majority of the execution time. The communication time for both parts shown in yellow depends on the number of columns m. For very skinny matrices it is of course lower than for wider matrices, due to the smaller message sizes. Looking at the communication cost for 1024 cores, it seems that the communication cost suddenly increases and dominates the execution time. However, investigating the communication time more closely and measuring every `MPI_Allreduce` call reveals that only very few `MPI_Allreduce` calls exhibit an above average communication time, leading to a strong increase of the total communication time. The longest `MPI_Allreduce` call for $n = 4194304, m = 256$ took 0.0193 s, whereas the average communication time over all reduction calls is 0.0035 s. For $m = n/2$ on the right side of Figure 6.9, this behaviour is even more significant: on average only 5 out of 8196 `MPI_Allreduce` calls are responsible for almost 40% of the total communication time. These calls occur randomly throughout the algorithm and are also independent of the message size. The average communication time is 0.0022 s with a standard deviation of 0.0326. Therefore, it is fair to assume that this large discrepancy for different `MPI_Allreduce` calls can be accounted to the network infrastructure.

6.6 Conclusion

We presented the parallel linear least squares solvers ARPLS-IR and ARPLS-MPIR which are based on semi-normal or normal equations and (mixed precision) iterative refinement. We compared two different strategies for the parallel computation of the QR factorisation required in this context (armGS and communication optimal TSQR). In the ARPLS solvers, all communication operations between participating processes are contained in all-reduce operations. Consequently, the ARPLS methods directly benefit from all improvements in such reduction operations (e. g. efficient implementation, optimised communication trees, fault tolerance or variants with localised communication).

The theoretical comparison of the communication cost of the relevant parallel QR factorisation methods revealed an asymptotically higher message count of armGS compared to communication optimal TSQR and CAQR. However, numerical experiments on several thousand cores of a high-performance cluster showed competitive runtime performance. We have also shown that the use of *mixed precision* IR in ARPLS-MPIR reduces both, the computation and communication costs. However, mixed precision solvers are limited to matrices with a condition number $\kappa(A) < 10^7$ due to the lower working precision. The experiments confirmed that ARPLS-IR also scales very well with the number of cores and outperforms the parallel dense LLS solvers in the state-of-the-art libraries DPLASMA and ScaLAPACK for several test cases. In particular, for tall and skinny matrices ARPLS-IR exploits the available computing power efficiently and scales very well with increasing processor count as illustrated by our experiments for up to 2048 cores on a high-performance cluster. ARPLS-IR achieves speed-ups up to 3820 on 2048 cores over the state-of-the-art solvers from DPLASMA and up to 26 on 2048 cores over ScaLAPACK for very tall and very skinny matrices. ARPLS-SNE-IR, which uses standard precision IR, achieves better accuracy than all other solvers and the results for $\kappa(A) > 10^8$ are improved by about two magnitudes. In contrast, the normal equation-based solver ARPLS-NE, although faster for well-conditioned problems, fails to compute a correct result for those ill-conditioned systems due to squaring the condition number when forming the cross product.

In the following chapter, we will use the approach developed in this chapter based on the semi-normal equations and normal equations to develop truly distributed algorithms for wireless sensor networks using gossiping as their communication protocol and arbitrary precision iterative refinement to reduce the communication cost and incorporate additional fault tolerant properties.

Chapter 7

Fault-Tolerant Linear Least Squares Solvers for Wireless Sensor Networks based on Gossiping

As already mentioned in Chapter 5, several applications require the distributed solution of a linear least squares (LLS) problem in loosely connected, decentralised sensor networks, e. g. target tracking [Say14], the reconstruction of physical fields [RMG12], localisation [RBTB06] or monitoring volcanic activity and solving the seismic tomography inversion problem [SSX*13]. In a fully decentralised environment, the sensors themselves have to be able to make decisions and can be combined with actuators to interact autonomously with the physical world. Wireless Sensor Networks (WSNs) typically consist of a large number of inexpensive sensor nodes which act autonomously but cooperate with each other to achieve a common goal. In contrast to the high-performance systems targeted in Chapter 6, the resources on typical sensor nodes are normally very restricted, especially their power supply and computational capabilities. Communication is one of the main sources of high power consumption. The energy required for communication is directly proportional to the communication range [SLS*12]. By restricting the communication to the immediate neighbourhood of a node, the power requirements can be reduced significantly, which is not only beneficial to the lifespan of the nodes, but also to the entire network.

Most distributed linear least squares (dLLS) solvers found in the literature require centralised fusion centres, cluster heads or multi-hop communication, all of which cannot be considered *truly distributed*. Multi-hop communication requires routing tables and setting those up requires additional communication. The overhead is particularly large if the routing tables have to be updated frequently to handle mobile nodes or nodes joining or leaving the network.

In this chapter, we propose a truly distributed approach for solving the dLLS problem

$$\min_x \|b - Ax\|_2$$

for $x \in \mathbb{R}^m$, where $A \in \mathbb{R}^{n \times m}$ with $n \geq m$ and $b \in \mathbb{R}^n$. The input data A and b is scattered over all participating nodes. In particular, we focus on situations where A is distributed row-wise over the N nodes of the network and the element $b(i)$ resides on the same node as the i^{th} row of A. For $n > N$, each node contains a block of consecutive rows of A. This distribution also corresponds to the before mentioned applications and many other big data problems, which naturally exhibit this data structure by having significantly more data points (n) than descriptors (m).

Our novel distributed solver GLS-IR (gossip-based least squares solver) for solving the dLLS problem is based on the method of semi-normal equations or normal equations, as discussed in Chapter 6. One of the innovations of GLS-IR is the adaptation of mixed precision iterative refinement to a truly distributed setup. In our algorithm, the communication is limited to the immediate neighbourhood and no fusion centre or multi-hop communication is required. By design, all internode communication in GLS-IR is contained in gossip-based reduction operations across the participating nodes and the solution $x \in \mathbb{R}^m$ is replicated across the nodes.

Gossip algorithms, also known as epidemic algorithms, spread their information by only communicating with the immediate neighbourhood and in each step the nodes randomly choose their communication partners. A prime example of a gossip aggregation algorithm is the push-sum algorithm [KDG03], which can calculate the sum or the average of values distributed over a set of nodes. At each point in time, each node has an estimate of the target solution which will converge to the correct result. Rumour spreading [Pit87] is another application of the gossip protocol where a node distributes information to all the other nodes in the network. It is also based on randomised communication and many variations exist which differ on the approach of distributing the information, either by push or pull operations. Both algorithms, push-sum and rumour spreading, are employed by our novel solver GLS-IR.

A very important factor in the design of a distributed algorithm for WSNs is *distributed fault tolerance*. Gossip-based algorithms already exist to handle some types of faults at the aggregation level. The push-flow algorithm [GNSSG13] is a fault-tolerant alternative to the push-sum method and is able to recover from silent message loss and temporary or permanent link failures. Furthermore, the use of IR itself already provides resilience against faults in the initial solution (see subsection 8.3.2). We will demonstrate experimentally the fault tolerance of GLS-IR based on push-flow against message loss.

In this chapter, we first summarise the related work from Chapter 5 and outline the differences between centralised, clustered and truly distributed approaches, emphasising the advantages of a truly distributed approach. In section 7.2 the mathematical basis for our novel dLLS solver is described, followed by the introduction of the GLS-IR algorithm and its components in section 7.3. section 7.4 provides an analysis of the communication cost and a comparison with existing dLLS solvers. In section 7.5 we present numerical experiments simulated in MPI to compare the performance of GLS-IR and existing dLLS solvers in terms of number of messages Special experiments investigate the fault tolerance properties of our algorithms and also analyse the benefits of using lower working precisions for the majority of the gossip-based reduction operations. Finally, section 7.6 concludes this chapter.

7.1 Summary of the Related Work

This section summarises the classification and the most relevant related work for distributed linear least squares solvers from Chapter 5. Most dLLS solvers found in the literature require centralised fusion centres, cluster heads or multi-hop communication. All these approaches cannot be considered truly distributed, as we will discuss in subsection 7.1.1. In subsection 7.1.2, we will focus our summary of the related work to *truly distributed* LLS solvers. For a more extensive discussion of dLLS solvers found in the literature, including centralised and clustered approaches, please refer to Chapter 5, where we already discussed our first truly distributed push-sum based dLLS solver PSDLS.

7.1.1 Classification

An extensively studied strategy for distributed computations is the **fusion centre approach**, where a central unit performs the computation for the entire network. The fusion centre approach first collects the data from all nodes in the network, then solves a problem at the fusion centre and finally distributes the result to all nodes. Both steps, collecting and distributing the data, require global communication for each node to reach the fusion centre. Multi-hop communication and setting up routing tables incur additional overhead. Challenges also arise with the positioning of the fusion centre which directly affects the communication cost and the scalability of the method (cf. [SSX*13]). Potential congestion effects (particularly around the fusion centre [KGH13]) can lead to delays and in the worst case to data loss. Last, but not least, the fusion centre is a single point of failure.

A first step towards a more decentralised setting than the fusion centre approach is based on **clustering**. The network is divided into clusters. In each cluster, one node acts as the cluster head, which often is a more powerful node than the other nodes in the cluster to handle the higher volume of messages received. Many techniques exist to form clusters, e. g. using the geographical location or setting a communication radius for the cluster head. The cluster heads act as intermediate fusion centres for the clusters. The nodes of a cluster only communicate with their cluster head and with nodes within the same cluster. Compared to the fusion centre approaches, a multi-tier model is used where only the cluster heads communicate with the fusion centre, reducing the communication cost and also the risk of congestion. Although clustering reduces the risk of a single point of failure affecting the entire network, it does not eliminate that risk completely. If a cluster head fails, the complete area covered by the cluster and its data are lost until a new cluster head takes over.

The most decentralised approach, the **truly distributed approach**, is to limit the communication of the nodes to their immediate neighbourhood (defined by their communication range). Each communication partner has to be reachable in a single hop as multi-hop communication would incur additional overhead through routing and thus increase the energy consumption of the resource restricted nodes. A truly distributed approach does not have the limitations of scalability seen in the fusion centre approach. There is no need for more powerful nodes to handle massive amounts of messages and the risk of congestion is limited by the low number of communication partners. A single node failure will not cause the failure of a part or even the entire network. Naturally, a node failure in any scenario will affect the computational results, but methods can be put into place to mitigate or eliminate these effects, as presented in this chapter with the use of fault-tolerant gossip algorithms and iterative refinement to recover from faults.

7.1.2 Truly Distributed LLS Solvers

Sayed et al. [Say14] proposed a diffusion-based least mean square estimator (diffLMS) using normal equations and steepest-descent iterations and limiting the communication to the neighbourhood. The data A and b are both distributed row-wise. The method delivers an estimate of the solution x on each node. However, simulations in section 5.6 have shown, that the accuracy achieved by diffLMS is usually very low.

The distributed least mean squares method (D-LMS) [SGRR08] also only uses neighbourhood communication. The method is based on Lagrange multipliers and uses the least squares residual and the difference between the estimates of x from the neighbourhood in a correction step to compute the least squares solution iteratively. The data distribution of A and b is again row-wise. At each step, an estimate for the solution x is available in each node.

In this chapter, we will provide further experimental results comparing our distributed gossip-based linear least squares solver GLS-IR to PSDLS and D-LMS.

7.2 Iterative Refinement for Least Squares Problems

Our approach for solving the dLLS problem is based on semi-normal equations (SNE) or normal equations (NE) in combination with iterative refinement (IR). The mathematical background and the numerical properties of our approach have already been discussed extensively in section 6.2. Therefore, we will limit this section to important additional remarks related to the distributed environment.

As already mentioned, mixed precision iterative refinement (MPIR) [LLL*06, BDK*08] is a special performance-oriented case of IR where the majority of operations is computed in single precision (SP) and only the critical parts are performed in double precision (DP). Using a lower working precision has many benefits. Processors can perform more lower precision operations per cycle and the data uses less storage, reducing the number of cache misses. In the context of distributed algorithms, as considered in this chapter, the communication cost is reduced by using mixed precision IR. The gossip algorithms require fewer rounds and therefore fewer messages to converge to the lower working precision. The amount of data sent per message is also reduced.

For LLS problems, the IR method [Gol65] is defined by

$$\min_{\Delta x} \|r - A\Delta x\|_2$$

for a given A and the residual vector of the LLS problem $r := b - A\hat{x}$, which satisfies $A^\top r = 0$ [Bjo96], with $\hat{x}$ being the initial approximate solution. In [Bjo67], the rate of convergence is shown to be roughly linearly dependent on the condition number $\kappa(A)$. To the best of our knowledge, the application of *mixed precision* IR in the context of dLLS solvers has not been studied so far. However, formulating the LLS problem as an augmented linear system [Bjo67] makes all IR methods developed for linear systems applicable to LLS problems. Therefore, the same proofs of convergence and numerical improvement apply to MPIR for dLLS problems.

7.3 A Truly Distributed LLS Solver

In this section, we discuss a distributed variant of the LLS solver based on SNE or NE and IR (as summarised in section 7.2). All communication of the resulting algorithm is contained in reduction operations across the participating nodes using gossip algorithms, and we therefore call our algorithm *gossip-based distributed least squares solver with iterative refinement* (*GLS-IR*). Through the use of fault tolerant reduction methods, such as push-flow [GNSSG13], the algorithm becomes resilient against silent communication failures. GLS-IR can directly benefit from any future improvements of the reduction operations, either in performance (e. g. by reducing the number of messages) or in fault tolerance.

GLS-IR consists of three main components: (*i*) a distributed QR factorisation for GLS-IR-SNE or distributed construction of the normal equations for GLS-IR-NE, (*ii*) a distributed matrix-vector multiplication followed by two local triangular solves to compute the solution to the dLLS problem, and (*iii*) IR to stabilise and improve the solution computed in the previous step, requiring a distributed matrix-vector multiplication and rumour spreading.

7.3.1 Distributed QR Factorisation

The first component required for the SNE is a distributed QR factorisation of the matrix A. We use a variation of the distributed modified Gram-Schmidt orthogonalisation (`dmGS`) from [SGZ12] which used push-sum for the reduction operations. `dmGS` only differs from sequential mGS in the distributed computation of two sums, one for the 2-norm of a vector and one for a dot product. No additional communication is necessary and the rest of the computations are performed locally. `dmGS` assumes that the matrix A is distributed row-wise across the computing nodes. Therefore, the part of A available locally at node u will be denoted by $A^{(u)}$. `dmGS` returns the factor Q distributed row-wise, the same distribution as A, and the upper-triangular matrix R which is fully available on every node ($R_u \in \mathbb{R}^{m\times m}$).

Adjustments were made for the use with the SNE and the communication cost was reduced by combining the communication steps in each iteration. For the SNE, the factor Q is not required to solve the LLS problem. The Q-less `dmGS` approach (denoted as `dmGSR` in the following) therefore only returns the full factor R of the QR factorisation and discards the computed columns of Q. This reduces the memory requirements of `dmGS` by $n(m-1)$ floating-point numbers as only one vector of length n instead of a full matrix Q is needed for the computation of R. Both methods, `dmGS` and `dmGSR`, can further be improved in terms of communication by postponing the scaling of the column of A by the diagonal element of R after the computation of the second distributed summation. The first summation of a scalar can be combined with the second distributed reduction operation by appending a single value to the vector. This reduces the communication cost from $2m-1$ to m messages and eliminates the overhead caused by the communication of a scalar value.

7.3.2 Distributed LLS Solver with IR

One of the standard methods for solving the LLS problem is the use of the QR factorisation to solve the linear system $Rx = Q^\top b$. The PSDLS algorithm [PSG14] computes the QR factorisation of A using `dmGS`, followed by a distributed matrix-vector multiplication `dmv` of ${Q^{(u)}}^\top$ and $b^{(u)}$. `dmv` first computes the product of the locally available factors and then forms the sum of the local results using a distributed reduction operation. The final step in PSDLS is the local back substitution using the full locally available factor R_u and the result z_u of `dmv`. Each node u then holds an approximation x_u of the solution x. Using the SNE, the process is identical in terms of communication. The steps for GLS-IR-SNE are shown in Algorithm 12 on lines 2–4. The Q-less mGS method, `dmGSR`, requires the same amount of computation and communication as `dmGS`, but only returns the factor R_u. `dmv` is used to compute ${A^{(u)}}^\top b^{(u)}$, with ${A^{(u)}}^\top$ having the same row-wise distribution as ${Q^{(u)}}^\top$ in PSDLS. Finally, the system is solved by a forward and back substitution using R_u, both operations being performed locally.

For the NE, the algorithm is very similar, the only changes being the factorisation in line 2 and using the Cholesky factor L instead of the QR factor R when solving the systems in lines 4 and 12. `dmGSR` is replaced with a distributed sum, `dsum`, to form the normal equations, followed by a local Cholesky factorisation of the distributed result. `dmGSR` requires m reduction operations (`dsum`), whereas the NE variant requires only a single reduction operation in line 2. Therefore, GLS-IR-NE requires less communication than PSDLS and GLS-IR-SNE, the number of reduction operations being independent of m.

After computing an initial solution x_u, it is improved using IR. The residual is computed locally (line 7 in Algorithm 12) and then ${A^{(u)}}^\top$ is applied using the distributed method `dmv`. Subsequently, a correction term Δx_u is computed in line 12 by solving the system using the already computed factor R_u or L_u. Finally, the solution is updated by the correction term in line 13. The process continues until a convergence criterion is met. GLS-IR uses two different precisions throughout the process, a higher target precision p_α (e. g. 10^{-15} for DP

Algorithm 12 GLS-IR-SNE and GLS-IR-NE

```
Input: $A \in \mathbb{R}^{n \times m}$ with $n > m$, $b \in \mathbb{R}^n$, both distributed row-wise over $N$ nodes
Output: $x \in \mathbb{R}^m$ on every node
 1: in each node $u$ do
 2:     $R_u \leftarrow$ dmGSR($A^{(u)}$) (SNE) or                          ▷ distributed, $p_\beta$
        $L_u \leftarrow$ cholesky(dsum($A^{(u)\top} A^{(u)}$)) (NE)   ▷ distributed sum, local Cholesky, $p_\beta$
 3:     $z_u \leftarrow$ dmv($A^{(u)\top}, b^{(u)}$)                              ▷ distributed, $p_\beta$
 4:     $x_u \leftarrow$ solve $R_u^\top R_u x_u = z_u$ (SNE) or $L_u L_u^\top x_u = z_u$ (NE)   ▷ local, $p_\beta$
 5:     for $i = 1$ : maxiter do
 6:         $x_u \leftarrow$ rumour_spreading($x_u$)                        ▷ distributed, $p_\alpha$
 7:         $r_u \leftarrow b^{(u)} - A^{(u)} x_u$                              ▷ local, $p_\alpha$
 8:         $s_u \leftarrow$ dmv($A^{(u)\top}, r_u$)                          ▷ distributed, $p_\alpha$
 9:         if $\|s_u\|_2 <$ tolerance then
10:             break $\rightarrow$ converged
11:         end if
12:         $\Delta x_u \leftarrow$ solve $R_u^\top R_u \Delta x_u = s_u$ (SNE) or $L_u L_u^\top \Delta x_u = s_u$ (NE)   ▷ local, $p_\beta$
13:         $x_u \leftarrow x_u + \Delta x_u$                                  ▷ local, $p_\alpha$
14:     end for
```

accuracy) and a lower working precision p_β (e. g. 10^{-8} for SP accuracy), leading to a *mixed precision* approach. The choice of the precisions directly affects the number of messages required by the gossip-based reduction operations to reach the requested accuracy. Most gossip-based reductions are performed during the factorisation of A in line 2 and the lower working precision p_β therefore affects the majority of the communication cost. For well-conditioned problems, IR requires very few iterations to converge (on average 2–3 iterations suffice, depending on p_β). Each IR step only requires one gossip-based reduction in line 8 which has to be accurate to the higher target precision p_α. In GLS-IR, the initial solution and the correction term can be computed completely in the lower working precision p_β, whereas computing the residual, applying $A^{(u)\top}$ to r_u and updating the solution requires the higher target precision p_α. Performing the majority of the computations and communication in p_β leads to fewer messages during the reduction operations and improved performance for the local computations.

Gossip-based reduction algorithms produce different approximations of the aggregation result at each node. In order for iterative refinement to work in a distributed setting, it is important that all nodes use the *same* approximation for x_u, at least to the accuracy p_α targeted for the solution, when computing the residual in the first step of each IR iteration (line 7 in Algorithm 12). Otherwise the computation of the correction term Δx will fail. This can either be achieved by computing an average of the approximate solution vectors x_u over all nodes, accurate to the targeted accuracy p_α, or by selecting one node to distribute its value of x_u to all other nodes in the network. In GLS-IR, a rumour spreading method is employed to achieve this condition (line 6 in Algorithm 12).

Table 7.1: Fault tolerance of IR: example of message loss during the initial factorisation.

Initial solution		IR iterations to reach p_α
$p_\beta = 10^{-8}$, $p_\alpha = 10^{-15}$	$\rightarrow$	2 iterations
$p_\beta = 10^{-4}$, $p_\alpha = 10^{-15}$	$\rightarrow$	3-4 iterations
$p_\beta = 10^{-8}$, $p_\alpha = 10^{-15}$ $\xrightarrow{\text{Message loss}}$ $p_\beta = 10^{-4}$, $p_\alpha = 10^{-15}$	$\rightarrow$	3-4 iterations

7.3.3 IR for Fault Tolerance

Aside from stabilising the solution of the semi-normal equations and improving an initial solution computed in a lower working precision, IR is one of the strategies employed in the GLS-IR algorithms to provide fault tolerance. IR is a naturally self-healing algorithm. In each iteration, a correction term Δx_u is computed, allowing for an improvement of a solution by at most $\log_{10} p_\beta$ digits. Before applying IR, the solution can either be of low quality due a low precision initial factorisation, as is the case of the mixed precision approach discussed and applied in this chapter, or due to a fault, e. g. a high amount of silent message loss or silent data corruption (i. e. bit-flips in the memory). Either way, the recovery process from such faults is identical to the initial solution already being computed in a lower precision corresponding to the accuracy of the result after a fault.

For example, consider the case of $p_\beta = 10^{-8}$ and $p_\alpha = 10^{-15}$. In this scenario, a high amount of message loss prevents push-sum to achieve the requested precision of p_β and only reaches 10^{-4}. This corresponds to the fault-free case of $p_\beta = 10^{-4}$. The message loss only affects the number of IR iterations required to reach the target precision p_α, which slightly increases with the reduction of the working precision. The number of iterations is still the same as if the initial problem had chosen $p_\beta = 10^{-4}$ as its input parameter. Therefore, IR can still improve the initial solution starting from 10^{-4} as if p_β was set to this precision initially. The steps of this example are shown in Table 7.1.

In subsection 7.5.2 we demonstrate the healing capabilities of IR if faults occurred during the initial factorisation of A and in subsection 7.5.3 we investigate the effects of different working precisions on the communication cost and the number of IR iterations required to reach the target precision $p_\alpha = 10^{-15}$.

7.4 Communication Cost Analysis

In this section, we analyse the communication cost for GLS-IR and compare the costs to existing distributed least squares solvers (see Table 7.2). For reasons of simplicity, we assume that the number of required messages is independent of the size of the data being transmitted.

PSDLS and GLS-IR-SNE have to compute a QR factorisation. `dmGS` or `dmGSR` both require $2m-1$ sum reduction operations and send the same amount of data $\frac{m(m+1)-2}{2} + 2m = O(m^2)$

Table 7.2: Comparison of the analytical communication cost per node. k and l denote the number of iterations required by IR and D-LMS, respectively. $|D_u|$denotes the number of neighbours of node u and $A \in \mathbb{R}^{n\times m}$.

LS method	**Number of messages sent per node**	**Total amount of data sent per node**
D-LMS [SMG09]	$(\,B(\lvert D_u\rvert) + \lvert D_u\rvert\,)\,l$	$(\,B(\lvert D_u\rvert) + \lvert D_u\rvert\,)\,ml$
PSDLS [PSG14]	$(m+1)R_\alpha$	$(\frac{m(m+1)-2}{2} + m)R_\alpha$
GLS-IR-SNE	$(m+1)R_\beta + R_{RS} + kR_\alpha$	$(\frac{m(m+1)-2}{2} + m)R_\beta + 2mkR_\alpha$
GLS-IR-NE	$2R_\beta + R_{RS} + kR_\alpha$	$(\frac{m(m+1)}{2} + m)R_\beta + 2mkR_\alpha$

scalars per node. In the distributed mGS methods, by postponing the scaling of the column of A by the diagonal element of R and combining the two reduction operations, the number of reduction operations can be further reduced to m. Solving the LLS problem requires one additional reduction operation for the matrix-vector product, resulting in a total of $m+1$ reduction operations for PSDLS and for computing the initial solution in GLS-IR-SNE. GLS-IR-NE only requires a single reduction operation to form the normal equations and one reduction operation for the matrix-vector product to solve the LLS problem. Each reduction operation is performed using a gossip-based aggregation algorithm, which communicates randomly and requires R rounds to reach a requested accuracy. We denote R_α and R_β as the number of rounds required to reach p_α and p_β, respectively. Note that in practice R may vary slightly for different push-sum calls even for reaching the same precision due to the randomised communication schedule. In each push-sum or push-flow call, the values and a weight have to be transmitted [KDG03].

Iterative refinement slightly increases the communication cost due to the additional sum reduction operation for each iteration in line 8 of Algorithm 12 and due to the rumour spreading in line 6. For the rumour spreading of x, we denote the number of rounds as R_{RS}, which includes the number of rounds for sending the rumour (i.e. the data) and the control messages for terminating the rumour spreading process. Compared to the `dmGS` QR factorisation, the communication cost of IR is negligible, since the number of iterations k is normally very small (usually, 2-3 iterations suffice). Each reduction operation forms the sum over vectors of length m. Using a lower working precision p_β, GLS-IR requires fewer rounds to reach p_β and GLS-IR-SNE also sends less data for the bulk of the communication performed in the QR factorisation. The effects of different choices for p_β in relation to p_α will be illustrated in section 7.5.

As shown in Chapter 5, the D-LMS method [SMG09] communicates twice in each iteration. First, a local broadcast to the neighbourhood D_u of a node u is required, distributing the vector x_u of size m to the neighbours. Then D-LMS sends $|D_u|$individual messages of size m to distribute a different correction term to each node in the neighbourhood (one-hop unicast). This results in $(B(|D_u|) + |D_u|)l$ messages and $(B(|D_u|) + |D_u|)ml$ data values sent per node, where $B(d)$ denotes the number of messages required for broadcasting to d

neighbours. In a WSN, $B(d) = 1$.

Comparing PSDLS and the GLS-IR variants analytically, GLS-IR-NE has the lowest communication cost due to the single reduction operation to form the NE instead of m operations for the QR factorisation. Considering that the number of rounds required for gossip-based reduction grows linearly with the logarithm of the accuracy, for $p_\beta = 10^{-8}$ only about half the number of messages are required compared to $p_\beta = 10^{-15}$. In this case, as long as $m \geq 5$, the lower working precision leads to a lower communication cost for GLS-IR-SNE than for PSDLS because the number of IR iterations k to reach p_α is very low (usually about 2). Moreover, the mixed precision approach can also benefit from transmitting smaller floating-point representations for the majority of the communication, leading to a lower communication cost even for (some) $m < 5$.

To compare the communication cost, information about the number of iterations l required by D-LMS and the number of rounds R required by PSDLS and GLS-IR is necessary. As our experiments in section 7.5 illustrate, these quantities differ significantly across the methods and also depend on the network topology.

7.5 Experiments

In this section, we present experimental performance results for our GLS-IR solvers. The simulation experiments are based on MPI implementations of the gossip-based aggregation algorithms push-sum (PS) and push-flow (PF) and were run on the Vienna Scientific Cluster VSC-2[5]. Using MPI implementations allows us to simulate large WSNs without the need of setting up and maintaining hundreds or thousands of physical sensor nodes. The gossip-based aggregation algorithms use asynchronous communication and in each round a node communicates with a single, randomly chosen neighbour. A remaining open question is how to terminate gossip-based algorithms efficiently in a distributed environment. However, this question is beyond the scope of this thesis.

