Dissertation

submitted
to the
Combined Faculty for the Natural Sciences and Mathematics
of

Heidelberg University, Germany

for the degree of
Doctor of Natural Sciences

Put forward by

Dipl.-Math Anke Mareike Schmidtobreick

born in Karlsruhe

Oral examination:

October 27, 2017

Parallel Asynchronous Matrix Multiplication

for a

Distributed Pipelined Neural Network

Advisors:

Prof. Dr. Vincent Heuveline
JProf. Dr. Holger Fröning

Either I will find a way, or I will make one.

(Philip Sidney)

Abstract

Machine learning is an approach to devise algorithms that compute an output without a given rule set but based on a self-learning concept. This approach is of great importance for several fields of applications in science and industry where traditional programming methods are not sufficient. In neural networks, a popular subclass of machine learning algorithms, commonly previous experience is used to train the network and produce good outputs for newly introduced inputs. By increasing the size of the network more complex problems can be solved which again rely on a huge amount of training data. Increasing the complexity also leads to higher computational demand and storage requirements and to the need for parallelization.
Several parallelization approaches of neural networks have already been considered. Most approaches use special purpose hardware whilst other work focuses on using standard hardware. Often these approaches target the problem by parallelizing the training data. In this work a new parallelization method named *poadSGD* is proposed for the parallelization of fully-connected, large-scale feedforward networks on a compute cluster with standard hardware. poadSGD is based on the stochastic gradient descent algorithm. A block-wise distribution of the network's layers to groups of processes and a pipelining scheme for batches of the training samples are used. The network is updated asynchronously without interrupting ongoing computations of subsequent batches. For this task a one-sided communication scheme is used. A main algorithmic part of the batch-wise pipelined version consists of matrix multiplications which occur for a special distributed setup, where each matrix is held by a different process group.
GASPI, a parallel programming model from the field of "Partitioned Global Address Spaces" (PGAS) models is introduced and compared to other models from this class. As it mainly relies on one-sided and asynchronous communication it is a perfect candidate for the asynchronous update task in the poadSGD algorithm. Therefore, the matrix multiplication is also implemented based GASPI. In order to efficiently handle upcoming synchronizations within the process groups and achieve a good workload distribution, a two-dimensional block-cyclic data distribution is applied for the matrices. Based on this distribution, the multiplication algorithm is computed by diagonally iterating over the sub blocks of the resulting matrix and computing the sub blocks in subgroups of the processes. The sub blocks are computed by sharing the workload between the process groups and communicating mostly in pairs or in subgroups. The communication in pairs is set up to be overlapped by other ongoing computations. The implementations provide a special challenge, since the asynchronous communication routines must be handled with care as to which processor is working at what point in time with which data in order to prevent an unintentional dual use of data.
The theoretical analysis shows the matrix multiplication to be superior to a naive implementation when the dimension of the sub blocks of the matrices exceeds 382. The performance achieved in the test runs did not withstand the expectations the theoretical analysis predicted. The algorithm is executed on up to 512 cores and for matrices up to a size of $131,072 \times 131,072$.
The implementation using the GASPI API was found not be straightforward but to provide a good potential for overlapping communication with computations whenever the data dependencies of an application allow for it. The matrix multiplication was successfully implemented and can be used within an implementation of the poadSGD method that is yet to come. The poadSGD method seems to be very promising, especially as nowadays, with the larger amount of data and the increased complexity of the applications, the approaches to parallelization of neural networks are increasingly of interest.

Zusammenfassung

Ein maschinelles Lernen Modell ist ein künstliches System, das ohne einen vorgegebenen Regelsatz basierend auf vorherigen Erfahrungen eigenständig lernt, zu unbekannten Eingaben passende Lösungen zu produzieren. Dieser Ansatz ist für mehrere Einsatzgebiete von großer Bedeutung sowohl in der Wissenschaft als auch der Industrie, wenn traditionelle Programmiermethoden nicht ausreichen. In neuronalen Netzwerken, einer beliebten Unterklasse der maschinellen Lernalgorithmen, werden vorgegebene Trainingsdaten bestehend aus Ein- und Ausgabewerten verwendet, um das Netzwerk basierend auf der Differenz zwischen dem vorgegebenen Wert und dem Ausgabewert des Netzwerkes zu trainieren. Durch größere Netze können komplexere Probleme abgebildet werden, die wiederum auf das Vorhandensein einer großen Anzahl von Trainingsdaten angewiesen sind. Die Erhöhung der Komplexität führt so zu höherem Rechenbedarf und höheren Speicheranforderungen, weshalb eine Parallelisierung des Trainings sinnvoll scheint.
Es wurden bereits mehrere Parallelisierungsansätze für neuronale Netze in Betracht gezogen. Viele Ansätze verwenden Spezialhardware, während sich andere Arbeiten auf die Verwendung von Standardhardware konzentrieren.
In dieser Arbeit wird eine neue Parallelisierungsmethode namens poadSGD vorgestellt, welche sich an vollständig verbundene, sehr große *Feedforward-Netzwerke* richtet, welche auf einem Rechencluster mit Standardhardware ausgeführt werden. PoadSGD basiert auf dem stochastischen Gradientenverfahren und verwendet eine blockweise Verteilung der Netzwerkschichten auf Gruppen von Prozessen, sowie ein Pipeline-Schema, das jeweils für eine Serie von Trainingsdaten ausgeführt wird. Das Netzwerk wird asynchron aktualisiert, ohne die noch laufenden Berechnungen der nachfolgenden Serien zu unterbrechen. Hierfür wird ein einseitiges Kommunikationsschema verwendet. Ein wesentlicher Teil des Algorithmus sind Matrixmultiplikationen, wobei für einen Teil davon der Fall auftritt, dass die Matrizen auf jeweils unterschiedliche Prozessgruppen verteilt sind.
In dieser Arbeit wird daher GASPI, ein paralleles Programmiermodell aus dem Bereich der *Partitioned Global Address Spaces* (PGAS) Modelle, vorgestellt und mit anderen Modellen dieser Klasse verglichen. Da es hauptsächlich auf einseitiger und asynchroner Kommunikation basiert, ist es ein perfekter Kandidat für die Umsetzung des asynchronen Updates im poadSGD-Algorithmus. Daher ist die Matrixmultiplikation auch auf der Basis von GASPI implementiert. Um eine gute Verteilung der Arbeitslast zu erreichen und möglichst effizient mit der Synchronisation, die durch die zusammenarbeitenden Prozessgruppen entsteht, umzugehen, wird für die Matrizen eine zweidimensionale blockzyklische Datenverteilung angewendet. Basierend auf dieser Verteilung wird der Algorithmus für die Matrix-Multiplikation durch eine diagonale Iteration über die Teilblöcke der Ergebnismatrix und die Berechnung dieser Teilblöcke in Untergruppen der Prozessgruppen umgesetzt. Die Rechenlast der Berechnung der Teilblöcke wird zwischen den Gruppen aufgeteilt und die Kommunikation findet hauptsächlich paarweise oder nur in Untergruppen statt. Die paarweise Kommunikation erfolgt so, dass sie von anderen laufenden Berechnungen überlappt wird. Die Implementierung des Algorithmus stellt eine besondere Herausforderung dar, da die asynchronen Kommunikationsroutinen mit Sorgfalt gehandhabt werden müssen. Es muss zu jedem Zeitpunkt klar sein, welcher Prozessor mit welchen Daten arbeitet oder wohin versendet, um eine unbeabsichtigte doppelte Nutzung von Daten zu verhindern.
Die theoretische Analyse zeigt, dass die hier präsentierte Matrixmultiplikation einer naiven Implementierung überlegen ist, wenn die Dimension der Teilblöcke der Matrizen größer als 382 ist. Diese Erwartung hat die in den Testläufen erzielte Performance nicht erfüllt. Der Algorithmus wurde auf bis zu 512 Kerne für Matrizen bis zu einer Größe von 131.072×131.072 ausgeführt. Die Im-

plementierung mit der GASPI-API wurde für nicht einfach befunden, wobei die API allerdings ein gutes Potenzial verspricht für die Umsetzung von Anwendungen ohne große Datenabhängigkeiten. Auch wenn die Performance nicht den Erwartung entsprach, so wurde die Matrixmultiplikation erfolgreich umgesetzt und kann für eine Implementierung der poadSGD-Methode verwendet werden. Der Ansatz der poadSGD-Methode ist sehr vielversprechend, zumal heutzutage mit den immer größeren Datensätzen und der erhöhten Komplexität der Anwendungen die Ansätze zur Parallelisierung von neuronalen Netzwerken zunehmend von Interesse sind.

Mathematical Contributions

The approach of machine learning is of great importance, both in science and in industry, when traditional algorithms are no longer sufficient.
In this work a parallelization method for large-scale feedforward neural networks is presented. The method used for its training thereby relies on the stochastical gradient descent method and is combined with a block-wise distribution of the network layers to groups of processes, as well as a pipelining scheme for batches of the training samples and an asynchronous update of the network.
A particular challenge of the proposed method is a matrix multiplication for very large matrices with a special parallel distribution of the matrix elements. For this particular configuration, a parallel algorithm is developed based on asynchronous communication mechanisms, which have an impact on the algorithmic implementation. In addition, the algorithm is analyzed with regards to its communication complexity and its potential for exploitation of overlapping communication and computation enabled by the asynchronous communication. Through the development and implementation of the matrix multiplication for the parallelization of neural networks, experience in the use of asynchronous communication for basic, numerical methods is gained.

Acknowledgments

Finally, the end of my PhD time has come, it had its ups and downs. During this time I had the pleasure to get to know and appreciate three different universities: Karlsruhe Institute of Technology (KIT), the University of Oxford and finally for most of the time the oldest university of the country: Heidelberg University, Ruperto-Carola.

Therefore my thanks go out to different locations: first of all to Prof. Dr. Vincent Heuveline first at KIT and then at the Faculty of Mathematics and Computer Science at Heidelberg University for giving me this great opportunity for my PhD. Also, I am very grateful for his continuous support and guidance during my time as a PhD student. In addition, I am grateful to JProf. Dr. Holger Fröning for his quick and eager agreement to support me as my second supervisor and thereby provide me with good advice. Furthermore, my thanks go to Prof. Endre Süli who welcomed me to the Numerical Analysis Group at the University of Oxford and made my research stay possible, it was a great experience.

Also thanks for all the encouragement and motivation boosts from all kinds of places: from the place opposite or next to mine (in all offices and trains), from neighboring offices, from family and friends, former and current colleagues, via e-mail or live – biggest thanks!

I would also like to acknowledge the "Karlsruhe House of Young Scientists (KHYS)" which funded my research stay at Oxford University and the support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1134-1 FUGG in terms of the computational resource that was used in this work.

Contents

Chapter 1 – Introduction 1
1.1 Contributions . . . 3
1.2 Structure . . . 4

Chapter 2 – Neural Networks 5
2.1 Introduction to Machine Learning . . . 5
2.2 Fundamentals of Neural Networks . . . 7
2.2.1 Components and Basic Functionality . . . 8
2.2.2 Training a Neural Network . . . 11
2.2.3 Different Network Models . . . 15
2.2.4 Challenges, Optimization and Regularization . . . 18
2.3 Parallelizing Neural Networks . . . 21
2.3.1 Hardware Employed . . . 22
2.3.2 Types of Parallelization . . . 22
2.3.3 Review of Parallelization on CPUs . . . 23
2.4 New Parallel Method poadSGD . . . 25
2.4.1 Key Ideas of poadSGD . . . 26
2.4.2 The poadSGD Algorithm . . . 29
2.4.3 Distribution of Weight Matrix . . . 30
2.4.4 Outlook . . . 32

Chapter 3 – Parallel Computing Models 35
3.1 Classifying The Parallel World . . . 35
3.1.1 Classification based on the Memory Architecture . . . 36
3.1.2 Classification based on Parallel Programming Models . . . 37
3.2 The PGAS Model . . . 39
3.2.1 Coarray Fortran (CAF) . . . 40
3.2.2 Unified Parallel C (UPC) . . . 42
3.2.3 Further PGAS-based languages . . . 44

Chapter 4 – Global Address Space Programming Interface (GASPI) 47
4.1 GASPI's Origin . . . 47
4.2 GASPI Specification and Functionalities . . . 48
4.2.1 Process and Memory Setup . . . 49
4.2.2 General Concepts of GASPI Functions . . . 51
4.2.3 Synchronization . . . 52
4.2.4 One-sided Communication . . . 52
4.2.5 Collectives . . . 56
4.2.6 Further GASPI Features . . . 57
4.2.7 Overlap of Communication and Computation . . . 58
4.3 Comparing Features: GASPI vs. CAF & UPC . . . 59

4.4 Related Work with GASPI 62
4.5 Further One-sided Communication APIs 64

Chapter 5 – Dense Matrix Multiplication in GASPI 65
5.1 Matrix Representation 66
5.2 A Matrix Multiplication Algorithm with GASPI 69
5.2.1 Related work 69
5.2.2 Implementation 71
5.2.3 Theoretical Analysis 78
5.3 Performance Tests 91
5.3.1 Test Setup 91
5.3.2 Results 97

Chapter 6 – Conclusion and Outlook 103

References 117

Chapter 1

Introduction

In a field where adequate rules are too difficult to devise for a given problem, classical programming approaches are not sufficient and instead machine learning algorithms are applied. These algorithms follow the concept of learning to solve a problem based on previous experiences. They are then able to generalize and to produce solutions for inputs they have never seen before. Machine learning has become of great importance in science and industry. Amongst others, it is used to solve problems from the fields of medical applications, image and voice processing, search engines and social networks [Sch14, Nie15].
A classical machine learning algorithm is the artificial neural network. Originally inspired by the human brain it is built up of several layers of neurons connected by synapses which transfer signals throughout the network and generate a solution when given a new input. Algorithms have been developed to optimize such a network by adapting the parameters of the network during a training phase based on previous training data. Increasing the size of these networks enables more complex problems to be solved. But this also leads to issues with the convergence of the training that need to be dealt with. Equally important is that the higher complexity also increases the computational demand and the storage requirements [PLWH03]. A solution to this is to parallelize these networks. However, to efficiently accomplish this task, several factors are of great importance: which hardware is selected, which parallel programming model suits the problem best and finally how exactly the algorithm is parallelized such that it scales well for differently sized networks and generates no unnecessary overhead.
For the training phase different parallelization approaches of neural networks have already been considered [RS97]. Most approaches use special purpose hardware such as GPUs [DCM+12] or specialized neuromorphic computers [FFA92, SBG+10]. But, as this special hardware is not always available in all research labs and universities, other works focus on using standard hardware. The parallelization techniques used include the running of several instances of the network in parallel [PLWH03]. Either these instances are started with the same initial state or different ones. Furthermore, they can all be trained with the same training input or with different parts of the data. After the separate training the results are reconciled. Other techniques parallelize the training inputs for a single network, pipeline the training samples or partition the network topology itself amongst several processes [NS92]. For parallelization on CPUs, MPI is often used for the communication between the processes, for example in [DMN08]. Furthermore, the update of the networks' parameters often occurs in a synchronized way by a central instance [DCM+12], limiting the efficiency.

This thesis devises an efficiently parallelized algorithm named *poadSGD* for the training of a neural network, when using a compute cluster with standard hardware. The focus is on the question whether an improvement of the performance can be achieved by using different communication schemes as provided by the partitioned global address space (PGAS) programming model. One such model is GASPI [Con13] which relies only on one-sided and asynchronous communication patterns. This leverages the idea of devising a batch-wise variant of the stochastic gradient descent method with a pipelining scheme where GASPI allows for an asynchronous update. The poadSGD algorithm maps the network layer-wise to different groups of processes, together achieving a pipeline which processes several batches of training samples directly after another. The updates are computed by averaging the updates of a batch and applying the result directly to the network. Updates are thereby sent asynchronously without interrupting ongoing computations of subsequent batches. The main algorithmic part of the pipelined version is the matrix multiplication which occurs for a special distributed setup, namely each matrix being held by a different process group. All in all, the main research question thereby is how to efficiently implement a scalable version of this matrix multiplication where both matrices are distributed amongst different process groups and when employing the asynchronous and one-sided communication scheme of GASPI.

In order to efficiently handle upcoming synchronizations within the process groups and achieve a good workload distribution, a two-dimensional block-cyclic data distribution is applied for the matrices. Thereby, a matrix is divided into sub blocks and these are mapped to individual processes by means of a process grid containing the processes associated with the matrix. A structure is defined which simplifies the determination of the exact location of each matrix element. As a result, the computation of each sub block of the matrix resulting from the matrix multiplication only depends on the processes in a row of the process grid of the first matrix and the processes in a column of the process grid of the second matrix. Therefore, the first computational scheme is to diagonally iterate over these sub blocks and compute the sub blocks which allows for a maximum number of blocks to be computed in parallel. At the heart of the method is then the *overlap algorithm*. Here, processes first work in pairs where one process is from each of the matrix groups. The algorithm is implemented with a focus on achieving an overlap of the communication between these pairs of processes with ongoing computations. Furthermore, based on the setup of the process grid, the algorithm makes use of the process grid topology to apply reduction operations on sub-groups of the process grid.