In our experiments, we terminate each gossip-based aggregation once the local approximation reaches a predefined accuracy compared to the exact value. All experiments are averaged over five random geometric topologies, which were generated using the igraph R package [igr16]. The diameter used to generate the topologies and the resulting average node degrees are shown in Table 7.3 and examples of the generated topologies are shown in Figure 7.6. The transmission radius r was chosen as $\sqrt{\log N/N}$, which leads to the vertex degree growing logarithmically in the number of nodes N [Pen03]. The maximum number of iterations for IR was set to 10, but this upper limit was never reached in the fault-free experiments. In all experiments, well-conditioned matrices were used and $\left\|A^\top r\right\|_2 \leq p_\alpha = 10^{-15}$ was achieved at termination. The convergence of D-LMS [SMG09] depends on a problem dependent step-size parameter μ which has a vast search space. The parameter μ is highly dependent on the input data, the network size and topology. Even for the same input size and

[5]http://vsc.ac.at/systems/vsc-2/

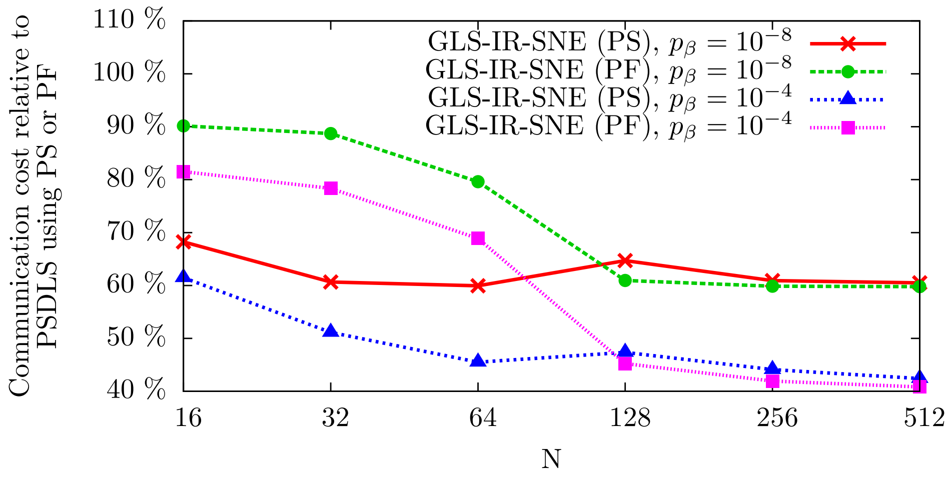
Table 7.3: Properties of the random geometric topologies

N	Diameter d	Average node degree
16	0.411497	7.07
32	0.325317	7.87
64	0.251989	10.18
128	0.192460	12.88
256	0.145486	15.33
512	0.109114	17.98

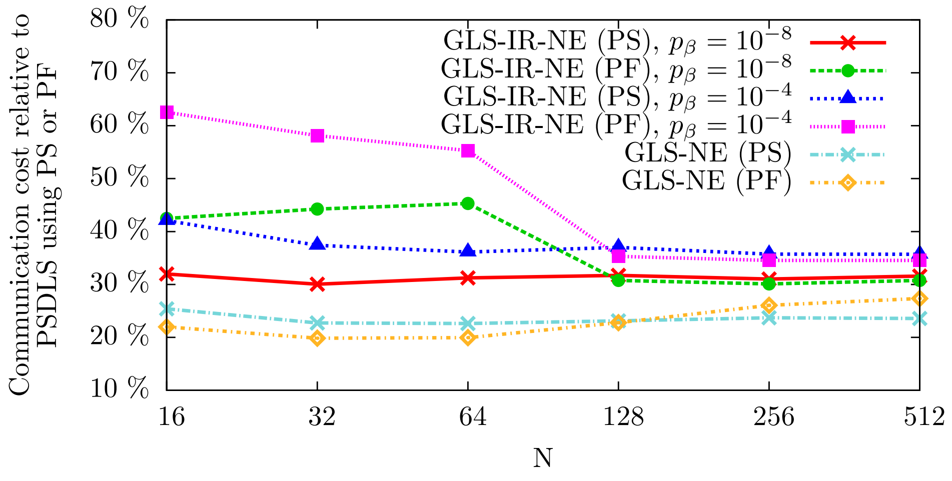
a matrix with the same condition number the step-size varies greatly. In order to achieve a fair comparison of the methods, we determined the best value for μ for each set of input data A and b, for each number of processors and for each topology. In the following figures, we report the average minimum number of messages required by D-LMS to reach the targeted accuracy p_α for each combination of the input parameters n, m, N and the topology.

7.5.1 Communication Cost

Figure 7.1 shows the communication cost, i. e. the average total number of messages sent per node, relative to PSDLS for different aggregation methods and different working precisions p_β to reach an accuracy of $p_\alpha = 10^{-15}$. Each node holds one row of A ($n = N$) and solves the dLLS problem for a skinny matrix with $m = 8$. For a larger choice of m, the GLS-IR methods achieve even higher improvements compared to PSDLS. `dmGS` has to compute m columns, which corresponds to m gossip-based aggregations. Reducing the communication cost for the QR factorisation through the use of lower working precisions becomes even more significant for larger m, while the IR costs are independent of m. However, due to the limited CPU resources on the VSC2, we limit our experiments to skinny matrices. On average, 2-3 iterations were necessary for IR to converge to the desired accuracy. All variants of GLS-IR need significantly fewer messages than PSDLS. Two different working precisions p_β are used, 10^{-8} and 10^{-4}. In all cases, IR achieves a final accuracy of $p_\alpha = 10^{-15}$, but GLS-IR-SNE with $p_\beta = 10^{-4}$ requires fewer rounds than PSDLS for both PS and PF. Using $p_\beta = 10^{-8}$, for large N GLS-IR-SNE requires about 60% of the messages of PSDLS for PS and PF. For $p_\beta = 10^{-4}$, the message count is further reduced to only 42%. This shows the significant advantage of IR and lower working precisions while still achieving the same accuracy as a solver without IR. GLS-NE without IR is the fastest method requiring about 20% of the messages of PSDLS for large N. For GLS-IR-NE, the communication cost of IR cannot be compensated by using lower working precisions to form the NE, but for growing N the overhead is less than 7% higher than GLS-NE (PF). This is a very low overhead while providing fault tolerance through PF *and* IR. In the case of message loss, the accuracy of the initial factorisation is reduced because PS is then unable to reach the required working precision p_β. This corresponds to a fault-free situation where the initial factorisation was



(a) Communication cost for GLS-IR-SNE

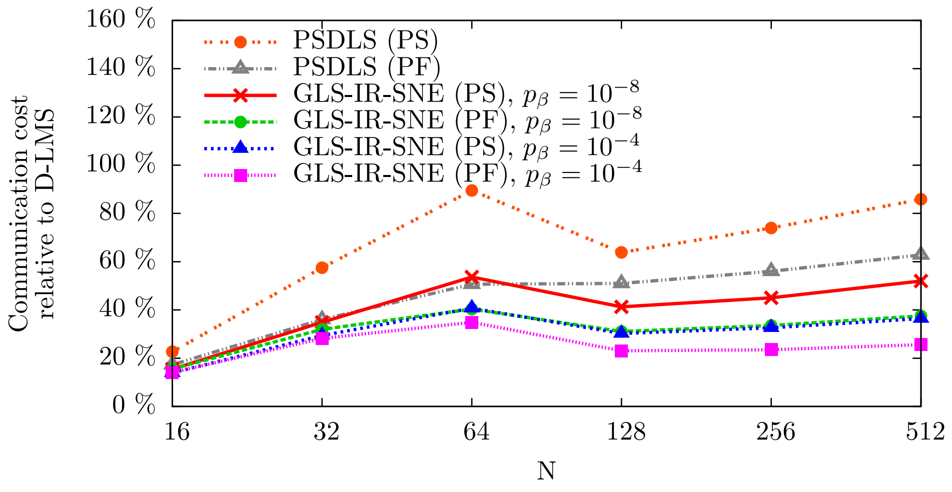


(b) Communication cost for GLS-IR-NE

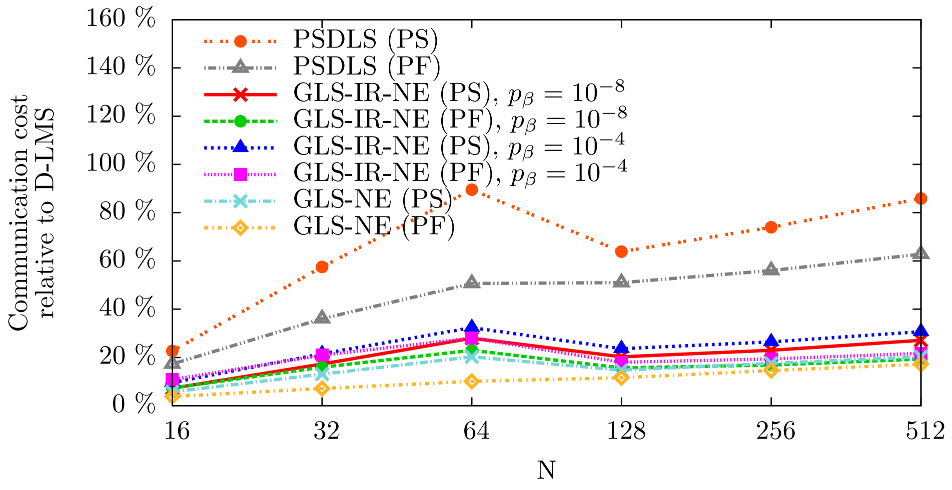
Figure 7.1: Communication cost for GLS-IR relative to PSDLS using PS or PF, respectively, on random geometric topologies for $n = N$, $m = 8$ and $p_\alpha = 10^{-15}$.

already computed in a working precision lower than p_β (e. g. 10^{-4} instead of 10^{-8}). IR can then improve the initial solution to the target precision p_α in exactly the same way as if the initial factorisation had already been computed in a lower p_β.

In Figure 7.2, the communication cost for PSDLS and all GLS-IR methods is shown relative to D-LMS. PSDLS using PS requires on average 75% of the messages used by D-LMS, but PSDLS using PF already comes close to GLS-IR-SNE using PS and $p_\beta = 10^{-8}$ with less than 62% and 52% of the messages used by D-LMS, respectively. GLS-IR-SNE using PF and $p_\beta = 10^{-8}$ and GLS-IR-SNE using PS and $p_\beta = 10^{-4}$ almost require the same amount of messages to converge, averaging on a third (34%) of the messages used by D-LMS. The best SNE method in Figure 7.2a is GLS-IR-SNE using PF and $p_\beta = 10^{-4}$ requiring only a quarter of the messages used by D-LMS to reach the target accuracy p_α. Figure 7.2b shows the communication cost of the GLS-IR-NE methods relative to D-LMS, almost all of which require fewer messages than the GLS-IR-SNE methods. Again, the algorithms using PF as



(a) Communication cost for GLS-IR-SNE



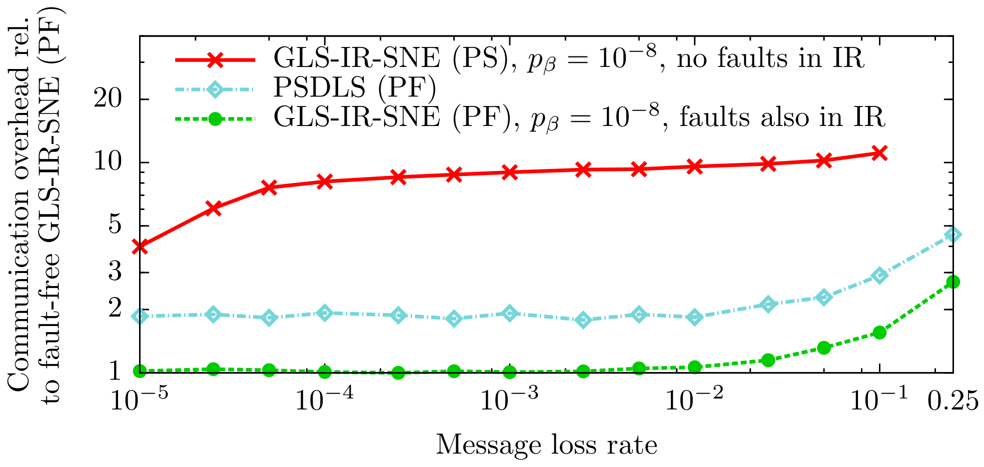
(b) Communication cost for GLS-IR-NE

Figure 7.2: Communication cost for PSDLS and GLS-IR relative to D-LMS on random geometric topologies for $n = N$, $m = 8$ and $p_\alpha = 10^{-15}$.

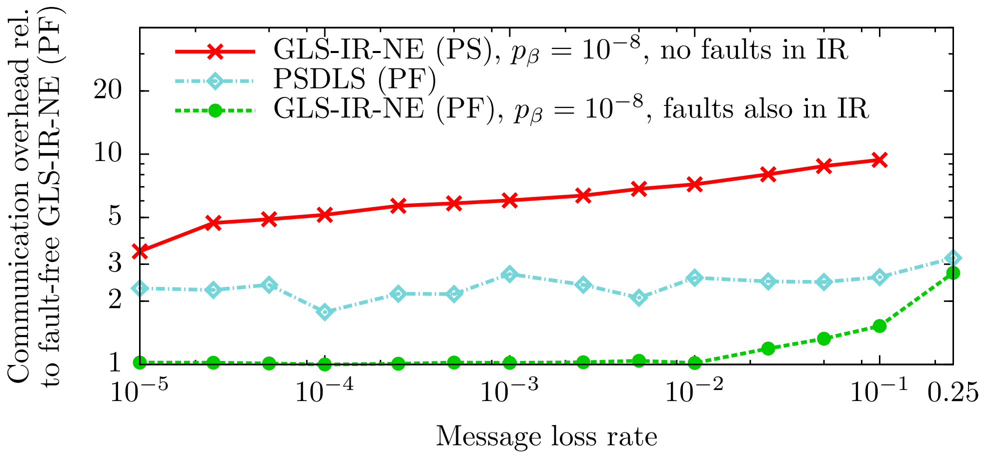
their aggregation method are always lower than their PS-based counterparts. GLS-NE using PF achieves the best performance compared to D-LMS, only requiring on average 15% of the messages to reach the target accuracy, being more than 6 times faster than D-LMS.

7.5.2 Fault Tolerance

To examine the fault tolerance properties of the GLS-IR solvers, a message loss probability was introduced. In these experiments, we focus on lost messages, but temporary and permanent link failures could also be modelled as a continuous message loss. In our simulation, for each received message it is randomly decided if it is processed or discarded. We tested various message loss probabilities between 10^{-5} and 0.25 for $N = 128$, again on 5 different random geometric topologies. m was fixed at 8 and the target precision p_α was again set to 10^{-15}. The maximum number of rounds per gossip-based aggregation was limited to 10000 to ensure termination even in the event of failures, which could cause the aggregation method to not



(a) Communication overhead relative to fault-free GLS-IR-SNE (PF)



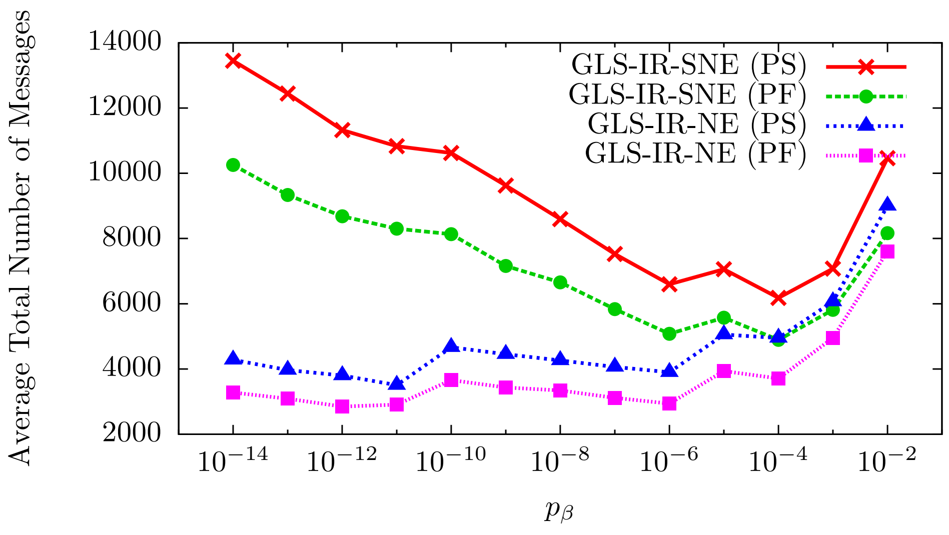
(b) Communication overhead relative to fault-free GLS-IR-NE (PF)

Figure 7.3: Communication overhead with $p_\beta = 10^{-8}$ for increasing message loss probability for $n = N = 128$, $m = 8$ and $p_\alpha = 10^{-15}$ on random geometric topologies.

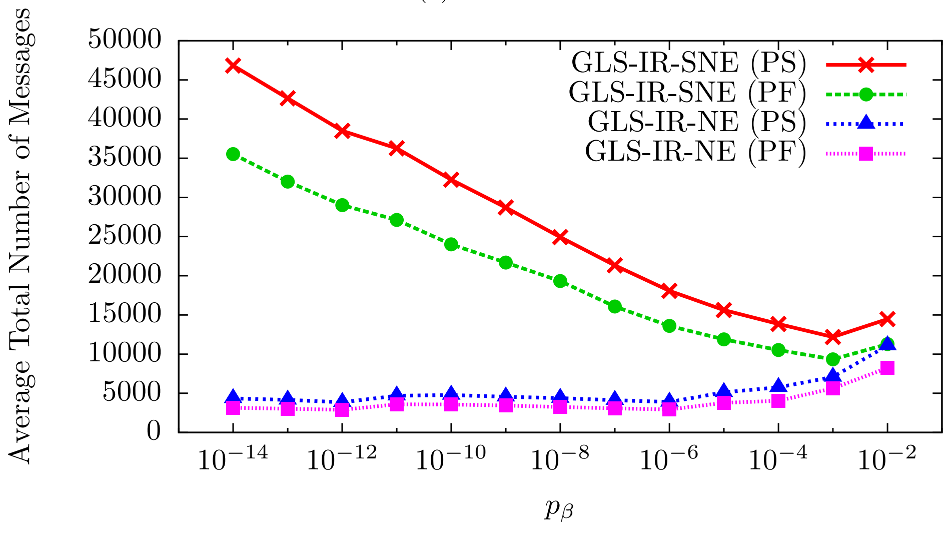
be able to converge to the prescribed accuracy. The maximum number of IR iterations was increased to 100 to tolerate slow improvements due to large errors in the initial computation. The results for GLS-IR-SNE are shown in Figure 7.3a and for GLS-IR-NE in Figure 7.3b.

PSDLS using the non-fault-tolerant PS is not able to handle any message loss, leading to an accuracy of less than 10^{-2} for the lowest message loss probability of 10^{-5} and decreasing for higher loss probabilities. GLS-IR-SNE and GLS-IR-NE using PS as their aggregation method also cannot achieve any meaningful results. However, restricting the message loss only to the initial solution (lines 2-4 in Algorithm 12) and running IR without dropping any messages allows GLS-IR-SNE and GLS-IR-NE with PS to achieve an accurate result. The number of IR iterations required to reach the desired target accuracy increases with the message loss probability, but only for the most extreme loss probability of 0.25 IR failed to improve the solution.

The fault-tolerant PF in combination with PSDLS or the GLS-IR methods can handle any message loss probabilities within the tested range. For the GLS-IR methods using PF,



(a) $m = 8$



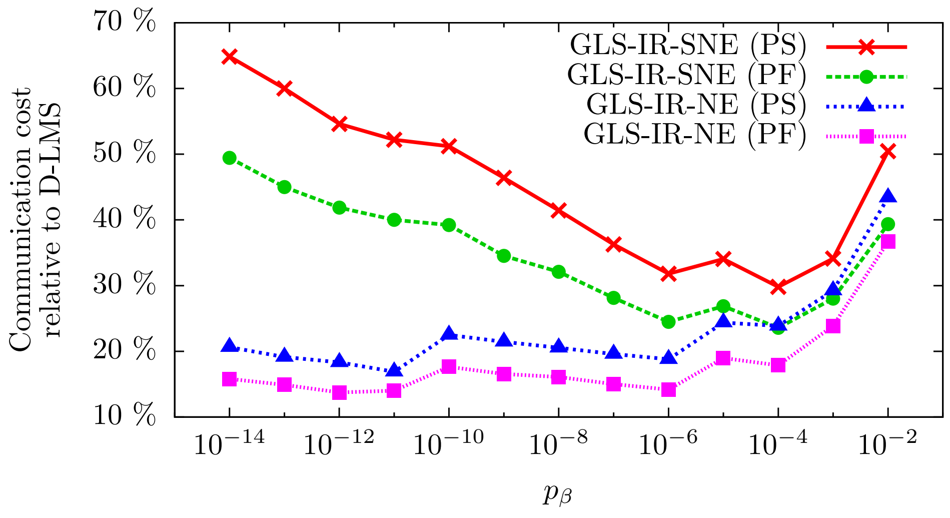
(b) $m = 32$

Figure 7.4: Average number of messages per node for GLS-IR for various working precisions p_β and different m with $n = N = 128$ and $p_\alpha = 10^{-15}$ on random geometric topologies.

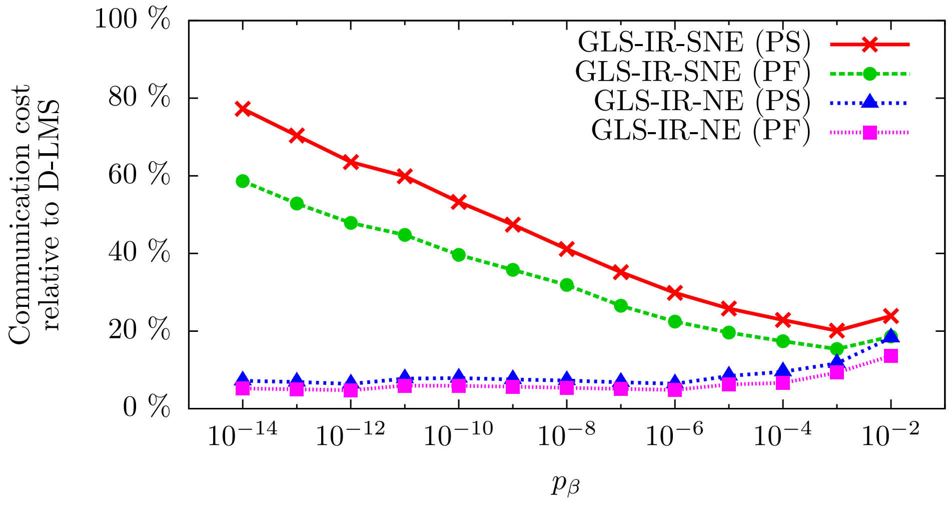
the number of IR iterations does not increase and the method converges to a solution accurate to 10^{-15} within only 2 IR iterations. The communication cost remained almost the same for all three methods using PF for message loss probabilities up to 10^{-2} in comparison to their fault-free runs. Only for very high message loss probabilities, PF required more messages to converge and for a message loss probability of 0.25, which on average corresponds to loosing every fourth message, the number of messages required to reach an accuracy of 10^{-15} tripled.

7.5.3 Working Precisions

In subsection 7.5.1 we compared the communication cost for two specific working precisions p_β, 10^{-8} and 10^{-4}. In this section, we analyse the range of working precisions $p_\beta = 10^i$ with $i = -14, \dots, -2$ and examine the reduction of the communication cost depending on the working precision used. The target precision p_α was set to 10^{-15} and was reached in all cases. The experiments were again run on 5 different random geometric topologies with $N = 128$ and fixed $m = 8$ (Figure 7.4a) or $m = 32$ (Figure 7.4b).



(a) $m = 8$



(b) $m = 32$

Figure 7.5: Communication cost for GLS-IR relative to D-LMS for various working precisions p_β and different m with $n = N = 128$ and $p_\alpha = 10^{-15}$ on random geometric topologies.

In both cases, the total number of messages sent by GLS-IR-NE remains almost the same up until $p_\beta = 10^{-5}$. Only for the lowest working precisions, the number of messages increases up to a factor of 2 in the case of $m = 8$ and 2.5 for $m = 32$ compared to GLS-IR-NE using the same precision throughout the calculation ($p_\beta = p_\alpha$). For GLS-IR-SNE using PS or PF, decreasing the working precision p_β continuously decreases the number of messages required to reach the target precision p_α. For $m = 8$, GLS-IR-SNE requires the least number of messages to reach p_α for $p_\beta = 10^{-4}$ and only uses 40% of the messages compared to the same algorithm using $p_\beta = p_\alpha = 10^{-15}$. In the second case $m = 32$, an even further reduction can be observed for GLS-IR-SNE. To reach p_α using $p_\beta = 10^{-3}$, GLS-IR-SNE using PS or PF requires less than 25% of the messages compared to the case of $p_\beta = p_\alpha$. In all cases shown in Figure 7.4, IR required 1-2 iterations for $p_\beta \in [10^{-14}, 10^{-6}]$, 3-5 iterations for $p_\beta \in [10^{-5}, 10^{-3}]$ and 7-8 iterations for $p_\beta = 10^{-2}$.

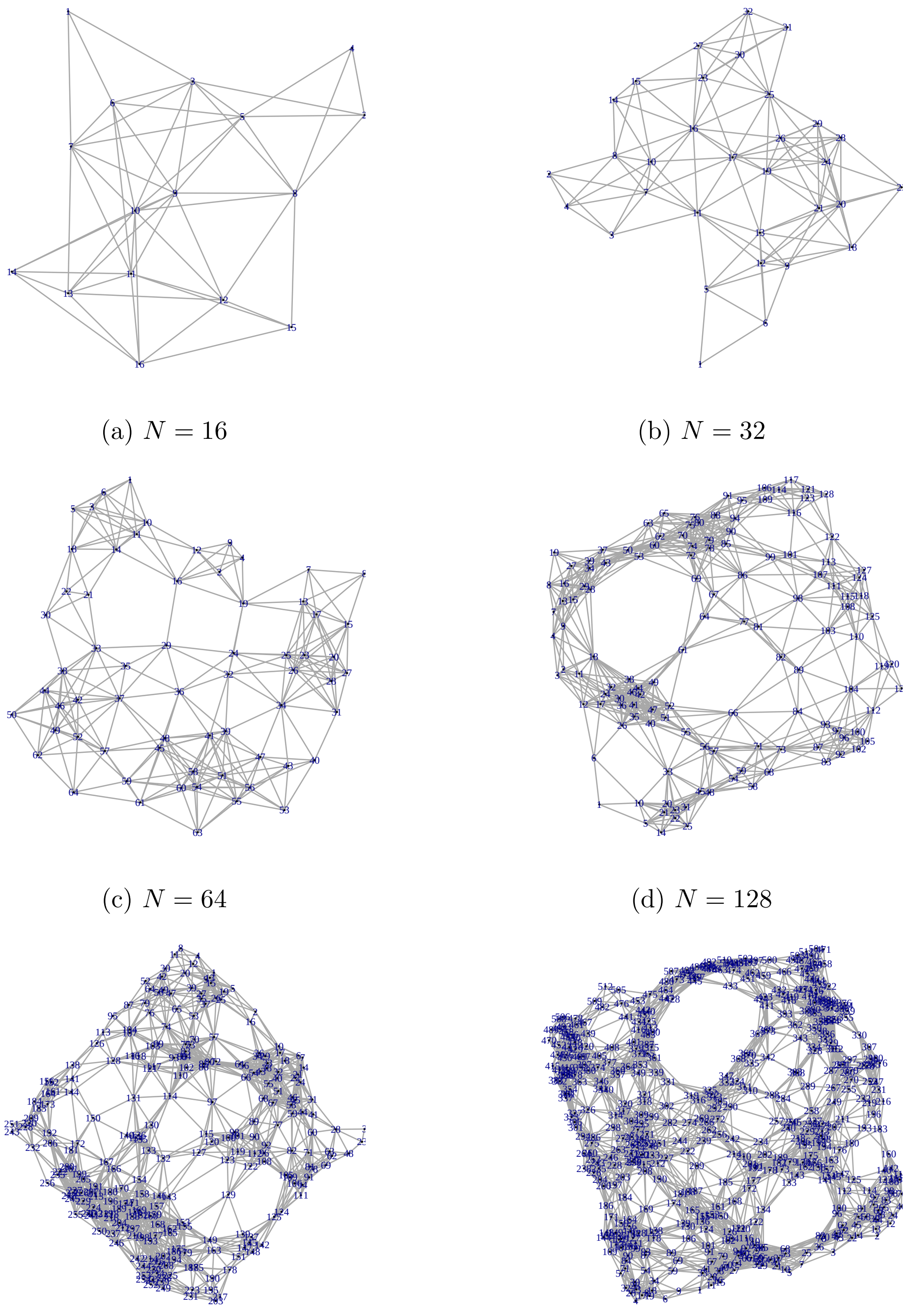
The communication cost of GLS-IR using various working precisions p_β relative to D-LMS is shown in Figure 7.5. For $m = 8$ (Figure 7.5a), GLS-IR-NE requires either 15% (PF) or 20% (PS) of the number of messages used by D-LMS for the majority of working precisions analysed in this section. GLS-IR-SNE starts off with 64% (PS) and 49% (PF) of the number of messages for $p_\beta = 10^{-14}$ and reaches 29% (PS) and 23% (PF) for $p_\beta = 10^{-4}$, the working precision requiring the least number of messages to reach p_α. For wider matrices, as shown for $m = 32$ in Figure 7.5b, even for the lowest working precision $p_\beta = 10^{-2}$ GLS-IR-SNE does not reach 20% of the messages used by D-LMS, making GLS-IR-SNE more than 5 times faster than D-LMS. Up until $p_\beta = 10^{-5}$, GLS-IR-SNE using PF requires only about 6% and using PS about 7% of the messages used by D-LMS to reach the target accuracy of $p_\alpha = 10^{-15}$. For GLS-IR-SNE, the communication cost steadily declines from 77% for PS and 58% for PF to about 20% and 15% using $p_\beta = 10^{-3}$ for PS and PF, respectively.