The result is an implementation of the matrix multiplication for dense matrices relying on GASPI's features with a special focus on its asynchronous communication scheme. The approach is analyzed theoretically with regard to its capability for overlap for its two main parts: the *diagonal iteration* over the blocks of the result matrix and the *overlap algorithm* which is the main part of the computation of these blocks. The diagonal iteration proves to always be superior to a naive implementation and the overlap algorithm is shown to improve performance when the dimension of the blocks exceeds 382. In addition, the strong and weak scalability on a high performance cluster is evaluated. As a first step, for each combination of possible number of processes and size of the matrices, the block size with the best performance is chosen for further tests. For these parameter combinations the matrix multiplication is then run on up to 512 cores and for matrices up to a size of $131,072 \times 131,072$ (≈ 17.2 billion elements). The performance achieved in the test runs did not withstand the expectations the theoretical analysis had promised. The speedup and efficiency highly depend on the matrix size used. They were determined with respect

to the results achieved on the smallest process grids with 4 processes each, such that the ideal speedup is 64 for the largest process grid. The efficiency is about 0.2 or better for matrices smaller than 1024×1024 but for larger matrices it drops drastically when increasing the number of processes.
The implementation using the GASPI API were found to be tedious as GASPI does not provide a black-box with easy-to-use communication calls, but provides a clearly structured interface that leaves many details to the user such as the management of the position of global data (in terms of offsets) and handling of buffers. Also correctly implementing with GASPI proves to be a challenge as it requires a deep understanding of the underlying functionality. However, the matrix multiplication was successfully implemented for the special setup of matrices distributed across multiple process groups and based on GASPI, although the results achieved by the test runs were below expectations. As a follow-up step a performance analysis tool could be employed in order to detect which points of the current algorithm could be optimized. Of further interest would be a complete implementation of the poadSGD method based on an asynchronous framework such as GASPI or to adapt an existing software framework for neural networks for this task.

1.1 Contributions

To provide a better overview, the author's contributions are listed in this section.

- An introduction to machine learning and neural networks is given, also providing an overview of parallelization techniques.

- The author presents a new pipelined method poadSGD for the parallel implementation of large-scale fully-connected feedforward networks which is based on one-sided and asynchronous communication schemes.

- As in literature parallel models are described differently, this work presents an overview of the different categorizations. This enables the user to understand the context of PGAS models and to better understand the shift from comparing computer architectures to comparing memory models of programming languages.

- An introduction to GASPI is presented, providing an understanding of its one-sided and asynchronous communication model and how it affects the development of an algorithm. Possibilities how GASPI's communication routines are used and how they can be employed to overlap communication with computation are demonstrated by small examples.

- A description of the matrix multiplication algorithm based on a PGAS-model with one-sided and asynchronous communication, written in C and GASPI, is presented. Divided into two main sub algorithms, an analysis of the combined arithmetical and communication complexity is given. Thereby the focus is on determining if or when overlap occurs and how it may increase the overall performance of the application.

- Moreover helpful approaches for future numerical implementations are given: ideas how to realize algorithmic steps and for the theoretical analysis of such algorithms, how data usage and active processors in each algorithmic step may be represented and that close attention has to be paid to the data dependencies.

- Possible extensions and improvements are put forward, both for the author's implementations to improve the current implementation and for the GASPI standard to lower the entry threshold for new users.

1.2 Structure

The structure of this thesis is divided into six parts.

Chapter I - Introduction The first chapter gives a short motivation of this thesis, addressing the class of machine learning algorithms and focusing on artificial neural networks and their significance in today's world. Furthermore, the need for parallelization of the training methods of these models is described.

Chapter II - Neural Networks A general introduction into machine learning is given based on the description of a simple feedforward network (FNN) and a presentation of a common learning method, the backpropagation. The task of parallelizing large-scale FNNs is put forward and different parallelization techniques are discussed. Finally a new parallelization approach is proposed: the poadSGD method. It is described and its key requirements regarding the asynchronous communication scheme and the main algorithmic part, the matrix multiplication for a special distributed case are explained.

Chapter III - Parallel Computing Models In this chapter different parallel computing models are introduced, leading to the definition of the PGAS model and ending with a brief presentation of the two main PGAS languages and a short overview of further PGAS languages.

Chapter IV - GASPI At the start of this chapter, the parallel programming model GASPI is introduced. A brief outlook on its history is given and then GASPI's main features are described. The introduction is illustrated by small, numerical examples. Finally GASPI's main characteristics are compared to those of two other PGAS languages, UPC and CAF.

Chapter V - Dense Matrix Multiplication in GASPI This chapter first introduces the data distribution and its implementational concept that is key to the implementation of the algorithms as it also is the basis for the later work distribution. Hereinafter the implementation of a matrix multiplication is presented which applies to the special case of the matrices residing on different process groups. It is followed by a theoretical analysis of its algorithmic parts. Finally, the results of different runs on a high performance cluster and the achieved scalability are discussed.

Chapter VI - Conclusion and Outlook The last chapter starts with an overview of the work accomplished in this thesis. The results are discussed and additionally, an overview of possible extensions and improvements is given. Finally, the work is concluded by giving an outlook on future development in parallel programming and the future significance of machine learning.

Chapter 2

Machine Learning and Neural Networks

After their first showdown in the 60s/70s, with the rediscovery of the backpropagation algorithm and the development of further network topologies, neural networks experienced a revival in science and industry.
This chapter gives a general introduction to machine learning, focusing hereby on the subclass of artificial neural networks. First, a simple version of a feedforward network and its learning method is introduced. The section thereafter presents variations to this network model and describes different optimization opportunities. This information is provided in order to give a more complete overview of this field but is not required in order to understand the whole of this thesis. Then, turning back to the basic feedforward network, the parallelization possibilities of neural networks are explained and an emphasis is put on the parallelization realized on generic compute clusters. Finally, a new parallelization approach is presented in order to achieve a pipelined, distributed version of a feed-forward neural network. The approach combines known with new parallelization techniques and leads to both a rethinking of the parallel paradigms used, as well as points out the need to further discuss a large-scale matrix multiplication.
The first section of this chapter is based on knowledge gained from various sources: the online book of Michael A. Nielsen [Nie15], a very detailed review paper in preprint form by Jürgen Schmidhuber [Sch14], the Coursera online course "Machine Learning" by Associate Professor Andrew Ng from Stanford University [Ng16] and a book on machine learning from Kevin P. Murphy [Mur12].

2.1 Introduction to Machine Learning

Learning to program most often one starts out with classic programming languages. A problem is given and an algorithm is devised to solve this problem. A prominent example from first year's programming class is to write code that imitates a calculator. Enter some numbers and operations and get the according results as an output. But a problem setup may also be much more complex and therefore require skilled and challenging programming code for solving it.
However, for the implementation of the calculator example, the rules for the computational steps are well known and taught at every school. But for other problems there may not be

such a simple recipe for solving, meaning they may not have a predefined set of explicit rules for the solution steps. An example are the tomography screens of a patient who has a brain tumor. Based on these pictures, writing an application that determines whether the tumor is malignant or benign is no straight-forward task but overly complex and it requires medical experience.
Here a different programming approach may step into place: *machine learning*. Arthur Samuel coined this term in 1959 and denotes machine learning[1] as the ...

> "...field of study that gives computers the ability to learn without being explicitly programmed."

This means learning how to solve a given problem without specifying any rules. Of course this does not mean that machine learning is a panacea with no effort or computational cost involved. In machine learning the algorithm is often learned based on previous experience. Input and output pairs are provided to the machine learning algorithm which then needs to decide on its own how to get from input to output.
Tom Mitchell gives a more formal definition [Mit97]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

So a doctor may provide several pictures of brain tumors and label each picture with the additional information whether it is malignant or benign. These labelled pictures are the previous experience E. The task T of the program would be to develop an algorithm which produces the risk classification for the current picture as well as newly added ones. Measuring the performance P would be equal to counting the number of correctly classified pictures. The derived algorithm would be acknowledged as "learning", if the number of correctly classified pictures increases.

Further applications arise from a wide range of topics, such as ([Sch14]):

- image recognition and classification,
- image segmentation,
- object detection,
- speech recognition,
- sequence recognition or
- propositional logic.

Concrete examples include the recognition of fingerprints, the detection of specific persons or faces in photographs or surveillance videos and the detection of fraud payments by credit card. Medical applications, such as the discovery of a tumor or its classification, can benefit from machine learning, too. Today, machine learning is already employed by miscellaneous companies, such as Google (search engine), Microsoft or Facebook (social network analysis) [Nie15].

[1] Note that this quote is cited multiple times throughout literature but the author could not track down the original source. It is sometimes referenced as a quotation from Samuel's paper on checkers [Sam59] (which is not true) and also cited as a quotation from an article in the "The New Yorker and Office Management" from 1959 [JEG16] to which the author had no access.

Classifications of Machine Learning Algorithms

The machine learning model is rather a generic concept. Algorithms following this model can be classified into three main categories: *supervised*, *unsupervised* and *reinforcement learning* [Mur12]. Depending on the literature the algorithms are categorized into more differentiated subcategories including semi-supervised and active learning.
For *supervised algorithms*[2] the previous experience E from Mitchell's model consists of a given data set together with correct outputs which the algorithm shall be able to reproduce most accurately. This type of algorithm can be applied to two different types of problems: *regression* problems for which the predicted result is continuous or *classification* problems where the output is discretely valued. An example for the first problem type could be the prediction of rental costs based on rentals of the previous years, whilst the formerly mentioned tumor classification turns out to be a classification problem.
Unsupervised algorithms[3] on the other hand only provide data input but no correct output or concrete goals. The rather general purpose often is to find some kind of structure within the input data. Again two different types of problems can be identified: *clustering* and *non-clustering* problems. As the name indicates, the clustering problem strives to find some kind of clustering of the data based on some relationships within the data. The non-clustering problem on the other hand looks for some other kind of structure. An example for the clustering problem is the grouping of news entries from different websites based on common topics as it is used by search engines. A different example may be the grouping of people in an election into groups of voters, undecided voters and non-voters based on a social analysis of their profile in a social network for further campaigns. A prominent non-clustering problem is the cocktail party problem which opts for filtering single voices from different sound recordings at a party.
For the third type of machine learning, *reinforcement learning*, no samples are given. Instead, learning occurs by means of encouragement, so it is based on rewards or punishment signals given by its environment throughout the learning process. This is what most closely resembles the way humans learn, for example when learning how to walk.
This thesis concentrates on the category of supervised learning.

2.2 Fundamentals of Neural Networks

There are several approaches to solving supervised machine learning problems. They range from learning based on a decision tree over applying support vector machines (SVM) or genetic algorithms to artificial neural networks and deep learning. In this section some concepts of machine learning are introduced based on the example of an *artificial neural network* (ANN), often also simply referred to as neural network, with a training algorithm used quite often, the backpropagation. The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous paper on that topic was published by David Rumelhart, Geoffrey Hinton, and Ronald Williams in 1986 (republished later as [RHW88]).

Biological Background

The original inspiration of neural networks was the information processing of the human brain. Why so? In comparison, computers already are quite large: they have up to and

[2] Also known as "predictive" algorithms [Mur12].
[3] Also known as "descriptive" algorithms [Mur12].

probably by today more than 10^9 transistors with a switching time of 10^{-9} seconds [Kri07]. However, a human brain has about 10^{11} neurons with a switching time of only 10^{-3} seconds. Furthermore, the brain with its rather simple layout can work massively parallel, can reorganize itself (some parts are able to take over tasks of other brain parts) and are fault tolerant against internal or external errors (up to a certain degree). Additionally, the brain is able to learn, either from training samples or by means of encouragement. A computer today is able to compute in parallel and there are certain means as to fault tolerance. But both are not as powerful as the human brain's performance and telling one part of a computer to cope for other failing parts is seemingly impossible. Picking up on these qualities, the artificial neural network, commonly known as neural network, was derived.
First, we take a step back and explore the "architecture" of a human brain in a simplified way and then we follow up on the artificial neural network. The information on the biological background was taken from [Kri07] and simplified a bit, focusing on the main characteristics of the brain's functionality. The major players of the human brain are the neurons. A neuron may be viewed as a switch with an information input and an information output. Even a fly already has about 10^5 neurons and humans with their 10^{11} neurons are again topped by elephants and some whale species who have twice as many. The neurons transfer information to other neurons via synapses. These can be electrical or chemical synapses. A chemical synapse is interrupted by a gap, the so-called synaptic cleft. In short, the incoming electrical signal is transformed into a chemical signal, some chemical processes occur in the cleft and the output is converted into an electrical signal again. The electrical signal then arrives at the neuron. The cell nucleus accumulates all the signals it receives from different synapses (from other neurons) into a single pulse. When the accumulated signal exceeds a certain threshold potential, an electrical pulse is sent on to other neurons.

2.2.1 Components and Basic Functionality of a Neural Network

The artificial neural network started out as a caricature of biology. The neurons, synapses and their functionality are the basic components of a neural network. Note, that there is not a single "neural network". There is a common basis of the structure and functionality and from there different neural networks have evolved. Next, a basic setup of a neural network is introduced.
Figure 2.1 shows an exemplary graphical representation of an artificial neural network. In the figure, the neurons are depicted as circles and are arranged in several vertical layers. Here, a distinction is made between the neurons in the first (most left) *input layer*, the *output layer* (most right) and the so-called *hidden* layers in-between. The connections between them correspond to the synapses. These are tagged with a certain *weight* value to account for the variability of the chemical processes in the synaptic cleft.
The basic functionality of a neural network can be viewed as the analysis of a given problem on the basis of several differently weighted decision levels. Consider the problem of identifying the fruit shown in a given picture (or set of pictures). The layers could represent different decision levels to this question. The first layer would take the picture in form of its pixels and hand this information on to the next hidden layer. This layer could, for example, detect the shape of the fruit. Based on the shape the next layer could decide on the main color within this shape. Then, another layer could categorize the texture of the shape and so on. The output of the last layer would be a number representing a certain kind of fruit.

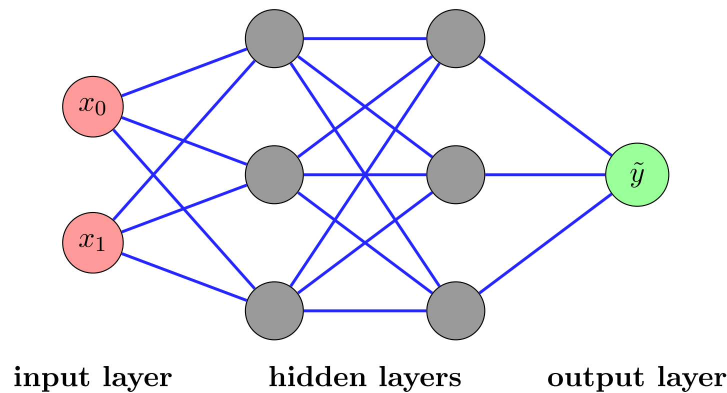


Figure 2.1: Depiction of a small neural network with two hidden layers.

Now how does this neural network work? In the 1950s/60s Frank Rosenblatt defined the network as above with perceptrons (artificial neurons) which take a binary input and output [Nie15]. Each perceptron (in one of the hidden or output layers) receives several input values via the incoming synapses which are additionally each provided with a weight. The binary output of a perceptron then depends on whether the sum of all weighted inputs exceeds a perceptron-specific threshold or not.

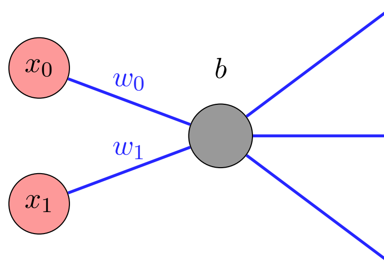


Figure 2.2: Single perceptron of a hidden layer within a neural network with weighted input synapses w_0 and w_1 and bias b.

The output of the perceptron in Figure 2.2 is then computed as

$$\text{output} = \begin{cases} 1, & \text{if } w_0x_0 + w_1x_1 \geq \text{threshold} \\ 0, & \text{if } w_0x_0 + w_1x_1 < \text{threshold} \end{cases}$$

or rewriting this equation as

$$\text{output} = \begin{cases} 1, & \text{if } \sum_j w_jx_j + b \geq 0 \\ 0, & \text{if } \sum_j w_jx_j + b < 0, \end{cases}$$

where the *bias* b acts as the negative threshold. The perceptron in this picture makes its decision by weighing up the results of the former layer, in this case the input layer.
However with these binary-valued perceptrons making small adjustments to some weights may result in large changes to the output. To avoid that, the *sigmoid neuron* may be used: its input and output values are real-valued and range from 0 to 1 and the function

determining the output is chosen such that small changes to the weights only cause small changes to the output. The corresponding function is referred to as *sigmoid* as it is often shaped like an "S" (see Figure 2.3). More generally, the sigmoid neuron is simply called

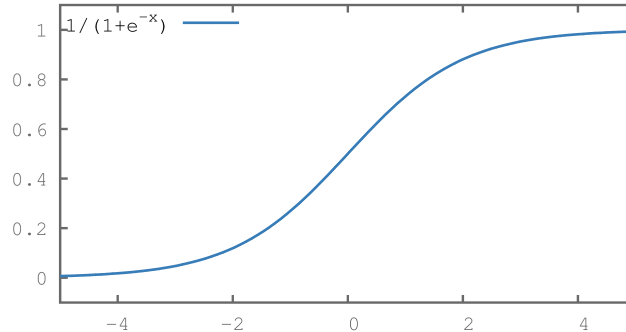


Figure 2.3: Sigmoid function often used in a neural network (see Equation (2.1)).

"the" neuron, not placing any restrictions to the output function which is then called *activation function* $a()$. Its task is more generally the limitation of the amplitude of the output of the neuron. Still, most commonly a sigmoid function is used, for example

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}, \tag{2.1}$$

where z is the sum of the weighted inputs.

Formal Definition of a Neural Network

An (artificial) neural network is a network consisting of multiple layers of neurons which are connected by weighted synapses and typically modelled as a graph. More formally the topology of the network is a triple (N, W, B) of neurons, weighted synapses and biases. The set of neurons is denoted layer-wise as a vector:

$$N := \{n^l \mid n^l_j \text{ variable denoting } j\text{-th neuron in } l\text{-th layer}, \ l \in [0, L]\}. \tag{2.2}$$

Similarly, the set of weighted synapses is a set of matrices where each matrix contains the weights on the synapses between the $(l-1)$-th and the l-th layer:

$$W := \{W^l \mid w^l_{ij} := (W^l)_{ij} \text{ is weight between } n^{l-1}_j \text{ and } n^l_i, \ l \in [1, L]\}. \tag{2.3}$$

Finally, each neuron is assigned a bias which is represented accordingly as a vector in each layer, too:

$$B := \{b^l \mid b^l_j \text{ is bias of neuron } n^l_j, \ l \in [1, L]\}. \tag{2.4}$$

The first column of neurons with $l = 0$ is the input layer with no incoming connections or biases. The last column $l = L$ is the output layer with no outgoing connections. For now it is assumed that neurons of neighboring layers are fully connected and the graph is

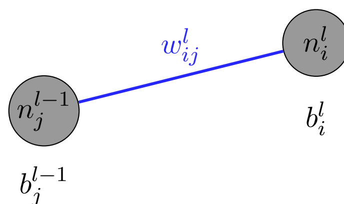


Figure 2.4: Two neurons from subsequent layers with weighted synapses and the corresponding biases.

acyclic, so without loops and synapses do not skip layers. Whilst the weights and biases are real values, n_j^l is only a variable used to clarify which node is meant. The indexing of these components is illustrated in Figure 2.4.
Each layer $l \in [1, L]$ corresponds to a function $f^{(l)}$ which for the neurons of each layer accumulates the weighted inputs and computes the activation function a for the input x:

$$x^l := f^{(l)}(x) = a(z^l), \quad \text{where} \tag{2.5}$$

$$z^l = W^l x^{l-1} + b^l. \tag{2.6}$$

Here, x^{l-1} denotes the input vector to the synapses leading to layer l and is equal to the output of the former layer. The dimension changes accordingly to the number of neurons in each layer. The input vector x^0 is usually simply denoted as vector $x \in \mathbb{R}^m$ (or in a normalized fashion $x \in [0,1]^m$) with m being the number of input elements for one sample. The same applies to the output which is displayed as $y \in [0,1]^{\overline{m}}$ with $\overline{m}$ specifying the number of output components. The auxiliary vector z is also named the weighted input. Note that the activation function is applied element-wise.
The neural network then is a function $f : x \mapsto \tilde{y}$ consisting of multiple chained functions

$$f(x) = f^{(L)}(f^{(L-1)}(...f^{(1)}(x)...)) = \tilde{y} \tag{2.7}$$

and dependent on the weights W, the biases B and the activation function a is applied[4]. Note that a network where the output of one layer corresponds to the input of the next is called a *feedforward neural network*. This particularly requires the graph to be acyclic.

2.2.2 Training a Neural Network

The basic concept of the neural network's supervised learning (or the user training it) is based on two steps: First the output of the network $f(x) = \tilde{y}$ is computed for a given training sample $\{(x, y)\}$. Based on a cost function that uses the difference between the network's output $\tilde{y}$ and the known solution y an optimization of the neural network is then computed. Thereby the parameters of the network are adapted, namely the weights W (see Equation (2.3)) and the biases B (see Equation (2.4)). This is carried out successively for each training sample given.
However, not all available training samples are used for the training of the network. Instead, the set of training data is subdivided into two (sometimes three) data sets: a training

[4] A mathematically better notation would be $f(W, B, a, x)$ and $f^{(l)}(W^l, b^l, a, x)$. However, the additional variables are omitted for a better readability.

set, a test set and sometimes also a validation set. The largest one, often two-third of the whole data set, is utilized for the training of the network. It is denoted as

$$(X,Y) = \{(x_j, y_j) \mid (x_j, y_j) \text{ is a pair of the training set with } j = 0, \ldots, n-1, \\ x_j \text{ is the input vector, } y_j \text{ the desired output}\}. \tag{2.8}$$

In case an iterative approach is used, i. e., the training algorithm may iterate several times over this training set. The validation set may then be applied to decide whether or not to continue training. Note, that the validation set is only applied after a supposedly successful training. Finally, to evaluate the performance of the trained network, the test set is used. The performance metric is often determined as the proportion of samples of the test set for which answers are given correctly. With the test set one can see how the model reacts to data it has not seen before.
As mentioned above, the outputs computed by the neural network are needed in order to set up the cost function. This is done by propagating the input information from each sample x_j of the training set through the network to get $\tilde{y}_j$. The formula for this *forward propagation* already was given in Equation (2.7) with the details noted in Equations (2.5) and (2.6).
In order to optimize the network, the *backward propagation* is employed. It was first defined by Rumelhart, Hinton and Williams in 1988 [RHW88]. They denoted it to be a procedure that ...

> "... repeatedly adjusts the weights of the connections in the network so as to minimize the measure of the difference between the actual output vector of the net and the desired input vector." [RHW88]

The measure of difference mentioned by Rumelhart et al. is what in this work is called a cost function and is used to determine when a network has improved. The cost function used in this work is the mean squared error of the outputs. It is given in Equation (2.9).

$$C(W, B, (X,Y)) = \frac{1}{2n} \sum_{x_j \in X} \|y_j - f(x_j)\|^2 \tag{2.9}$$

This cost function is non-negative, quadratic and convex, improving the probability of finding a global minimum. Optimizing the neural network then corresponds to minimizing the cost function.
As a means of minimizing the cost function, the *gradient descent* (GD) method can be applied. In order to minimize a function $g(v)$ with parameters $v = (v_1, v_2)$ the gradient of the function

$$\nabla g = \left(\frac{\partial g}{\partial v_1}, \frac{\partial g}{\partial v_2} \right)$$

is computed. The gradient then determines "the direction to go" to find the minimum. The GD law of motion [Nie15] then follows as:

$$\Delta v = -\eta \nabla g, \tag{2.10}$$

where η is a small and positive value which is called the *learning rate*.

Applying this law to the cost function (2.9) it yields the following update rules for the weights and biases:

$$w_{ij}^{l,new} = w_{ij}^{l,old} - \eta \frac{\nabla C}{\nabla w_{ij}^l}, \tag{2.11}$$

$$b_i^{l,new} = b_i^{l,old} - \eta \frac{\nabla C}{\nabla b_i^l}. \tag{2.12}$$

The gradient of C (2.9) however, can also be seen as an averaged gradient of the subfunctions $C_x = \frac{1}{2}\|y - f(x)\|^2$. These subfunctions can be computed with the help of the error δ_j^l in the j-th neuron of the l-th layer which is defined as follows:

$$\delta_j^l := \frac{\partial C_x}{\partial z_j^l}. \tag{2.13}$$

The error δ_j^l accounts for a change of the weighted input z_j^l in layer l and therefore also for changes in the weights and biases. Applying the chain rule to Equation (2.13) the error for the last layer L can be computed as

$$\delta_j^L = \frac{\partial C_x}{\partial f^{(L)}(x)_j} a'(z_j^L) = \frac{\partial C_x}{\partial \tilde{y}_j} a'(z_j^L). \tag{2.14}$$

This can also be written in vectorized form as

$$\delta^L = D^L \nabla_{x^L} C, \tag{2.15}$$

where D^L is diagonal matrix containing the entries $a'(z_i^L)$ and $\nabla_x^L C$ the gradient vector of C in layer L with respect to the sample x:

$$D^L = \operatorname{diag}(a'(z^L)) = \begin{pmatrix} a'(z_0^L) & & 0 \\ & \ddots & \\ 0 & & a'(z_m^L) \end{pmatrix},$$

$$\nabla_x C = \begin{pmatrix} \frac{\partial C_x}{\partial f^{(L)}(x)_0} \\ \vdots \\ \frac{\partial C_x}{\partial f^{(L)}(x)_m} \end{pmatrix}.$$

This error is then propagated backwards through the neural net, similarly to the forward propagation, yielding δ^l for $l = L-1, \dots, 1$ as presented in vectorized form in Equation (2.16).

$$\delta^l = \left((W^{l+1})^T \delta^{l+1}\right) \odot a'(z^l), \quad l = L-1, \dots, 1 \tag{2.16}$$

Here, $\odot$ is the Hadamard product which computes the product of two vectors element-wise. The propagated error δ^l can then be related to the gradients of C_x as follows:

$$\frac{\partial C_x}{\partial b^l} = \delta^l, \tag{2.17}$$

$$\frac{\partial C_x}{\partial W^l} = f^{(l-1)}(x)\delta^l. \tag{2.18}$$

Based on these equations the new weights and biases of the neural network can be computed. All in all, Algorithm 1 describes one learning step of the neural network based on the GD method.

It is known, that GD does not always perform well. For example, methods also making use of the second derivative converge a lot faster. However, only using the first derivative makes computation simpler and can be implemented in an easier way [RHW88]. More details about the issues of GD are explained in Section 2.2.4.

```
   // INPUT: Neural network (N, W, B) and training set (X, Y).
   // Initialization of network
 1 setup network topology
 2 initialize weights and biases
   // Iterate over training set
 3 for (x, y)_j ∈ (X, Y) do
       // Feedforward of input
 4     for l = 1, 2, ..., L do
 5         compute the weighted input z^l as in Equation (2.6);
 6         compute the output of this layer x^l as in Equation (2.5);
 7     end
       // Backward propagation
 8     compute the error δ^L in the neurons of layer L as in Equation (2.14);
 9     update weights W^L and biases b^L as in Equation (2.11) and (2.12) ;
10     for l = L − 1, ..., 1 do
11         compute error δ^l in the neurons of layer l as in Equation (2.16);
12         update weights W^l and biases b^l as in Equation (2.11) and (2.12) ;
13     end
14 end
```

Algorithm 1: Application of forward and backward propagation to a training set based on gradient descent.

In the setup of Algorithm 1, the network is trained with the same data (possibly in the same order) over and over again. Unfortunately, the backpropagation algorithm cannot be shown to converge [Hay09]. Therefore, some kind of stopping criterion is required to determine when the training is assumed to suffice. The computation of the gradient for the whole training set may take quite long [Nie15]. A variation of GD, the *stochastic gradient descent* (SGD) method therefore follows a different path to speed up the computation. The idea is to randomly divide the training set into smaller subsets and then train with each such *mini-batch* one after the other. After having trained the network with each mini-batch, the validation set is applied and the percentage of correct answers determined. This validation step avoids overfitting. If the result is not good enough, another such *epoch* is started. Different mini-batches are randomly created and again trained one by one. Algorithm 2 shows the stochastic GD method. The size of the mini-batches s and the number of epochs E are freely selectable. If the size s is chosen to be one, the method is called *on-line learning*.

```
   // INPUT: Neural network (N,W,B) and training set (X,Y).
   // epoch
 1 for e = 1,...,E do
 2 |   randomly shuffle training data;
 3 |   create mini-batches of size s: (X,Y)^m = {(x_i,y_i) ∈ (X,Y) | i = ms,...,ms+s};
   |   // training of network on mini-batches
 4 |   for m = 1,...,M do
 5 |   |   apply forward and backward propagation as in Algorithm 1 for neural
   |   |   network (N,W,B) and training set (X,Y)^m;
 6 |   end
   |   // evaluate neural network
 7 |   evaluate neural network with validation data;
 8 |   if percentage of correct results high enough then
 9 |   |   stop;
10 |   end
11 end
```

Algorithm 2: Stochastic gradient descent (SGD).

2.2.3 Characterizing Different Neural Network Models

In Section 2.2.1 a certain model of a neural network was introduced. But not all networks use the same topology or work the same way. In general, artificial neural networks can be characterized by three different properties [Lip88]:

- network topology
- node characteristics
- training rules.

The first aspect refers to the number and type of layers of the neural network and to the connectivity structure in the neural net's graph. Also, for some networks the topology may not be fixed but dynamically adaptable. The node characteristics are represented by the internal threshold (the bias) and the way a node processes its given inputs, including the non-linearity (the activation function) that is applied. Additionally, the node characteristics include the type of neuron activation, representing the moment when a neuron is activated. This can be done synchronously for all neurons of a network at once or asynchronously following the topological order or in a random order [Kri07]. The last aspect is the training rules. This is a broad field, ranging from choosing of the training set, initializing the network's parameters, choosing the loss function, determining how and when the network is updated down to deciding about the stopping criterion for the training.
Some network models not only assume a certain topology but may also fix the activation function or the training algorithm to be used. This work focuses on the usage of feedforward neural networks, like those encountered in Section 2.2.1. Beyond that, this section gives a short overview on the characteristics of different network models, as well as further activation functions and training algorithms for the interested reader. This overview is not necessarily required in order to understand the rest of this work but completes the basic overview of this topic. For more information on models and types of neural networks,

the author refers to a very detailed review paper from Jürgen Schmidhuber [Sch14], currently published as a preprint at this point. It contains a very detailed overview on the evolvement of neural networks up till 2014.

Models of Neural Networks

In Section 2.2.1, the *feedforward neural network (FNN)* was introduced. It is a multi-layer network with synapses only pointing in the direction of the subsequent layer. Thereby the output of one layer is always the input to the subsequent layer, no information is ever fed back. In other words, the corresponding directed network graph is acyclic. The network always consists of an input layer, a number of hidden layers and an output layer. By increasing the number of hidden layers, the network may extract higher-order statistics from its input [Hay09]. The connectivity structure between layers and neurons depends on the network's topology. A network is called *fully connected* when all neurons in a layer are connected to all neurons in the subsequent layer, otherwise it is called *partially connected*. At times, an FNN may also include shortcut connections: In this case a neuron is connected not only to the subsequent layer but also to another one closer to the output layer.

A special type of FNN is the *convolutional neural network (CNN)*. In addition to the fully connected layers known from the FNN, it contains special layers such as convolutional layers (from which the name is derived) together with feature maps or subsampling layers [BRSS15]. The convolutional layer extracts local features from its given input. This locality is implemented by connecting the neurons inside such a layer only to a few neurons of the previous layer which form a so-called local receptive field. Neurons which are part of feature maps share their weights, which leads to an invariance to shifts or certain transformations of the input data. This also reduces the number of parameters needed in the network. Pooling or subsampling layers perform a kind of non-linear down-sampling, thereby reducing the transferred data. The most common one is the max pooling. Combinations of convolutional and pooling layers are also referred to as MPCNN networks.
In general, a convolutional network is well suited for pattern classification and therefore often applied for image or video processing and operations involving the natural language. Some popular CNNs include the "LeNet", the "AlexNet" and the LSTM network [Nie15].

Another network type is the *recurrent neural network (RNN)*. The main difference between an RNN and an FNN is the removal of the feedforward restriction. In an RNN the synapses may also point backwards to a previous layer or back to a neuron itself. In other words, the corresponding directed network graph allows for backward connections and therefore cycles. Otherwise it is a static multilayer perceptron network [Hay09].
How does this affect the behavior of the network? Interpreting this in the biological sense, the neuron fires only for a limited time span [Nie15]. Firing backwards therefore does not affect other neurons instantaneously but in a different time step, avoiding dependencies.
So whilst a CNN performs a convolution in space, the RNN performs a convolution in time. The final output of the network then depends on the current input as well as all (or some) previous inputs. In addition to input, hidden and output layers an RNN also contains an element called a time-delay unit to delay the firing.
Such a network can also work in a "generative mode". That means it can produce new elements by sampling from the output probabilities. RNNs are often applied to the processing of natural language or speech and in general to applications which predict certain

time series such as the weather forecast. An example for an application to the natural language is the task of predicting the probability of the next word in a sentence or generating a scientific paper based on previous ones.

The following information about further network models in this section is based on a book from David Kriesel [Kri07]. Another supervised learning model is the *Hopfield network* that features completely linked neurons without any self-connections but with symmetric weights. The Hopfield network originates from simulating the behavior of particles in a magnetic field. Other possible models from the class of unsupervised learning are self-organizing maps, adaptive resonance theory or radial basis function networks. A short overview of these three is given in order to provide a better understanding of other network types.

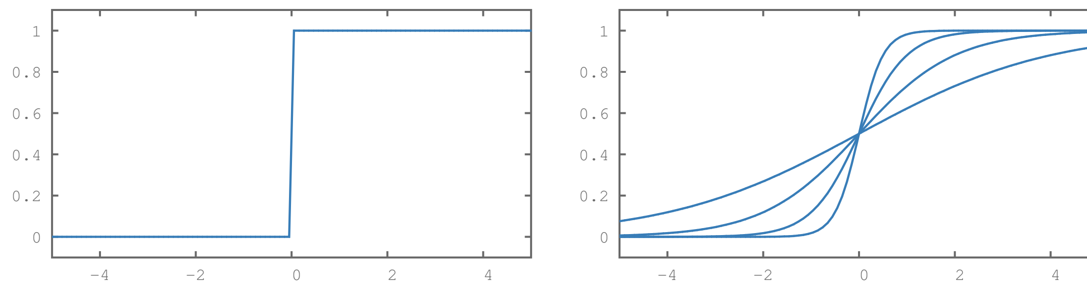
A *self-organizing map (SOM)* is basically a neural network where the output is not a vector of values but the state of the network itself. This can be compared to the state of our brain when storing memories. The brain has a concept which is learned and adapted given new impressions each day. When a new external input (a new impression) is given, it is stored in some way and the output is the new state of the brain.
The idea of an *adaptive resonance theory network (ART)* is to classify a given input by returning a 1-out-of-n output, following the winner-takes-all scheme. It only consists of two layers: the input and the resonance layer. These are fully connected in both directions, propagating activities in one layer back to the other one, leading to a resonance. Training an ART network therefore means adapting both weight matrices. First the activity in the input layer is propagated to the resonance layer, where the strongest corresponding neuron wins. Then the weights of the according weight matrix are updated to enhance the output even more. Afterwards, only the weights of the winning neuron are conveyed back to the input layer.
Finally, *radial basis function networks (RBF)* are again more similar to FNNs. However, the number of hidden layers is fixed to exactly one and different computational rules apply to RBF and the output layer. Firstly, neurons in the RBF layer do not have any input weights (or are constantly set equal to 1). Instead they compute the euclidean distance between the input and the position of the current neuron and feed this information into a radial basis function (the activation function). Secondly, neurons in the output layer do not apply an activation function (or only apply the identity function). So they basically compute a linear combination of the RBF layer's outputs. RBF networks originate from approximation theory [Dre05].