These results demonstrate the benefits of using lower working precisions for the majority of the aggregation operations, while still being able to achieve the high target accuracy through the use of IR. Furthermore, the recovery capabilities of IR can be seen for various potential accuracies of the initial factorisation. As already mentioned in subsection 8.3.2, the recovery from a fault only increases the number of iterations required by IR slightly. Due to the low computational complexity of the IR iterations, the effect of a fault on the total computation time will be very low compared to the factorisation of the matrix.

7.6 Conclusion

We presented the distributed gossip-based linear least squares solver GLS-IR based on semi-normal equations or normal equations and mixed precision iterative refinement. In this solver, all communication operations between participating nodes are contained in gossip-based reduction operations. Consequently, GLS-IR directly benefits from all improvements in such reduction operations. The fault-tolerance of the GLS-IR algorithms is achieved through the use of the fault-tolerant gossip algorithm push-flow and employing the correcting properties of iterative refinement. Thus, the algorithms become fault-tolerant against silent message loss and temporary or permanent node failures.

The experiments demonstrated that GLS-IR significantly reduces the number of messages compared to existing dLLS solvers. The use of lower working precisions has been shown to further reduce the communication costs without loss of accuracy. IR not only stabilises the method of SNE, but itself provides resilience against faults that occurred during the QR factorisation or the formation of the normal equations. The resilience of GLS-IR is further improved through the use of push-flow, which has been illustrated to handle high message loss rates in the context of our dLLS solver at a very low communication overhead.



(a) $N = 16$

(b) $N = 32$

(c) $N = 64$

(d) $N = 128$

(e) $N = 256$

(f) $N = 512$

Figure 7.6: Examples for random geometric topologies from $N = 16$ to $N = 512$ as defined in Table 7.3.

Part III

Algorithm-Based Fault Tolerance

Chapter 8

Algorithm-Based Fault Tolerance

Investigating the fault tolerance approaches for linear least squares solvers, we found many different methods to provide fault tolerance against silent data corruption (also known as single event upset or bit-flips) to many different algorithms. One method that stood out from the others was algorithm-based fault tolerance (ABFT) for matrix multiplication. While intriguing, the handling of bit-flips was limited to specific regions within the floating-point representation. In the literature, the occurrence of bit-flips was artificially restricted to the mantissa and faults in the exponent of the value were omitted, an extremely unrealistic scenario severely limiting its application to real-world problems. Investigating the reasons further showed the limitations of current ABFT methods when recovering from a fault caused by a bit-flip. We saw an opportunity to solve some significant shortcomings of ABFT to correct bit-flips in any position of the floating-point representation.

In the following chapters, we discuss the existing approaches to resilience against silent data corruption in general and ABFT in particular. This led to the design of the fault tolerant APIR method (FTAPIR), which uses components and ideas from ABFT. FTAPIR has a very low overhead to protect the algorithm from silent bit-flips. As we shifted our focus to ABFT for matrix multiplications, we required a method to systematically evaluate our improvements of ABFT experimentally. We therefore developed a fault injector (see Chapter 9), which can simulate bit-flips in any given data array and any bit position in a floating-point number. Our approach also works with existing libraries, e. g. an optimised BLAS library, without the need of recompilation. In Chapter 10, we improve the fault resilience of ABFT and present a 2.5D fault tolerant matrix multiplication (2.5D FTMM) using our improved ABFT method, having a very low overhead to correct any type of bit-flip encountered at any position in a floating-point number of the result matrix.

8.1 Introduction

With the existence of petascale computing systems today and the objective of reaching exascale systems in the not-too-distant future, fault tolerance is a very important aspect of large high-performance systems which is attracting more and more interest [SWA*13, CGG*14]. Many large applications run for days or weeks on such systems and it is important to ensure that the valuable computing time is not wasted due to faults during the computation. Nowadays it is no longer sufficient to design and implement an algorithm with the aim of high performance and good scalability, but these algorithms also need to be able to handle faults without having to re-compute the entire solution. There are a variety of fault types which have to be considered, from hardware faults to node crashes as well as soft errors like bit-flips or message loss. In this and the following chapters, we focus on silent *bit-flips* in the registers, the cache as well as in the main memory while the computation is in progress, because these are among the hardest types of faults to cope with.

There are many reasons why bit-flips occur [CGG*14]. For example, they can be caused by the data being corrupted while travelling from the memory to the processing units or the result of cosmic rays passing through the components. Effects of neutron radiation on computer components have been extensively studied at the ground level [Nor96, ZCM*96] and in higher altitudes [BY15] for decades. Additional radiation sources have to be taken into consideration in space environments. Galactic cosmic rays and protons and heavy ions from solar winds [BDS03] exhibit much higher energies than the neutron radiation experienced on Earth. These sources affect space missions around Earth, but also on other planets. Mars has a much weaker magnetosphere than Earth. Therefore, significantly more charged particles reach the planet's surface and will affect the components of any equipment used in the exploration missions at a higher rate than these components would be exposed to on Earth.

Measurements presented in [SPW09] illustrate that over many ten-thousands of machines about a third of the machines exhibited at least one (soft or hard) memory error per year. Extrapolating this data to a system with millions of computing units, the mean time to failure would be less than a minute. This example already demonstrates that memory errors can occur with high frequency and therefore pose an important challenge on large-scale systems. Another example is given in [HP10], where 50 000 GPGPUs running on the Folding@home network were examined for their susceptibility to bit-flips. The results reveal that soft memory errors occurred on an overwhelming two-thirds of the tested GPUs. These numbers further emphasise the importance of fault tolerance on large-scale systems. These systems also use GPUs and FPGAs as accelerator cards to increase their performance.

The use of error correcting codes (ECC) in memory chips is very common in high-performance systems. However, higher fault rates would have to be conquered by adding redundancy in the circuits or by using more powerful (and more costly) ECCs to protect the memory words [CGG*14]. According to [SWA*13], these improvements would lead to an increase of 20% in circuitry and energy consumption. Furthermore, the cost of the components and their development would be higher due to not having a high demand on the general market. Building exascale systems out of commodity components without ECCs would lower the costs but at the same time also lower the reliability levels of the hardware. Therefore, other techniques will be required to defend against higher fault rates.

Various approaches exist which can cope with at least some fault types. The most widely used method in high-performance computing (HPC) is checkpoint-restart (C/R) [SPD*05, CGG*14], which saves the state of a computation at specific intervals and can recover from detected faults by rolling back to a check-pointed state. Its popularity is largely due to its general applicability. However, this approach tends to be relatively expensive in terms of overhead. Checkpoints generate a significant amount of I/O traffic and often block the progression of the application [CGG*14]. Another problem is that checkpointing only works if the error can be detected. Although several improvements have been developed to make C/R also usable on large systems [WMES10, ZNK12], its efficiency decreases with increasing system size [FME*12]. Predictions on exascale systems expect a doubling of the total execution time of an application when C/R is being used [FSL*11]. Another method for handling faults is replication, where the application is executed either in parallel or sequentially multiple times. An example for this approach is triple modular redundancy (TMR) [LV62], which masks any single fault through majority voting. Naturally, the high resource cost in terms of either replicated hardware or multiple consecutive executions can be a major drawback in the context of HPC. Nevertheless, it has been shown that process replication strategies can outperform traditional C/R approaches for a certain range of system parameters [FSL*11]. Moreover, several benefits of *combining* C/R methods with redundancy (process replication) have been illustrated [EKF*12, CRVZ15].

An alternative approach, and the one we employ in this and the following chapters, is to design the algorithms themselves to be aware of silent faults. A prominent representative of this type of algorithms is *algorithm-based fault tolerance* (ABFT) [HA84], which basically adds a small amount of redundant data, the checksums, to the input to allow for detection and correction of a corresponding number of silent faults either during or at the end of the computation. Compared to C/R and replication techniques, ABFT achieves fault tolerance with much lower overhead and thus higher performance. Moreover, in contrast to C/R it can handle silent bit-flips occurring during the computation.

8.2 Related Work

The majority of the research on ABFT has been conducted for matrix multiplication, but the approach has also been applied to other linear algebra methods, including LU, Cholesky and QR factorisation as well as Hessenberg reduction. In this section, we will review the related work for these ABFT methods.

8.2.1 ABFT for Matrix Multiplication

ABFT [HA84] handles faults at the algorithmic level by adding rows or columns to the matrices containing checksum encoded information about the data. An operation has to be checksum preserving to be able to use ABFT. Most research about ABFT focuses on node failures. In this variant, sometimes called *global ABFT*, the checksums are stored in additional processor rows and columns dedicated to the checksum encoded values. A fault tolerant MPI implementation can be used to handle node crashes (e. g. User Level Failure Mitigation [BBH*13]).

In the ABFT approach for recovering from bit-flips, the result of an operation is checked for faults by recomputing the checksums after the operation has completed and comparing the new checksums with the ones returned in the result. This indicates an important limitation: usually, only the final result of a matrix operation can be corrected. An exception has been presented in [BDDL09], where an outer product matrix multiplication uses global ABFT for fault detection and correction *during* the computation instead of only in the final result. The outer product preserves the checksum relationship at every step of the computation, unlike other matrix multiplication algorithms where the checksums are only consistent with the result after the computation is complete.

Another important aspect are numerical problems which can arise due to the growth of checksums with the problem size [NA88]. It has been shown that the numerical accuracy of ABFT can be improved by using checksums for blocks of data instead of global checksums (*local* ABFT [RJ92]). Although the blocked variant improves the handling of bit-flips, it cannot recover from node failures due to the missing global checksums. However, the detection and correction of faults is confined to local operations on the computing node itself and therefore does not incur any communication overhead.

In [DA96], the authors focus on the reduction of false positives due to numerical round-off errors and on the detection of faults in the lowest bits of the mantissa. They introduce "mantissa-preserving" checksums which are additional integer checksums of the mantissa. However, their approach can only protect multiplicative and not additive operations and therefore does not protect the complete matrix multiplication. Furthermore, the authors neither consider nor test bit-flips in the exponent.

Typical scientific applications spend the majority of their execution time in methods which can be protected by ABFT. However, other sections taking place in between these function calls remain unprotected. Therefore, Bosilca et al. [BBH*14] suggest combining the best of two worlds, to use ABFT protected methods whenever possible and otherwise use C/R. During the execution of ABFT methods, C/R would be disabled and only protects sections of the application otherwise prone to faults. Based on their performance model, the scalability is significantly better using this combined approach than only using C/R, while protecting the entire application.

8.2.2 ABFT for LU Factorisation

The algorithm-based fault-tolerant LU factorisation (ABFT-LU) was first presented in the initial paper introducing ABFT [HA84]. The basic idea of ABFT is to add a small amount of redundant data, the checksums, to the input to allow for detection and correction of silent faults. In [HA84], the detection and correction take place at the end of the computation. In the case of an LU factorisation this is very problematic because faults are propagated throughout the algorithm, leading to large parts of the factors L and U being incorrect. To prevent the faults from propagating, *online* ABFT methods [DC13] have been developed which handle faults during the computation. Both methods only add row checksums to the input which only protect the operations on the factor U, whereas the operations on L are at the column level and remain unprotected. Therefore, the authors in [DLD11] and [YZC*15] proposed to add row and column checksums to protect all operations in the online ABFT method. The number of checksums d added to the input data determines how many faults can be handled. If both row and column checksums are used, $2d$ faults can be detected and d faults can be corrected.

In [DLD11], bit-flips are treated as rank-one perturbations and the Sherman-Morrison formula, which calculates the inverse of the sum of an invertible matrix and the outer product of two vectors, is used to recover the solution x by applying a low-complexity update procedure on the perturbed result vector. The protection of the left factor is also extended using diskless checkpointing to work with LU with partial pivoting. With no faults present, the overhead is shown to be about $1-2\%$ compared to the same solver with no fault protection. The overhead for recovering from a single fault reaches about 3% for larger matrices. Their work ensures that the solution vector x is correct, but does not protect the complete factorisation itself. Multiple faults have not yet been considered. The authors suggest that the same methodology can also be applied to QR and Cholesky factorisations.

8.2.3 ABFT for Other Linear Algebra Methods

Wu et al. [WC14] have developed online ABFT methods that recover from errors during the computation for Cholesky, QR, and LU factorisations and released FT-ScaLAPACK, a fault tolerant version of ScaLAPACK implementing their results for these three linear algebra methods. Their aim is to not only protect the result of the operation, but to also ensure that the factorisations themselves are correct. The interface of the library is identical to ScaLAPACK. It can therefore be used as a drop-in replacement for ScaLAPACK to enable fault tolerance in those three algorithms by linking to the new library. In the paper, the authors mention that their algorithms can handle multiple faults. However, this refers to the total execution of the algorithm, but only a single fault can be handled in each recovery step of the methods due to the use of a single checksum row or column being used.

A Cholesky factorisation using ABFT to recover from fail-stop errors (e. g. node failures) has been presented in [HWC14]. The work focuses on the challenges which arise by adding column and row checksums to the positive definite matrix A. Due to the checksums being linear combinations of the rows or columns of A, the positive definite property of A is lost. The authors discuss different approaches to handle this problem. The first one is a bordered Cholesky algorithm, which only updates the checksums in the last iteration to ensure they are correct *after* the computation completes. The second approach uses an outer product method, which maintains the row and column checksums during the computation and skips the last iteration (i. e. the $n+1$ iteration), which would otherwise use the checksums breaking the positive definiteness of the matrix. The final method discussed by the authors is the right-looking algorithm which exploits the symmetry of the matrix A and therefore can only maintain the column checksums during the computation.

ABFT has also been used to protect the Hessenberg reduction [JBLD13] from a node failure. An algorithm, FT-Hess, was designed using ABFT in combination with diskless checkpointing. The matrix A was extended by row checksums and the algorithm could therefore tolerate one failure per row of processors in a 2D processor grid. Unlike other global ABFT methods, FT-Hess does not use dedicated checksum processors but stores two copies of the row checksums of A on two different processors, one being the same processor that already holds the data. The authors base their implementation on the panel-based block-level Hessenberg method available in ScaLAPACK. In ScaLAPACK, first a sequence of Householder transformations is used to reduce a panel of the matrix A. Then the trailing submatrix is updated by accumulating the Householder reflectors and applying them to the trailing matrix together (using matrix multiplications). These operations are executed for each panel. In [JBLD13], ABFT is used to protect the trailing matrix which is modified frequently and therefore would not benefit from checkpointing due to the high I/O overhead. The lower left part of the matrix is not protected by ABFT during the reduction operation. Before starting the block column factorisation, the diskless checkpointing therefore takes a snapshot of the panel to protect the result before updating the trailing matrix. The row checksums for the block columns are valid at the end of each iteration.

8.3 Revisiting APIR in the Context of ABFT

In this section, we investigate the effects of silent bit-flips on iterative refinement (IR). IR is a naturally fault resilient algorithm at the cost of additional iterations in the event of a fault in the solution vector x. However, using lower working precisions in MPIR and APIR, more precisely smaller exponent ranges $e_\beta < e_\alpha$ (cf. nomenclature in Chapter 4), leads to IR not being able to handle bit-flips in higher exponent bits of e_α. We therefore propose an approach to make APIR (from Chapter 4) resilient against these kind of faults at a very low overhead. We present and discuss the *fault tolerant APIR* algorithm *FTAPIR* (see Algorithm 13) which can handle bit-flips at the algorithmic level. We consider bit-flips occurring in the solution vector x, while the input matrix A and the factors L and U are protected by the checksums from ABFT-LU. The intermediate results, the residual r and the solutions of the linear system with r as their right-hand side, z and Δx, are also allowed to be affected by silent data corruptions. These faults would propagate to the approximation of x and would therefore be detected as well. As a fault model, we assume an exponential distribution in time and a uniform distribution in space to ensure every position within the data structure, which implies every position of the floating-point representation, is equally probable.

8.3.1 Fault Tolerant LU Factorisation

The first component of APIR that has to be protected against bit-flips is the LU factorisation (line 1 in Algorithm 1). We use the algorithm-based fault-tolerant LU factorisation (ABFT-LU) presented in [HA84] (see subsection 8.2.2 for details). In this section, we use the well-studied ABFT-LU factorisation as a black-box. Our goal is to detect and correct bit-flips in the subsequent IR process. If the exponent ranges for the target and working precisions are the same, faults in x can be handled automatically by IR at the cost of an increase in the number of iterations. However, correcting bit-flips in IR methods using different exponent ranges for the target and working precisions has, to the best of our knowledge, not yet been done.

8.3.2 Fault Tolerant Iterative Refinement

IR is a naturally self-healing algorithm. Standard IR (SIR), where $\beta = \alpha$, can handle almost any bit-flip in x at any position of the FP representation. In each iteration, the correction term Δx would reduce a fault by the maximum possible accuracy of the FP representation, e. g. DP would reduce the error by approximately 10^{15} and SP by 10^{7}. Considering this limit of improvement per iteration, bit-flips in the higher exponent bits, which increase the power of the affected value, naturally lead to an increase of the number of iterations required to reach the threshold τ_r. The ratio of the logarithm of the maximum erroneous value and the maximum number of digits recoverable per iteration by the FP representation provides the additional number of iterations required after a bit-flip. If the bit affected by the bit-flip falls under the termination threshold, IR would naturally not be able to detect the fault. It would be considered a numerical error instead of a fault. However, numerical errors below the threshold would already have been considered an acceptable loss and faults in those low bits would not have a negative impact on the accuracy. The only case that cannot be handled by IR without an additional detection step are bit-flips which cause values to become NaNs. As we will see, our fault tolerance approach is able to handle this case to also make SIR fault tolerant against such bit-flips.

In the case of APIR, more specifically, in the case of β having a smaller exponent range than α ($e_\beta < e_\alpha$), bit-flips in the higher exponent bits can no longer be handled implicitly by IR. Bit-flips which reduce the exponent of a number or only affect the mantissa or sign bits are automatically detected when comparing $\|r\|_2$ with the threshold τ_r (line 8 in Algorithm 1) and are then handled in the following iterations. Casting a value with an exponent larger than the maximum exponent of e_β to the lower FP format β will lead to an FP overflow. In accordance with the IEEE 754-2008 [IEE08] standard, the value then becomes infinity.

Using this value while solving $Lz = r$ for z (line 5 in Algorithm 13) will lead to the values of z being NaN.

The IR process can be made fault tolerant to protect the solution vector x against silent data corruption by replacing line 5 in Algorithm 1 by lines 5-15 in Algorithm 13, leading to our fault tolerant APIR method. Solving the linear system $LU\Delta x = r$ is split into two steps and a fault detection and correction process is added in between. First, z has to be checked for NaNs as the indicator for a fault which cannot be recovered by IR itself. It is sufficient to test the last element of z, $z(n)$, because a NaN value would propagate to the end of the vector in the forward substitution process in line 5 of Algorithm 13. The correction process then has to determine the indices of x which have been affected by bit-flips. The affected values are significantly larger than the other values in x, otherwise the faults would not have caused an FP overflow and could have been corrected by IR without any additional help. Therefore, all elements in x which are either larger than $\tau_{\max_\beta}$ or NaN are set to zero. The threshold $\tau_{\max_\beta}$ depends on the working precision β and is the largest representable number of the FP representation. A further benefit from setting the faulty value to zero is the decrease of the required number of iterations to reach the threshold τ_r compared to continuing the improvement process with the potentially very large faulty value and a low improvement per iteration due to using β. A smaller exponent range e_β compared to e_α allows a large increase in the exponent to be caught earlier in line 6. Setting the faulty value to zero only requires the same number of iterations as required after the LU factorisation to reach the same accuracy as at the time of the fault. Depending on the problem setting, one could consider lower exponent ranges for e_β which would allow faults to be detected earlier and increase the convergence speed after faults occur.

After eliminating the faulty values, the residual r and the linear system solution for z are recomputed and the iterative improvement can continue (see lines 9 and 10 in Algorithm 13). The final check in line 11 is a safeguard against additional faults occurring during the detection and correction process. If another fault occurred during this process, which is very short especially compared to the LU factorisation, then most likely the fault rate is too high to compute any step correctly. However, it is highly unlikely that multiple faults would occur during the IR process, even though the fault detection and IR itself can handle multiple faults. Such an unreliable system would have already failed to compute correct factors L and U in ABFT-LU due to its higher complexity. Assuming the same fault rate throughout the algorithm, for multiple faults to occur during the IR process of $O(n^2)$, at least a factor of n faults would have to occur during the LU factorisation of $O(n^3)$. This would require the number of checksums d for ABFT-LU to be at least n, which would significantly increase the overhead of the ABFT-LU factorisation. Therefore, even an increase in the number of iterations of IR would not be considered a likely source for multiple transient bit-flips.

The last question to be considered is what happens if a fault occurs in x while the residual in line 17 is being computed. As long as the IR process is running, faults will be handled by the detection and correction process. The only way to ensure that the solution x returned after

Algorithm 13 Fault Tolerant APIR (FTAPIR)

```
Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$
1: $[L, U] \leftarrow$ abftlu($A$)                              ▷ *fault tolerant* factorisation in $\beta$
2: Solve $LUx = b$                                               ▷ solve in $\beta$
3: $r \leftarrow Ax - b$                                         ▷ compute residual in $\alpha$
4: for $i = 0 : i_{max}$ do
5:     $z \leftarrow$ solve $Lz = r$                             ▷ solve in $\beta$
6:     if $z(n)$ is NaN then
7:         $I_x = \{j \in [1, n] | |x(j)| \geq \tau_{\max_\beta}$ or $x(j)$ is NaN$\}$
8:         $x(I_x) = 0$                                          ▷ set faulty values to zero
9:         $r \leftarrow Ax - b$                                 ▷ re-compute residual in $\alpha$
10:        $z \leftarrow$ solve $Lz = r$                         ▷ solve in $\beta$
11:        if $z(n)$ is NaN then
12:            return $\rightarrow$ too many faults occurred
13:        end if
14:    end if
15:    $\Delta x \leftarrow$ solve $U\Delta x = z$               ▷ solve in $\beta$
16:    $x \leftarrow x + \Delta x$                               ▷ update $x$ in $\alpha$
17:    $r \leftarrow Ax - b$                                     ▷ compute residual in $\alpha$
18:    if $\|r\|_2 < \tau_r$ then
19:        $r \leftarrow Ax - b$                                 ▷ compute residual in $\alpha$, fault free
20:        if $\|r\|_2 < \tau_r$ then                            ▷ fault free
21:            break $\rightarrow$ converged and correct
22:        end if
23:    end if
24: end for
```

the IR process is correct is by recomputing the residual and comparing its norm to τ_r while ensuring that no further faults will occur (lines 19 and 20 in Algorithm 13). Consequently, these are the only two steps required to run fault free and are only required *after* the IR process has already reached the termination criterion in line 18. For these two steps, reliable hardware or other strategies like triple modular redundancy [LV62] could be used. If faults are detected, the IR process can be continued to recover from the fault, again without any restrictions where or when faults are allowed to occur in x.

The overhead of the fault detection is almost non-existent, only evaluating a conditional expression in each iteration (line 6 in Algorithm 13). In the event of a fault, the correction itself (lines 7-10) costs less than an additional iteration of IR. The fault-free computation of the residual (line 19) requires n^2 operations in precision α to ensure that the returned vector x is correct.

8.4 Conclusion

In this chapter, we reviewed the related work on algorithm-based fault tolerance. We considered silent bit-flips occurring during the computation and proposed a fault tolerant APIR method (FTAPIR), which is resilient against these types of faults. Bit-flips are allowed to occur anywhere throughout the algorithm. Only a very small part of the algorithm, which takes place after the IR process has converged, is required to run fault-free or has to be protected by other approaches.

In Chapter 10, we analyse the shortcomings of ABFT for matrix multiplications and improve the method to handle bit-flips in any bit of a floating-point representation. We further demonstrate the use of ABFT in a high-performance matrix multiplication leading to our method 2.5D FTMM. Our algorithm is able to recover from bit-flips occurring in the result matrix at a very low overhead. Before we can run experiments to evaluate the fault resilient properties of our algorithms, we require a fault injector, that can simulate all scenarios that we are investigating. In the following chapter, we present FaITh, a thread-based fault injector which fulfils our needs for granular testing of our improvements and simulate bit-flips in any set of bits of a floating-point number, additionally having a very low impact on the performance of our algorithms.

Chapter 9

Fault Injector

As discussed in Chapter 8, fault tolerant algorithms will become more and more important with the growing size of high-performance systems. Bit-flips are one of the main concerns which lead to corrupted memory and erroneous results and can be caused by various sources, e. g. neutron radiation. Preparing for the increased probability of calculations being affected by bit-flips is therefore of paramount importance. Extensive testing and verification of newly developed methods is essential to ensure their capabilities of handling these faults.

9.1 Introduction

Analysing the properties and the resilience of fault tolerant algorithms requires the simulation of bit-flips. These simulations should be as close to a realistic use case as possible and correctly mimic the effects of real bit-flips, which sooner or later will mainly result in a memory corruption. In the worst case the effects could lead to the failure of the affected system. However, such fail-stop faults are not the focus of this thesis and we focus on the silent data corruptions that a program can recover from and continue its work.

Naturally, hardware induced faults are the closest to real-world events but they also have many disadvantages. Physically caused faults, e. g. using some form of radiation to trigger a bit-flip, require special equipment and are not reproducible. In most cases, due to the chip density one has very limited control over the position of a fault. Additionally, many of the techniques also pose the danger of permanently damaging the hardware devices. Therefore software-based fault injection is preferable as it produces repeatable results and experiments can be run on a much larger scale. Software-based fault injectors are ideal to demonstrate the fault tolerant properties and examine the behaviour of an algorithm when a silent data corruption occurs.

In this chapter, we will define our requirements of a software-based bit-flip fault injector and summarise our contributions. In section 9.2, we summarise the relevant related work on software-based fault injection techniques. section 9.3 discusses the design ideas behind our novel thread-based fault injector, FaITh. In section 9.4, we describe the basic usage of FaITh and the options provided by our library. The overhead of the fault injector will be demonstrated with single-core and multi-core experiments in section 9.5 using `dgemm` from a highly optimised BLAS library. The possibilities of our fault injector will be demonstrated further in Chapter 10 to examine the capabilities of the ABFT method for matrix multiplication. Finally, section 9.6 concludes this chapter.

9.1.1 Requirements

For our analysis of the effects of bit-flips in fault tolerant algorithms, we defined the following requirements for a software-based bit-flip fault injector.

1. The fault injector should require, at most, minimal code modifications, which furthermore should not be required within the actual algorithm itself. Any additional changes within the algorithmic structure would cause unforeseen side-effects. It would influence the performance of the algorithm and could lead to compiler optimisation techniques to be, at the very least, less efficient. For example, adding instructions to the inner-most loop of a matrix multiplication would significantly affect the execution of the algorithm. Even if the instructions only have to check, if a fault should be injected, this step would have to be performed $O(n^3)$ times, an unacceptable increase in the number of operations. Minimal code modifications are necessary to ensure that the probe effect of the software-based fault injector is as small as possible.

2. Another requirement, which is very closely related to the previous requirement, is the possibility to inject faults into external functions and libraries. This includes, for example, highly optimised BLAS libraries, which should not have to be recompiled to examine the effect of bit-flips. This requirement also leads to the possibility of using commercial or closed-source libraries, e. g. the Intel MKL BLAS library.

3. For many algorithms, e. g. algorithm-based fault tolerance (ABFT), it is also necessary to have fine-grained control, where a bit-flip is allowed to occur within the data structure. For example, to investigate special problems, one should be able to restrict the affected bits in a floating-point representation to the exponent or to the mantissa. Some effects of a bit-flip can only be observed in the higher bits of the exponent, others might be focused on a single bit position in a data structure. In most cases, a bit-flip in the lowest bit of a floating-point number will have almost no effect on the computation whereas a bit-flip in the exponent could affect the entire result. A fault injector has to provide the ability to limit the location of a bit-flip to analyse an algorithm and its capability to handle bit-flips with varying degrees of severity.