Activation Functions

Some of the activation functions were already introduced in Section 2.2.1 or mentioned alongside with the models of neural networks in the previous paragraphs. Here, three popular activation functions applied in standard neural network models such as the FNN are presented. These three functions are illustrated in Figure 2.5.
The *threshold function* is used by a binary-neuron, also referred to as McCulloch-Pitts model [Hay09]. For the weighted input of a given neuron, the output is computed as:

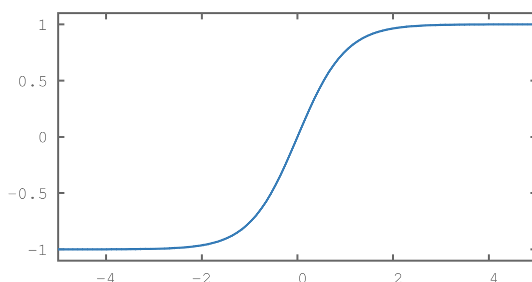
$$a(z) = \begin{cases} 1, & \text{if } z \geq \text{threshold} \\ 0, & \text{if } z < \text{threshold} \end{cases}$$

It has the all-or-none property, in this case $a(z) \in \{0, 1\}$.



(a) Threshold function used in McCulloch-Pitts model.

(b) Logistic or Fermi function $\sigma_\alpha(z)$ for different values of α.



(c) Hyperbolic tangens function $\tanh(z)$.

Figure 2.5: Examples of activation functions.

The most commonly used activation function is the *sigmoid function* which was introduced in Section 2.2.1. It has an S-shaped form and its output values are in the range $[0, 1]$. Contrary to the threshold function, the sigmoid function is differentiable. The more general version of it, the *logistic* [Hay09] or *Fermi function* [Kri07], allows for the usage of different slopes. It is defined as

$$\sigma_\alpha(z) = \frac{1}{1 + \mathrm{e}^{-\alpha z}}, \tag{2.19}$$

including the slope parameter α. As α approaches infinity, the function becomes the threshold function. Figure 2.5b shows the sigmoid function for different values of α.
In order to get values from a different range, the *hyperbolic tangens function* tanh can be used:

$$\tanh(z) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}}.$$

The activation function then ranges from -1 to $+1$. This may be desirable for certain cases [Hay09]. The function is illustrated in Figure 2.5c.

2.2.4 Challenges of Gradient Descent, Optimization and Regularization

For multilayer FNNs, the classical backpropagation was the first successful algorithm to be used [CGBFRAB06]. However, this version encountered various problems which are presented in the next paragraph. Thereafter different optimization and regularization techniques for these problems are briefly introduced and further literature is listed.

Challenges of the Gradient Descent Method

The depth of a neural network is determined by its number of layers L. A neural network is considered *deep* if it has a high number of layers. Using more layers means increasing the complexity of a problem. Each layer can be interpreted to be a decision step concerned with a different level of abstraction compared to the original problem. So a deeper network may give more precise results than a shallow one or be more general (i. e., better cope with new, yet unknown samples).

However, when using a deeper feedforward network with a gradient-based learning method and backward propagation, an instability in the learning process may be experienced [Nie15]. For either the earlier or the later layers of the network learning process proceeds noticeably slow.

Reflecting on the computations of the backward propagation, we notice that the gradient of the cost function is computed by multiplying the outputs of the activation function according to the chain rule. Using an activation function with absolute values smaller than one thus results in very small to almost zero gradients. Hence, the gradients for the first layers are very small which makes the learning progress there slow. By some researchers this is known as the vanishing gradient problem and was first described by Sepp Hochreiter in his diploma thesis [Hoc91]. If instead an activation function is used with significantly large gradients, learning in the first layers is considerably faster than in the later layers.

As shown by this example, the basic setup of learning for a deep neural network has got some obstacles. In general, other problems may occur as well. Classical backpropagation has been observed to be a rather slow learning algorithm [Roj96, CGBFRAB06]. If some parameters are not chosen well enough, it might become even more slow. The learning of an ANN being NP-complete, results in the worst case scenario having the computational effort for computing the parameters increase exponentially with the number of unknown parameters [Roj96].

Other than that the success of the gradient descent method highly depends on the initialization of the weights and the choice of the learning rate. If the learning rate is too small, the updates computed by backpropagation may get stuck in a local minimum of the nonlinear error function. If otherwise the learning rate is too large, the gradient direction may be trapped in a canyon of the error surface, oscillating frequently.

Finally, the success of classical backpropagation also highly depends on the problem it is applied to. Even though a network setup may provide good results, often a learning task can be found which makes the same network perform much worse [Roj96].

Neural Networks can also encounter a point in training where the training result first improves but then saturates. The result may be an *overfitting* or overtraining of the network. A reason may be that the network may be too complex and too few training samples may have been provided. The network may then adjust to noise in the data or simply memorize the input samples [Dre05]. As a result, it is not able to generalize well to new unseen samples.

The opposite can also happen: when the model is too simple, it will not be able to adapt to data with a high complexity. For example a model represented by a linear function will not be able to learn to map input and output pairs from a parabolic function.

Optimization and Regularization

Today, many variations of the backward propagation algorithm are used or applied a priori in order to cope with the challenges presented in the last section. Typically, not only one technique is used but a combination of them [Roj96]. However, finding out which techniques work best for a given problem is often only achieved by trial-and-error.
The choice of the initialization of the weights influences the convergence speed of the method applied. Castillo [CGBFRAB06] summarizes some of these methods, from statistically controlling the weights to initializing them with vector quantization prototypes.
In order to avoid the gradient direction leading to wide oscillations in a narrow valley of the error surface, a momentum term can be added [Roj96, Hay09]. Instead of solely using the negative gradient direction, a combination of both the current gradient and the previously computed update is used:

$$\Delta w_{ij}(t) = \eta \frac{\nabla C}{\nabla w_{ij}} - \mu \Delta w_{ij}(t-1).$$

The addition of the previous gradient direction is controlled by a scalar μ, the momentum rate. With its introduction there are now two parameters that need to be set appropriately, depending on the learning task.
A similar idea is to add a small constant ϵ to the derivative of the sigmoid [Roj96]. This fixed offset term then can help to move out of a too flat region of the error surface. The term is only added when moving across relative flat surfaces, otherwise the exact gradient direction is used.
Clipping the derivatives of the sigmoid function, for example by setting $\sigma(z) \geq 0.01$, is an approach proposed to avoid the vanishing gradient described in the previous section [Roj96]. The computation then does not use the actual derivative anymore but the vanishing gradient would be avoided.
Although the backpropagation algorithm is highly sensitive to the precision and the range of numbers which are being used [NS92], the floating-point operations also make the algorithm quite expensive. Therefore, the goal of the this technique is to reduce the number of floating-point operations [Roj96]. As the activation function is typically based on exponential functions (sigmoid and hyperbolic tangens), an idea is to not compute them explicitly but instead store the output values in tables and only look-up the result. An alternative is to use fixed-point arithmetic.
The step size taken by the standard backpropagation method is fixed. Contrary to this, the adaptive step algorithms follow the idea of adapting the step size by changing the learning rate based on additional information [Roj96]. The idea is to increase the step size whenever the direction leads further down towards the minimum and to decrease the step size whenever the algorithm oversteps a minimum. An example is the dynamic adaption algorithm which first generates two points and then moves to the point with the lower error [Roj96]. In case the learning rate is not global but each weight has its own local learning rate, the learning rates can be optimized individually to better correct the direction of the negative gradient. Further methods mentioned by Castillo et al. are the self-determination of the learning rate, a nonlinear adaptive momentum scheme for fast stochastic gradient descent and other new methods [CGBFRAB06].
Another set of algorithms aiming at optimizing the learning of a neural network are the second-order algorithms. Their basic idea is to find a better gradient direction and increase the speed of convergence by also including information about the curvature of the error

function in terms of its second derivatives [Roj96, Hay09]. Instead of evaluating the Hessian matrix and computing its inverse at each step, which is computationally expensive, a quadratic approximation is used or heuristics applied. Examples include the quasi-Newton methods DFP and BFGS which employ an approximation of the inverse, the Levenberg-Marquardt method and the conjugate gradient algorithms [CGBFRAB06]. In general, second-order methods are more efficient than methods based on gradient descent but they are not as useful concerning larger networks when trained in batch mode [CGBFRAB06].

Relaxation methods utilize the perturbation of weights [Roj96]. Instead of computing the gradient of the error function explicitly, it is discretely approximated. A small value is added to a weight and the update of the parameters is then based on the difference between the error of the gradient with and without the disturbance. Thereafter the next weight is randomly selected and the same procedure applied.

In their work Castillo et al. also show a sensitivity-based linear method for a two-layered FNN [CGBFRAB06]. The weights are learned by solving a system of linear equations and based on a sensitivity analysis.

The general approach to avoid overfitting of a model is to either increase the amount of training data or to reduce the model's parameters. In situations with large networks where overfitting cannot be avoided, regularization methods are applied. The methods can be divided into two classes, as described by Dreyfus [Dre05]. The first is the class of *early stopping* methods. The basic idea here is to stop the training before the minimization of the loss function is finished. The training may still improve for the given training set but may not generalize well anymore, if training was to be continued. An example for a stopping criterion is to monitor the variation of the standard prediction error of a validation set and to stop when that increases. However, the second class of regularization methods is often preferred: the *penalty methods*. A penalty term is added to the cost function that penalizes overly complex models. The most popular term is the *weight decay* which prevents the weight parameters from increasing too much.

2.3 Parallelizing Neural Networks

The advancement of computing power enables larger and more complex neural networks to be implemented. Such large-scale networks can contain several thousand neurons per layer, leading to networks with several billions of parameters [DCM+12]. Other networks increase the depth of the network in order to increase the complexity of the initial problem. All in all, the higher complexity leads to a higher computational demand, especially in the training phase [PLWH03]. As neural networks inherit the natural parallelism of their biological origin [Sei04], it is only natural to move onward with their parallelization.

The following sections give an overview of the kind of hardware employed for parallelizing neural networks, in what way parallelization techniques for neural networks were categorized so far and a short review on how these techniques were implemented in the past. The final section then proposes a new way of parallelizing neural networks: firstly, by employing a new parallelization method and secondly, by radically switching to a new parallel programming paradigm which makes use of its one-sided, asynchronous communication scheme.

2.3.1 Hardware Employed for Parallelization

Parallelizing the training phase of neural networks has been vastly considered [RS97]. Neural networks have been implemented on different kinds of hardware:
Most approaches focus on special purpose hardware such as GPUs which have the ability of performing computations on huge data sets massively in parallel, especially when batch training is used. In particular, GPUs apply well to convolutional neural networks. Now, although GPUs are well placed for these parallel computations, they are limited in their amount of memory when used as a standalone processor. Not all models fit into the memory of a single GPU [DCM+12]. Therefore, some researchers try to reduce the size of the data sets or to decrease the size of the network itself but often this is then less attractive for the problem it is applied to. Others consider using multiple GPUs. Already some software frameworks support the usage of up to 4 GPUs [AAB+15, JSD+14, BLP+12]. However, efficiently parallelizing large-scale neural networks on multiple GPUs remains more difficult to implement successfully.
Other researchers aim at using computers which are specifically designed for neural networks: different neurocomputers [FFA92] or neuromorphic hybrid systems such as the BrainScaleS [SBG+10]. These systems are closer to the biological origin of neural networks but are less well able to be applied to more general problems.
Also note, that not all computing labs or research institutions have access to advanced special-purpose hardware. Hence, being able to parallelize well when resorting to standard hardware may be of great advantage. Of course, using common computing clusters entails other drawbacks, due to the often high network latency and low bandwidth. Therefore, several different approaches have been made in order to avoid the occurring communication [DMN08]. Nevertheless, the high availability of computing clusters in many cases outweighs the potential limitation of speed-up. In addition, it was shown that the performance of an implementation on a compute cluster can also surpass that of an implementation on a GPU [DCM+12].
It remains to be said that there is no "best way" of parallelizing a neural network implementation. The best parallelization method depends on the type of the problem and the availability of hardware. This thesis rises to the challenge of using compute clusters with standard off-the-shelf hardware.

2.3.2 Types of Parallelization

Artificial neural networks offer many possibilities of parallelizing them. In their paper from 1992, Nordström and Svensson describe six different methods of parallelization [NS92] when using several processing elements at once. Not all types provide equally useful speed-up and may also work differently well on different types of hardware. Later authors usually chose only one or two of these methods.
The first parallelization type in the hierarchy described by Nordström and Svensson is the *parallelization of the training session*. Here, each processing element receives a copy of the complete neural network and the training set. However, each network is initialized differently and may also train with different learning rates. After training them individually for a certain number of epochs, the best trained network is selected. This method makes it possible to try a lot of different network types without having to restart a training session over and over again.
The next type of parallelization is based solely on the distribution of the training set: the *training example parallelization*. Also referred to as sample parallelism [BP95], ex-

emplar parallelism [RS97] or pattern parallelism, the overall term *data parallelism* is self-explanatory. Again each processing element holds a replica of the neural network, only this time the networks have identical parameters to start with. The training set is then distributed across the processing elements. Either each subset is chosen to be disjoint of the others or the subsets are selected randomly [DMN08]. Each process then trains its replica based on its sample subset. At the end of an epoch, the weight changes are accumulated and all network copies synchronized. In which way the weight exchange and update occurs, is implementation-dependent. Pethick et al. [PLWH03] accomplish this by exchanging the data with one master process whilst Rogers and Skillicorn [RS97] implement a tree method.
Layer parallelization, also called *Forward-Backward Parallelism* is based on the idea of pipelining the computations by sending one sample directly after the last through the network, so that several are processed simultaneously in different layers. Nordström and Svensson considered this type of parallelization not to have enough effect to be further considered.
More popular in being used are the following two types of model (or network) parallelism: neuron parallelism and weight parallelism. *Neuron* or *node parallelism*[5] distributes a network across several processing elements such that all incoming weights to a neuron are held by one processing element. Related to the weight matrix of one layer this is equivalent to mapping a row of the weight matrix, corresponding to all synapses leading to one neuron, to a process. Thereby, all weighted sums and activations in a layer are computed concurrently. According to Dahl et al. [DMN08] and Dean et al. [DCM+12], this type of parallelization only then yields good performance when the distributed neural network being used is large enough to diminish the effect of dominating communication costs.
The second way of distributing a network across the processing elements is the *weight parallelism*. The weights are distributed such that all inputs to a neuron can be processed in parallel. This way the neurons of a layer are mapped to different processes. For the weight matrix this corresponds to a column-wise distribution to the processes.
A combination of the latter two parallelization types not mentioned by Nordström and Svensson but considered by Rogers and Skillicorn [RS97] is the *block parallelism*. Rogers and Skillicorn here map the neurons of the network to the processes in a block-wise fashion.
The last parallelization type Nordström and Svensson introduce is the *bit parallelism*. It refers to processing the bits of a value simultaneously and was found not to increase speed-up enough to be worth further consideration.

2.3.3 Review of Parallelization of Neural Networks on CPUs

Over the years, parallelizing a neural network for a computing cluster was done based on the different types of parallelization described in the previous section. Here we present some of the research that has been carried out in this field throughout the years.
As mentioned before, Nordström and Svensson [NS92] introduced different types of parallelization. Additionally, they described all kinds of different parallel computers that were specifically designed for or at least employed to implement neural networks.
Further special-purpose built neurocomputers are mentioned by Fujimoto et al. [FFA92]. They also propose two new parallel architectures for neurocomputers: the toroidal lattice architecture (TLA) and the planar lattice architecture. The key idea behind these archi-

[5] Also called *unit parallelism*.

tectures is to split the neuron model into its synapses and dendrites, i. e., the weighted sum of the inputs, and the cell body which computes the activation function. These basic functionalities are then mapped to hardware, leading to dedicated synapse and cell processors, as well as processors specialized for input and/or output purposes. Fujimoto et al. find the performance of their prototype of the TLA neurocomputer, when applied to the traveling salesman problem and the identity mapping, to be almost proportional to the numbers of processors used.

Besch and Pohl approach the parallel implementation of an FNN using the BP training algorithm by encapsulating functional units such as the computation of the gradient, the calculation of the error and the update of the weights [BP95]. These functional units are then parallelized and are re-used as black boxes within the code. The communication within the units is accomplished by a distributed logarithmic tree.

Rogers and Skillicorn on the other hand compare different parallelization techniques: examplar (training example), block and neuron parallelism [RS97]. Contrary to the neuron parallelism described by Nordström and Svensson [NS92], Rogers and Skillicorn choose to map the neurons of the neural network randomly to the processing elements. Furthermore, they reduce the block parallelism soon to layer parallelism, by choosing the width of the rectangle to be one. These three models are compared by viewing them as bulk synchronous parallelism (BSP) programs. BSP programs consist of sequentially executed *supersteps* which contain three phases: local computation, global communication and a synchronization step. Based on this program model, Rogers and Skillicorn find all three parallelism types to require the same amount of time for computation but that the exemplar parallelism outperforms the other two considering the communication cost.

Again different implementations of the backpropagation on a cluster computer are compared by Pethick et al. [PLWH03]. They focus on exemplar and node parallelism whereas for the latter they also discuss whether to update after each sample or only after computing a whole batch. For both parallelism schemes communication is performed by sending data back and forth with a master process. They use one hidden layer and vary the number of neurons per layer (ranging from 125 to 2000), the size of the training set (between 100 and 20,000 samples) and using different numbers of processes (up to 32). Pethick et al. conclude that node parallelism tends to outperform exemplar parallelism for either small training sets or large networks. If it is the other way around, exemplar parallelism wins.

Similarly, Dahl et al. parallelize an artificial network based on backpropagation for an eight-node cluster with pattern parallel training [DMN08]. However, here the subgroup of patterns for each processing element are chosen randomly, thus opening the possibility that not all samples have been processed in the end. Also, in contrast to Pethick et al., in this publication the global synchronization is not performed via a master process but by using the MPI Allgather function which is basically a synchronous broadcast call. For the local computations, the open source library FANN [Nis03] is employed. In total they get a best speed-up factor of 10.6 for 8 nodes being used.

A parallel implementation of the stochastic gradient descent algorithm is performed by Zinkevich et al. [ZWLS10]. Their algorithm *SimuParallelSGD* distributes the training set to the processing elements which all have a copy of the network. Each network is trained with its batch of data and computes the updates of the gradient independently. After the batch is processed the weights from all processes are aggregated and averaged weights are computed. Zinkevich et al. praise their algorithm for its small amount of communication due to communication only being necessary at the very end of the training phase.

In their paper Dean and his team introduce the software framework DistBelief [DCM+12].

Within this framework they developed two new algorithms for simulating neural networks: *Downpour SGD* and *Sandblaster L-BFGS*. Downpour SGD is an asynchronous variant of the stochastic gradient descent algorithm whilst Sandblaster L-BFGS relies on a distributed batch optimization. Both algorithms combine both model and data parallelism and follow the concept of a centralized sharded parameter server. These parameter servers each hold a part of the network parameters and run independently from each other. Other groups of processes each fetch the current network parameters, train on a subset of the training set and send their weight updates to the central servers. For the Sandblaster algorithm there is also an additional coordinator process. The updates occur asynchronously and at any time parameters retrieved at a given time may already be outdated. Dean et al. apply their algorithms to speech recognition and visual object recognition applications with 5 layers and 2560 hidden nodes each for the first and 3 stages with 21,000 nodes in the output layer for the second application. The training sets contain 1.1 billion labeled samples and 16 million images, respectively. As a final result, they find that "given access to sufficient CPU resources, both Sandblaster L-BFGS and Downpour SGD with Adagrad can train models substantially faster than a high performance GPU" [DCM$^+$12].
Keuper and Pfreundt propose the algorithm *ASGD* which is based on stochastic gradient descent and which especially relies on an asynchronous one-sided communication paradigm [KP15]. Following a scheme similar to SimuParallelSGD, the algorithm initializes the same network on all nodes and provides each with a subset of the training samples. For a certain number of iterations, the algorithm then proceeds with its computations locally thread-parallelized and independently from the other nodes. Each node randomly chooses a mini-batch from the subset and computes the aggregated update of the samples of this batch. The weight update consists not only of the locally computed results but is combined with an update computed by another node at a different iteration step. To ensure the combination improves the result, the *Parzen-Window* optimization [KP15] is applied. After completing the local update, the node sends its the result to another randomly chosen node, writing it into a buffer of the receiving node. The receiving node then uses this result again for its own weight update. The communication occurs asynchronously, thereby opening the possibility of overwriting updates in the buffer that have not been used yet or working with only partially written data. Keuper and Pfreundt refer to this as a lock-free approach which scales quite well on up to 1024 cores.

2.4 New Parallel Method for Neural Networks: poadSGD

The network this work focuses on is a large-scale fully-connected feedforward neural network featuring a high number of hidden layers with a large number of neurons each. For easiness of use it is assumed to have the same number of neurons in each hidden layer. As a consequence, the network has a very high total number of parameters which do not fit into a single processor's RAM, leading to the need for its parallelization. As mentioned before, various types of hardware can be employed for this task. However, in lack of special-purpose hardware, the implementation shall be applicable to a compute cluster consisting of standard commodity hardware. The author therefore proposes the training algorithm *poadSGD*, a **p**ipelined, **o**ne-sided and **a**synchronously updated, **d**istributed variant of **SGD**.
The next Section 2.4.1 introduces the key ideas of the algorithm. The algorithm itself is introduced in Section 2.4.2. The special focus of Section 2.4.3 is on the distribution of the network data. The last Section 2.4.4 about the poadSGD algorithm concludes with

an outlook emphasizing the most crucial points needed when implementing this algorithm, concerning the programming model and the numerical algorithms.

2.4.1 Key Ideas

The author's idea of parallelizing the training for this network is based on picking up on characteristics from the existent concepts of parallelism mentioned in Section 2.3 and combining them with new ideas.
The first step of the poadSGD method is to distribute several successive layers across groups of processes. Grouping the L layers into groups of γ_l successive layers and dividing the $\mathtt{P}$ processes into just as many groups of size γ_p, they are mapped as follows:

$$\begin{array}{lcl} \textbf{\textit{layers}} & & \textbf{\textit{processes}} \\ \{1,\dots,\gamma_l\} & \mapsto & \{1,\dots,\gamma_p\} \\ \{\gamma_l+1,\dots,2\gamma_l\} & \mapsto & \{\gamma_p+1,\dots,2\gamma_p\} \\ & \vdots & \\ \{L-\gamma_l+1,\dots,L\} & \mapsto & \{\mathtt{P}-\gamma_p+1,\dots,\mathtt{P}\} \end{array}$$

This is a variant of the block parallelism described by Rogers and Skillicorn [RS97]. They organized the network's neurons into blocks of depth x (in the direction of crossing layers) and width y (the direction within one layer). In the case of poadSGD the depth x coincides with the number of successive layers in a group γ_l and the width y is simply chosen to be the complete length of each layer m. An exemplary distribution is shown in Figure 2.6.

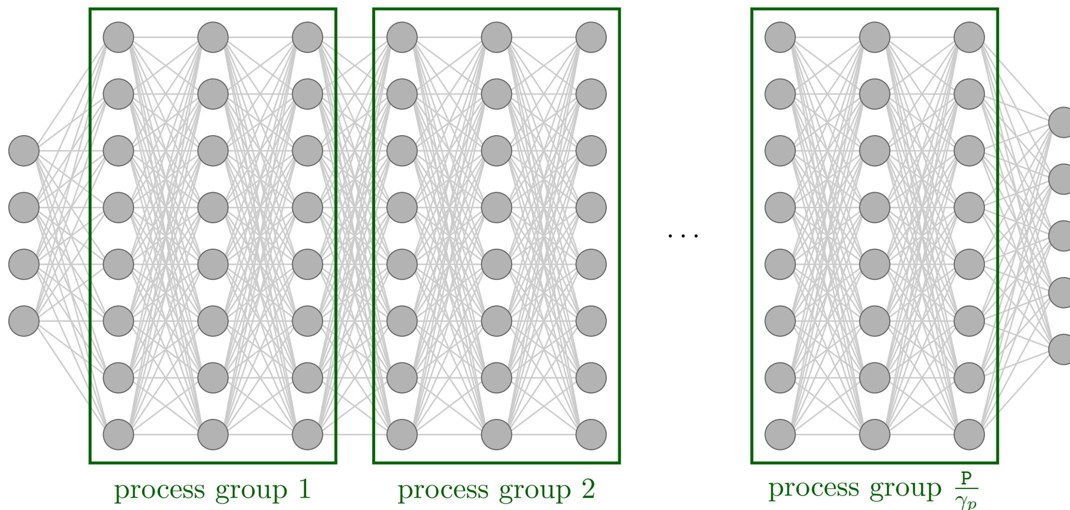


Figure 2.6: Mapping of $\gamma_l = 3$ successive layers to process groups.

Next an idea of the layer parallelism mentioned in [NS92] is employed: the *pipelined processing* of samples. Here, the first group of processing elements proceeds with the computations corresponding to its layers. When the first group has processed its layers, the feedforward step from one layer group to the next is computed together by these two process groups. Hereafter, the second process group continues computing its layers' contributions and accomplishes the last step again together with the following process

group and so on. To avoid the first process group being unemployed, the samples of the training set are sent through one by one. Once the pipeline is filled, all process groups are continuously active and work independently and in parallel. The only exception is the border case of the layer groups in which case two process groups have to work together. In order to directly proceed with the backpropagation, a *reverse copy* of the network is attached to the end of the pipeline. An exemplarily visualization based on a quite small, not yet distributed network is shown in Figure 2.7. After the output of the feedforward propagation has been computed it is directly used to start the backpropagation of the error.

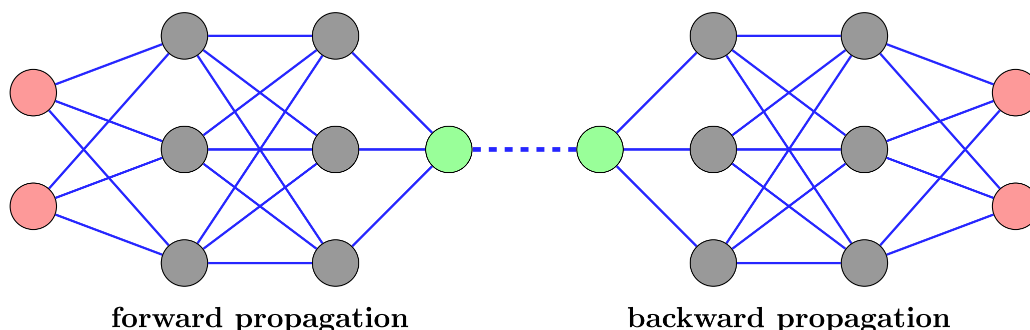


Figure 2.7: Attaching reverse copy of network: forward and backward propagation.

Another idea that is adopted is a *batch-wise* operation similar to the SGD algorithm given in Algorithm 2. Instead of performing the feedforward and backward propagation steps only for one sample at a time as shown in Equation (2.6), a batch of samples is propagated simultaneously. The batches

$$(X,Y)^k \quad \text{with} \quad |X^k| = |Y^k| = n+1 \quad \text{and} \quad k = 1,\dots,\mathtt{N}/(n+1)$$

are chosen either randomly from the complete training set (X,Y) – with $|X| = \mathtt{N}$ (see Equation (2.8)) such that they are disjoint – or else fully randomly without any specifications except for the size of the batch.
The computation of the feedforward and backward propagation is accomplished by adapting their rules to work with all samples of a batch at once, thereby performing matrix matrix multiplications instead of matrix vector multiplications. In the feedforward algorithm – Equations (2.5) and (2.6) – computing the weighted sum and executing the activation function for the batch X^k is then reformulated as follows:

$$X^l := f^{(l)}(X^k) = a(Z^l)\,, \quad \text{where} \tag{2.20}$$

$$Z^l = W^l X^{l-1} + B^l \tag{2.21}$$

with

$$X^l = (\, x_0^l \,|\, x_1^l \,|\, \dots \,|\, x_n^l \,),$$
$$Z^l = (\, z_0^l \,|\, z_1^l \,|\, \dots \,|\, z_n^l \,),$$
$$B^l = (\, b^l \,|\, b^l \,|\, \dots \,|\, b^l \,),$$

$l = 1,\dots,L$ and W^l as the weight matrix between layer $l-1$ and l as specified before. Note, that x_j^l denotes the output vector of layer l when computed in the feedforward

propagation using the j-th sample of X^k as network input. In other words, element X^l_{ij} is the output of neuron i of the l-th layer when considering sample j.
The same conversion is done with the equations of the backward propagation for the error of samples processed in a batch. The equations for the computation of the error in each layer for a single sample were given in Equations (2.15) and (2.16). The error is represented as an error matrix

$$D^l = \left(\delta^l_0 \,|\, \cdots \,|\, \delta^l_n \right),$$

where $l = L, \ldots, 1$ and δ^l_j is the error vector in the l-th layer for sample j of the batch. Based on these, the error D^L in the final layer L yields

$$D^L = G \odot A^L, \tag{2.22}$$

where G represents the matrix of gradient vectors and A holds the derivative vectors of the activation function:

$$G = \left(\nabla_{x_0} C \,|\, \cdots \,|\, \nabla_{x_n} C \right)$$
$$A^l = \left(a'(z^l_0) \,|\, \cdots \,|\, a'(z^l_n) \right).$$

In this case the entry A^l_{ij} contains the derivative of the activation function for the weighted input of the i-th neuron of the j-th sample in layer l. Again the Hadamard product $\odot$ is used to denote the element-wise multiplication.
Likewise, the computation of the error D^l in the other layers is derived:

$$D^l = \left((W^{l+1})^\top D^{l+1} \right) \odot A^l, \quad l = L-1, \ldots, 1 \tag{2.23}$$

Both main Equations (2.21) and (2.23) of the feedforward and the backward propagation are now mainly based on the matrix-matrix multiplication of the weight matrix with another matrix. The backward step uses the transpose of the weight matrix. However, in order to use the same implementation of the matrix multiplication, for the backward part of the network the weighted matrix W^k is directly stored in transposed form. The corresponding computation then simply omits the transpose in Equation (2.23). For batches of the same dimension as the length of a layer, these matrix multiplications are performed for two squared matrices. In general, this is not necessarily the case.
After computing the gradient updates for a given batch, these updates are averaged and applied to both the forward and backward network. Since a global synchronization of all processes (of both networks) would undermine the pipelining effect, the update of the weights occurs asynchronously. A concept of the ASGD algorithm from Keuper and Pfreundt [KP15] is used: relying on asynchronous and one-sided communication for this tasks. However, unlike Keuper and Pfreundt the updates are not asynchronously written to a random node that started out with the same initial network. Instead, there is only one distributed network that is to be updated. This update occurs similar to the asynchronous Downpour SGD of Dean et al. [DCM+12] whenever an update of a batch has been computed. Performing this in a one-sided and non-blocking way, it does not interfere with the ongoing computations of other batches by the receiving processes. The main difference to the asynchronous update of Downpour SGD is that with poadSGD the network is not stored by a central instance but by the processing elements themselves. However, they share the fact that the processing of a single batch may not occur with up-to-date parameters and parameters may even be changed whilst being used. This process

is continued until one of the following applies: either a certain threshold is reached, e. g. after a certain number of batches all processes halt at a global barrier and the training is evaluated or a specified number of batches has been processed.

2.4.2 The poadSGD Algorithm

All of these characteristics of the poadSGD algorithm are gathered in Algorithm 3.

```
   // INPUT: Large-scale fully-connected neural network (N, W, B),
      training set (X, Y) and groups of processes {P_0, ..., P_P}.
   // Setup of network
1  initialize_network(N, W, B);
2  attach_reverse_copy(N, W, B);
3  distribute_layers(W, P);
   // Train for s iterations
4  for i = 0, ..., s − 1 do
      // First process group: Start Pipeline
5     if mygroup == P_0 then
6        pick_batch((X, Y), rand, n + 1);
7        inner_feedforward();
8        notify_process(P_1);
9        right_border_feedforward(P_1) ;
10    end
      // Inner process groups: Pipelined Computations
11    if mygroup ≠ {P_0, P_{P−1}} then
12       wait_on_notify(P_{mygroup−1});
13       left_border_feedforward_or_backward (P_{mygroup−1}) ;
14       inner_feedforward_or_backward();
         // Last Process group of Pipeline
15       if mygroup ≠ P_{P−1} then
16          notify_process(P_{mygroup+1});
17          right_border_feedforward_or_backward(P_{mygroup+1}) ;
18       end
19    end
      // Last process group: Compute Network Update and Distribute
20    if mygroup == P_P then
21       wait_on_notify(P_{P−1});
22       compute_weight_updates();
23       async_update(P_0, ..., P_{P−1}) ;
24    end
25 end
```

Algorithm 3: Application of forward and backward propagation to a training set based on gradient descent.

As the characteristics were already mentioned before, the steps are only briefly commented. In Line 1 and 2 the network is set up, the topology is created, attaching the reverse network, and the weights and biases are initialized. The layers are distributed across the process groups in groups of subsequent layers as shown in Figure 2.6. The exact mapping is discussed in more detail later in this section. Then the training starts with the feedforward and backward processing Line 4. In this case, as a stop criterion a fixed number of iterations was chosen. The algorithm decides on what work is to be accomplished based on the process group a process belongs to. The first process group P_0 starts the pipeline in Line 6 by picking the batch samples for this iteration from the whole training set and computing the first forward propagation of its layers in Line 7 based on Equations (2.21) and (2.20).
The general setup for the forward and backward computational steps is then as follows: Reducing the view to a single process group P_i either in the forward or backward network, the processes of P_i first compute their first layer step from their left-most layer together with process group P_{i-1}. This joint calculation is referred to as a "border" step (either feedforward or backward) as in Lines 8, 12 and 16. The right or left depends on the position of the layer within the local layer group. In order to start a border case step only when both process groups are ready, a notification is sent from P_{i-1} to P_i. Only then does the computation start. Afterwards, the next layer steps are computed. This all occurs only within the process group P_i, it is an "inner" feedforward or backward step (Line 7 and 14). After the final outputs of the last layer are computed, the process group P_i notifies the next process group P_{i+1} holding the next group of layers. Again a synchronization between the two process groups occurs as they get together to compute the next forward or backward step together.
When a batch has passed all forward and backward layers, from Line 19 on, the updates are computed (averaging them as mentioned before) and finally applied to the network. In order not to interfere with the ongoing computations of the other process groups which already process the next batches, the update shall be accomplished by using one-sided and asynchronous communication. That is, the final process group sends the weight updates to the other process groups without interrupting their ongoing computations or the processes needing to post some kind of communication call in order to receive the data. The sending and the arrival of the updates is an asynchronous process. Of course, the parallel programming language used for implementation needs to support these communication schemes.

2.4.3 Distribution of Weight Matrix

Now, one question that has not been addressed yet, is how exactly the layers are mapped to the processes. To be more specific, the question is how the weights of the weight matrix are distributed within its assigned group of processes P_i. In neuron and weight parallelization, the weights are distributed such that all incoming weights to a neuron or all outgoing weights from a neuron are held by a processing element. The new approach of the poadSGD method is, not to distribute the weights based on where the weights go to or come from a certain neuron. Instead the distribution of the weights depends on their position in the weight matrix of the later computations in Equation (2.21) and (2.23).
The idea is to distribute the matrix in a two-dimensional block-cyclic fashion based on a process grid. A more detailed explanation of the distribution is given in Ch. 5.1. For an exemplary illustration a process group is assumed to consist of four processing elements: the blue, the red, the yellow and the green one. The distribution of the weight matrix is

then accomplished as shown in Figure 2.8.