4. Modelling the occurrence of a fault is another essential requirement in the analysis of the effects of bit-flips on an algorithm. Therefore, a fault injector has to be able to accept various fault models for the time and position when and where bit-flips can occur. In a parallel environment, it should also be possible to simulate different fault

models on different nodes. This could be used to simulate components of different ages and measure the effects of varying fault rates on the algorithmic techniques used to handle these types of faults.

5. The fault injector should have no special dependencies. It should not be limited to a special compiler or require an external parser. Furthermore, it should work on any hardware configuration and not be limited to a specialised processor. Ideally, it should not require any external libraries, but should use standard interfaces instead.

6. Last but not least, the fault injector has to have a very low overhead to ensure minimal interference in the performance of the investigated algorithms. The injection of the fault is not allowed to halt the execution of the algorithm to inject a bit-flip.

9.1.2 Contributions

The goal of our fault injector is to be as close to a physical real-world experiment as possible. We focus on silent data corruption, injecting bit-flips into the memory locations that are accessible from the software-level. We do not aim to modify the instruction buffers of the processors or the address busses because sooner or later any fault will affect the memory and cause data corruption.

Our novel thread-based bit-flip fault injector FaITh (**Fa**ult **I**njector **Th**reads) fulfils all the requirements described in subsection 9.1.1. FaITh has no special dependencies and is purely written in the C++11 standard. It does not require any external libraries and only uses the standard C++11 libraries and interfaces. Only minimal code modifications are required to use FaITh which can even be restricted to the main method. The fault injector only requires access to the pointers of the data structures that should be affected by the bit-flips. It works with any existing high-performance library without the need to recompile these libraries, as will be shown with unmodified ATLAS BLAS and OpenBLAS libraries in our experiments in section 9.5. Our fault injector approach provides the granularity needed to examine the behaviour of algorithms if bit-flips can only occur at specific positions in a floating-point representation or any other data type. FaITh acts completely independent on all processors, which allows for the use of different fault models on different processors, e. g. simulating older components which could be more error prone than newer components. Furthermore, all these requirements have been met with the fault injector incurring a very low overhead also compared to existing approaches. On average, the overhead is only about 1% of the total execution time, independent of the number of cores used.

9.2 Related Work

In this section we will discuss the advantages and disadvantages of hardware fault injection techniques and summarise the related work about software-based fault injectors.

9.2.1 Hardware Fault Injection

In [KFA*98], three physical fault injection techniques were examined. Heavy-ion radiation injects a bit-flip at internal locations in integrated circuits (ICs). The experiment has to be conducted in vacuum and the protective covers of the chips have to be removed. However, in the majority of cases, heavy-ion radiation causes only a single bit-flip and is therefore comparable to the effects of ground-level neutron radiation [STW00]. The second method described in the paper is a pin level injection, where faults are injected directly to the pins of ICs. This method can therefore also simulate permanent faults and has the highest degree of controllability and repeatability. Electro-magnetic interference was simulated by generating bursts of electro-magnetic waves either to the entire chip or localised parts of a chip. In general, hardware fault injection approaches require specialised (expensive) equipment and can often only examine the effects of radiation on specially designed hardware components, which limits the portability of these methods. Their advantages include assessment of hardware locations which otherwise could not be accessed (e. g. by software-based methods) and that the experiments can be carried out in real-time, without creating an overhead influencing the execution or performance of a program, aside from the injected fault. However, they also carry the risk of permanently damaging the exposed hardware. With ever increasing chip density, it is also more difficult to accurately inject a fault into the components. In [AC03], the authors compared the beforementioned hardware fault injection techniques to a software-based fault injector. In their case, the faults were injected into the machine code before executing the program. They came to the conclusion, that software-based bit-flips are able to generate similar errors as the physical techniques.

9.2.2 Software-Based Fault Injection

Many software-based fault injection tools have been developed over the years, but many of them are restricted to certain architectures or even specific chips. In this section, we will not discuss simulation-based fault injectors, which run the targeted application in specially modelled simulation environments. These approaches have a high observability and controllability where, when and how a fault is injected. However, they entail a high development cost and require a model of every targeted system and processor. The experiments cannot be run in real-time and the quality of the analysis depends on the quality of the abstracted model of a complex system.

FERRARI [KKA95] injects faults by altering the execution state of a targeted program. It runs two processes, one for the fault injector which then spawns the second process for the target program. The target application process is controlled using the Unix command ptrace. FERRARI uses software traps to inject faults into the CPU registers and the memory. The fault injector process waits for an interrupt handler and then injects a fault through the communication between the two processes. FERRARI is platform-specific and its injector module and data analysis and collection module both require modifications to be ported to any new platform.

Xception [CMS98] uses the performance monitoring and debugging features already available in most processors. It uses various types of hardware exceptions instead of software trap instructions. When a hardware exception is triggered, an exception handler is called which injects the fault. The faults can be specified individually according to the exception type and require the definition of the fault location (e. g. the instruction execution control unit, the floating-point unit, . . .) and the fault type. For example, to inject a fault into the processor's registers, the size of the register and the register map need to be known. This information is provided by Xception by requiring the application to be run twice, once without any fault injection as a reference and, after defining the faults, rerunning the application with the required instruction or breakpoint triggers activated. Xception uses an interface to define the faults to ensure portability over the range of supported processors.

FlipIt [COS14] focuses on faults that arise in the processor. It injects faults by including additional instructions at compile-time using an LLVM parser step in the compiler toolchain. The user can specify the names of the functions that should be affected by faults and the probability of these faults. The explicit code modifications are minimal and can be contained to the main method, but the additional instructions can have a high performance impact on the application. Every time the program reaches the targeted instruction, additional instructions are executed to check the probability function and to determine if a bit-flip should be injected. Furthermore, any part of the code that should be affected by bit-flips has to be recompiled. It is not possible to inject faults into existing high-performance libraries without recompilation. The authors report very high increases in the execution time of an application using FlipIt with a slow-down of up to 123 on a single core. This slow-down decreases to 21 for 4096 cores, but only due to the communication time dominating their example application.

An overview of other software-based fault injection techniques, which are also not limited to bit-flips, can be found in [CMC*13, ZAV04]. Sadly, all of the fault injectors mentioned in this section are not publicly available and we are therefore unable to compare FaITh to existing software-based fault injection methods.

9.3 Design

We use an additional thread per process that has access to all data structures where the user allows bit-flips to occur. A user-defined time distribution determines the duration for the thread to sleep whereas a user-defined data distribution determines which bit should be affected. The fault injector thread sleeps until the next injection is scheduled and switches between different sleep methods to achieve a high accuracy.

Our fault injector does not depend on any external libraries or compilers, as seen with other fault injectors in the related work, and is not limited to a specific architecture or processor. The only requirement is a C++ compiler with support for the C++11 standard.

Specifically, the `random` and `mutex` APIs from the C++11 standard are used. The computational kernels do not have to be modified for silent data corruption to occur during the execution. This enables the use of FaITh with existing external libraries without the need of recompilation. Functions from high-performance libraries, like BLAS and LAPACK, can be used directly without the need for any modifications. As shown in the following section, only minimal code modifications are necessary to use the FaITh library and these modifications can be completely contained in the main method, as long as the method has access to the pointers of the data which should be affected by bit-flips.

The results are reproducible by setting the same seeds for the time and data distributions. However, naturally fluctuations can lead to a program running slower or faster than a previous run. Therefore, the last fault injections could be omitted because they would have occurred at a later time or additional injections could occur because the program runs longer. Normally, this should not be a problem. If the probability of an injection is very high and the runtime of the program varies greatly between runs, then naturally the effect would be higher. However, for more realistic failure rates this would not affect the results.

By design, our thread-based injection approach is very close to a realistic setup where faults can occur at any time during the execution. In order to limit the influence of the fault injection on the computation as much as possible, the fault injector thread runs asynchronously to the main algorithm and does not halt the main thread during the modification of the value. A bit-flip in the data structure is injected into the register of the processor core and then automatically synchronised to all other memory hierarchies. The injector thread reads a data value from the main memory into a local memory variable, modifies the value by flipping a bit and then writes the modified value back to the main memory. In very rare situations, this can lead to a race-condition with the main thread, which may be operating on the same value selected by the fault injector. The main thread would then write the correct result from the multiplication back to the main memory, where it would subsequently be overwritten by the fault injector thread with the original value containing a single bit-flip. From the point of view of the main thread, the value would have been exposed to more than one bit-flip even though the fault injector only changed a single bit. In the context of algorithm-based fault tolerant algorithms, which was the initial motivation for the development of this fault injector, it is irrelevant how many faults occur in the same value. The faults would still only be detected as a single faulty value by the algorithm.

9.4 Functionality and Usage

In this section, we will show how to use the FaITh fault injector and which options are available to control the injection of the bit-flips. A basic example of the usage of FaITh is shown in Listing 9.1.

```
#include "faultinjector.h"
#include "fijdistributions.h"
using namespace FIj;
```

```
 4 ...
 5 try {
 6    DataDistribution *dDist = new UniformDataDistribution();
 7    FaultInjector::setDataDistribution(dDist);
 8    TimeDistribution *tDist = new ExponentialTimeDistribution(MTTF);
 9    FaultInjector::setTimeDistribution(tDist);
10
11    FaultInjector::addData(n*n, C);
12
13    FaultInjector::run();
14
15    ... // faults will be injected during the computations
16
17    FaultInjector::terminate();
18
19    FaultInjector::printLog(stdout);
20 } catch ( const std::exception* e ) {
21    printf("Exception caught: %s\n", e->what());
22 }
23 ...
```

Listing 9.1: Basic usage of the FaITh library

First, the data and time distributions have to be set (see lines 6-9). For the data distribution we chose a uniform distribution to ensure every position within the entire data, which implies every position of the floating-point representation, is equally probable. The discrete probability function of the uniformly distribution on the interval $[a, b]$ is defined as

$$P(i|a, b) = \frac{1}{b - a + 1} \quad .$$

The time distribution uses an exponential model with the probability density function

$$P(x|\lambda) = \lambda e^{-\lambda x} \quad .$$

The constructor requires a single parameter defining the mean time to failure (MTTF), which is used in the calculation of $\lambda = N/\text{MTTF}$, where N is the total number of bits which are allowed to be affected in all data structures registered with the fault injector. These distributions are defined in the header `fijdistributions.h` but any user defined model can be implemented and used with the fault injector. Naturally, the seeds for the random generators of each distribution can be defined independently using the method `srand` of the distribution classes.

In the next step, Line 11, the data, that should be affected by the silent data corruptions, is added to the fault injector by providing the pointer of the array and its length. These are all the preparations that are required before starting the fault injector. After calling the `run` method (in Line 13), the fault injection thread will be started and bit-flips will be injected into the data structures according to the chosen distributions. FaITh will inject faults during all computation steps until the `terminate` method is called (see Line 17), which ensures that no further faults are injected and kills the fault injection thread. It is also possible to only pause the fault injection, which will disable any bit-flips until the next call of the `run` method. This is especially helpful if no faults are allowed to occur in a certain block of code. By pausing the fault injector, the code is guaranteed to run fault free, allowing the critical section of an algorithm to succeed or a verification step to be executed. While the fault injector is paused, settings can also be modified which is not permitted while the fault injector is running. This includes changing the distributions, resetting the number of injections and the log data, or removing data from the fault injector using the method `removeData`.

All activities of FaITh are logged: starting, pausing and terminating the injector thread, the addition and removal of data and the injection itself. The relative and absolute positions of each bit-flip are recorded as well as the time of the injection. Aside from logging all events, it is also possible to see how an element has been affected by the silent data corruption in more detail. By providing a function handler of the form

```
typedef std::string (*elementToString_t) ( unsigned char* elm );
```

to the method `setElementToString`, the values before and after the injection will be logged as well. One should be aware that this function is called twice during every injection of the bit-flip and should therefore be implemented very efficiently. The class `FaultyData` contains a method `getElementInBinary` to record the binary representation of any element. Alternatively, a data type specific function can also be used, as shown in the example in Listing 9.2 for double precision values.

```
std::string elementToDouble ( unsigned char* p ) {
   std::ostringstream strs;
   strs << std::setprecision(16) << std::scientific << *((double*)p);
   return strs.str();
}
```

Listing 9.2: Returns the current value of an element for the log

The entire log can be printed to a file (including `stdout`) as seen in Line 19. If a function was set with `setElementToString` then the values are also included in the output. However, it is also possible to disable logging all together by calling `disableLogging`.

This simple example demonstrated the basic functionality of FaITh, but many additional features are also available. FaITh provides fine-grained control to specify which bits should be affected by the fault injector. A bit mask can be provided either globally using the method `setDefaultFlipRange` or independently for each data structure as the last parameter to the `addData` method. For example, a bit mask could define that only a set of bits, a certain pattern, or even only a single bit is allowed to be affected by the fault injection. The effects of bit-flips on different parts of the data elements could be simulated using a corresponding bit mask, e. g. only the highest bits in the exponent of a floating-point number. For the floating-point data types, single and double precision, predefined bit masks can be used to specify which parts of the floating-point representation should be affected. This includes the sign, the exponent and the mantissa, but also any combination of these three options. If no bit mask is set then bit-flips can occur at any position in the data structure. In Listing 9.3, the different options available to set a range or bit mask are shown.

```
unsigned long long bitmask = 0
    b1000000011110000000000000000000000000000000000000000000000001111ULL;
// Set the default flip range
FaultInjector::setDefaultFlipRange(FIj::fliprange::SIGN|FIj::fliprange::
    MANTISSA);
// use default flip range
FaultInjector::addData(n*n, A);
// sets a different flip range
FaultInjector::addData(n*n, B, FIj::fliprange::EXPONENT);
// sets the bit mask explicitly
FaultInjector::addData(n*n, C, &bitmask);
```

Listing 9.3: Examples of setting the range of the bit-flips

9.5 Experimental Evaluation

As already mentioned in section 9.2, most software-based fault injectors described in the literature are not available publicly and therefore cannot be compared to FaITh. Furthermore, very few fault injectors provide performance results of their approaches. FlipIt [COS14] is aimed at high-performance applications. The authors report very high increases in the execution time of an application using FlipIt with a slow-down of up to 123 on a single core. This slow-down decreases to 21 for 4096 cores, but only due to the communication time dominating their example application.

Our experiments were run on a shared memory machine with four AMD Opteron 6174 processors with 12 cores each running at 2.2 GHz and 256 GB RAM. The time interval between two fault injections is exponentially distributed with a specified mean time to failure (MTTF) per byte, as described in section 9.4. The position at which a fault is injected is uniformly distributed, ensuring that all bits have the same chance of being flipped.

For the experiments, we used the highly optimised `dgemm` method from the OpenBLAS library (version 0.2.3) and flipped bits during its execution in all three matrices A, B and C. Various MTTFs were tested to cover a wide range of fault rates and the results were averaged over multiple runs.

First, we wanted to observe the influence of the additional thread of the fault injector on the performance of the algorithm. However, the difference in the execution times with and without the fault injector were hardly measurable and showed that the algorithms are not influenced by FaITh if present but not injecting any faults.

We analysed the impact of FaITh on a single core run of `dgemm` with the fault injector actively running and injecting bit-flips into the memory. In Figure 9.1, the overhead is plotted on the y-axis compared to a fault free run of the algorithm without the fault injector being active. On the x-axis, the average number of affected bits per second is shown. We tested three different matrix sizes $n = \{1000, 2000, 5000\}$ and the results are all averaged over 10 runs. FaITh hardly influences the execution of the matrix multiplication up until 10^3 bits are affected per second. The overhead of the fault injector is less than 0.5% for injecting up to

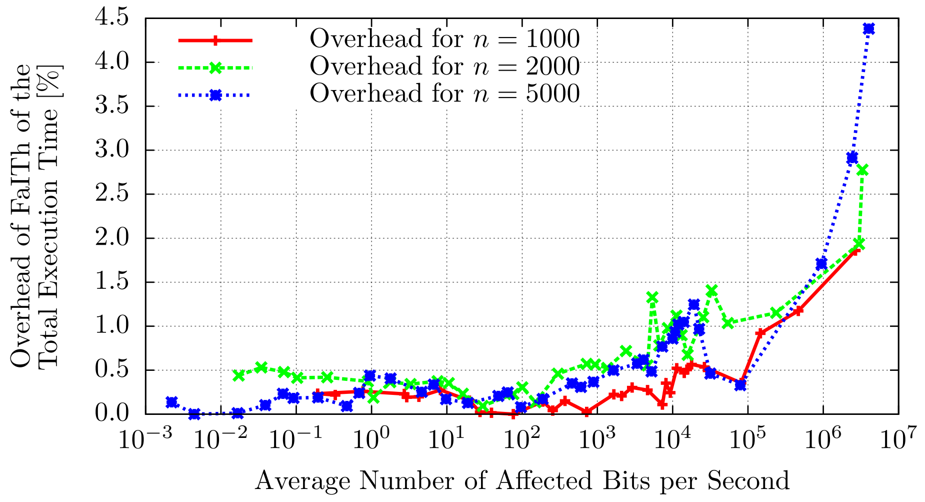


Figure 9.1: The overhead due to the use of the fault injector is less than 1% of the total execution time up to about 10^4 bit injections.

a 1000 bit-flips per second. Naturally, the overhead grows for higher fault rates, but for 10^4, the overhead is still less than 1%. Even for 10^6 bit-flips FaITh only causes an overhead of less than 2%. For $n = 1000$, with faults being injected into all three matrices, on average this corresponds to every third value being affected by a bit-flip. This would be a very high and (hopefully) unrealistic fault rate for any computer system, where a path to recovery would seem improbable.

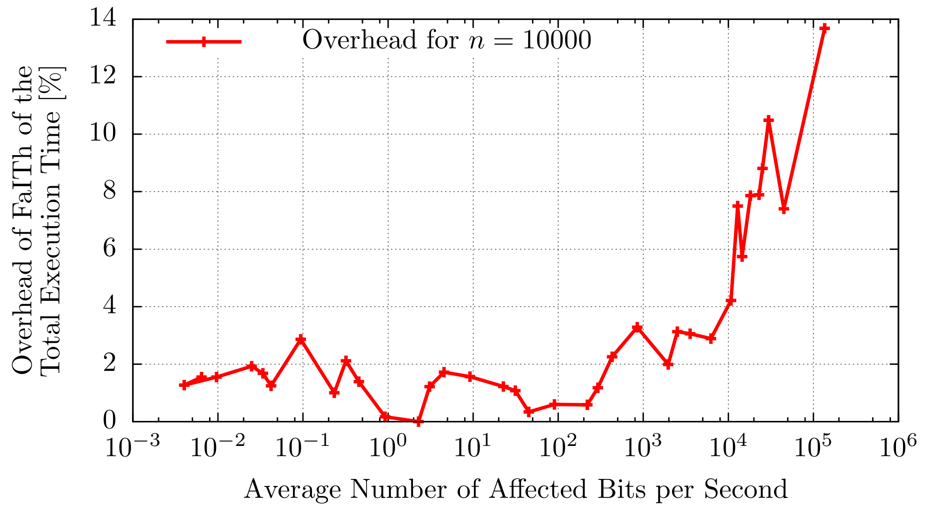


Figure 9.2: `dgemm` was run on 48 cores using OpenMP. The overhead due to the use of the fault injector is about 4% of the total execution time up to about 10^4 bit injections per second.

In Figure 9.2, the matrix multiplication was executed on 48 cores on a shared memory machine using OpenMP. The results are averaged over 5 runs, with the matrix size n being set to 10000. Again, the overhead compared to a fault-free run without an active fault injector is shown on the y-axis and the number of bit-flips on the x-axis. Up to almost 10^3 bit-flips, the overhead is about 2% and remains under 4% for fault rates that cause up to 10^4 bit-flips per second.

Further evaluation of our fault injector will be shown in Chapter 10, where FaITh will be used to examine the capabilities of the algorithm-based fault tolerance algorithm for matrix multiplications, where an overhead of up to 1% was recorded over all conducted experiments.

9.6 Conclusion

In this chapter, we presented FaITh, a novel thread-based fault-injector to simulate silent data corruption, i. e. bit-flips. FaITh is purely written using C++11 standard interfaces and requires only minimal code modifications. It can be used with existing high-performance libraries without the need for recompilation. Fine-grained control is provided to define the position of the bit-flips and only allow faults to occur in specific parts of a data type representation. Furthermore, user-defined fault models can be used. Despite the vast possibilities provided by our fault injector, this strategy has a very low overhead and therefore hardly influences the performance results of the tested algorithms. On average, the overhead only makes up about 1% of the total execution time, independent of the number of cores used. This compares favourably to other published fault injection strategies, e. g. FlipIt [COS14] where a significantly higher overhead has been reported (slow-down factors up to 123).

In Chapter 10, we will use FaITh to simulate bit-flips to analyse the properties of algorithm-based fault tolerant algorithms for matrix multiplications.

Chapter 10

Fault Tolerant Communication-Optimal 2.5D Matrix Multiplication

Algorithm-based fault tolerance (ABFT) protects an algorithm from silent data corruption at the algorithmic level. Although it requires the adaptation of each algorithm and is therefore not as universally applicable as other fault-tolerant methods such as NMR or Checkpoint/Restart (C/R), the low overhead of ABFT compared to these methods significantly outweighs the initial design effort. NMR has at least a 100% overhead for executing a computation twice and in the event of an error, the overhead is increased to 200% having to repeat the calculation for a third time. C/R has a high I/O overhead and limitations on scalability. Furthermore, C/R cannot be used to protect an algorithm from silent bit-flips as all faults have to be detectable by C/R in order to initiate a recovery process. ABFT only adds a small amount of redundant data to the input values and can recover from bit-flips at a very low cost. Furthermore, the overhead in a fault-free run is negligible, as the amount of redundant data is very low in relation to the total amount of data, hardly influencing the performance of an algorithm while providing safety measures in case bit-flips occur.

As described in Chapter 8, ABFT can be applied to a wide range of algorithms. We will focus on a core component of many linear algebra algorithms, the matrix multiplication. It is imperative that any fault tolerant techniques employed to handle silent data corruption do not significantly impact the performance of such a crucial building block. We will show that fault tolerance and high performance do not contradict each other.

In this chapter, we first discuss the limitations of classical ABFT, especially with respect to handling bit-flips in the exponent of a floating-point (FP) number (section 10.2). As we will show, such faults can cause current ABFT methods to fail. We resolve these issues with our improved ABFT method, called *dABFT*, which protects *all* bits of a FP number without significant overhead (section 10.3). The improvement even reduces the overhead compared to classical ABFT due to the efficient handling of the special floating-point values NaN and infinity during the fault detection step. We also derive fault detection conditions for multiple checksum encoding vectors, which up until now were not considered in the analysis of the error bound available in the literature. In section 10.4, we provide a detailed analysis of the resilience of dABFT. We then combine the fault tolerance properties of dABFT with the high performance of 2.5D matrix multiplication methods [SD11, GGDS*12] to receive our *fault tolerant 2.5D matrix multiplication* (*2.5D FTMM*) (section 10.5). For very low failure rates we show that we can further reduce the overhead of the fault tolerant method in the context of 2.5D algorithms. To demonstrate the fault tolerance of our approach, we use our previously described fault injector FaITh (see Chapter 9) which asynchronously injects random bit-flips during the computation with a very low overhead of on average just 1% of the total execution time. In section 10.6 we illustrate the high scalability and low overhead of our 2.5D FTMM algorithms on a high-performance cluster.

10.1 Related Work on Parallel Matrix Multiplication

For our method 2.5D FTMM, we combine two key components: a high-performance parallel matrix multiplication and fault tolerance at the algorithmic level. The related work about ABFT (and other fault tolerant techniques) has already been discussed in Chapter 8. In the following, we discuss the state-of-the-art of parallel matrix multiplications.

Cannon's algorithm [Can69] for 2D meshes is one of the earliest parallel algorithms for matrix multiplication. However, the method is hard to generalise for rectangular grids and matrices. It has therefore since been superseded by SUMMA (scalable universal matrix multiply algorithm) [VDGW97], which has overcome the restrictions imposed by Cannon and uses blocked computations and pipelining to improve the performance. SUMMA is also the algorithm implemented in ScaLAPACK.

In recent years, communication-avoiding algorithms have been developed and a communication-optimal parallel 2.5D matrix multiplication (2.5D MM) has been presented [SD11]. This algorithm is based on Cannon's algorithm but uses extra memory to store multiple replicates of the matrices to asymptotically reduce the communication cost. 2.5D MM is actually a generalisation of 2D and 3D methods, which either store a single copy (2D) or $q^{1/3}$ copies (3D) of the matrices, where q is the total number of processes. The 2.5D MM chooses a value c for the number of replications with $1 \leq c \leq q^{1/3}$. The chosen c aims to use the available memory optimally to achieve the lowest communication cost, reducing the bandwidth cost by $c^{1/2}$ and the latency cost by $c^{3/2}$ compared to the 2D version, but increases the memory cost by a factor of c. For rectangular grids and matrices, other methods have been developed, including 2.5D SUMMA [GGDS*12], 3D SUMMA [SPvdG12], hierarchical SUMMA [QHL13] and communication-optimal parallel recursive rectangular matrix multiplication (CARMA) [DEF*13].

In this chapter, we will discuss the numerical problems arising in ABFT due to bit-flips in the exponent and how they can be handled. These important questions have not yet been discussed in the literature. We will show how to protect 2.5D Cannon and 2.5D SUMMA against bit-flips using ABFT. Our novel method 2.5D FTMM can recover from faults after every local matrix multiplication. CARMA will not be discussed in this chapter due to its non-static data layout, which results from the recursive approach. The efficient combination of ABFT with CARMA is left for future research.

10.2 ABFT for Matrix Multiplication

In this section, we first review the classical ABFT method for matrix multiplication and then discuss the current limitations of the existing ABFT methods.

10.2.1 Review of ABFT for Matrix Multiplication

ABFT methods are verification-based and it is assumed that no faults occur during the verification process. This process consists of the following steps: *(i)* recompute the checksums, *(ii)* compare the checksums with the available checksums in the result of the operation, and *(iii)* if a fault is detected, solve a single least squares problem for all discrepancies.

The ABFT matrix multiplication to compute $C = AB$ with $A, B, C \in \mathbb{R}^{n \times n}$ is described in Algorithm 14. First, augmented matrices $A^c \in \mathbb{R}^{(n+d)\times n}$ and $B^r \in \mathbb{R}^{n\times(n+d)}$ are defined as

$$A^c = \begin{pmatrix} A \\ W^\top A \end{pmatrix} \quad \text{and} \quad B^r = \begin{pmatrix} B & BW \end{pmatrix} \quad . \tag{10.1}$$

$W \in \mathbb{R}^{n\times d}$ is the weight matrix and d denotes the number of checksums and therefore also the number of rows and columns that are added to the input matrices A and B. In the original publication of ABFT [HA84], only a single vector (i. e. $d = 1$) with all entries equal to one is used to compute the checksums and detect errors. This choice corresponds to forming the sum of all elements in a row or column. In the literature, a second vector consisting of powers of two is often added, leading to the following weight matrix for $d = 2$:

$$W^\top = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 2^0 & 2^1 & \dots & 2^{n-1} \end{pmatrix} \quad .$$

However, as one can easily deduce, the entries of $W^\top(2,:)$ grow exponentially fast with n and in floating-point arithmetic the large coefficients will make it impossible to detect many errors [RJ94]. In [JA86], the weighted checksum encoding scheme is proposed, which uses d encoding vectors to detect and correct multiple faults. With d row or column checksums ABFT can guarantee the detection of up to d errors and the correction of up to $\lfloor d/2 \rfloor$ errors per row or column.