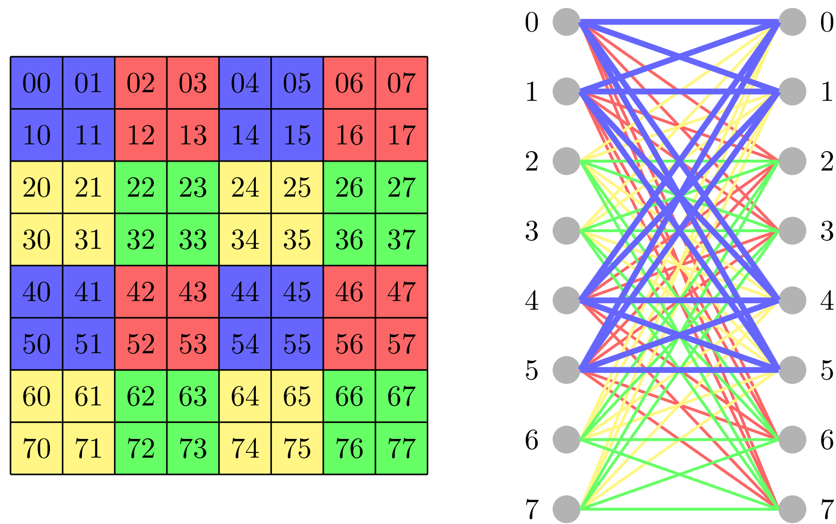


Figure 2.8: On the left: Two-dimensional block-cyclic distribution of a weight matrix to a group of four processing elements (blue, red, yellow, green) and on the right the corresponding mapped weights between two hidden layers.

This distribution of the weight matrix mainly affects the computational steps of the poad-SGD algorithm: the inner feedforward or backward steps and the border feedforward and backward steps. Inner steps mean matrix-multiplications within a block of layers that are held by the same process group, as presented in Figure 2.9.

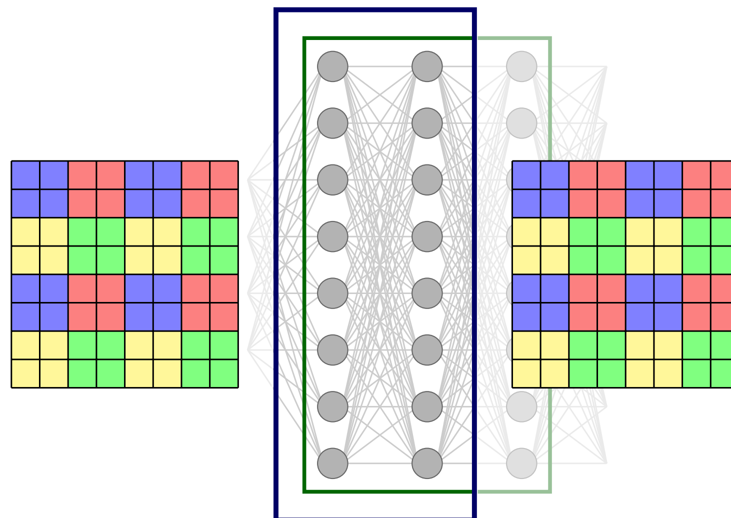


Figure 2.9: Inner case of the layer grouping: Matrix multiplications within a block layer. Both matrices are held by the same process group.

On the other hand, matrices occurring in the computational step in between two group-

ings of layers, in the border case, are held by separate groups of processes as shown in Figure 2.10. This proves to be a real challenge due to the increase of communication

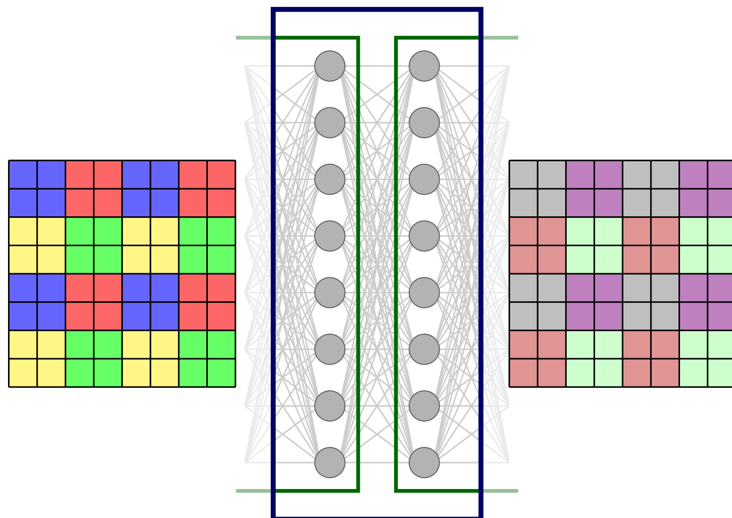


Figure 2.10: Border case of the layer grouping: Matrix multiplications across block layers. The matrices are held by disjoint process groups.

necessary in order to compute the matrix-matrix multiplication of the border case with both groups together.

2.4.4 Outlook

The poadSGD algorithm introduced in this section is based on several key points which are visualized in Figure 2.11.

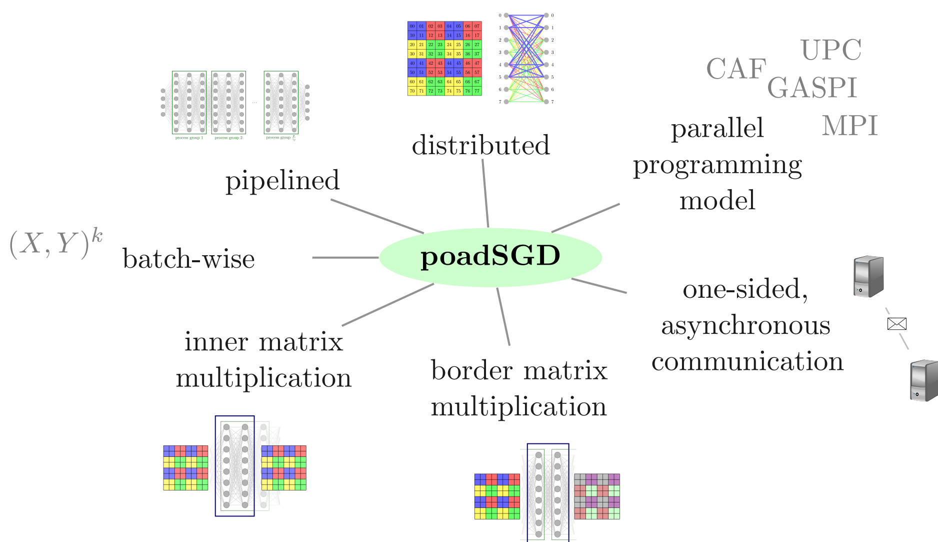


Figure 2.11: Key characteristics of poadSGD algorithm for large-scale FNNs.

Aiming at an efficient realization on large-scale systems, one crucial point is the implementation of the “border case” of the matrix multiplication in the feedforward and backward step. That the matrices involved are held by disjoint groups of processes poses a challenge of additional data exchange and distribution of the work amongst the processes.
Next to thinking about how to algorithmically implement this task, another major question is how to deal with the additional communication. This involves not only the data exchange between the two process groups in the border case but also opens the question of how to update the network's parameters without introducing unnecessary global synchronizations or disturbing ongoing computations. The author proposes to circumvent this problem by not working with standard synchronous communication mechanisms as often know from MPI [Mes12] but to look into the area of programming models based on asynchronous communication and a globally addressable address space for the asynchronous update of the weights.
In the following we will focus on these challenges proposed, believing unless feasible solution to these key points are found, the approach of the *poadSGD* method may not work efficiently. The next chapter (Ch. 3) will give an overview of different parallel programming models and indicate a new parallel programming model *GASPI* which is introduced in Ch. 4. The proposed implementation of the matrix-matrix multiplication for the border case of the poadSGD is presented in Ch. 5. An implementation of the complete poadSGD algorithm is beyond the scope of this thesis.

Chapter 3

Parallel Computing Models

This chapter covers the background on parallel architectures and parallel programming models by presenting different ways of classifying them. This includes a classification based on the system's physical memory architecture and a differentiation based on the view on memory which is enforced by the programming model used. Therefore, first the classification of the *tightly* vs. the *coupled system* architecture is presented. Then, the differentiation between the *shared* and the *distributed programming model* is described, together with a brief presentation of their respective dominating programming languages. This leads to another parallel programming model, the *partitioned global address space* (PGAS) programming model which is described in the section thereafter.
In order to better illustrate the PGAS model, two PGAS language extensions and their handling of distributed data are described. They are also often considered to be the standard ones in the PGAS realm: Unified Parallel C (UPC) [UPC13] and Coarray Fortran (CAF) [NR98]. Finally, a basic overview of further languages and language extensions, current and dispersing, which are associated with the PGAS model is given.

3.1 Classifying The Parallel World

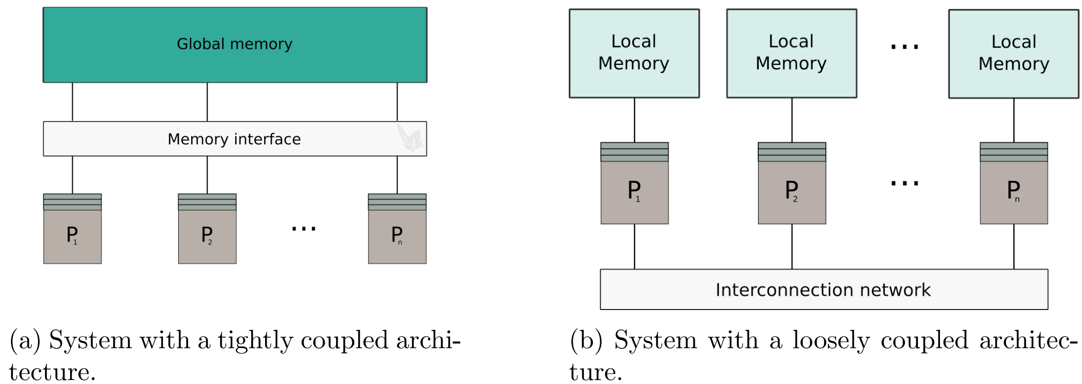
The employment of parallel computers nowadays is a well-established approach to scientific problems requiring high performance computing. There are some basic components that each of these parallel computing systems consists of such as the processing element(s), the memory, caches, registers and the some kind of interconnection network. However, no standard setup or structure for a parallel computer or parallel computing model exists. Instead, there are different ways of classifying parallel computers [Bar14].
A classical differentiation is the hardware classification by Flynn [Fly72]. Here computers are differentiated by the number of their instruction and data streams. He sets up four categories referring to whether a computer has a **S**ingle or **M**ultiple **I**nstruction streams and a **S**ingle or **M**ultiple **d**ata streams. In an MIMD[6] system for example there are multiple instruction and multiple data streams, so that each processor processes its individual set of instructions as well as its individual data stream. However, most of today's parallel architectures belong to the category of MIMD systems, reducing this classification's significance.

[6] multiple instruction, multiple data

Other classifications are based on the way memory is organized and accessed [RR10]. Thereby, the most commonly used classification is based on the memory architecture of the computing system, referring to the physical distribution of memory with regard to its processes. This classification of tightly and loosely coupled systems is described in the following Section 3.1.1. As today's architectures seldomly fit into one of these concepts, another classification type is introduced.
Hereby the view on memory is not based on its physical distribution but is based on the way that the memory appears to the user, preset by the programming model used. The section thereafter, Section 3.1.2, therefore deals with the classification of parallel systems from a programmer's point of view, differentiating between shared, distributed and partitioned global address space models.
Moreover, classifications may be based on the granularity of the architecture or the topology of the processors [ALO02]. Other, newer propositions for classifications have been made as well. Duncan, for example, made a proposal for a classification which excludes low-level parallelism and instead defines an informal taxonomy based on high-level principles [Dun90]. He suggests a new high-level taxonomy with the three categories of synchronous (such as vector or processor array architectures), MIMD and MIMD paradigms parallel architectures. However, so far these newer proposals did not receive as much attention.
Note that so far no comment was made on the type of processing elements used. Of course, there is the possibility of using compute resources other than the standard central processing unit (CPU). Already a lot of large-scale systems today include other processing units such as GPUs or FPGAs. However, this work focuses on more general computing systems solely involving standard CPUs.

3.1.1 Classification based on the Memory Architecture

Parallel computing systems may also be differentiated by their memory structure. There exist two broad categories: the *tightly coupled* and the *loosely coupled systems*. In a loosely coupled system the memory is physically distributed, usually by means of each processing unit having its own local memory, whereas a physically shared memory is the main characteristic of a tightly coupled system. This categorization is only concerned with the physical memory structure as depicted in Figure 3.1 and does not take into account any structural organization of the interconnection network (hypercubes etc.) or in what way the memory is to be accessed or communication occurs. An example for a loosely coupled system is a multiple-processor computer system, where several processors reside in a single computing system and share a common memory. A tightly coupled system would be represented by a cluster which is a collection of multiple standalone commodity computers with, for example, one processor each, connected via an interconnection network. In a cluster, these single computers are called "nodes". Note, that these two types of computer architectures are also often simply referred to as *shared* and *distributed memory systems*. However, this categorization is a rather limiting description, as today's architectures seldomly appear in this simple way: Following the continuous evolution of technology, computing systems today most often contain multi-core processors instead of single-core processors. These multi-core processors contain several processing units (cores) which all share the same physical memory and peripherals. In case of compute nodes of a distributed system with multi-core processors, the cores of one compute node may not be able to access memory on a different computing node, resulting in a distributed shared memory system (DSM). Computing systems of this mixed type are also called NUMA architectures re-



(a) System with a tightly coupled architecture.

(b) System with a loosely coupled architecture.

Figure 3.1: Two types of computer architectures with a tightly (left) and a loosely (right) coupled architecture, also referred to as shared and distributed memory systems.

ferring to the non-uniformally accessible memory modules (local vs. different remote) or ccNUMA in case of additional cache coherency. If the memory were uniformly accessible (as is the case in a UMA system where the memory is truly physically shared), all processes would have the same, uniform latency and bandwidth for accessing the memory. An example of such a ccNUMA architecture is shown in Figure 3.2.

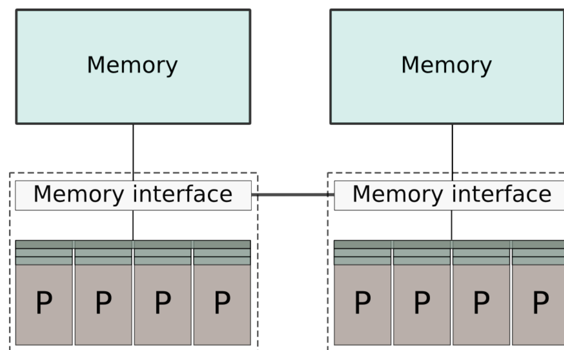


Figure 3.2: A ccNUMA system with two sockets, where processors on one socket share the L3 cache and the memory interfaces are connected via a coherent link [HW10].

A further case may be a computing system with a physically shared memory which is divided into memory modules each being assigned to a certain processor and thereby supporting a distributed memory model.
Then again, another computing system with an appropriate network logic may make distributed memory modules appear as one single address space to the user [HW10], mimicking a shared memory.
All of these cases are hard to be matched to the loosely or the tightly coupled architectures exclusively.

3.1.2 Classification based on Parallel Programming Models

As described in Section 3.1.1, the categorization of parallel computers solely based on their memory setup may be inadequate. Therefore, it may be preferable to categorize

these computer systems from the programmer's point of view, by comparing the view on memory the parallel programming models impose, instead of architectures.

A **shared memory programming model** is given when each processor has access to every existing part of the memory, no matter its location (excluding caches and registers). Data exchange between processors thereby occurs via the shared memory, e. g., via shared variables.
OpenMP ("Open Multi-Processing") [Ope13] is often thought of as the main representative for shared memory programming model. In OpenMP directives[7] are used to mark special parts of the code which are to be processed in parallel. At the beginning of such a section the thread running the program, called the master thread, forks a specified number of other threads, called slave threads, in order to share the workload between them. All of the created threads have access to the shared memory and may accomplish their thread-specific work on any data in the memory. At the end of this section the slave threads are put to sleep again and the master thread continues on through the code. Such a work distribution is called the fork-join model. Both at the beginning and the end of such a section, synchronization implicitly occurs. But explicit synchronization may be triggered by the user as well. However, scaling beyond a few dozen processors with OpenMP is difficult as the interconnection network is strained trying to provide fast access to global memory for each thread [RR10], especially for distributed shared memory. Furthermore, OpenMP does not provide any directives for exploiting locality [Ope13, CH08].
It should also be mentioned that OpenMP requires cache coherency, as remote data is cached and therefore needs to be explicitly updated when it is remotely changed. Therefore, NUMA architectures with an Infiniband network[8] may not be able to run OpenMP unless by special means. For example, "vSMP" aggregates multiple servers into one single virtual system and presents a NUMA SMP architecture to the guest O/S [STW+10] on which OpenMP may be run.