Additionally, the correction matrix $H \in \mathbb{R}^{d\times(n+d)}$ defined by $H := \begin{pmatrix} W^\top & -I_d \end{pmatrix}$ is required, where $I_d \in \mathbb{R}^{d\times d}$ is the identity matrix of dimension d. The only condition imposed on the encoding vectors is that every possible combination of $d - 1$ columns of H has to be

Algorithm 14 Classical ABFT for Matrix Multiplication (ABFT)

```
Input: $A^c \in \mathbb{R}^{(n+d)\times n}, B^r \in \mathbb{R}^{n\times(n+d)}, W \in \mathbb{R}^{n\times d}$
Output: $C^f \in \mathbb{R}^{(n+d)\times(n+d)}$
1: Unreliable $C^f = A^c B^r$
2: Set all NaN and infinity in $C^f$ to 0
3: $C_1 = W^\top C^f(1:n, 1:n+d)$
4: $C_2 = C^f(1:n+d, 1:n)W$
5: $S_1 = C_1 - C^f(n+1:n+d, :)$
6: $S_2 = C_2 - C^f(:, n+1:n+d)$
7: $I_1 = \{j \in [1,n] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B(:,j)\|_p} > 2(2+\mu_n)\mu_n\} \bigcup$
       $\{k_c \in [1,d] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$
                                        ▷ see Equation 10.8 and Equation 10.10
8: $I_2 = \{i \in [1,n] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|A(i,:)\|_p \|B\|_p \|W(:,k_c)\|_p} > 2(2+\mu_n)\mu_n\} \bigcup$
       $\{k_r \in [1,d] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$
                                        ▷ see Equation 10.7 and Equation 10.9
9: if $I_1 \neq \{\}$ and $I_2 \neq \{\}$ then
10:     Solve $H(:, I_2) \cdot \Delta C = S_1(:, I_1)$ for $\Delta C$
11:     $C^f(I_2, I_1) = C^f(I_2, I_1) - \Delta C$
12: end if
```

linearly independent [JA86]. Multiplying the augmented matrices Equation 10.1 results in the extended matrix

$$C^f = A^c B^r = \begin{pmatrix} AB & ABW \\ W^\top AB & W^\top ABW \end{pmatrix} . \tag{10.2}$$

The upper-left block of C^f is identical to C and augmented by row and column checksums with checksums of the checksums residing in the lower-right block. Since C^f has full checksums, which corresponds to d row *and* d column checksums, $2d$ faults can be detected and d faults can be corrected.

In classical ABFT as shown in Algorithm 14, the multiplication in line 1 is the only operation which is allowed to be unreliable. As we will show in subsection 10.5.2, this restriction is not always necessary.

After the unreliable matrix computation, a fault detection and, if necessary, a correction step are performed. The detection process is shown in lines 3-8 in Algorithm 14. Based on the augmented result matrix C^f, the matrices S_1 and S_2 are computed for the row and column checksums of C^f, respectively. First, W is applied (from the left or the right) to a submatrix of C^f which recomputes the checksums and subsequently the subtractions in lines 5 and 6 form the difference between them and the checksums stored in C^f. The sets of indices I_1 and I_2 computed in lines 7 and 8 indicate the faulty rows and columns detected by evaluating the novel fault detection conditions, which will be discussed in detail in subsection 10.2.2. If faulty values are found, the correction process (lines 9-12 in Algorithm 14) computes a correction matrix ΔC for each faulty value by solving an overdetermined linear least squares problem (line 10). In *exact* arithmetic, as long as the number of faults is less than or equal to d, the erroneous matrix C^f can be fully corrected. If more than d faults occur, the least squares problem is underdetermined and the corrupted elements in C^f cannot be recovered. Finally, the correction matrix ΔC is subtracted from the corrupted values in C^f at the intersection of the indices I_1 and I_2.

10.2.2 ABFT in Floating-Point Arithmetic

The distinction between bit-flips and numerical round-off errors is difficult. As long as the fault detection condition is not too strict, a limited number of false positives can be handled (depending on the number of checksums d). In [WDC*11], fault detection conditions have been defined for the case $d = 1$ based on the standard round-off error bound of the matrix product. It is stated in [WDC*11] that no bit-flips have occurred in row i of C^f and deviations in the checksum are only due to round-off errors, if

$$\left|\Sigma_{j=1}^{n} C^f(i,j) - C^f(i,n+1)\right| \leq \mu_n \left\|A^c\right\|_\infty \left\|B^r\right\|_\infty$$

and analogously, no bit-flips have occurred in column j of C^f if

$$\left|\Sigma_{i=1}^{n} C^f(i,j) - C^f(n+1,j)\right| \leq \mu_n \left\|A^c\right\|_1 \left\|B^r\right\|_1$$

where $\mu_n = \frac{nu}{1-nu}$ and u is the unit round-off error.

The fact that the computation of the checksums itself is also affected by numerical round-off errors has not been considered in the literature, mainly due to most of the time only a single weight vector being used whose elements are all equal to one. We therefore generalise the previous fault detection conditions for $d \geq 1$ encoding vectors, which requires the consideration of the weight matrix W on both sides of the inequalities. To derive this generalisation, we use the well-known error bound of the matrix product in floating-point arithmetic described in [Hig02]. There, the computed result $\hat{C}$ is defined as $\hat{C} = [A + \Delta A]B$ and the exact result as $\bar{C} = AB$. For ΔA, it is shown that

$$\left\|\Delta A\right\|_p \leq \mu_n \left\|A\right\|_p \tag{10.3}$$

where $p = 1, \infty, F$. The error bound is then stated as

$$\left\|\bar{C} - \hat{C}\right\|_p = \left\|AB - [A + \Delta A]B\right\|_p = \left\|\Delta AB\right\|_p \leq \mu_n \left\|A\right\|_p \left\|B\right\|_p \quad .$$

We now apply the same steps to derive the error bounds for each individual checksum block of C^f in Equation 10.2.

We denote the exact result without numerical errors and unaffected by faults by $\bar{C}^f$ and the computed result with numerical errors but no faults due to bit-flips by $\hat{C}^f$. C^f denotes the computed result from the unreliable matrix multiplication (line 1 in Algorithm 14), which, in addition to numerical errors, can also be affected by bit-flips. For any row $k_r \in [1, d]$, each representing a single column-checksum entry, and any column $j \in [1, n]$, located in the lower left block $W^\top AB$ of C^f, we have

$$\begin{aligned}\hat{C}^f(n+k_r, j) =& \left[W^\top(k_r,:) + \Delta W^\top(k_r,:)\right][A + \Delta A]B(:,j) \\ =& \, W^\top(k_r,:)\cdot A \cdot B(:,j) + \left[W^\top(k_r,:) + \Delta W^\top(k_r,:)\right]\Delta A \cdot B(:,j) + \\ & \Delta W^\top(k_r,:)A\cdot B(:,j) \quad .\end{aligned}$$

The error bound for a single column-checksum entry is then defined as

$$\begin{aligned}\left|\bar{C}^f(n+k_r,j) - \hat{C}^f(n+k_r,j)\right| \leq& \left\|W^\top(k_r,:) + \Delta W^\top(k_r,:)\right\|_p \mu_n \|A\|_p \|B(:,j)\|_p + \\ & \mu_n \left\|W^\top(k_r,:)\right\|_p \|A\|_p \|B(:,j)\|_p \\ \leq& \, (2+\mu_n)\,\mu_n \left\|W^\top(k_r,:)\right\|_p \|A\|_p \|B(:,j)\|_p \quad . \end{aligned} \tag{10.4}$$

Analogously, for any column $k_c \in [1, d]$, each representing a single row-checksum entry, and any row $i \in [1, n]$, located in the upper right block ABW of C^f, we obtain the error bound

$$\left|\bar{C}^f(i, n+k_c) - \hat{C}^f(i, n+k_c)\right| \leq (2+\mu_n)\,\mu_n \|A(i,:)\|_p \|B\|_p \|W(:,k_c)\|_p \quad . \tag{10.5}$$

For the lower right block $W^\top ABW$ of C^f, which contains the checksums of the checksums, we have

$$\begin{aligned}\hat{C}^f(n+k_r, n+k_c) =& \left[W^\top(k_r,:) + \Delta W^\top(k_r,:)\right][A+\Delta A]\,B\,[W(:,k_c) + \Delta W(:,k_c)] \\ =& \, W^\top(k_r,:)\cdot A\cdot B\cdot W(:,k_c) + \\ & \Delta W^\top(k_r,:)\,[A+\Delta A]\cdot B\,[W(:,k_c)+\Delta W(:,k_c)] + \\ & W^\top(k_r,:)\Delta AB\,[W(:,k_c)+\Delta W(:,k_c)] + W^\top(k_r,:)AB\Delta W(:,k_c)\end{aligned}$$

where $k_r, k_c \in [1, d]$. The corresponding error bound is expressed as follows:

$$\left|\bar{C}^f(n+k_r, n+k_c) - \hat{C}^f(n+k_r, n+k_c)\right| \leq \xi \tag{10.6}$$

where

$$\begin{aligned}\xi = {} & \mu_n \left\| W^\top(k_r,:) \right\|_p (1+\mu_n) \left\| A \right\|_p \left\| B \right\|_p (1+\mu_n) \left\| W(:,k_c) \right\|_p + \\ & \mu_n \left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B \right\|_p (1+\mu_n) \left\| W(:,k_c) \right\|_p + \\ & \mu_n \left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B \right\|_p \left\| W(:,k_c) \right\|_p \\ = {} & \mu_n(3+3\mu_n+\mu_n^2) \left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B \right\|_p \left\| W(:,k_c) \right\|_p \quad .\end{aligned}$$

Transforming inequalities Equation 10.4, Equation 10.5 and Equation 10.6 to fault detection conditions leads to the following four conditions for the different blocks of C^f. Naturally, we do not have access to the exact result $\bar{C}^f$. Therefore, we use the re-computed checksums from lines 3 and 4 in Algorithm 14 as the values of $\bar{C}^f$. The checksums from the unreliable matrix multiplication in line 1 of Algorithm 14 are used for $\hat{C}^f$. Both checksums are computed in finite precision. They use the same operations but only differ in their order. In the worst case, the second computation of the checksums could double the numerical error. Therefore, the numerical error for both checksums is considered in the four fault detection conditions by multiplying the right-hand side by 2.

Based on Equation 10.5, for the row checksums, i. e. the upper right block of C^f, a fault is detected in row $i \in [1,n]$ if

$$\max_{k_c \in [1,d]} \frac{\left| \Sigma_{j=1}^n C^f(i,j)W(j,k_c) - C^f(i,n+k_c) \right|}{\left\| A(i,:) \right\|_p \left\| B \right\|_p \left\| W(:,k_c) \right\|_p} > 2\,(2+\mu_n)\,\mu_n \quad . \tag{10.7}$$

The lower left block of C^f contains the column checksums and based on Equation 10.4, a fault is detected in column $j \in [1,n]$ if

$$\max_{k_r \in [1,d]} \frac{\left| \Sigma_{i=1}^n W^\top(k_r,i)C^f(i,j) - C^f(n+k_r,j) \right|}{\left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B(:,j) \right\|_p} > 2\,(2+\mu_n)\,\mu_n \quad . \tag{10.8}$$

To detect faults in the checksums of the checksums in the lower right block of C^f the following two different conditions are formulated, depending on which checksums are used in the comparison. Based on Equation 10.6, for the column checksum $k_r \in [1,d]$ a fault is detected if

$$\max_{k_c \in [1,d]} \frac{\left| \Sigma_{j=1}^n C^f(n+k_r,j)W(j,k_c) - C^f(n+k_r,n+k_c) \right|}{\left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B \right\|_p \left\| W(:,k_c) \right\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2) \tag{10.9}$$

and for the row checksum $k_c \in [1,d]$ a fault is detected if

$$\max_{k_r \in [1,d]} \frac{\left| \Sigma_{i=1}^n W^\top(k_r,i)C^f(i,n+k_c) - C^f(n+k_r,n+k_c) \right|}{\left\| W^\top(k_r,:) \right\|_p \left\| A \right\|_p \left\| B \right\|_p \left\| W(:,k_c) \right\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2) \quad . \tag{10.10}$$

Both conditions have to be evaluated to ensure the detection of all faults in the checksums of the checksums. The numerator of Equation 10.7 and Equation 10.9 corresponds to $|S_2(i, k_c)|$ (see line 6 in Algorithm 14) and of Equation 10.8 and Equation 10.10 to $|S_1(k_r, j)|$ (see line 5 in Algorithm 14).

The conditions only have to hold for one of the checksums in k_c or k_r to indicate the existence of a faulty value in the corresponding row or column, respectively. For example, a fault is detected in row i of C^f if condition Equation 10.7 holds for at least one $k_c \in [1, d]$. Both conditions Equation 10.7 and Equation 10.8 (or Equation 10.9 and Equation 10.10 for the checksums of the checksums) have to be fulfilled to retrieve the column indices I_1 and the row indices I_2. The locations of the faulty values are given by the Cartesian product of I_1 and I_2 (see lines 7-8 in Algorithm 14). If only a column or only a row index is found, then this fault is treated as a numerical rounding error.

In subsection 10.4.2, we will illustrate experimentally that Conditions Equation 10.7-Equation 10.10 guarantee highly accurate fault detection despite bit-flips.

10.2.3 Limitations of Existing ABFT Methods

Unfortunately, in all exiting ABFT methods the ability to correct faults is limited. More specifically, the limitations can be distinguished along two main dimensions – temporal and spatial limitations.

Temporal limitations In the literature, faults are generally only allowed to occur during the matrix multiplication (line 1 in Algorithm 14) and not during the detection and correction process. More precisely, after an element of the matrix C^f has been checked for faults and is considered to be correct, no faults are allowed to occur in that element. This implies that faults are also allowed to occur in lines 3 and 4 of Algorithm 14. In floating-point arithmetic, this is only possible as long as the fault does not change the value to NaN. Any faults occurring during or after the correction process would not be detected or corrected and lead to an incorrect result in C. This limitation cannot be completely eliminated, but in the context of the 2.5D matrix multiplication it can be postponed until after the last distributed computation, as we will show in subsection 10.5.2.

Spatial limitations In floating-point arithmetic, classical ABFT can only handle faults in the mantissa or sign bits, but in practice faults can obviously occur at any position of the floating-point representation. Bit-flips in the exponent bits are much harder to correct due to numerical aspects. For example, consider the special case where all exponents of the original data are zero and a bit-flip changes one exponent to $\beta > 0$, then in the worst case (in double precision) the error caused by the fault can be reduced at most to $O(10^{-16+\beta})$ due to the cancellation of the least significant bits during the computation of S_1 and S_2. For $\beta \geq 16$ this leads to all bits being incorrect. This also implies that classical ABFT does not work if the magnitudes of the elements of the input data differ widely.

In the following, we provide a simple example in more detail for demonstrating the spatial limitation of the fault correction of classical ABFT. Without loss of generality, we use the decimal system for easier illustration, with a maximum of 16 decimal digits. Consider the matrix C as a scalar with $C = 1.23999$ being the correct value before a fault has occurred. Using $W = 1$, the checksum is then also 1.23999. Due to a fault, the value of C is changed to $1.1 \cdot 10^{14}$. To correct the fault, this erroneous value is subtracted from the checksum. During this subtraction, the checksum value is normalised to the same exponent as the new value of C and therefore truncated to 1.23. The truncation error is $O(10^{-2})$ (corresponding to the case of $\beta = 14$ in the previous paragraph) and results in a corrected C of 1.23.

10.3 Improving the Resilience of ABFT

In the literature, bit-flips are implicitly modelled to only affect the mantissa of a floating-point number. Our experiments demonstrate that, due to the spatial limitations mentioned in subsection 10.2.3, classical ABFT cannot handle all bit-flips occurring in the exponent (see subsection 10.4.2). In this section, we improve the existing ABFT methods for matrix multiplication to be able to detect and correct bit-flips in all parts of a floating-point number, in particular allowing bit-flips also to occur in the exponent.

A large change in an exponent dominates the checksum of a faulty row or column. The correction can only reduce the fault by the maximum accuracy of the floating-point representation (about 16 decimal digits in double precision). Our novel idea is to compute the correction matrix ΔC *without* using the corrupted values. Therefore, we set all faulty elements in C^f to zero (line 9 in Algorithm 15) and recompute the corresponding part of the matrices S_1 and S_2 (lines 10-13). This enables our improved ABFT method to handle any bit-flips in the result, including all bits of the exponent, without significant overhead compared to classical ABFT (see section 10.4). We call our novel method *direct ABFT* (*dABFT*), motivated by the way the correction matrix is computed. The resulting algorithm for dABFT matrix multiplication is shown in Algorithm 15.

The formal proof of correctness of ABFT is not affected by the modifications introduced here for dABFT. Each detected faulty value is just replaced with zero, which is in itself just a different faulty value. Instead of computing the difference between the faulty value and the checksums (like in classical ABFT), dABFT computes the correct value directly. By setting the faulty value to zero, we reduce all possible fault scenarios to a single fault scenario, one that can already be handled by classical ABFT.

We return to the simple example discussed in subsection 10.2.3 to illustrate that classical ABFT can only guarantee a correction up to $O(10^{-16+\beta})$, and now show the effect of our modifications to ABFT. In dABFT, the faulty value $C = 1.1 \cdot 10^{14}$ is set to zero, which leads to the checksum value not being truncated and therefore the faulty value being correctly retrieved using the checksum value. Thus, $C = 1.23999$, having full accuracy after the correction.

Algorithm 15 Direct ABFT for Matrix Multiplication (dABFT)

```
Input: $A^c \in \mathbb{R}^{(n+d)\times n}, B^r \in \mathbb{R}^{n\times(n+d)}, W \in \mathbb{R}^{n\times d}$
Output: $C^f \in \mathbb{R}^{(n+d)\times(n+d)}$
 1: Unreliable $C^f = A^c B^r$
 2: $C_1 = W^\top C^f(1:n, 1:n+d)$
 3: $C_2 = C^f(1:n+d, 1:n)W$
 4: $S_1 = C_1 - C^f(n+1:n+d, :)$
 5: $S_2 = C_2 - C^f(:, n+1:n+d)$
 6: $I_1 = \{j \in [1,n] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\left\|W^\top(k_r,:)\right\|_p \|A\|_p \|B(:,j)\|_p} > 2\left(2+\mu_n\right)\mu_n\} \bigcup$
        $\{k_c \in [1,d] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\left\|W^\top(k_r,:)\right\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$
                                                        ▷ see Equation 10.8 and Equation 10.10
 7: $I_2 = \{i \in [1,n] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|A(i,:)\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\left(2+\mu_n\right)\mu_n\} \bigcup$
        $\{k_r \in [1,d] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\left\|W^\top(k_r,:)\right\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$
                                                        ▷ see Equation 10.7 and Equation 10.9
 8: if $I_1 \neq \{\}$ and $I_2 \neq \{\}$ then
 9:     $C^f(I_2, I_1) = 0$
10:     $C_1(:, I_1) = W^\top C^f(:, I_1)$
11:     $C_2(I_2, :) = C^f(I_2, :)W$
12:     $S_1(:, I_1) = C_1 - C^f(n+1:n+d, I_1)$
13:     $S_2(I_2, :) = C_2 - C^f(I_2, n+1:n+d)$
14:     Solve $H(:, I_2) \cdot C^f(I_2, I_1) = -S_1(:, I_1)$ for $C^f(I_2, I_1)$
15: end if
```

Aside from the correction of bit-flips in the exponent, dABFT can also handle the special floating-point values NaN and infinity in C^f implicitly. The conditions for I_1 and I_2 in lines 6 and 7 of Algorithm 15 can be implemented to be true automatically for any of these special values, due to the definition of the comparison operators for these values in the IEEE-754 specification [IEE08] (by reformulating the conditions to check whether the left side is "$\not\leq$" the right side). This strategy is useless when applied to classical ABFT since NaN and infinity already influence the computation of S_1 and S_2. Therefore, classical ABFT has to check for the existence of these special values in C^f explicitly before starting the fault detection process (see line 2 in Algorithm 14). dABFT can combine both steps through the proper implementation of the conditions and therefore reduces the overhead compared to classical ABFT. This is possible because S_1 and S_2 are recomputed after setting all faulty values (including NaN and infinity) to zero (see lines 12 and 13 in Algorithm 15). The performance increase due to this advantage is shown in subsection 10.4.3, where we conduct experiments for both variants of ABFT.

The performance of classical ABFT and dABFT are dominated by the matrix multiplication of order $O\left(n^3 + n^2 d\right)$ in line 1 of both algorithms. The detection and correction steps of classical ABFT and lines 2-7 in Algorithm 15 for dABFT are of order $O\left(n^2 d\right)$, an order of magnitude lower than the dominating operation (as d is normally much smaller than n). The additional detection and correction steps in dABFT (lines 9-13) are only of order

$O\left(nd^2\right)$ because they only recompute the parts of the factors S_1 and S_2 which used the faulty values in lines 4 and 5. Moreover, as already mentioned, dABFT does not require the explicit checking for NaN and infinity (see line 2 in Algorithm 14), which saves $O\left(n^2\right)$ comparison operations. Overall, the performance benefits from the implicit handling of the special floating-point values.

Compared to classical ABFT, dABFT does not introduce additional numerical errors but actually improves the numerical accuracy. Any numerical floating-point errors caused by lines 9-13 in Algorithm 15 are guaranteed to be lower than in lines 2-7 due to the faulty values being removed from the matrix. The values of S_1 and S_2 are overwritten with the new values in lines 12 and 13 and are therefore more accurate than the ones computed with the faulty values in lines 4 and 5.

As mentioned in subsection 10.2.3 (cf. temporal limitations), faults are also allowed to occur in lines 2 and 3 of Algorithm 15, as long as they occur before the value of C^f has been checked for faults. In contrast to classical ABFT, dABFT can also handle the case of a fault changing a value to NaN, as the value would subsequently be set to zero during the detection phase and not influence the computation of the least squares problem in line 14.

10.4 Experimental Evaluation of dABFT

In this section, we experimentally demonstrate the fault resilience of dABFT. All experiments were run on a single core of an AMD Opteron Processor 6174 and OpenBLAS (version 0.2.14) was used as the optimised BLAS library. For all experiments, the checksum encoding matrix W is generated using random, uniformly distributed values between 0 and 1. Faults are injected randomly using our approach summarised in subsection 10.4.1. The resilience of classical ABFT and dABFT is compared in subsection 10.4.2 and the low overhead of both ABFT variants is shown in subsection 10.4.3.

10.4.1 Fault Injection

As described in Chapter 9, we designed our own fault injector FaITh for the experiments in subsection 10.4.2 to inject faults randomly in time and space. The time interval between two fault injections is exponentially distributed with a specified mean time to failure (MTTF) per byte. The position at which a fault is injected is uniformly distributed, ensuring that all bits have the same chance of being flipped. For the comparison of classical ABFT and dABFT, the specification of the injection range is very important. FaITh can specify the injection range to cover any bit of a floating-point number or can restrict the range to the mantissa, the exponent or the sign bit. Due to the injector threads running asynchronously, the injections can also take place in library calls, e. g. BLAS `dgemm`, without any source code modifications or recompilation.

Our fault injector has a very low overhead and hardly influences the performance results of the tested algorithms. In these experiments, the overhead is on average only about 1% of the total execution time, independent of the number of cores used.

10.4.2 Resilience of ABFT Variants

In this section, we compare dABFT as presented in section 10.3 to classical ABFT in terms of resilience. It suffices to demonstrate the resilience properties of the ABFT variants on a single core, because the 2.5D algorithms presented in section 10.5 only use ABFT locally.

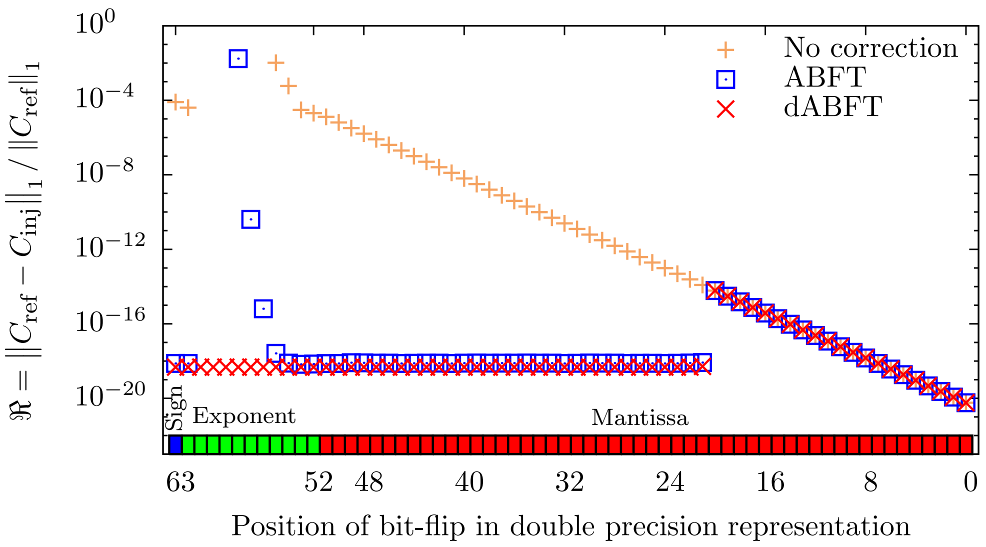


Figure 10.1: Error of the matrix product $C = AB$ after correcting a single bit-flip for different positions of the bit-flip.

We begin with injecting a single bit-flip deterministically into the element (1,1) of the result of the matrix multiplication $C = AB$. Subsequently, classical ABFT and dABFT recover from the fault. Figure 10.1 shows an average over 100 experiments with the position of the bit-flip on the x-axis and the resulting relative error $\Re := \|C_{\text{ref}} - C_{\text{inj}}\|_1 \,/\, \|C_{\text{ref}}\|_1$ on the y-axis, where the result matrix without any bit-flips is denoted by C_{ref} and the result matrix after injecting and correcting a bit-flip by C_{inj}.

As one can see in Figure 10.1, when flipping one of the least significant bits of the mantissa, the error increases with the position of the bit-flip for all methods. The effects of these faults are so small that they are below the thresholds of the fault detection conditions Equation 10.7-Equation 10.10. Starting with the 21st bit of the mantissa, the conditions are satisfied and both methods actually correct the injected bit-flip. The quality of the correction of classical ABFT and dABFT are equal up to the 4th bit of the exponent. Then, the error of classical ABFT increases, in the worst case, exponentially. For a bit-flip in the 7th to 10th bit of the exponent all digits of the result produced by classical ABFT are incorrect. This is due to the fact that it tries to correct the faulty data by subtracting a correction value. This correction value can only be correct up to 16 digits due to the limited numerical representation of double precision. For the 11th bit in the exponent, the result of classical ABFT is again correct because in all our experiments the bit-flip *decreased* the value (the 11th bit of the exponent was originally 1). Consequently, for classical ABFT, the quality of the correction strongly depends on the location of the bit-flip. In contrast, dABFT sets the faulty value to zero and computes the correct value directly, which results in an accurate result in all cases, regardless of the position of the bit-flip. In these experiments, only a bit-flip in the most significant bit of the exponent decreases the value of the floating-point number. In general, a bit-flip in the exponent can either decrease or increase the value (depending on whether the flipped bit is 1 or 0) and therefore it can cause small as well as large faults. Thus, the effect of the fault does not only depend on the position of the flipped bit but also on the floating-point number in which the bit-flip occurred (for details see [EHM16]).

The next step is an exhaustive investigation of the resilience of both ABFT methods against any location of bit-flips occurring during the matrix multiplication using our non-deterministic fault injection method described in subsection 10.4.1. We ran more than 12 000 matrix multiplications with problem size $n = 1000$ for each ABFT variant. The experiments covered different numbers of checksums $d \in \{1, 3, 5, 10, 15, 20, 25, 50, 100\}$ and different MTTFs, always ensuring that at most d faults were injected. Figure 10.2 illustrates the achieved correction quality by a cumulative distribution of the resulting relative errors. For both ABFT variants, we use exactly the same set of random injection points in time and position to make the results as comparable as possible. However, due to the asynchronicity of the fault injection thread it cannot be guaranteed that different individual runs are identical. Nevertheless, the huge number of experiments leads to a high confidence in the comparison of the two methods.