A **distributed memory programming model** on the other hand is characterized by each memory module being attributed as private and therefore being only accessible by the associated processor. In order to exchange information, processes have to send messages via the network, known as *message passing*.
The "Message Passing Interface" (MPI) [Mes12] is regarded as the main representative of the distributed programming model. MPI's first specification was established in 1994, with new versions continuously being published since then, together with several commercial and open-source implementations, varying in implementational details if permitted by the specification. The MPI standard has undergone quite some changes in its development over the years. Three major versions were published, namely MPI 1.0 [Mes94], MPI 2.2 [Mes09] and MPI 3.0 [Mes12]. These versions will be referred to as MPI-1, MPI-2 and MPI-3, for simplicity.
With MPI-1, communication was basically focused on two-sided communication calls. They require both the sending and the receiving process to post a matching communication call. Both calls come in two basic versions. First, they can be blocking, allowing the sending process only to return from its call when the data from the send buffer has been transferred and the send buffer itself is reusable again. Secondly, they can be non-blocking, where the communication procedure may return even though the communication has not

[7]Special commands for the compiler.
[8]Interconnect technology allowing for direct memory accesses [Pad11].

been completed, allowing for other computation to take place (hardware-dependent also in parallel) before posting a communication completion call to make the local buffer reusable again. Whether the data is transferred directly to the receiver's memory or first copied to a temporary system buffer is implementation-dependent or may depend on the underlying communication layer. In the second case the system buffer may have to store all messages from pending communication requests. Consequently, for too many processes and communication calls the system's resources may not be sufficient. Additionally, the synchronization coming alongside with the synchronous communication mode prevents implementations from making use of a possible overlap of communication and independent computation. MPI-2 then introduced one-sided communication calls, with the hope of tackling these problems. With it came parallel I/O and a dynamic process management. In MPI-3 collective operations and extensions to one-sided operations were included together with non-blocking versions of them. However, including all new functionalities of the MPI-3 standard into the MPI implementations takes some time. As of November 2014 for example several main implementations had not yet implemented all features [For15], by mid-2016 almost all MPI implementations had included them [For16]. Still most non-sophisticated applications do not yet make use of these functionalities.
Another approach to parallel computing is the **PGAS programming model**, relying on a **P**artioned but **G**lobal **A**ddress **S**pace for which first programming languages and APIs emerged in the mid-90s. It is designated for distributed memory systems but provides a virtually shared memory [RR10]. Thereby it allows for combining the benefits of a tightly coupled with a loosely coupled architecture, as it provides a globally addressable memory space but due to its distribution is still scalable. Often, shared memory systems may be used, too, although performance may decrease. As this thesis is mainly concerned with the PGAS model, a more thorough introduction to the PGAS concept and associated languages is given separately in the following section.

3.2 The PGAS Model

In this section an introduction to the PGAS concept is given and two well-known representatives of the PGAS-languages are presented, including their features and functionalities. A short overview of other existent PGAS languages and interfaces is given thereafter. The **P**artioned but **G**lobal **A**ddress **S**pace (PGAS) model is a parallel programming model based on distributed (loosely coupled) memory [Pad11]. To the user the memory appears as one globally addressable memory space. Additionally, each single part of this memory region has an affinity to one of the processors, leading to a quicker memory access for this process compared to remote accesses of others. The according setup of such an architecture is illustrated in Figure 3.3.
These three components "set of processors/nodes", "global partitioned memory" and "affinity of shared memory" is what a PGAS model consists of [Pad11].
There exist several languages, language extensions and APIs which build up upon these three components. However, although all of these are based on the PGAS model, they differ in several important points. Language-dependent, work assignment can be influenced by the user or data dependencies can be automatically resolved. Furthermore, each language may have different types of synchronization mechanisms and handle memory consistency differently. They may also have a different way of accessing global, remote data, from simple referencing of variables to providing detailed information about the memory location. Eventually, each process may have its own local memory to which access is

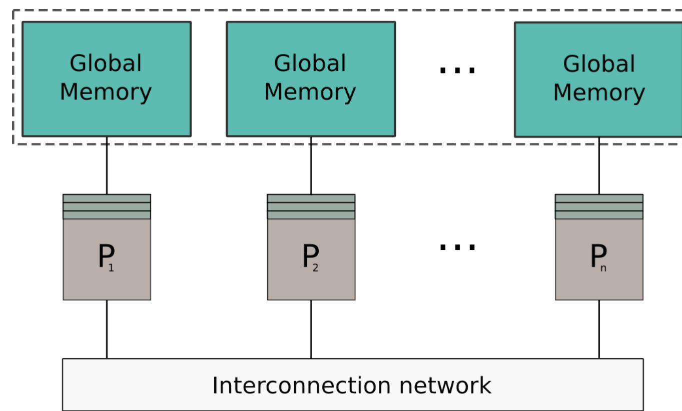


Figure 3.3: Architecture with local memory modules appearing as one global address space.

restricted solely to the local process.
For a better understanding of the fundamentals of the usage of a partitioned global address space, a short introduction of two PGAS-based language extensions and their notion of shared data is given in the following sections. This chapter then concludes with an overview of further PGAS-based languages and language extensions.

3.2.1 Coarray Fortran (CAF)

Coarray **F**ortran (also known as CAF) is a parallel programming model based on Fortran and is the first parallel programming model known to be added to a programming language [Pad11]. It started out in the 1990s under the name F^{--} as a syntactic extension of Fortran 95, intended for parallel computing. The extension was then renamed to CAF, extending Fortran 2003, until it was integrated into Fortran 2008 as a standard feature [Rei10]. Today, Fortran 2008 and therefore also CAF is developed and maintained by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) through their Joint Technical Committee 1 (JTC1).

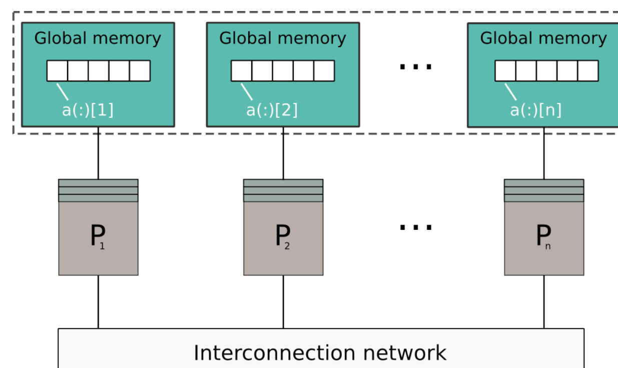


Figure 3.4: Coarray $a(:)[*]$ in CAF located on an arbitrary number n of images.

The coarray programming model is a PGAS model and follows the single-program multiple-data (SPMD) parallelization scheme. The program to be executed is replicated multiple

times and runs asynchronously on each process. The replications of the program are called *images* and can be identified by their unique *image index*. Each image has its own independent execution state as well as its own data objects, in- and outputs.
Syntactically CAF extends Fortran by adding an extra trailing subscript in square brackets to a data object. This addition indicates a remote memory element and the subscript, the *codimension*, specifies the remote image index. Thereby access to remote data can be achieved by simple references, omitting the need for explicit communication calls.
For example,

```
1 real :: a(5)[*]
```

declares a single precision floating point coarray a which appears as a local vector with 5 elements on each image. Figure 3.4 illustrates it's data distribution.
Remote accesses occur then as follows:

```
1 real :: a(5)[*]
2 real :: b(5)
3
4 b(:) = a(:)[2]
```

In line 4 of the above example, each image copies the data from vector a located on the second image into its local vector b. For the image with index 2 this is a simple, local reference. The codimension may also be multidimensional, creating a multidimensional layout for the data and their images.

```
1 real :: c(5)[2,*]
```

The above declaration of the shared array c would then create a local array of length five on each image, organizing the images in a grid with the first dimension being two and the second dimension depending on the total number of images executing the program (grid may remain incomplete).

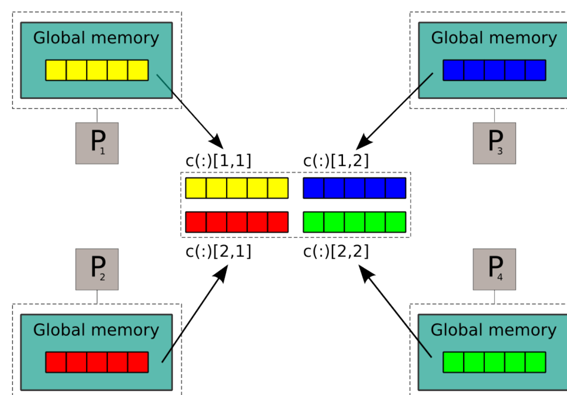


Figure 3.5: Data structure of coarray $c(5)[2,*]$ with a codimension of rank two for four images.

To ensure data availability synchronization statements can be included in between data accesses. These can be barriers combined with memory synchronizations which apply to all images together or only to designated images. In addition, CAF provides lock and unlock statements as well as critical sections which limit access to a piece of memory or code, respectively, to one image at a time.

Compilers supporting coarrays include the commercial Cray and Intel Compiler and the free OpenUH and G95 compilers [FBC+14]. However coarrays are often not yet or only partially supported by other Fortran compilers [CS14].
Additionally, scientists from Rice University in Houston, Texas developed Coarray Fortran 2.0 (CAF 2.0) [MCASJ09]. CAF 2.0 is a runtime library and is based on the coarray programming model but includes additional features.
More details on CAF's features are discussed in Section 4.3.

3.2.2 Unified Parallel C (UPC)

Unified **P**arallel **C**, or UPC for short, is a parallel extension to the C standard and is based on the PGAS programming model [UPC13]. Its first specification version was published in May 1999 [RR10] at the Institute for Defense Analyses Center for Computing Sciences. The next version to be published, version 1.0, was a cooperate effort of industry, government and academia in 2001. Further versions followed with the latest specification being version 1.3, which was disclosed in November 2013. Developing UPC was based on experience gained from "its distributed shared memory C compilers such as Split-C [KCD+93], AC and PCP" [CSC+05].
The general setup in UPC is as follows: First of all, instances of execution in UPC are called threads. Each thread then is assigned local memory which is divided into two parts. One part of it is declared to be private memory to which no other thread has access. The other part is the thread's portion of the shared memory space, to which this thread has an affinity to and which any other thread may directly access. Note, that 'shared memory' only for the context of this section refers to the partitioned, global address space, not a tightly coupled memory architecture.
One feature of UPC is that it provides simple statements for remote memory accesses. Data to be shared is simply declared as a `shared` type. Internally, the data is then distributed blockwise across all threads in a round robin fashion, if not declared otherwise. A shared variable is declared as follows:

```
1 shared <type> <variable>[<length of array>];
```

In the next example an array a of length $2*$`THREADS` and of double precision floating point type is declared. The type supplementation `shared` declares the location of the array to be within the global address space.

```
1 shared double a[2*THREADS];
```

As a shared array is distributed across all threads and the length of the array in this case corresponds to two times the number of threads existent, each thread would store two of its values in its local, but globally accessible memory space, as depicted in Figure 3.6. Each thread may then access each element of this shared array by simply calling the array with the index of the array, handling it as one continuous array:

```
1 shared double a[THREADS];
2 double d;
3
4 d = a[2];
```

Note that the thread does not need to know where the data is located exactly, but only needs to name the index within the array, which is two in the code example above.
Shared arrays may also be distributed in chunks of a given size, determined in squared brackets after the `shared` qualifier:

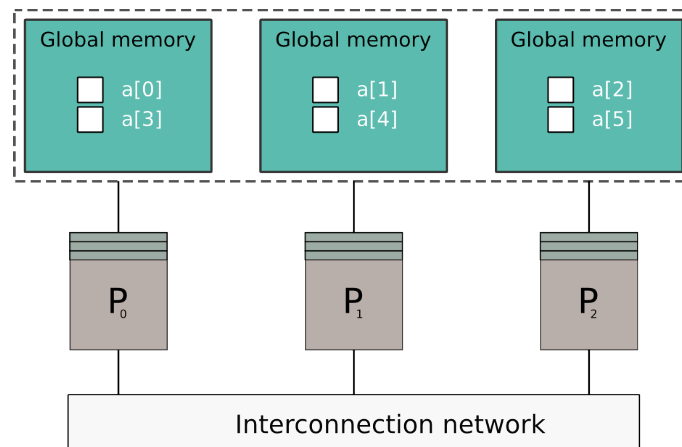


Figure 3.6: Round robin distribution of a shared array declared as `shared int a[6]` across 3 threads.

```
1 shared [2] double b[10];
```

Each thread holds the same amount of memory for this shared variable or array:

$$c * \lceil \lceil l/c \rceil / \texttt{THREADS} \rceil ,$$

where c is the chunk size, l the original global length of the array and `THREADS` the number of threads existent. The actual amount of memory held may include some padding if the array distribution does not add up to the number of threads available, as shown in Figure 3.7.

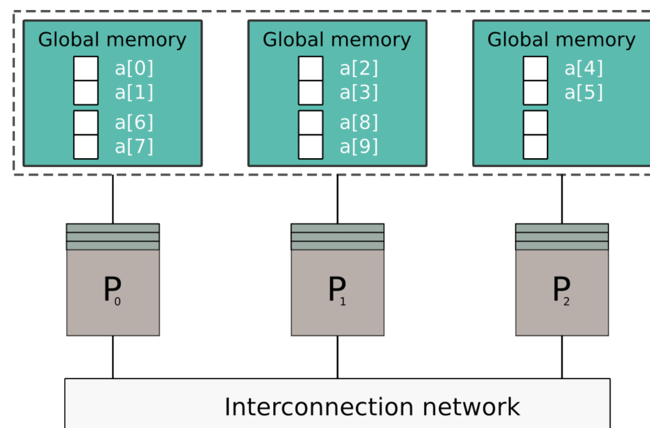


Figure 3.7: Distribution of a shared array declared as `shared [2] int b[10]` across 3 threads with a block size of 2.

UPC also provides various mechanisms for memory synchronization. Code sections or single memory accesses can be labelled as strict or relaxed memory consistency modes. Strict memory consistency means that statements within a code section marked as such have to be executed without re-ordering, while relaxed code sections allow for compiler

optimizations by changing the order of statements. Furthermore UPC provides locks to prevent multiple concurrent accesses from different threads and blocking barriers as well as split-barriers for general thread synchronization.
Other than the barrier, no collective operations are specified in the UPC Language Specification. However there is an additional specification for collective operations [Con03], as well as one for parallel I/O with UPC [EGCS+06].
Besides, there are several more UPC-specific features, such as shared pointers, explicit work distribution on each thread via a special for-loop depending on shared data, different shared memory allocation methods and copying methods that transfer blocks of data from private to shared, vice versa and shared to shared memory.
There are several implementations or compiler extensions for UPC. These include proprietary versions like HP UPC from Hewlett-Packard which is only conform to the UPC Language Specification v.1.2 [HPDC09], Cray UPC [BYEG06] and SGI UPC [SGI13], as well as open-source versions like Berkeley's UPC compiler [pro14], MuPC UPC from Michigan Tech University [ZSS06] and GNU UPC [FV13].
More details on UPC's features are discussed in Section 4.3.

3.2.3 Further PGAS-based languages

Similar to UPC and CAF, Titanium [YSP+98] is another PGAS-based programming model which is designed to extend an existing sequential high-level language, in this case Java, and builds upon a static parallelism model. It extends the Java language by immutable classes, synchronization and communication routines, local and global references and a possibility for the user to control memory management [YSP+98]. Titanium was developed at U.C. Berkeley and uses the Berkeley Titanium compiler which translates the Java dialect source-to-source directly to C with calls to the communication layer GASNet [Bon02].
Another course of development was that of the HPCS languages, named so for their origin. In 2002, the U.S. Defense Advanced Research Projects Agency (DARPA) offered funding for research leading towards novel languages in the HPC community as part of the program High Productivity Computing Systems (HPCS). Funding in the first phase went to Hewlett-Packard, SGI, Cray, IBM Corp. and Sun Microsystems, Inc., although only Fortress from SUN [ACH+08], Chapel from Cray [CCZ07] and X10 from IBM [STG+14] made it into the second funding phase with the latter two being the only to remain in the third phase. These three languages were developed with a different focus than UPC, CAF and Titanium. From the start they were not designed as extensions to existing sequential languages, but as autonomous languages, aiming at productivity, which is declared by DARPA as "a combination of performance, programmability, portability and robustness" [Wei07].
Fortress for example introduced a new syntax, mimicking mathematical notations in order to ease the use of Fortress for mathematical scientists. However in 2012 work on Fortress was discontinued[9].
X10 uses Java syntax, as well as its types and data structures, however it uses only a subset of Java, i. e., leaving out arrays and instead including parallelism supporting constructs.The concept of X10 is based on the asynchronous PGAS (APGAS) model [SAB+10] and includes the possibility to spawn a new thread which runs asynchronously to the others and can remotely fulfill specified tasks on remote data [STG+14]. Thereby, remote memory accesses are moved into remote tasks, avoiding the need of direct remote mem-

[9] `https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up`

ory accesses [Bre12]. Atomic operations enforce local node synchronization and therefore guarantee memory synchronization. X10's latest version 2.6.0 was published in June 2016.
Chapel, the Cascade High Productivity Language, builds on syntax from many previous languages, such as C, Fortran, Java, Modula and Ada [CCZ07, CCZ04]. Just as X10 it is based upon the APGAS model, including a dynamic multi-threaded execution model. Its main feature is the *locale*, an execution unit which combines computational capabilities and uniform memory access. Chapel's parallel features are high-level abstractions that were influenced by ZPL, HPF and Cray's multi-threaded extensions to C and Fortran [BAdA+08]. They include atomic sections, synchronization variables, parallel statements such as *forall* and the spawning of computations via section declarations. Additionally new types were introduced, e.g. *domains* which are index sets and build the basis to array definition and manipulation [Wei07]. Chapel's latest version 1.13.1 was published in June 2016.
All three HPCS languages have in common that they combine the PGAS model with dynamic multithreading.
There also exist several APIs which are based on or contain PGAS-motives. This thesis is based on one of them: GASPI, which will be described in detail in the following Chapter 4. In its last section (4.5), the other PGAS-APIs will be listed with some additional information.

Chapter 4

Global Address Space Programming Interface (GASPI)

This chapter gives an overall introduction to GASPI, the **G**lobal **A**ddress **S**pace **P**rogramming **I**nterface – an API based on the PGAS model.

It begins with a brief outline of the development of GASPI. Then an overview of the concepts and features of GASPI is given, illustrated by small code examples.