We have seen in Figure 10.1 that the quality of classical ABFT strongly depends on the position of the bit-flip. Therefore, the results for the accuracy of classical ABFT in Figure 10.2 are split into three groups: faults in any bit of the floating-point number, faults only in the mantissa or faults only in the exponent. Focusing on the fault detection condition, Figure 10.2 shows the error of dABFT being less than 10^{-13} in all tested cases. In comparison, for classical ABFT this is only true for bit-flips occurring in the mantissa. If faults are also injected into the exponent, the error can become unacceptably large. For example, a bit-flip in the exponent can result in the faulty value being smaller than the original number, a case which can also be handled by classical ABFT. However, in the worst case, if a bit-flip increases the affected matrix element, classical ABFT can reduce the error by a factor of $O(10^{-16})$ and will fail to produce a correct result, as discussed in subsection 10.2.3 (cf. spatial limitations). For bit-flips at any position in the floating-point number, the result of classical ABFT was incorrect in all digits in more than 20% of all tested cases (see Figure 10.2). Therefore, in a realistic system, where bit-flips can occur in any bit, the classical ABFT method cannot be used for fault tolerant matrix multiplications. The step-like pattern in Figure 10.2 for classical ABFT (e. g. at 10^{-13}) illustrates that the error varies strongly due to the influence of different bit-flips in the exponent. From one bit to the next, a single fault can potentially

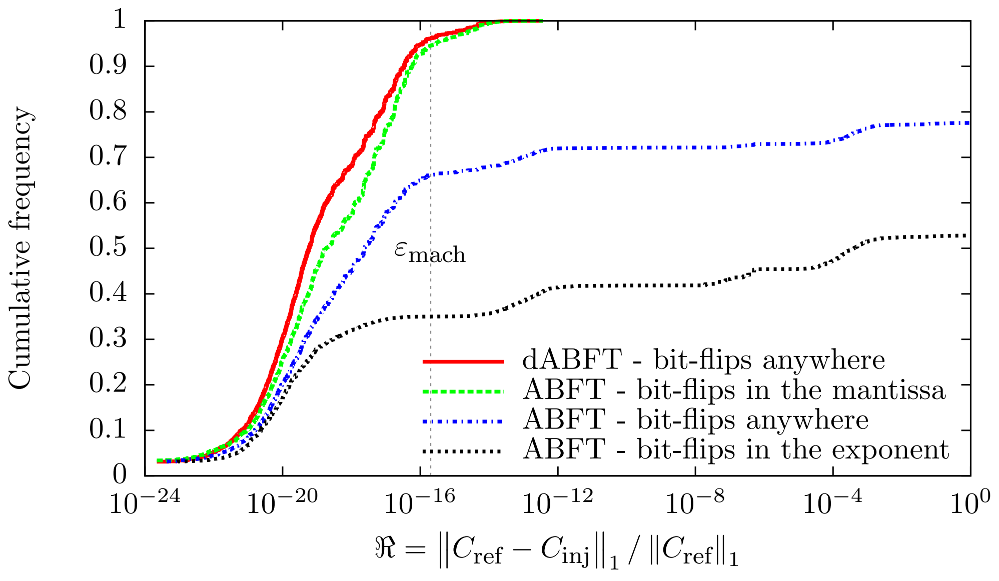


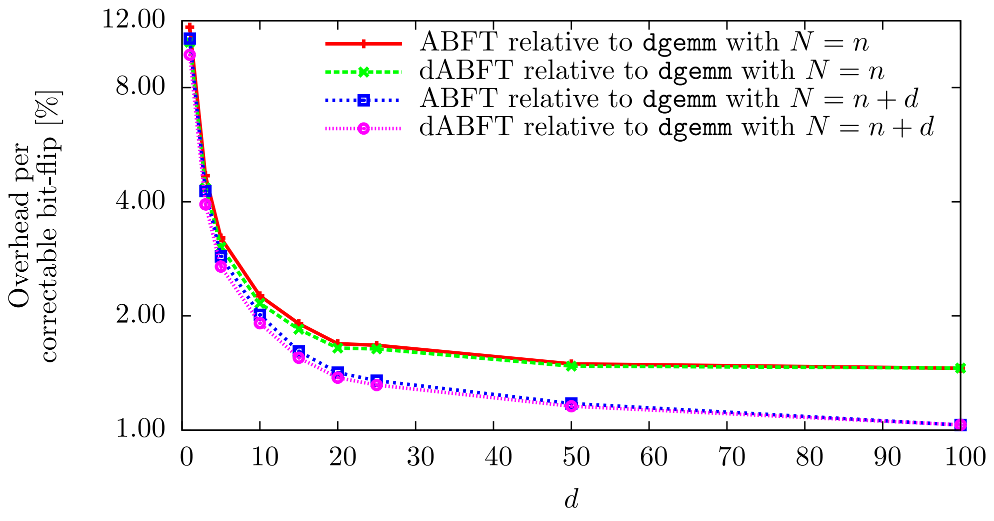
Figure 10.2: Cumulative distribution of $\Re$ for classical ABFT and dABFT with $n = 1000$. “bit-flips anywhere” means that bit-flips were injected in all bits of the floating-point numbers, including the sign bit. These results include single as well as multiple bit-flips per matrix multiplication.

double the exponent of the error. Moreover, as discussed at the end of section 10.3, dABFT handles bit-flips in the mantissa numerically more accurately than classical ABFT because setting the faulty value to zero basically reduces the number of floating-point operations (an addition or multiplication in floating-point arithmetic with one operand being zero always has an error of zero). However, this effect can only really be seen if the residual is smaller than the machine epsilon ε.

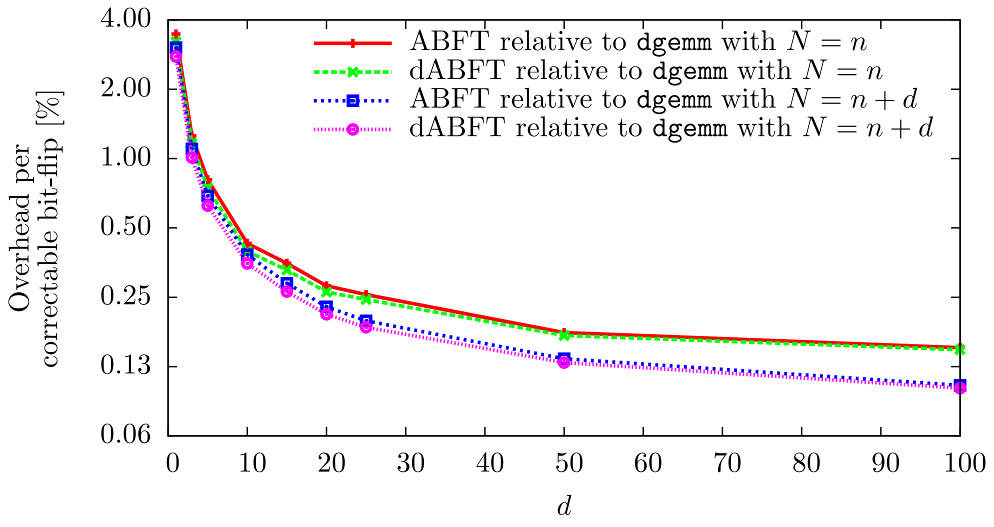
10.4.3 Runtime Performance of ABFT Variants

In the previous section, we demonstrated the superiority of dABFT over classical ABFT in terms of resilience. Now, we compare both methods in terms of runtime performance. In Figure 10.3, the runtime overhead of classical ABFT and dABFT is compared to a standard (non-fault tolerant) matrix multiplication of two $N \times N$ matrices (using `dgemm` from OpenBLAS). On the x-axis, the number of checksums d per row or column is shown. The overhead per correctable bit-flip is depicted on the y-axis. The first set of lines refer to the overhead caused by ABFT compared to `dgemm` with the original matrix dimension $N = n$. The second set of lines shows the overhead of the correction and detection steps of ABFT by comparing it to `dgemm` with $N = n + d$, the same size as the augmented matrices A^c and B^r. In this comparison, the overhead caused by the larger matrices is ignored and only the additional steps of ABFT are considered.

dABFT is always faster than classical ABFT due to the fact that it does not need to check for NaNs or infinity explicitly, as explained in section 10.3. For $n = 1000$ (Figure 10.3a), the



(a) Overhead for $n = 1000$



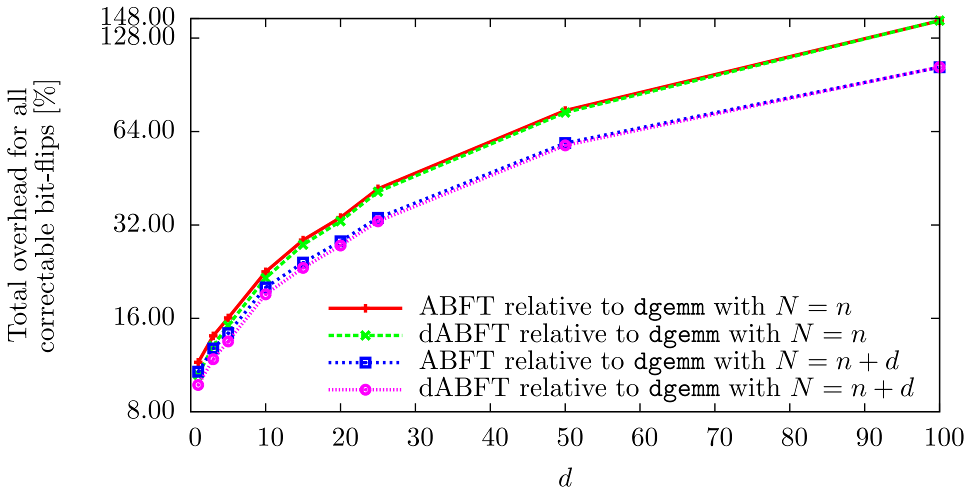
(b) Overhead for $n = 5000$

Figure 10.3: Overhead per correctable bit-flip of ABFT and dABFT relative to BLAS `dgemm`.

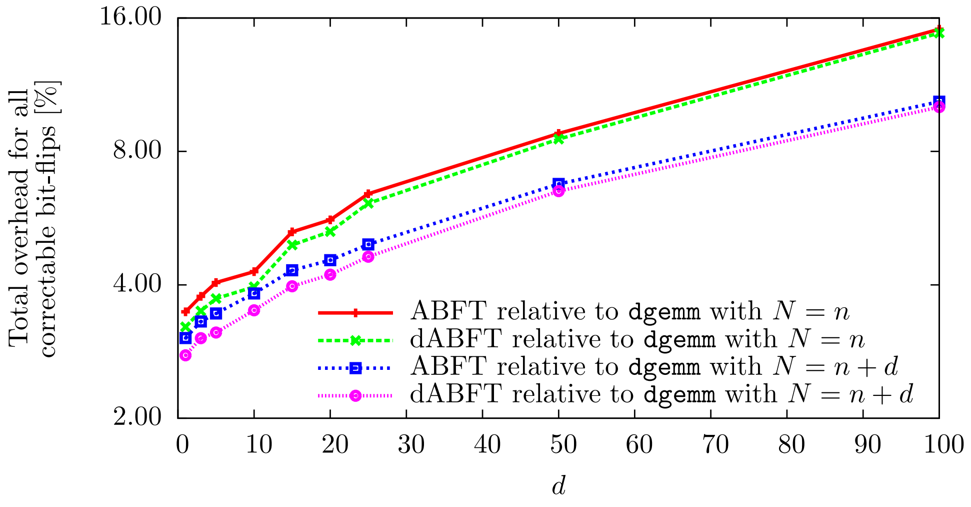
overhead per correctable bit-flip decreases in all cases and reaches a value of about 1.45% for the entire overhead ($N = n$) and 1.03% for the detection and correction process ($N = n + d$). About half of the overhead is caused by the increased matrix size and the other half by the detection and correction process. A more detailed analysis shows that the correction itself is only responsible for up to 10% of the total overhead because the least squares problems to be solved are at most of size d.

Figure 10.3b shows the same analysis of the overhead for larger matrices with $n = 5000$. As expected, the larger matrix size significantly decreases the overhead due to the impact of the number of checksums being smaller relative to the matrix size. In this case, the entire overhead for classical ABFT and dABFT is only about 0.15%. For the detection and correction process it is only about 0.1%, again about half of the total overhead. These results highlight the extremely low overhead, especially for large matrices and high fault rates.

The total overhead for all correctable bit-flips (shown in Figure 10.4) naturally increases with an increasing failure rate for a fixed n, but the overhead does not grow as fast as the number of checksums. Furthermore, for the same failure rates, the overhead caused by the increase in checksums decreases with the increase of the matrix size. For dABFT and $n = 1000$ (see Figure 10.4a), the total overhead is up to 7.23 times lower than the increase of d from 1 to 100, with 10.5% for $d = 1$ and 145% for $d = 100$. Increasing the matrix size to $n = 5000$ (shown in Figure 10.4b), the total overhead is even up to 21.73 times lower than the increase of d, with 3.22% for $d = 1$ and 14.8% for $d = 100$. This is already a significant decrease compared to $n = 1000$ and for the same fault rates the overhead will decrease even more for larger matrices.



(a) Total overhead for $n = 1000$



(b) Total overhead for $n = 5000$

Figure 10.4: Total overhead for all correctable bit-flips of ABFT and dABFT relative to BLAS `dgemm`.

Algorithm 16 2.5D SUMMA

Input: $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}$

Output: $C \in \mathbb{R}^{n \times n}$

```
1: for each process $(i, j, \kappa) \in \Pi$ do
2:     Broadcast $A_{i,j}$, $B_{i,j}$ to $\Pi(i, j, :)$
3:     for $t = (\kappa - 1)\sqrt{q/c^3} + 1 : \kappa\sqrt{q/c^3}$ do
4:         $A_{\text{local}}$: Broadcast $A_{t,j}$ to $\Pi(i, :, \kappa)$
5:         $B_{\text{local}}$: Broadcast $B_{i,t}$ to $\Pi(:, j, \kappa)$
6:         $C_{\text{local}} = C_{\text{local}} + A_{\text{local}} B_{\text{local}}$
7:     end for
8:     $C_{i,j} \leftarrow$ Sum-reduction of $C_{\text{local}}$ over $\Pi(i, j, :)$
9: end for
```

10.5 Fault Tolerant 2.5D Matrix Multiplication

In this section, we present and discuss the *2.5D FTMM* algorithm, which results from integrating ABFT into 2.5D MM. As discussed in section 10.1, 2.5D Cannon [SD11] is restricted to square matrices, whereas 2.5D SUMMA [GGDS*12] can handle rectangular grids and matrices. For simplicity, we demonstrate our approach for the 2.5D SUMMA algorithm. However, all modifications can also be applied to 2.5D Cannon.

10.5.1 2.5D Matrix Multiplication

For the 2.5D MM we define a three-dimensional process grid $\Pi \in \mathbb{N}^{\sqrt{q/c} \times \sqrt{q/c} \times c}$, where q is the number of processes and c the size of the third dimension. In 2.5D SUMMA (see Algorithm 16), c is the number of copies of the input matrices A and B that are distributed along the third dimension of the grid in the first step of the algorithm. Initially, process $\Pi(i, j, 1)$ stores blocks $A_{i,j} := A((i-1)b+1 : ib, (j-1)b+1 : jb)$ and $B_{i,j} := B((i-1)b+1 : ib, (j-1)b+1 : jb)$, where $b = n/\sqrt{q/c}$ is the local block size. First, these blocks are broadcast along the third dimension. Then, iteratively, along each column of $\Pi(:, :, \kappa)$ one block of A and along each row of $\Pi(:, :, \kappa)$ one block of B is broadcast. The received blocks A_{local} and B_{local} are multiplied locally and added to C_{local}. Finally, the local results are summed along the third dimension $\Pi(i, j, :)$, so that $\Pi(i, j, 1)$ receives the sum $C_{i,j}$, which corresponds to the block of the result matrix C.

10.5.2 Combining dABFT and 2.5D SUMMA

In 2.5D SUMMA, the local matrix multiplication in line 6 of Algorithm 16 can be replaced by dABFT to allow bit-flips to be detected and corrected in the parallel algorithm without requiring any additional messages. To achieve this, the checksums are not added to the global matrix but to each local block A^c_{local} and B^r_{local}. This also leads to the detection and correction process being performed locally on each node. Therefore, no additional communication is incurred, thus preserving the communication-avoiding properties of the 2.5D algorithms.

Naturally, the local computation time is slightly increased due to the augmented system size, but the overhead is negligible for a small number of checksums d. Slightly more data has to be sent due to the additional rows and columns in the augmented blocks A^c_{local} and B^r_{local}. For small d, the additional data is very small in relation to the protected data and becomes negligible for growing matrix dimensions.

Applying the checksums to local blocks instead of the global matrix also improves the numerical properties of the fault detection. Rexford and Jha [RJ94] showed that, depending on the choice of the encoding vectors, numerical problems become more severe with the size of the data. They therefore proposed the partitioned encoding scheme to reduce the numerical inaccuracies occurring in the computation and comparison of the checksums. Naturally, from a global perspective, more memory is required for adding $2d \cdot n/b$ elements to the local matrix block. However, each block of the matrix is protected by d checksums which also allows more faults to be tolerated compared to the global approach. If the same fault rate is considered for both approaches, faults are expected less often in smaller amounts of data. Therefore, the number of checksums per block can be reduced, which decreases the overhead per block. The authors in [RJ94] also showed that all matrix operations discussed in [HA84] preserve the partitioned checksum property. Therefore, existing ABFT algorithms can directly benefit from the use of block-level checksums without having to be modified. The partitioned encoding scheme improves the upper bound on the round-off error by a factor of $O(\|W^{(n)}\|_2 / \|W^{(b)}\|_2)$, where $W^{(n)} \in \mathbb{R}^{n\times d}$ is the encoding matrix for global checksums and $W^{(b)} \in \mathbb{R}^{b\times d}$ for block-level checksums. The ability to detect faults is also improved, because the maximum tolerated error is decreased by this factor. Consequently, the blocked encoding scheme does not increase the number of false positives and at the same time allows for more faults at lower magnitudes to be detected than the global approach.

Our fault-tolerant 2.5D matrix multiplication (2.5D FTMM) is shown in Algorithm 17. The data distribution is the same as described in subsection 10.5.1 for 2.5D SUMMA. The only difference is that each local block of A and B is augmented by the block-level checksums. The extended blocks are broadcast and used for the local ABFT matrix multiplication (lines 6-10 in Algorithm 17). Furthermore, the resulting local block of C is also extended by full checksums (rows and columns) at the block-level. Replacing the local matrix multiplication by an ABFT-based matrix multiplication ensures that bit-flips occurring during the multiplication can be corrected. A final detection and correction step consisting of lines 2-15 in Algorithm 15 has to be inserted at the end (line 13 in Algorithm 17) to also cover faults that occur during the reduction process.

As mentioned in subsection 10.2.3 (cf. temporal limitations), the detection and correction process in classical ABFT has to run fault free because it runs at the end of the computation and the returned result matrix C^f has to be guaranteed to be correct. In contrast, in 2.5D FTMM this constraint can be lifted for the multiple local matrix multiplications which are performed throughout the method. Therefore, except for the final detection and correction process in line 13, 2.5D FTMM is *completely fault tolerant*. There are no further restrictions

Algorithm 17 Fault-tolerant 2.5D matrix multiplication (2.5D FTMM)

```
Input: $A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times n}, W \in \mathbb{R}^{b\times d}$, frequency $F$
Output: $C \in \mathbb{R}^{n\times n}$
1:  for each process $(i, j, \kappa) \in \Pi$ do
2:      Broadcast $A^c_{i,j}$, $B^r_{i,j}$ to $\Pi(i, j, :)$
3:      for $t = (\kappa - 1)\sqrt{q/c^3} + 1 : \kappa\sqrt{q/c^3}$ do
4:          $A^c_{\text{local}}$: Broadcast $A^c_{t,j}$ to $\Pi(i, :, \kappa)$
5:          $B^r_{\text{local}}$: Broadcast $B^r_{i,t}$ to $\Pi(:, j, \kappa)$
6:          if $(t - (\kappa - 1)\sqrt{q/c^3}) \bmod F = 0$ then
7:              $C^f_{\text{local}} = C^f_{\text{local}} + dABFT(A^c_{\text{local}}, B^r_{\text{local}})$
8:          else
9:              $C^f_{\text{local}} = C^f_{\text{local}} + A^c_{\text{local}} B^r_{\text{local}}$
10:         end if
11:     end for
12:     $C_{i,j} \leftarrow$ Sum-reduction of $C_{\text{local}}$ over $\Pi(i, j, :)$
13:     dABFT fault detection and correction in $C^f_{i,j}$
14: end for
```

on the time when or place where faults can be tolerated. This includes all detection and correction steps taking place during the local matrix multiplications. Although the final detection and correction step in line 13 of Algorithm 17 has to be reliable, this additional step has very little influence on the total execution time and, like all other ABFT steps, is performed locally at each node without any additional communication. As we will see in section 10.6, this computation is very fast and therefore could be performed on reliable hardware at low cost.

10.5.3 Improvements for very low fault rates

The choice of the number of checksums d depends on the expected fault rate. For very low fault rates, 2.5D FTMM does not necessarily have to perform a detection and correction step after each local matrix multiplication. Very low fault rates would theoretically only require $d < 1$ checksum encoding vectors. Naturally, to protect the result from faults, d has to be at least 1, but the 2.5D matrix multiplication can be improved by not wasting valuable resources continuously searching for faults that are highly unlikely to occur in each consecutive local matrix multiplication. We introduce the parameter $F \geq 1$ which defines the frequency at which the ABFT steps are performed. For $F = \nu$ the ABFT steps are executed only after every ν^{th} local matrix multiplication. If F is larger than the total number of local matrix multiplications, only the last detection and correction process in line 13 would be performed. In section 10.6, we will demonstrate the performance benefits that can be achieved for different values of F. As long as the number of faults during F subsequent local matrix multiplications does not exceed d, the accuracy of the result is not influenced at all by this parameter.

The best choice for the parameters d and F strongly depends on the system, the expected

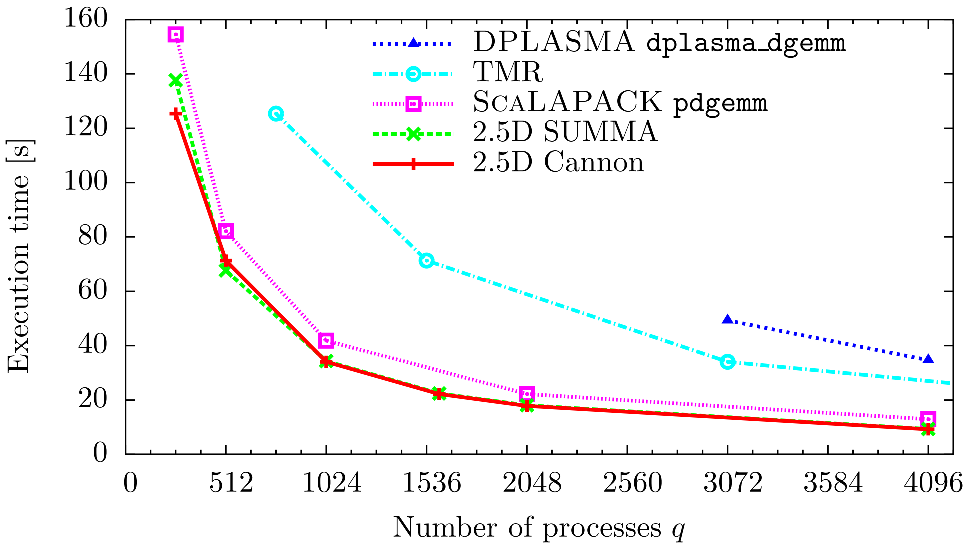


Figure 10.5: Runtime of our implementations of 2.5D SUMMA and 2.5D Cannon compared to state-of-the-art libraries and an optimal (fictitious) triple modular redundancy (TMR) for $n = 50\,000$.

fault rate and whether 2.5D FTMM is combined with other fault tolerance techniques. This could include C/R, which is necessary for fail-stop errors and can also help to recover from faults that can be detected but not corrected by ABFT. However, in terms of floating-point operations, it is less efficient to increase d and F than to reduce both parameters. The number of floating-point operations in the detection and correction step is about $dn(n+d)/F$. If both parameters d and F are increased by a factor of ν, the number of operations is increased to $\nu dn(n + \nu d)/(\nu F) = dn(n + \nu d)/F$.

10.6 Experimental Evaluation of 2.5D FTMM

In this section, we present performance results for our 2.5D FTMM algorithm for matrices up to $n = 64\,000$ on 4096 cores. All experiments were run on the Vienna Scientific Cluster VSC-2[6] consisting of 1314 nodes. Every node has two AMD Opteron 6132HE processors with eight cores each and 32 GB of main memory. The nodes are connected via QDR InfiniBand using a fat tree topology.

We compare the performance of 2.5D FTMM to state-of-the-art parallel matrix multiplication routines from DPLASMA [BBD*10] (version 2.0.0) and ScaLAPACK (version 2.0.0) and show the benefits of reducing the overhead of ABFT in the context of 2.5D MM. For all methods, MVAPICH2 (version 1.9) was used as the MPI library and OpenBLAS (version 0.2.14) as the optimised BLAS library.

In Figure 10.5, we compare our implementations of the *non-fault tolerant* 2.5D algorithms, 2.5D SUMMA and 2.5D Cannon, with the parallel matrix multiplication routines available

[6]http://vsc.ac.at/systems/vsc-2/

Table 10.1: All possible values of c for the given number of processes q and the values of c which achieved the best performance shown in Figure 10.5 for both 2.5D algorithms and $n = 50\,000$. Additionally, the corresponding block sizes b are also reported.

Number of processes q	**Possible values for** c	**Best performance 2.5D Canon**	**Best performance 2.5D SUMMA**
256	$\{1,4\}$	$c = 1$, $b = 3125$	$c = 1$, $b = 3125$
512	$\{2,8\}$	$c = 2$, $b = 3125$	$c = 2$, $b = 3125$
1024	$\{1,4\}$	$c = 1$, $b = 1563$	$c = 4$, $b = 3125$
1600	$\{1,4\}$	$c = 1$, $b = 1250$	$c = 4$, $b = 2500$
2048	$\{2,8\}$	$c = 2$, $b = 1563$	$c = 8$, $b = 3125$
4096	$\{1,4,16\}$	$c = 1$, $b = 782$	$c = 4$, $b = 1563$

in ScaLAPACK and DPLASMA for a matrix size $n = 50\,000$. To determine the optimal factor c of the 2.5D algorithms, we ran all possible values of c for each number of processes q and only report the results with the best performance. The local block size b is determined by q and c and was between 782 and 3125 (for details on c and b see Table 10.1). For ScaLAPACK we tested a wide range of block sizes and show the results with the best performance for each number of processors in Figure 10.5. For DPLASMA, we chose the parameters to be optimal for large processor counts and therefore do not show the runtimes of DPLASMA for smaller processor counts in Figure 10.5. DPLASMA is executed using one process per node, with each node on the VSC-2 having 16 cores available. DPLASMA provides many different parameters to tune its routines for high performance, including various block sizes (tile, supertile and inner blocking), parameters for defining the process grid and the type and size of the high and low-level reduction trees. All variants of the reduction trees were tested using various tree sizes, but the best results were achieved using the default greedy tree settings. We also tested various block sizes and setting the tile size to 400 returned the best results. For all other block sizes, the default settings achieved the best performance on the VSC-2.

Our implementations of the 2.5D algorithms outperform both libraries, DPLASMA and ScaLAPACK. This demonstrates the high efficiency of our implementations of the communication-avoiding matrix multiplications. Furthermore, a lower bound on the runtime of triple modular redundancy (TMR) is also included in Figure 10.5. We use a fictitious TMR by assuming optimal scalability and absolutely no overhead. We set the runtime of this fictitious TMR with q processes equal to the runtime of the best 2.5D MM with $q/3$ processes. This guarantees a more than fair comparison with our approach by avoiding inefficient implementations of TMR or simply multiplying the runtime by three.

As discussed in subsection 10.5.2, integrating dABFT into 2.5D MM incurs a small overhead due to the local operations becoming more expensive and slightly larger messages having to be sent. However, the number of messages stays the same. In Figure 10.6a, both 2.5D

FTMM variants, 2.5D FTSUMMA and 2.5D FTCannon, are compared for different numbers of checksums d per local block. The overhead (in %) is shown relative to non-fault tolerant 2.5D Cannon since this was identified as the non-fault tolerant reference implementation with the highest performance in Figure 10.5. For all methods, the matrix dimension n increases with the number of processes q, from $n = 8\,000$ for $q = 256$ to $n = 64\,000$ for $q = 4096$. The local block size b for the best performance of the 2.5D algorithms varies with q due to the parameter c. In Figure 10.6, b was between 1000 and 4000. In our experiments, 2.5D SUMMA is on average 4% slower than 2.5D Cannon. For $d = 1$, the overhead for protecting 2.5D Cannon against a single bit-flip per local block is about 3%. The overhead naturally increases with d and reaches about 7% for 2.5D FTCannon with $d = 10$ and is slightly higher for 2.5D FTSUMMA.

As mentioned in subsection 10.5.2, we can significantly reduce the overhead caused by ABFT in the context of the 2.5D algorithms by reducing the number of detection and correction steps during the computation. The resulting improvements are shown in Figure 10.6b for different fault detection and correction frequencies F (see Algorithm 17), again relative to 2.5D Cannon. The best results are achieved for $F = \infty$, where each local matrix multiplication does not check for faults and the detection and correction only takes place in the last step of 2.5D FTMM after the reduction operation. In this case, the overhead has been reduced to less than 1% for 2.5D FTCannon. As already discussed in subsection 10.5.2, $F = \infty$ can only be used if the fault rate is very low. For $F = 7$, which checks for faults after every 7th local matrix multiplication, the overhead still remains very low while providing resilience against higher fault rates. Even though Figure 10.6 shows 2.5D FTSUMMA having a higher overhead due to also including the overhead compared to 2.5D Cannon, similar effects can be observed.

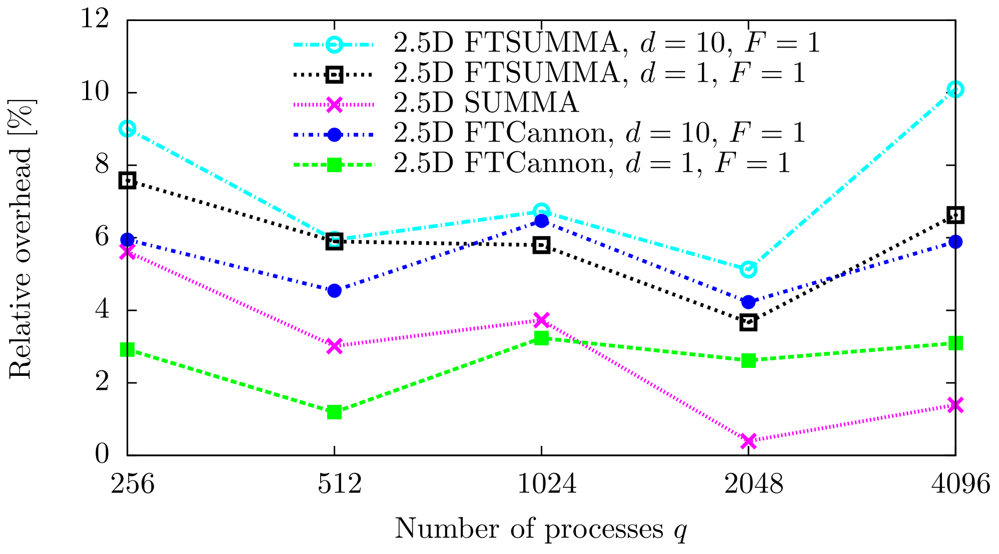
The last detection and correction step of 2.5D FTMM has to be performed fault-free to guarantee a correct result, as mentioned in subsection 10.5.2. For this step, reliable hardware or other strategies like TMR could be used. However, in our experiments this single step costs on average only 0.5% of the entire execution time of the matrix multiplication and is only performed locally. Therefore, employing more expensive strategies for this step would not influence the total runtime significantly.

10.7 Conclusion

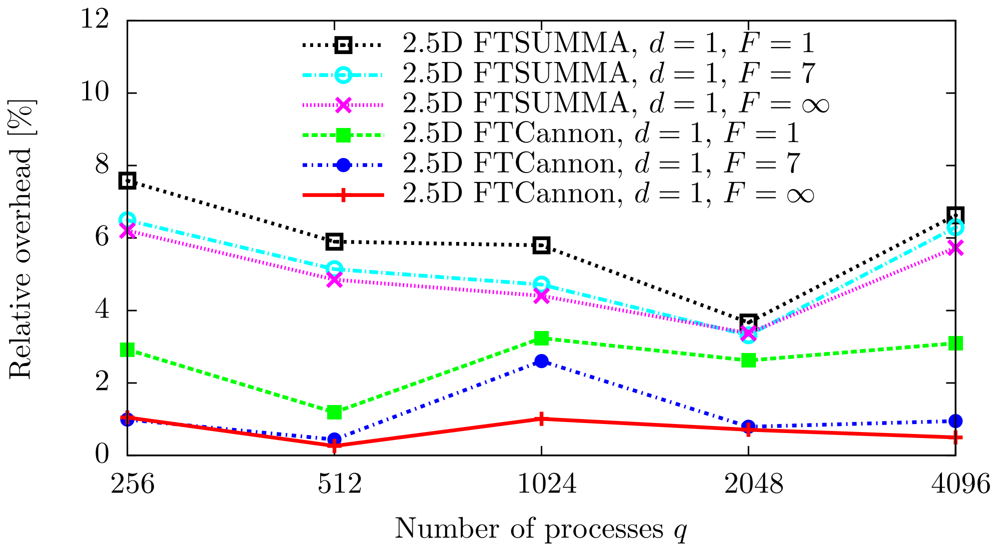
We illustrated that classical ABFT as described in the literature is not able to correct all possible bit-flips and therefore cannot guarantee a correct result. Bit-flips can only be corrected if they are restricted to the mantissa and some of the least significant bits of the exponent, an unrealistic limitation in real-world applications. We proposed a novel improved ABFT method for matrix multiplication called dABFT, which removes this constraint and can handle bit-flips at any position, a necessity for fault tolerant matrix multiplications on real-world systems. We also derived novel fault detection conditions which, for the first time in the literature, are suitable for multiple checksum encoding vectors. Aside from the increased resilience, the performance of ABFT has also been improved due to the efficient handling of the special floating-point values NaN and infinity. We experimentally confirmed that our new method handles completely random bit-flips using our non-deterministic fault injector FaITh and conducting a large number of experiments, covering a wide range of fault rates, where an accurate result was always returned. Furthermore, we showed that the relative overhead of classical ABFT and dABFT per correctable bit-flip decreases with the number of checksums d, which makes these approaches very efficient for increasing matrix sizes and on systems with high fault rates.

Based on dABFT, we investigated how to integrate ABFT concepts into state-of-the-art high-performance matrix multiplication algorithms. We introduced the fault tolerant matrix multiplication 2.5D FTMM, which combines the fault tolerance properties of dABFT with the high performance of 2.5D algorithms. In 2.5D FTMM, faults are allowed to occur throughout the entire algorithm, only requiring the last step – the final detection and correction step after the reduction operation – to be computed fault-free to ensure a correct result, drastically reducing the temporal limitations imposed by classical ABFT. This means that the vast majority of the computation – every local matrix multiplication and all detection and correction steps up until the reduction operation – can be computed on unreliable hardware. Only a very small part of the algorithm, which in our experimental evaluation only requires on average 0.5% of the total execution time, has to be made fault tolerant by other approaches. However, the cost of the fault protection of this part hardly influences the performance of the overall fault tolerant algorithm. Therefore, even methods which tend to be rather expensive in general, e. g. triple modular redundancy, can be used for this part.

Additionally, for very low fault rates we were able to reduce the overhead of ABFT in the 2.5D algorithms by reducing the frequency of the detection and correction steps. Without any loss of accuracy, this reduced the overhead to less than 1% compared to a non-fault tolerant computation, a very small price to pay for making high-performance matrix multiplication resilient against silent bit-flips.



(a) Varying number of checksums d



(b) Varying frequencies F of fault detection and correction

Figure 10.6: Overhead relative to non-fault tolerant 2.5D Cannon for $n = 8\,000$ ($q = 256$) to $n = 64\,000$ ($q = 4096$).

Chapter 11

Conclusion

Throughout this thesis, we strived to protect algorithms from silent data corruption in different environments while still achieving high performance and high accuracy. Our goals were achieved with the fault tolerant truly distributed linear least squares solver for wireless sensor networks and a parallel matrix multiplication protected from bit-flips by improving algorithm-based fault tolerance to cover any bit in a floating-point representation.

We started out by investigating iterative refinement, a method to improve round-off errors and increase accuracy. By returning to the origin of iterative refinement, we first developed a more efficient iterative refinement algorithm for eigenvalue problems. It requires less floating-point operations than the existing method by Dongarra, Moler and Wilkinson [Don82] and is able to refine a single eigenpair independently from the other eigenpairs.

Significant performance gains can be achieved by generalising mixed precision IR [BDK*08] to *arbitrary precision* arithmetic. Using arbitrary precision, we are no longer restricted to the standard IEEE 754 FP formats. We analysed the possibilities and advantages of using significantly lower working precision than the double precision target precision and presented a performance model for implementations on FPGAs, where the arbitrary precisions can be implemented efficiently due to the reconfigurable hardware. We showed that APIR can outperform direct LU solvers and other IR algorithms (SIR, MPIR, EPIR), while still achieving a very similar accuracy compared to the other algorithms, even if APIR uses working precisions far below single precision. The projected speedups are expected to be more than 20 for a linear system with $n = 10000$ compared to a direct LU solver.

In a survey of existing distributed LLS solvers, we analysed the communication patterns and identified the *truly distributed* solvers in the literature, which do not require a fusion centre or clustering to solve the LLS problem. Instead, the nodes only require communication with their neighbours within a single hop to achieve their goal. Our target platform for the development of our truly distributed LLS solver is a wireless sensor network, a large number of inexpensive sensor nodes which cooperate with each other to achieve a common goal. Aside from other advantages like avoiding congestion and not requiring routing tables in a non-static network, limiting the communication range to the local neighbourhood also conserves energy.

The power supply of the nodes in a WSN is normally severely limited and it is therefore preferable to reduce the power requirements which increases the lifetime of the nodes and in turn of the entire network.

The development of LLS solvers that could be used in such a distributed environment required careful design choices which considered the restrictive properties of WSNs. We based our LLS solvers on the method of semi-normal equations (SNE) and normal equations. The SNE method is not backward stable, but adding a step of iterative refinement stabilises the algorithm. All communication operations between participating nodes are contained in gossip-based reduction operations using the fault-tolerant gossip algorithm push-flow to handle message loss at a very low communication overhead. The knowledge gained through the use of lower working precision in IR not only enabled us to improve the performance of our LLS solver by reducing the communication cost, but also introduced a secondary fault tolerance technique through the natural ability of IR to recover from faults by iteratively improving the result and converging to the correct result, even if the initial values were completely corrupted (e. g. by bit-flips in the exponent).

Restricting the communication to the immediate neighbourhood is not limited to WSNs, but can also become beneficial in other types of future high-performance computing systems as pointed out in [CGG*14, Don12, VFR10]. On future extreme-scale supercomputers reliable communication connections and global control mechanisms can become infeasible or impossible. We demonstrated that our LLS solvers based on all-to-all reduction operations outperformed parallel dense LLS solvers in the state-of-the-art libraries DPLASMA and ScaLAPACK on high-performance supercomputers for several test cases. Despite a higher message count compared to communication-optimal methods, numerical experiments on several thousand cores of a high-performance cluster showed competitive runtime performance. In particular, for tall and skinny matrices ARPLS-IR scales very well with increasing processor count and achieves speed-ups up to 3820 on 2048 cores over the solvers from DPLASMA (despite intensive parameter optimisation of the library) and up to 26 on 2048 cores over ScaLAPACK for very tall and very skinny matrices. We also analysed the numerical accuracy for well- and ill-conditioned problems for several variations of our LLS algorithms and demonstrated the ability of IR to improve the accuracy by about two orders of magnitudes for ill-conditioned systems.

The methods to incorporate fault tolerance in LLS were applied at the algorithmic level. During our investigation of possible fault tolerance approaches against silent data corruption for our LLS solvers, we examined *algorithm-based fault tolerance* (ABFT) for matrix multiplication, which adds a small amount of redundant data, the checksums, to the input data to detect and correct erroneous results. While studying this method, we discovered shortcomings which resulted in artificial restriction to specific bits within a floating-point representation, where a fault was allowed to occur. Bit-flips in most exponent bits could not be handled, an unrealistic scenario which prevented its use in real-world applications.

We developed *direct ABFT* (dABFT) to overcome the effects of large changes in an exponent which would dominate the checksum of a faulty row or column. The standard correction process would be prevented from sufficiently improving the result due to such faults and could at most reduce the error by the maximum accuracy of the floating-point representation. Our novel idea is to compute the correction matrix without using the corrupted values and therefore avoiding the limits presented by the floating-point arithmetic. For the first time in the literature, we also derived fault detection conditions for multiple checksum encoding vectors.

In order to test and verify our improvements to ABFT, we required a method to simulate bit-flips and therefore developed FaITh, a thread-based fault injector. FaITh has a very low overhead to limit the effect on the performance of critical algorithms such as the matrix multiplication. It requires only minimal, high-level code modifications and can be used with existing high-performance libraries without the need for recompilation. Our fault injector offers fine-grained control over the occurrence of bit-flips in time and space to evaluate the problems of ABFT methods and verify the correctness of our improved method.

Finally, we introduced the fault tolerant matrix multiplication 2.5D FTMM, which combines the fault tolerance properties of dABFT with the high performance of 2.5D algorithms. In 2.5D FTMM, faults are allowed to occur throughout the entire algorithm, only requiring the last step to be computed fault-free to ensure a correct result. This restricts the temporal limitations imposed by ABFT to the final detection and correction step after the reduction operation. The vast majority of the computation can be computed on unreliable hardware and thus even rather expensive methods such as triple modular redundancy can be used to protect the final step from faults, which on average only makes up 0.5% of the total execution time. Additionally, for very low fault rates we were able to reduce the overhead of ABFT methods in the 2.5D algorithms by reducing the frequency of the detection and correction steps without any loss of accuracy. The overhead was reduced to less than 1% compared to a non-fault tolerant computation demonstrating the viability of efficiently protecting a high-performance matrix multiplication from silent bit-flips.

Bibliography

[ABB*99] Anderson E., Bai Z., Bischof C., Blackford L. S., Demmel J., Dongarra J. J., Du Croz J., Hammarling S., Greenbaum A., McKenney A., Sorensen D.: *LAPACK Users' Guide*, 3rd ed. SIAM, 1999.

[AC03] Arlat J., Crouzet Y.: Comparison of Physical and Software-implemented Fault Injection Techniques. *IEEE Computers 52*, 9 (2003), 1115–1133. doi:10.1109/TC.2003.1228509.

[ACD*10] Agullo E., Coti C., Dongarra J., Herault T., Langem J.: QR Factorization of Tall and Skinny Matrices in a Grid Computing Environment. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), IPDPS, pp. 1–11. doi:10.1109/IPDPS.2010.5470475.

[ADD*09] Agullo E., Demmel J., Dongarra J., Hadri B., Kurzak J., Langou J., Ltaief H., Luszczek P., Tomov S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *Journal of Physics: Conference Series 180*, 1 (2009), 012037. doi:10.1088/1742-6596/180/1/012037.

[AFB*06] Angskun T., Fagg G. E., Bosilca G., Pjesivac-Grbovic J., Dongarra J. J.: Scalable Fault Tolerant Protocol for Parallel Runtime Environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2006), vol. 4192 of *EuroPVM/MPI, Lecture Notes in Computer Science*, Springer. doi:10.1007/11846802_25.

[AFR*14] Aupy G., Faverge M., Robert Y., Kurzak J., Luszczek P., Dongarra J.: Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC. In *Euro-Par 2013: Parallel Processing Workshops*, vol. 8374 of *Lecture Notes in Computer Science*. Springer, 2014, pp. 657–667. doi:10.1007/978-3-642-54420-0_64.

[AK06] Alkurdi A., Kincaid D.: LU-decomposition with Iterative Refinement for Solving Sparse Linear Systems. *Journal of Computational and Applied Mathematics 185*, 2 (2006), 391–403. doi:10.1016/j.cam.2005.03.018.

[Alt10] Altera: Achieving One TeraFLOPS with 28-nm FPGAs, 2010. WP-01142-1.0. URL: http://www.altera.com/literature/wp/wp-01142-teraflops.pdf.

[Alt11] Altera: *Stratix IV Device Handbook: DSP Blocks in Stratix IV Devices*, vol. 1. Altera, 2011. sec. 1, ch. 4. URL: https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-iv/stx4_siv51004.pdf.

[Alt13] Altera: *Radar Processing: FPGAs or GPUs*. Tech. rep., Altera Corporation, 2013. WP-01197-2.0.

[ARH10] Anzt H., Rocker B., Heuveline V.: Energy Efficiency of Mixed Precision Iterative Refinement Methods using Hybrid Hardware Platforms. *Computer Science - Research and Development 25*, 3-4 (2010), 141–148. doi:10.1007/s00450-010-0124-2.

[ASY*10] Aqeel-ur-Rehman, Shaikh Z. A., Yousuf H., Nawaz F., Kirmani M., Kiran S.: Crop Irrigation Control using Wireless Sensor and Actuator Network (WSAN). In *International Conference on Information and Emerging Technologies* (2010), ICIET, pp. 1–5. doi:10.1109/ICIET.2010.5625669.

[BBD*09] Baboulin M., Buttari A., Dongarra J., Kurzak J., Langou J., Langou J., Luszczek P., Tomov S.: Accelerating Scientific Computations with Mixed Precision Algorithms. *Computer Physics Communications 180*, 12 (2009), 2526–2533. doi:10.1016/j.cpc.2008.11.005.

[BBD*10] Bosilca G., Bouteiller A., Danalis A., Faverge M., Haidar H., Herault T., Kurzak J., Langou J., Lemariner P., Ltaief H., Luszczek P., YarKhan A., Dongarra J.: *Distributed Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA*. Tech. rep., University of Tennessee Computer Science, UT-CS-10-660, 2010.

[BBD*11] 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 *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011), IPDPSW, pp. 1432–1441. doi:10.1109/IPDPS.2011.299.

[BBH*13] Bland W., Bouteiller A., Herault T., Bosilca G., Dongarra J.: Post-failure Recovery of MPI Communication Capability: Design and Rationale. *International Journal of High Performance Computing Applications 27*, 3 (2013), 244–254. doi:10.1177/1094342013488238.

[BBH*14] Bosilca G., Bouteiller A., Herault T., Robert Y., Dongarra J.: Assessing the Impact of ABFT and Checkpoint Composite Strategies. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops* (2014), IPDPSW '14, IEEE, pp. 679–688. doi:10.1109/IPDPSW.2014.79.

[BCC*97] Blackford L. S., Choi J., Cleary A., D'Azevedo E., Demmel J., Dhillon I., Dongarra J., Hammarling S., Henry G., Petitet A., Stanley K., Walker D., Whaley R. C.: *ScaLAPACK Users' Guide*. SIAM, 1997.

[BDDL09] Bosilca G., Delmas R., Dongarra J., Langou J.: Algorithm-based Fault Tolerance Applied to High Performance Computing. *Journal of Parallel and Distributed Computing 69*, 4 (2009), 410–416. doi:10.1016/j.jpdc.2008.12.002.

[BDK*08] Buttari A., Dongarra J., Kurzak J., Luszczek P., Tomov S.: Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. *ACM Transactions on Mathematical Software (TOMS) 34*, 4 (2008), 17:1–17:22. doi:10.1145/1377596.1377597.

[BDL*07] Buttari A., Dongarra J., Langou J., Langou J., Luszczek P., Kurzak J.: Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *The International Journal of High Performance Computing Applications 21*, 4 (2007), 457–466. doi:10.1177/1094342007084026.

[BDS03] Barth J. L., Dyer C. S., Stassinopoulos E. G.: Space, Atmospheric, and Terrestrial Radiation Environments. *IEEE Transactions on Nuclear Science 50*, 3 (2003), 466–482. doi:10.1109/TNS.2003.813131.

[BG65] Businger P., Golub G. H.: Linear Least Squares Solutions by Householder Transformations. *Numerische Mathematik 7*, 3 (1965), 269–276. doi:10.1007/BF01436084.

[BGL05] Benzi M., Golub G. H., Liesen J.: Numerical Solution of Saddle Point Problems. *Acta Numerica 14* (2005), 1–137. doi:10.1017/S0962492904000212.

[Bjo67] Bjorck A.: Iterative Refinement of Linear Least Squares Solutions I. *BIT Numerical Mathematics 7*, 4 (1967), 257–278. doi:10.1007/BF01939321.

[Bjo87] Bjorck A.: Stability Analysis of the Method of Semi-normal Equations for Linear Least Squares Problems. *Linear Algebra and its Applications 88–89* (1987), 31–48. doi:10.1016/0024-3795(87)90101-7.

[Bjo96] Bjorck A.: *Numerical Methods for Least Squares Problems*. SIAM, 1996. doi:10.1137/1.9781611971484.

[BLKD08] Buttari A., Langou J., Kurzak J., Dongarra J.: Parallel Tiled QR Factorization for Multicore Architectures. *Concurrency and Computation: Practice and Experience 20*, 13 (2008), 1573–1590. doi:10.1007/978-3-540-68111-3_67.

[BSLT09] Behnke R., Salzmann J., Lieckfeldt D., Timmermann D.: sDLS - Distributed Least Squares Localization for Large Wireless Sensor Networks. In *International Conference on Ultra Modern Telecommunications & Workshops* (2009), pp. 1–6. doi:10.1109/ICUMT.2009.5345330.

[BY15] Barak J., Yitzhak N. M.: SEU Rate in Avionics: From Sea Level to High Altitudes. *IEEE Transactions on Nuclear Science 62*, 6 (2015), 3369–3380. doi:10.1109/TNS.2015.2495324.

[Can69] Cannon L. E.: *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[CGG*14] Cappello F., Geist A., Gropp W. D., Kale S., Kramer B., Snir M.: Toward Exascale Resilience: 2014 Update. *Supercomputing Frontiers and Innovations 1*, 1 (2014), 1–28. doi:10.14529/jsfi140101.

[CMC*13] Cho H., Mirkhani S., Cher C.-Y., Abraham J. a., Mitra S.: Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Proceedings of the 50th Annual Design Automation Conference* (2013), DAC '13, pp. 101:1–101:10. doi:10.1145/2463209.2488859.

[CMM97] Cai Z., Manteuffel T. A., McCormick S. F.: First-Order System Least Squares for Second-Order Partial Differential Equations: Part II. *SIAM Journal on Numerical Analysis 34*, 2 (1997), 425–454. doi:10.1137/S0036142994266066.

[CMS98] Carreira J. a., Madeira H., Silva J. a. G.: Xception: Software Fault Injection and Monitoring in Processor Functional Units. *IEEE Transactions on Software Engineering 24*, 2 (1998), 125–136. doi:10.1109/32.666826.

[COS14] Calhoun J., Olson L., Snir M.: FlipIt: An LLVM Based Fault Injector for HPC. In *Euro-Par 2014: Parallel Processing Workshops*, vol. 8805 of *Lecture Notes in Computer Science*. Springer, 2014, pp. 547–558. doi:10.1007/978-3-319-14325-5_47.

[CRVZ15] Casanova H., Robert Y., Vivien F., Zaidouni D.: On the Impact of Process Replication on Executions of Large-scale Parallel Applications with

Coordinated Checkpointing. *Future Generation Computer Systems 51* (2015), 7–19. doi:10.1016/j.future.2015.04.003.

[CS10] Cattivelli F. S., Sayed A. H.: Diffusion LMS Strategies for Distributed Estimation. *IEEE Transactions on Signal Processing 58*, 3 (2010), 1035–1048. doi:10.1109/TSP.2009.2033729.

[DA96] Dutt S., Assaad F. T.: Mantissa-preserving Operations and Robust Algorithm Based Fault Tolerance for Matrix Computations. *IEEE Transactions on Computers 45*, 4 (1996), 408–424. doi:10.1109/12.494099.

[Dat10] Datta B. N.: *Numerical Linear Algebra and Applications*, 2nd ed. SIAM, 2010.

[DC13] Davies T., Chen Z.: Correcting Soft Errors Online in LU Factorization. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing* (2013), HPDC '13, ACM, pp. 167–178. doi:10.1145/2462902.2462920.

[DDM00] Demmel J., Diament B., Malajovich G.: On the Complexity of Computing Error Bounds. *Foundations of Computational Mathematics 1*, 1 (2000), 101–125. doi:10.1007/s10208001004.

[DEF*13] Demmel J., Eliahu D., Fox A., Kamil S., Lipshitz B., Schwartz O., Spillinger O.: Communication-optimal Parallel Recursive Rectangular Matrix Multiplication. In *IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), IPDPS, pp. 261–272. doi:10.1109/IPDPS.2013.80.

[DGHL08] Demmel J., Grigori L., Hoemmen M. F., Langou J.: *Communication-optimal Parallel and Sequential QR and LU Factorizations*. Tech. Rep. UCB/EECS-2008-89, EECS Department, University of California, Berkeley, 2008.

[DGHL12] Demmel J., Grigori L., Hoemmen M., Langou J.: Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing 34*, 1 (2012), 206–239. doi:10.1137/080731992.

[DHK*06] Demmel J., Hida Y., Kahan W., Li X. S., Mukherjee S., Riedy E. J.: Error Bounds from Extra-Precise Iterative Refinement. *ACM Transactions on Mathematical Software (TOMS) 32*, 2 (2006), 325–351. doi:10.1145/1141885.1141894.

[DHRL09] Demmel J., Hida Y., Riedy E. J., Li X. S.: Extra-Precise Iterative Refinement for Overdetermined Least Squares Problems. *ACM*

Transactions on Mathematical Software (TOMS) 35, 4 (2009), 28:1–28:32. doi:10.1145/1462173.1462177.

[DHS*07] Dinh T. L., Hu W., Sikka P., Corke P., Overs L., Brosnan S.: Design and Deployment of a Remote Robust Sensor Network: Experiences from an Outdoor Water Quality Monitoring Network. In *32nd IEEE Conference on Local Computer Networks* (2007), LCN, pp. 799–806. doi:10.1109/LCN.2007.39.

[DHT01] Davies P. I., Higham N. J., Tisseur F.: Analysis of the Cholesky Method with Iterative Refinement for Solving the Symmetric Definite Generalized Eigenproblem. *SIAM Journal on Matrix Analysis and Applications 23* (2001), 472–493. doi:10.1137/S0895479800373498.

[DLD11] Du P., Luszczek P., Dongarra J.: High Performance Dense Linear System Solver with Soft Error Resilience. In *Proceedings of the IEEE International Conference on Cluster Computing* (2011), CLUSTER, pp. 272–280. doi:10.1109/CLUSTER.2011.38.

[DMW83] Dongarra J. J., Moler C. B., Wilkinson J. H.: Improving the Accuracy of Computed Eigenvalues and Eigenvectors. *SIAM Journal on Numerical Analysis 20*, 1 (1983), 23–45. doi:10.1137/0720002.

[Don82] Dongarra J. J.: Algorithm 589: SICEDR: A FORTRAN Subroutine for Improving the Accuracy of Computed Matrix Eigenvalues. *ACM Transactions on Mathematical Software (TOMS) 8*, 4 (1982), 371–375. doi:10.1145/356012.356016.

[Don12] Dongarra J.: Algorithmic and Software Challenges when Moving Towards Exascale. In *Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?* (2012), ATIP '12, A*STAR Computational Resource Centre, pp. 17:1–17:35. URL: http://dl.acm.org/citation.cfm?id=2346696.2346717.

[EG00] Elmroth E., Gustavson F.: Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. *IBM Journal of Research and Development 44* (2000), 605–624. doi:10.1147/rd.444.0605.

[EHM16] Elliott J., Hoemmen M., Mueller F.: Exploiting Data Representation for Fault Tolerance. *Journal of Computational Science 14* (2016), 51–60. doi:10.1016/j.jocs.2015.12.002.

[EKF*12] Elliott J., Kharbas K., Fiala D., Mueller F., Ferreira K., Engelmann C.: Combining Partial Redundancy and Checkpointing for HPC.

In *IEEE 32nd International Conference on Distributed Computing Systems* (2012), ICDCS, pp. 615–626. doi:10.1109/ICDCS.2012.56.

[FHL*07] Fousse L., Hanrot G., Lefèvre V., Pélissier P., Zimmermann P.: MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Transactions on Mathematical Software (TOMS) 33*, 2 (2007). doi:10.1145/1236463.1236468.

[FL16] Fang X., Leeser M.: Open-Source Variable-Precision Floating-Point Library for Major Commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 9*, 3 (2016), 20:1–20:17. doi:10.1145/2851507.

[FME*12] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 78:1–78:12. doi:10.1109/SC.2012.49.

[FSL*11] Ferreira K., Stearley J., Laros III J. H., Oldfield R., Pedretti K., Brightwell R., Riesen R., Bridges P. G., Arnold D.: Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), SC '11, pp. 44:1–44:12. doi:10.1145/2063384.2063443.

[GCP*04] Govindu G., Choi S., Prasanna V., Daga V., Gangadharpalli S., Sridhar V.: A High-Performance and Energy-Efficient Architecture for Floating-Point based LU Decomposition on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (2004), IPDPS, pp. 149–156. doi:10.1109/IPDPS.2004.1303134.

[GGDS*12] Georganas E., González-Domínguez J., Solomonik E., Zheng Y., Touriño J., Yelick K.: Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 1–11. doi:10.1109/SC.2012.32.

[GNSSG13] Gansterer W. N., Niederbrucker G., Straková H., Schulze-Grotthoff S.: Scalable and Fault Tolerant Orthogonalization Based on Randomized Distributed Data Aggregation. *Journal of Computational Science 4*, 6 (2013), 480–488. doi:10.1016/j.jocs.2013.01.006.

[Gol65] Golub G.: Numerical Methods for Solving Linear Least Squares Problems. *Numerische Mathematik 7*, 3 (1965), 206–216. doi:10.1007/BF01436075.

[GST07] Göddeke D., Strzodka R., Turek S.: Performance and Accuracy of Hardware-Oriented Native-, Emulated- and Mixed-Precision Solvers in FEM Simulations. *International Journal of Parallel, Emergent and Distributed Systems 22*, 4 (2007), 221–256. doi:10.1080/17445760601122076.

[GSU03] Gansterer W. N., Schneid J., Ueberhuber C. W.: *Mathematical Properties of Equilibrium Systems*. Technical report, Department of Distributed and Multimedia Systems, University of Vienna, 2003.

[Gul94] Gulliksson M.: Iterative Refinement for Constrained and Weighted Linear Least Squares. *BIT Numerical Mathematics 34*, 2 (1994), 239–253. doi:10.1007/BF01955871.

[GVL13] Golub G. H., Van Loan C. F.: *Matrix Computations*, 4th ed. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[GW66] Golub G., Wilkinson J.: Note on the Iterative Refinement of Least Squares Solution. *Numerische Mathematik 9* (1966), 139–148. doi:10.1007/BF02166032.

[HA84] Huang K., Abraham J.: Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers C-33*, 6 (1984), 518–528. doi:10.1109/TC.1984.1676475.

[HAZ*18] Haidar A., Abdelfattah A., Zounon M., Wu P., Pranesh S., Tomov S., Dongarra J.: The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques. In *Computational Science – ICCS 2018* (2018), pp. 586–600. doi:10.1007/978-3-319-93698-7_45.

[HGB01] Hong X., Gerla M., Bagrodia R.: The Mars Sensor Network: Efficient, Energy Aware Communications. In *Communications for Network-Centric Operations: Creating the Information Force. IEEE Military Communications Conference* (2001), MILCOM, pp. 418–422. doi:10.1109/MILCOM.2001.985830.

[Hig91] Higham N. J.: Iterative Refinement Enhances the Stability of QR Factorization Methods for Solving Linear Equations. *BIT Numerical Mathematics 31*, 3 (1991), 447–468. doi:10.1007/BF01933262.

[Hig97] Higham N. J.: Iterative Refinement for Linear Systems and LAPACK. *IMA Journal of Numerical Analysis 17*, 4 (1997), 495–509. doi:10.1093/imanum/17.4.495.

[Hig02] Higham N. J.: *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.

[HLAD10] Hadri B., Ltaief H., Agullo E., Dongarra J.: Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. In *IEEE International Symposium on Parallel Distributed Processing* (2010), IPDPS, pp. 1–10. doi:10.1109/IPDPS.2010.5470443.

[HP10] Haque I. S., Pande V. S.: Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), CCGrid '10, IEEE, pp. 691–696. doi:10.1109/CCGRID.2010.84.

[HPZ19] Higham N. J., Pranesh S., Zounon M.: Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems. *SIAM Journal on Scientific Computing 41*, 4 (2019), A2536–A2551. doi:10.1137/18M1229511.

[HS10] Hogg J. D., Scott J. A.: A Fast and Robust Mixed-Precision Solver for the Solution of Sparse Symmetric Linear Systems. *ACM Transactions on Mathematical Software (TOMS) 37*, 2 (2010), 1–24. doi:10.1145/1731022.1731027.

[HTDH18] Haidar A., Tomov S., Dongarra J., Higham N. J.: Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), SC '18, pp. 603–613. doi:10.1109/SC.2018.00050.

[HWC14] Hakkarinen D., Wu P., Chen Z.: Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition. *IEEE Transactions on Parallel and Distributed Systems 26*, 5 (2014), 1323–1335. doi:10.1109/TPDS.2014.2320502.

[HWTD17] Haidar A., Wu P., Tomov S., Dongarra J.: Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (2017), ScalA '17, pp. 10:1–10:8. doi:10.1145/3148226.3148237.

[IEE08] IEEE: *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008)*. Tech. rep., IEEE, 2008. doi:10.1109/IEEESTD.2008.4610935.

[igr16] igraph R package, 2016. retrieved February, 16th 2016. URL: http://igraph.org/r/.

[Int07] Intel: *Intel SSE4 Programming Reference*, 2007. URL: https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf.

[Int16] INTEL: *Intel Architecture Instruction Set Extensions Programming Reference*, 2016. URL: https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[Int17a] INTEL: *Intel Stratix 10 Variable Precision DSP Blocks User Guide*, 2017. UG-S10-DSP 2017.05.08. URL: https://www.intel.com/content/www/us/en/programmable/documentation/kly1436148709581.html.

[Int17b] INTEL: *Understanding Peak Floating-Point Performance Claims*. Tech. rep., Intel Corporation, 2017. WP-01222-1.1. URL: https://www.altera.com/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf.

[JA86] JOU J.-Y., ABRAHAM J. A.: Fault-tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures. *Proceedings of the IEEE 74*, 5 (1986), 732–741. doi:10.1109/PROC.1986.13535.

[JBLD13] JIA Y., BOSILCA G., LUSZCZEK P., DONGARRA J.: Parallel Reduction to Hessenberg Form with Algorithm-based Fault Tolerance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), SC '13, pp. 88:1–88:11. doi:10.1145/2503210.2503249.

[KA11] KVSSRSS S., AVADHANI P.: Public Key Cryptosystem based on Pell's Equation Using The GNU MP Library. *International Journal on Computer Science and Engineering 3*, 2 (2011), 739–743.

[KBD08] KURZAK J., BUTTARI A., DONGARRA J.: Solving Systems of Linear Equations on the CELL Processor using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems 19*, 9 (2008), 1175–1186. doi:10.1109/TPDS.2007.70813.

[KD07] KURZAK J., DONGARRA J.: Implementation of Mixed Precision in Solving Systems of Linear Equations on the Cell Processor. *Concurrency and Computation: Practice and Experience 19*, 10 (2007), 1371–1385. doi:10.1002/cpe.v19:10.

[KDG03] KEMPE D., DOBRA A., GEHRKE J.: Gossip-based Computation of Aggregate Information. *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science* (2003), 482–491. doi:10.1109/SFCS.2003.1238221.

[KFA*98] KARLSSON J., FOLKESSON P., ARLAT J., YVES CROUZET, LEBER G., REISINGER J.: Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture. *Dependable Computing and Fault Tolerant Systems 10* (1998), 267–288. doi:10.1.1.62.5641.

[KGH13] Khan M. I., Gansterer W. N., Haring G.: Static vs. Mobile Sink: The Influence of Basic Parameters on Energy Efficiency in Wireless Sensor Networks. *Computer Communications 36*, 9 (2013), 965–978. doi:10.1016/j.comcom.2012.10.010.

[Kie81] Kiełbasiński A.: Iterative Refinement for Linear Systems in Variable-precision Arithmetic. *BIT Numerical Mathematics 21*, 1 (1981), 97–103. doi:10.1007/BF01934074.

[KKA95] Kanawati G. a., Kanawati N. a., Abraham J. a.: FERRARI: A Flexible Software-based Fault and Error Injection System. *IEEE Transactions on Computers 44*, 2 (1995), 248–260. doi:10.1109/12.364536.

[Kor01] Koren I.: *Computer Arithmetic Algorithms*. A. K. Peters, Ltd., 2001.

[LBNAS10] Le Borgne Y.-A., Nowe A., Abughalieh N., Steenhaut K.: Distributed Regression for High-level Feature Extraction in Wireless Sensor Networks. In *Seventh International Conference on Networked Sensing Systems* (2010), INSS, pp. 249–252. doi:10.1109/INSS.2010.5572212.

[LLL*06] Langou J., Langou J., Luszczek P., Kurzak J., Buttari A., Dongarra J.: Exploiting the Performance of 32 Bit Floating Point Arithmetic in Obtaining 64 Bit Accuracy (Revisiting Iterative Refinement for Linear Systems). In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006), SC '06, ACM, pp. 1–17. doi:10.1145/1188455.1188573.

[LLL*13] Lipson M., Loh P.-R., Levin A., Reich D., Patterson N., Berger B.: Efficient Moment-Based Inference of Admixture Parameters and Sources of Gene Flow. *Molecular Biology and Evolution 30*, 8 (2013), 1788–1802. doi:10.1093/molbev/mst099.

[LP11] Lee J. K., Peterson G. D.: Iterative Refinement on FPGAs. In *Symposium on Application Accelerators in High-Performance Computing* (2011), SAAHPC, pp. 8–13. doi:10.1109/SAAHPC.2011.19.

[LV62] Lyons R. E., Vanderkulk W.: The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development 6*, 2 (1962), 200–209. doi:10.1147/rd.62.0200.

[LV08] Langhammer M., VanCourt T.: Accelerating Floating Point DGEMM on FPGAs. In *Proceedings of the 12th Workshop on High Performance Embedded Computing* (2008), HPEC 2008.

[MBdD*10] Muller J.-M., Brisebarre N., de Dinechin F., Jeannerod C.-P., Lefèvre V., Melquiond G., Revol N., Stehlé D., Torres S.: *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[Mes12] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[MLG11] Mücke M., Lesser B., Gansterer W. N.: Peak Performance Model for a Custom Precision Floating-Point Dot Product on FPGAs. In *Euro-Par Parallel Processing Workshops*, vol. 6586 of *Lecture Notes in Computer Science*. Springer, 2011, pp. 399–406. doi:10.1007/978-3-642-21878-1_49.

[Mol67] Moler C. B.: Iterative Refinement in Floating Point. *Journal of the ACM 14*, 2 (1967), 316–321. doi:10.1145/321386.321394.

[MRAMBJ12] Matta N., Ranhim-Amoud R., Merghem-Boulahia L., Jrad A.: A Wireless Sensor Network for Substation Monitoring and Control in the Smart Grid. In *IEEE International Conference on Green Computing and Communications* (2012), GreenCom, IEEE, pp. 203–209. doi:10.1109/GreenCom.2012.39.

[MRG*08] Ma Y., Richards M., Ghanem M., Guo Y., Hassard J.: Air Pollution Monitoring and Mining Based on Sensor Grid in London. *Sensors 8*, 6 (2008), 3601–3623. doi:10.3390/s8063601.

[MSG09] Mateos G., Schizas I. D., Giannakis G. B.: Performance Analysis of the Consensus-based Distributed LMS Algorithm. *EURASIP Journal on Advances in Signal Processing 2009*, 1 (2009), 68:6–68:6. doi:10.1155/2009/981030.

[NA88] Nair V., Abraham J.: General Linear Codes for Fault-tolerant Matrix Operations on Processor Arrays. In *Eighteenth International Symposium on Fault-Tolerant Computing* (1988), FTCS-18, pp. 180–185. doi:10.1109/FTCS.1988.5317.

[NDI18] Nakasato N., Daisaka H., Ishikawa T.: High Performance High-Precision Floating-Point Operations on FPGAs Using OpenCL. In *International Conference on Field-Programmable Technology* (2018), FPT, pp. 262–265. doi:10.1109/FPT.2018.00049.

[NO10] Nedic A., Ozdaglar A.: Cooperative Distributed Multi-agent Optimization. In *Convex Optimization in Signal Processing and Communications*. Cambridge University Press, 2010, pp. 340–386. doi:10.1017/CBO9780511804458.011.

[Nor96] Normand E.: Single Event Upset at Ground Level. *IEEE Transactions on Nuclear Science 43*, 6 (1996), 2742–2750. doi:10.1109/23.556861.

[NSG12] Niederbrucker G., Strakova H., Gansterer W. N.: Improving Fault Tolerance and Accuracy of a Distributed Reduction Algorithm. In *SC Companion: High Performance Computing, Networking, Storage and Analysis* (2012), SCC, pp. 643–651. doi:10.1109/SC.Companion.2012.89.

[NVI18] NVIDIA: NVIDIA Tesla V100 Data Center GPU, 2018. Accessed: 2019-06-07. URL: https://www.nvidia.com/en-us/data-center/tesla-v100/.

[NVI19] NVIDIA: CUDA Toolkit Documentation - Programming Guide, 2019. CUDA v10.1.168. Section 5.4.1 Arithmetic Instructions. Accessed: 2019-06-07. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#arithmetic-instructions.

[OOR09] Oishi S., Ogita T., Rump S.: Iterative Refinement for Ill-Conditioned Linear Systems. *Japan Journal of Industrial and Applied Mathematics 26* (2009), 465–476. doi:10.1007/BF03186544.

[Par00] Parhami B.: *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[Pen03] Penrose M.: *Random Geometric Graphs*. Oxford University Press, 2003.

[PG13] Prikopa K. E., Gansterer W. N.: On Mixed Precision Iterative Refinement for Eigenvalue Problems. *Procedia Computer Science 18*, 0 (2013), 2647–2650. International Conference on Computational Science (ICCS). doi:10.1016/j.procs.2013.06.002.

[Pit87] Pittel B.: On Spreading a Rumor. *SIAM Journal on Applied Mathematics 47*, 1 (1987), 213–223. doi:10.1137/0147013.

[PPPM12] Panigrahi T., Pradhan P. M., Panda G., Mulgrew B.: Block Least Mean Squares Algorithm over Distributed Wireless Sensor Network. *Journal of Computer Networks and Communications 2012* (2012), 1–13. doi:10.1155/2012/601287.

[Pri11] Prikopa K. E.: *Analysis and Evaluation of Binary Cascade Iterative Refinement and Comparison to other Iterative Refinement Algorithms for Solving Linear Systems*. Master's thesis, University of Vienna, 2011. URL: http://othes.univie.ac.at/16030/.

[PSG14] Prikopa K. E., Straková H., Gansterer W. N.: Analysis and Comparison of Truly Distributed Solvers for Linear Least Squares Problems on Wireless Sensor Networks. In *Euro-Par 2014 Parallel Processing*, vol. 8632 of *Lecture Notes in Computer Science*. Springer, 2014, pp. 403–414. doi:10.1007/978-3-319-09873-9_34.

[QHL13] Quintin J.-N., Hasanov K., Lastovetsky A.: Hierarchical Parallel Matrix Multiplication on Large-Scale Distributed Memory Platforms. In *42nd International Conference on Parallel Processing* (2013), ICPP '13, IEEE, pp. 754–762. doi:10.1109/ICPP.2013.89.

[Rö03] Röck H.: *Finding an Eigenvector and Eigenvalue, with Newtons Method for Solving Systems of Nonlinear Equations*. Tech. rep., Department of Scientific Computing, University of Salzburg, Salzburg, 2003.

[RBTB06] Reichenbach F., Born A., Timmermann D., Bill R.: A Distributed Linear Least Squares Method for Precise Localization with Low Complexity in Wireless Sensor Networks. In *Proceedings of the Second IEEE international conference on Distributed Computing in Sensor Systems* (2006), DCOSS 2006, Springer, pp. 514–528. doi:10.1007/11776178_31.

[Ren98] Renaut R. A.: A Parallel Multisplitting Solution of the Least Squares Problem. *Numerical Linear Algebra with Applications 5*, 1 (1998), 11–31. doi:10.1002/(SICI)1099-1506(199801/02)5:1<11::AID-NLA123>3.0.CO;2-F.

[Ric81] Rice J.: *Matrix computations and mathematical software*. McGraw-Hill, 1981.

[RJ92] Rexford J., Jha N. K.: Algorithm-based Fault Tolerance for Floating-point Operations in Massively Parallel Systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems* (1992), vol. 2 of *ISCAS '92*, IEEE, pp. 649–652. doi:10.1109/ISCAS.1992.230168.

[RJ94] Rexford J., Jha N. K.: Partitioned Encoding Schemes for Algorithm-based Fault Tolerance in Massively Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems 5*, 6 (1994), 649–653. doi:10.1109/71.285610.

[RMG12] Reise G., Matz G., Grochenig K.: Distributed Field Reconstruction in Wireless Sensor Networks Based on Hybrid Shift-Invariant Spaces. *IEEE Transactions on Signal Processing 60*, 10 (2012), 5426–5439. doi:10.1109/TSP.2012.2205918.

[Say14] Sayed A. H.: Diffusion Adaptation over Networks. In *Academic Press Library in Signal Processing, vol. 3*. Academic Press, Elsevier, 2014, ch. 9, pp. 323–454. doi:10.1016/B978-0-12-411597-2.00009-6.

[SC10] Shakibian H., Charkari N. M.: MMS-PSO for Distributed Regression over Sensor Networks. In *IEEE Conference on Multisensor Fusion and Integration for Intelligent Systems* (2010), MFI, pp. 68–73. doi:10.1109/MFI.2010.5604476.

[SD11] Solomonik E., Demmel J.: Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Proceedings of the 17th International Conference on Parallel Processing* (2011), Euro-Par '11, Springer, pp. 90–109. doi:10.1007/978-3-642-23397-5_10.

[SG13] Straková H., Gansterer W. N.: A Distributed Eigensolver for Loosely Coupled Networks. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (2013), PDP, pp. 51–57. doi:10.1109/PDP.2013.18.

[SGRR08] Schizas I. D., Giannakis G. B., Roumeliotis S. I., Ribeiro A.: Consensus in Ad Hoc WSNs with Noisy Links - Part II: Distributed Estimation and Smoothing of Random Signals. *IEEE Transactions on Signal Processing 56*, 4 (2008), 1650–1666. doi:10.1109/TSP.2007.908943.

[SGZ12] Straková H., Gansterer W. N., Zemen T.: Distributed QR Factorization Based on Randomized Algorithms. In *Parallel Processing and Applied Mathematics (PPAM)* (2012), vol. 7203 of *Lecture Notes in Computer Science*, Springer, pp. 235–244. doi:10.1007/978-3-642-31464-3_24.

[Sie91] Siewiorek D. P.: Architecture of Fault-Tolerant Computers: An Historical Perspective. *Proceedings of the IEEE 79*, 12 (1991), 1710–1734. doi:10.1109/5.119549.

[SJZW09] Shao Y., Jiang L., Zhao Q., Wang Y.: High Performance and Parallel Model for LU Decomposition on FPGAs. In *Fourth International Conference on Frontier of Computer Science and Technology* (2009), FCST '09, pp. 75–79. doi:10.1109/FCST.2009.66.

[SLS*12] Sivasankari H., Leelavathi R., Shaila K., Venugopal K. R., Iyengar S. S., Patnaik L. M.: Dynamic Cooperative Routing (DCR) in Wireless Sensor Networks. In *Advances in Communication, Network, and Computing* (2012), CNC 2012, Springer, pp. 87–92. doi:10.1007/978-3-642-35615-5_13.

[SMG09] Schizas I. D., Mateos G., Giannakis G. B.: Distributed LMS for Consensus-Based In-Network Adaptive Processing. *IEEE Transactions on Signal Processing 57*, 6 (2009), 2365–2382. doi:10.1109/TSP.2009.2016226.

[SNV10] Sundhar Ram S., Nedić A., Veeravalli V. V.: Distributed Stochastic Subgradient Projection Algorithms for Convex Optimization. *Journal of Optimization Theory and Applications 147*, 3 (2010), 516–545. doi:10.1007/s10957-010-9737-7.

[SPD*05] Sancho J. C., Petrini F., Davis K., Gioiosa R., Jiang S.: Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium* (2005), IPDPS '05, IEEE. CD-ROM. doi:10.1109/IPDPS.2005.157.

[SPD*09] Srikanth S. V., Pramod P. J., Dileep K. P., Tapas S., Patil M. U., Sarat C. B. N.: Design and Implementation of a Prototype Smart PARKing (SPARK) System Using Wireless Sensor Networks. In *International Conference on Advanced Information Networking and Applications Workshops* (2009), WAINA '09, pp. 401–406. doi:10.1109/WAINA.2009.53.

[SPS08] Sun J., Peterson G. D., Storaasli O. O.: High-Performance Mixed-Precision Linear Solver for FPGAs. *IEEE Transactions on Computers 57*, 12 (2008), 1614–1623. doi:10.1109/TC.2008.89.

[SPvdG12] Schatz M., Poulson J., van de Geijn R.: *Parallel Matrix Multiplication: 2D and 3D*. Tech. Rep. TR-12-13, The University of Texas at Austin, Department of Computer Sciences, 2012. FLAME Working Note #62.

[SPW09] Schroeder B., Pinheiro E., Weber W.-D.: DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (2009), SIGMETRICS '09, ACM, pp. 193–204. doi:10.1145/2492101.1555372.

[SR13] Sluciak O., Rupp M.: Network Size Estimation Using Distributed Orthogonalization. *IEEE Signal Processing Letters 20*, 4 (2013), 347–350. doi:10.1109/LSP.2013.2247756.

[SSK*13] Shi L., Song W.-Z., Kamath G., Xing G., Liu X.: Distributed Least-Squares Iterative Methods in Networks: A Survey. Submitted to Computing Journal, 2013.

[SSW10] Strenski D., Sundararajan P., Wittig R.: The Expanding Floating-Point Performance Gap Between FPGAs and Microprocessors. *HPCWire* (2010). URL: http://www.hpcwire.com/features/The-Expanding-Floating-Point-Performance-Gap-Between-FPGAs-and-Microprocessors-109982029.html.

[SSX*13] Shi L., Song W.-Z., Xu M., Xiao Q., Kamath G., Lees J. M., Xing G.: Imaging Seismic Tomography in Sensor Network. In *IEEE International Conference on Distributed Computing in Sensor Systems* (2013), DCOSS, pp. 304–306. doi:10.1109/DCOSS.2013.19.

[STIH11] Schöne R., Tschüter R., Ilsche T., Hackenberg D.: The VampirTrace Plugin Counter Interface: Introduction and Examples. In *Proceedings of the 2010 Conference on Parallel Processing* (2011), Euro-Par 2010, Springer, pp. 501–511. doi:10.1007/978-3-642-21878-1_62.

[STW00] Satoh S., Tosaka Y., Wender S. A.: Geometric Effect of Multiple-bit Soft Errors Induced by Cosmic Ray Neutrons on DRAM's. *IEEE Electron Device Letters 21*, 6 (2000), 310–312. doi:10.1109/55.843160.

[SW80] Symm H. J., Wilkinson J. H.: Realistic Error Bounds for a Simple Eigenvalue and its Associated Eigenvector. *Journal Numerische Mathematik 35*, 2 (1980), 371–375. doi:10.1007/BF01396310.

[SWA*13] Snir M., Wisniewski R. W., Abraham J. A., Adve S. V., Bagchi S., Balaji P., Belak J., Bose P., Cappello F., Carlson B., Chien A. A., Coteus P., Debardeleben N. A., Diniz P., Engelmann C., Erez M., Fazzari S., Geist A., Gupta R., Johnson F., Krishnamoorthy S., Leyffer S., Liberty D., Mitra S., Munson T. S., Schreiber R., Stearley J., Hensbergen E. V.: Addressing Failures in Exascale Computing. *International Journal of High Performance Computing* (2013). doi:10.1177/1094342014522573.

[TBA86] Tsitsiklis J., Bertsekas D., Athans M.: Distributed Asynchronous Deterministic and Stochastic Gradient Optimization Algorithms. *IEEE Transactions on Automatic Control 31*, 9 (1986), 803–812. doi:10.1109/TAC.1986.1104412.

[TGTB13] Thajchayapong S., Garcia-Trevino E. S., Barria J. A.: Distributed Classification of Traffic Anomalies Using Microscopic Traffic Variables. *IEEE Transactions on Intelligent Transportation Systems 14*, 1 (2013), 448–458. doi:10.1109/TITS.2012.2220964.

[TRG05] Thakur R., Rabenseifner R., Gropp W.: Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66. doi:10.1177/1094342005051521.

[TS12] Tu S.-Y., Sayed A. H.: Diffusion Strategies Outperform Consensus Strategies for Distributed Estimation Over Adaptive Networks. *IEEE Transactions on Signal Processing 60*, 12 (2012), 6217–6234. doi:10.1109/TSP.2012.2217338.

[TSS07] Tubaishat M., Shang Y., Shi H.: Adaptive Traffic Light Control with Wireless Sensor Networks. In *4th IEEE Consumer Commu-*

nications and Networking Conference (2007), CCNC 2007, pp. 187–191. doi:10.1109/CCNC.2007.44.

[UYA03] Ulmer C., Yalamanchili S., Alkalai L.: *Wireless Distributed Sensor Networks for In-situ Exploration of Mars*. Tech. rep., Georgia Institute of Technology and California Institute of Technology, 2003.

[Var59] Varga R.: *Factorization and Normalized Iterative Methods*. Tech. rep., Westinghouse Electric Corp. Bettis Plant, Pittsburgh, 1959.

[Vav94] Vavasis S. A.: Stable Numerical Algorithms for Equilibrium Systems. *SIAM Journal on Matrix Analysis and Applications 15*, 4 (1994), 1108–1131. doi:10.1137/S0895479892230948.

[VDGW97] Van De Geijn R. A., Watts J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience 9*, 4 (1997), 255–274. doi:10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2.

[VFR10] Varela M. R., Ferreira K. B., Riesen R.: Fault-tolerance for Exascale Systems. In *IEEE International Conference On Cluster Computing Workshops and Posters* (2010), CLUSTER WORKSHOPS, pp. 1–4. doi:10.1109/CLUSTERWKSP.2010.5613081.

[WAJR*05] Werner-Allen G., Johnson J., Ruiz M., Lees J., Welsh M.: Monitoring Volcanic Eruptions with a Wireless Sensor Network. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks* (2005), IEEE, pp. 108–120. doi:10.1109/EWSN.2005.1462003.

[WALW*06] Werner-Allen G., Lorincz K., Welsh M., Marcillo O., Johnson J., Ruiz M., Lees J.: Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing 10*, 2 (2006), 18–25. doi:10.1109/MIC.2006.26.

[WC14] Wu P., Chen Z.: FT-ScaLAPACK: Correcting Soft Errors On-line for ScaLAPACK Cholesky, QR, and LU Factorization Routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (2014), HPDC '14, ACM, pp. 49–60. doi:10.1145/2600212.2600232.

[WDC*11] 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* (2011), ScalA '11, pp. 25–28. doi:10.1145/2133173.2133185.

[WDL*09] Wu G., Dou Y., Lei Y., Zhou J., Wang M., Jiang J.: A Fine-Grained Pipelined Implementation of the LINPACK Benchmark on FPGAs. In *17th IEEE Symposium on Field Programmable Custom Computing Machines* (2009), FCCM, pp. 183–190. doi:10.1109/FCCM.2009.11.

[Wil63] Wilkinson J. H.: *Rounding Errors in Algebraic Processes*. Her Majesty's Stationery Office, London, 1963.

[Wil65] Wilkinson J. H.: *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.

[WMES10] Wang C., Mueller F., Engelmann C., Scott S. L.: Hybrid Checkpointing for MPI Jobs in HPC Environments. In *IEEE 16th International Conference on Parallel and Distributed Systems* (2010), ICPADS, pp. 524–533. doi:10.1109/ICPADS.2010.48.

[YB04] Yang L. T., Brent R. P.: Parallel MCGLS and ICGLS Methods for Least Squares Problems on Distributed Memory Architectures. *The Journal of Supercomputing 29*, 2 (2004), 145–156. doi:10.1023/B:SUPE.0000026847.75355.69.

[YZC*15] Yao E., Zhang J., Chen M., Tan G., Sun N.: Detection of Soft Errors in LU Decomposition with Partial Pivoting using Algorithm-Based Fault Tolerance. *International Journal of High Performance Computing Applications 29*, 4 (2015), 422–436. doi:10.1177/1094342015578487.

[ZAV04] Ziade H., Ayoubi R., Velazco R.: A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology 1*, 2 (2004), 171–186.

[ZCM*96] Ziegler J. F., Curtis H. W., Muhlfeld H. P., Montrose C. J., Chin B., Nicewicz M., Russell C. A., Wang W. Y., Freeman L. B., Hosier P., LaFave L. E., Walsh J. L., Orro J. M., Unger G. J., Ross J. M., O'Gorman T. J., Messina B., Sullivan T. D., Sykes A. J., Yourke H., Enger T. A., Tolat V., Scott T. S., Taber A. H., Sussman R. J., Klein W. A., Wahaus C. W.: IBM Experiments in Soft Fails in Computer Electronics (1978–1994). *IBM Journal of Research and Development 40*, 1 (1996), 3–18. doi:10.1147/rd.401.0003.

[ZKHP11] Zhou Q., Kar S., Huie L., Poor H. V.: Robust Distributed Least-Squares Estimation in Sensor Networks with Node Failures. In *IEEE Global Telecommunications Conference* (2011), GLOBECOM 2011, pp. 1–6. doi:10.1109/GLOCOM.2011.6133690.

[ZNK12] Zheng G., Ni X., Kale L. V.: A Scalable Double In-memory Checkpoint and Restart Scheme Towards Exascale. In *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops* (2012), DSN-W, pp. 1–6. doi:10.1109/DSNW.2012.6264677.

[ZP06] Zhuo L., Prasanna V.: High-Performance and Parameterized Matrix Factorization on FPGAs. In *International Conference on Field Programmable Logic and Applications* (2006), FPL '06, pp. 1–6. doi:10.1109/FPL.2006.311238.

Listings

List of Figures

List of Tables

List of Algorithms