Subsequently, in order to gain a better understanding of these features, GASPI is compared to the two PGAS-languages which were introduced in Sections 3.2.1 and 3.2.2; UPC and CAF. They were chosen for comparison as they are among the most well-known representatives of the PGAS programming model and as the PGAS memory model is very clearly present with these two. The focus is hereby on setup, data exchange and synchronization mechanisms. The chapter concludes with a short overview of other programming languages and APIs which are based on one-sided communication.

4.1 GASPI's Origin

From 2005 on, the Fraunhofer Institute for Industrial Mathematics ITWM developed the Fraunhofer Virtual Machine (FVM), a communication library and runtime system influenced by the RDMA[10] model and targeting Infiniband architectures [ML09]. The FVM was superseded by GPI[11] from 2007 on, again ITWM software with the same goals as

[10] remote direct memory access

[11] **G**lobal Address Space **P**rogramming **I**nterface, not to be confused with GASPI

the FVM and so far an in-house development being employed in some industrial companies [ITW, Pfr10].
In 2011, the GASPI project was launched with the main goal to design a new PGAS API on the strength of the concepts and mechanisms of GPI and to enhance these. Furthermore, the project's goals were to develop an open-source, highly portable implementation with additional numerical libraries to extend the capabilities available with GASPI and finally, to provide a performance profiling interface and tool. Hence, GASPI is based on the partitioned global address space, its one-sided and optionally asynchronous communication and a fine-grained synchronization mechanism, the so-called *notifications*.
GASPI was a joint project of several research institutions: Dresden University of Technology, Fraunhofer Society, scapos AG, T-Systems Solutions for Research GmbH, German Aerospace Center (DLR), Jülich Research Centre, Karlsruhe Institute of Technology (till April 2013) , Heidelberg University (from May 2013 on) and the German Meteorological Service (DWD). For a period of three years, it was funded by the "Bundesministerium für Bildung und Forschung" within the funding program "IKT 2020 - Research for Innovations".
As a member of the Engineering Mathematics and Computing Lab (EMCL) at Karlsruhe Institute of Technology and later Heidelberg University the author was part of the GASPI project from the start, actively involved in the development of the API specification, as well as carrying out research for numerical libraries based on GASPI with a focus on dense linear algebra.
The further work of this thesis is based on the eminent outcomes of this project: the GASPI specification v.1.01 [Con13] and its open-source implementation GPI-2 v.1.0 [WJJ13].
Today the project partners and further interested parties established the GASPI Forum [12].
The Forum's members have been meeting once or twice a year since September 2014.

4.2 GASPI Specification and Functionalities

This section is mainly based on the GASPI specification v.1.0.1 published November 14th, 2014 [Con13]. Later versions did not significantly change the functionalities until lately. The latest version is version 17.1 and was published only recently on February 7th, 2017 [GAS17]. The main characteristics mentioned in this section have not significantly changed, but some functionalities were added. Whenever mentioned in this section, these additional functionalities are pointed out explicitly. Furthermore, this section is based on the author's own experience with GASPI, if not mentioned otherwise.
Within the code examples throughout this section some variables are continuously reused. They are therefore declared once in code block 4.1 and then used without further declaration in order to avoid repetition and to gain a better insight. Also, the complete syntax is kept in C-notation, as all applications in this thesis were implemented in C.

```
1 gaspi_rank_t myrank;        // the identification ID of the calling
2                             // process
3 gaspi_offset_t offset;      // a length of memory in bytes
4 gaspi_pointer_t seg_ptr;    // void pointer to the start of a segment
5 gaspi_queue_id_t queue_id;  // the ID of a local queue
6 gaspi_return_t ret;         // return value of a GASPI function
7 gaspi_segment_id_t seg_id;  // the ID of a segment
8 gaspi_size_t datasize;      // size of data in bytes
9 gaspi_timeout_t timeout;    // the timeout of a function in seconds
```

[12] http://www.gaspi.de/

```
10  gaspi_notification_id_t not_id; // the ID of a local notification
11  gaspi_notification_t not_val;    // the value of a notification
```

Code 4.1: Declarations of variables which will be used without further declaration throughout this section.

Furthermore, code blocks occurring later in this section may depend on previous code blocks. Therefore, code sections which also appear in preceding codes will not be reprinted but only mentioned in a comment. For example, the initialization of GASPI or the general infrastructure setup will not be repeated each time but is shown only once.

4.2.1 Process and Memory Setup

As mentioned before, GASPI's programming model is based on the PGAS model and allows for SPMD as well as MPMD applications. Its memory is split up into two regions known as globally accessible (RDMA) and private regions. They distinguish themselves by allowing remote access or by restricting the access solely to the local process and its threads. The according setup of the architecture with these two memory regions is illustrated in Figure 4.1. Although GASPI may also be run on shared memory systems,

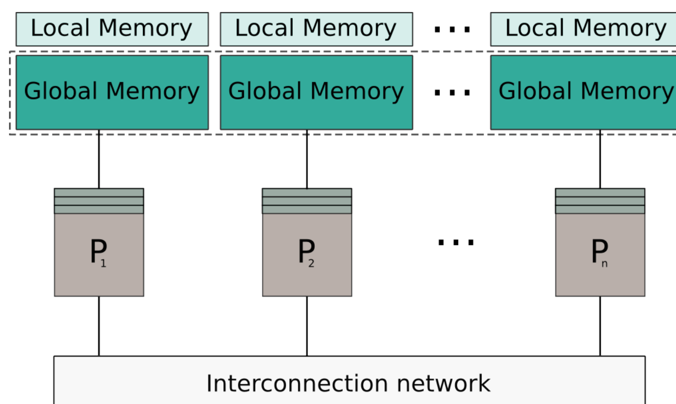


Figure 4.1: PGAS architecture with local memory divided into private and global (shared) regions.

its main focus is on distributed systems, leaving open the possibility to combine GASPI with a shared memory programming API for local parallelization. For example, GASPI may be run together with OpenMP [BSH14], ITWM's MCTP threads [Pfr10] or POSIX threads.

The term *process* will be used from now on to represent a processing element which may be part of a multiprocessor or a single computing unit. Note, that this does not specify whether a process has local memory or not. However, in order to work with remote data, each process must possess a portion of the partitioned global address space.

In GASPI each such process is uniquely identified by a process ID, called its GASPI *rank*. With this unique rank, code parts may be restricted to certain processes and thereby workload may be distributed based on the rank of the processes. Each process may inquire its own rank and the number of participating processes as shown in Code 4.2 which showcases the standard "Hello, World!" example in GASPI.

```
#include <iostream>
#include <GASPI.h>

int main() {
  gaspi_rank_t myrank;
  gaspi_number_t numprocs;

  // initiate GASPI
  gaspi_proc_init( timeout );

  gaspi_proc_rank( &myrank );
  gaspi_proc_num( &numprocs );

  printf("Hello, world! My rank is %d from %d\n", myrank, numprocs);

  gaspi_proc_term( timeout );
  return 0;
}
```

Code 4.2: "Hello, World!" in GASPI.

In this example `gaspi_proc_init` creates the GASPI processes, sets predefined environmental parameters or uses default ones and may set up the communication infrastructure which is required for data exchange. The initialization procedure is furthermore given a *timeout* parameter which will be explained in more detail in the following Section 4.2.2. The functions `gaspi_proc_rank()` and `gaspi_proc_num()` return the information on the rank of the calling process and the total number of GASPI processes running in this program. Processes with rank r will from now on be named Pr, e. g. P0 stands for the process with rank 0.

In GASPI the globally accessible memory regions mentioned before are called *segments*. Segments are normally allocated at the start of the program. However this is not necessary and they may be created dynamically later when needed. The user is responsible for defining their number and size (in Kb) which both may vary from process to process. As a result a process may host more than just one segment. These segments may also be mapped to a variety of memory types, which may differ in terms of bandwidth and latency of data accesses [Con13]. Thereby GASPI segments may also be allocated on SSDs[13], memory of GPUs[14], MICs[15] and NUMA[16] partitions. An example for the mapping of a GASPI segment to GPU memory is described in [Ode13].

Segments are allocated locally with a given, locally unique segment ID and then need to be individually registered at processes which are to have remote access. These two steps can also be performed in one go via `gaspi_segment_create()`, with automatic registration of the segment with all members of a given group of processes as shown in Code 4.3. From this point on the segment is globally accessible for all threads of the processes where it was registered at.

```
// create segment for each process
gaspi_alloc_t alloc_policy = GASPI_MEM_INITIALIZED;
gaspi_segment_create(seg_id
                    ,datalength * sizeof(int)
                    ,...
```

[13]solid state devices
[14]graphical processing units
[15]many integrated cores
[16]non-uniform memory access

```
6                        ,alloc_policy);
7
8  gaspi_segment_ptr(seg_id, &seg_ptr);
```

Code 4.3: Setup of segments (i. e., globally accessible memory) in GASPI.

In order to set up data within a segment, the *segment pointer*, a void pointer pointing to the start of the segment, can be acquired with a call to `gaspi_segment_ptr()`. With its help local store and load operations may occur, whereas remote access is possible via pre-defined read or write accesses.
The GASPI specification from 2017 [GAS17] contains the additional calls `gaspi_segment_bind()` and `gaspi_segment_use()` that allow the user to allocate memory himself and re-use that memory as a GASPI segment. This allows the user to position a segment wherever wanted.

4.2.2 General Concepts of GASPI Functions

GASPI also provides the possibility to *group* processes for the use in collective operations or in other functions involving several but not all processes at once. One such group is always predefined: `GASPI_GROUP_ALL` contains all GASPI processes available at startup per default. All subsets of this group can form a new group which is identified by its only locally valid group ID. The number of groups an application may create depends on the GASPI implementation used. GPI-2 has a limitation of 32 groups.
Next, GASPI also features *time-based blocking*: All non-local functions which may be *blocking* include a *timeout* parameter. A blocking function hereby indicates that a process may not return from such a function unless the operation has completed. The timeout parameter of a timeout-based function then specifies the time in milliseconds the function will remain in the operation, e. g. waiting on data from other processes, before it returns, unless the function has completed earlier already. If the timeout takes effect, the function immediately returns. However, this does not necessarily include its successful completion. A contrario, a process may return whilst the operation still continues on in the background or may simply have halted.
Depending whether the function is synchronous or asynchronous, this may be synonymous with the function making progress or not. Table 6.1 and 6.2 in the appendix give an overview on all GASPI procedures, noting whether they are synchronous or asynchronous, local or non-local and if they are time-based blocking. `gaspi_write()` for example is an asynchronous non-local time-based blocking procedure, while collective operations are time-based blocking and may be implemented as synchronous or asynchronous functions.
There are special timeout parameters: `GASPI_BLOCK` stands for a blocking call or a timeout of value infinity. `GASPI_TEST` denotes another special timeout, requiring a process to only remain inside the function as long as is necessary to complete all local work.
Note that in the GPI-2 implementation, if a process enters a read or write procedure, no remote data has to be waited on. The information about the data to be transferred is put into a local queue and the process may return (more information on the concept of queues in Section 4.2.4). Hence calling such a procedure with the timeout `GASPI_BLOCK` or `GASPI_TEST` may seem to be interchangeable. But when using GASPI together with a threading package, only one thread will accomplish the local operations. Therefore whilst with `gaspi_write(..., GASPI_TEST)` other threads calling the same function will return with `GASPI_TIMEOUT`, with `gaspi_write(..., GASPI_BLOCK)` all threads will wait

together for the one thread accomplishing the local work. Other implementations of the GASPI standard may of course differ, in this point.
Another intention of GASPI is to enable fault tolerant programming. Most GASPI functions have the same return type: `gaspi_return_t`. Function-dependent it returns either `GASPI_SUCCESS` if the operation was completed, `GASPI_TIMEOUT` if the function did not finish within the specified timeout or with `GASPI_ERROR` or a different user-defined error code stating that an error has occurred.

4.2.3 Synchronization

From the start, all GASPI processes run independently from each other per default. However, GASPI also provides several synchronization mechanisms.
As many other parallel programming languages, GASPI provides a *barrier*, namely `gaspi_barrier`. It takes two input parameters: a group and a timeout. A process of the specified group which reaches this point has to wait until all processes in the named group have reached the barrier. Alternatively the process will wait for timeout milliseconds and return immediately, whatever event occurs first. Note that this synchronization does not include a memory fence, i. e., no synchronization of memory is guaranteed and data transfer may even occur in the background across the barrier.
GASPI also features real global atomics. These are global, shared variables whose values can be fetched, increased or swapped and compared by all processes, but only by one process at a time. A process trying to access an atomic which is occupied by another process will wait until the atomic has been freed again. Atomics can be used to synchronize processes. For example when one process has written data it can increase the atomic by one. A master process sees that all data has been written when the value of the atomic corresponds to the number of processes participating. Otherwise they may be used as global shared variables.
A weak synchronization mechanism GASPI provides is the *notification*. Each process has several local notifications that the other processes may set to a new value. These are similar to global atomics in terms of their global setting. However they belong to a certain process and remote processes only have write access. So a process may notify another process by writing a certain value to a predetermined notification. The receiving side then can check on a single notification or a set of notifications whenever it is ready to.

```
1 // set notification ID 3 on rank 1
2 if( myrank == 0 )
3   gaspi_notify(remote_seg_id, 1, 3, not_val, queue_id, GASPI_BLOCK);
4   ...
5 // check on notifications 2,3,4,5
6 if( myrank == 1 )
7   gaspi_notify_waitsome(local_seg_id, 2, 4, &first_id, GASPI_BLOCK);
```

Code 4.4: Notification mechanism in GASPI.

In Example 4.4, P0 sets the notification with ID 3 on rank 1 to the value `not_val`. Process P1 checks on 4 notifications starting from 2 and returns the ID of the first notification to be non-zero.

4.2.4 One-sided Communication

The main building block of GASPI is its *one-sided* and *asynchronous* communication. With a one-sided communication mechanism one process may write to the globally ac-

cessible part of a remote process without any interaction of that process. This ability to access data on a remote process's memory is also called remote direct memory access (RDMA) and needs to be supported by the hardware.
Asynchronity specifies the ability of a function to continue making progress although the process already has returned from that function. In other words, progress of an asynchronous function may be achieved without involvement of the CPU of the calling process. Meanwhile the network system takes care of that task while the process itself may turn its attention to other tasks which are independent of the ongoing communication. Next the available communication mechanisms are described, an example of how their asynchronity may be usefully deployed is given in Section 4.2.7.
GASPI in principle provides two point-to-point (or global address space to global address space) communication or rather data transfer mechanisms: `gaspi_write()` and `gaspi_read()`. These DMA requests are directed to one of the local *queues* which stores all information on what type of request was made. Then the underlying network infrastructure takes care of the actual data transfer, allowing the process to return. To gain information on the local status of the request posted, the queue can be monitored. Either the length of the queue may be inquired or `gaspi_wait()` may be called on the queue in question. This forces the process to wait until the complete queue has been processed or the specified timeout applies. If `gaspi_wait()` returns successfully, the local data is reusable. Different data requests may be sent to different queues, thereby allowing the synchronization of a set of communication requests separately from others. All DMA requests posted to a queue will maintain the input order, so they are guaranteed to not overtake. DMA requests posted to different queues however are not ordered in their execution line, although fairness of transfers is to be guaranteed across queues by the implementation.
The information that is posted into the queue includes the source and the destination location of the data within the global address space, the size of the data, the queue ID and the timeout for this function, as shown in Code 4.5.

```
1  // write data of size ``datasize'' to remote rank
2  gaspi_write(seg_id_local
3             ,offset_local
4             ,rank_remote
5             ,seg_id_remote
6             ,offset_remote
7             ,datasize
8             ,queue_id
9             ,timeout);
10
11 // wait until local data is reusable
12 gaspi_wait(queue_id, timeout);
```

Code 4.5: Write and wait function in GASPI.

The location of data on the source or destination process consists of the following triplet:

$$(\text{segment_ID}, \text{offset}, \text{remote_rank}).$$

The segment ID has to be given, as each process may have multiple segments. In addition, the offset then specifies the exact location within the segment by giving the byte offset from the start address of the segment. Last, the remote rank determines the process where the data is written to/read from. As can be perceived from their specifications, both data locations, remote and local, need to be within the global address space.

It is important to know that before posting a communication request it should be checked that the queue has space enough for another communication request, as the length of a queue is limited. This is done by querying the queue size and flushing the queue if needed. Otherwise, if the queue is already full, the behavior of the program is undefined. The latest GASPI specification [GAS17] defines a new return code for a communication request: `GASPI_QUEUE_FULL`. This code is returned if the queue is already full and the communication call could not be posted. Checking for this code avoids waiting on every communication request to ensure that it can be posted. This functionality was not yet considered in the implementation in this work.
As mentioned, these simple `gaspi_write` and `gaspi_read` calls only involve contribution of the caller's side. However sometimes at least some kind of notification is wanted on the remote process's side, e. g. on the receiving side of a `gaspi_write`. Then a `gaspi_notify` can be triggered right after a `gaspi_write` which writes a notification into an internal notification buffer on the remote process. If the notification request is put into the same queue as the communication call, it will be executed only after the writing request. This procedure is enclosed in a shortcut as `gaspi_write_notify` which posts the communication request and an associated notification to the same queue as is demonstrated in Code 4.6.

```
1  // write data of size ``datasize'' to remote rank
2  // and additionally notify the rank afterwards
3  gaspi_write_notify(seg_id_local
4                    ,offset_local
5                    ,rank_remote
6                    ,seg_id_remote
7                    ,offset_remote
8                    ,datasize
9                    ,not_id
10                   ,not_val
11                   ,queue_id
12                   ,timeout);
```

Code 4.6: Combination of write and notify function in GASPI.

The following Code 4.7 is an example showcasing the functions of GASPI the sections have covered so far. It was shortened for better readability and hence excludes self-explanatory variable declarations or declarations already made in Code 4.1 and some other necessities. The complete example can be found in the appendix, Code 6.1.
The code is written for exactly two GASPI processes. The setup is as follows: two vectors are divided among two processes so that each process holds half of the vector. More precisely the first half of both vectors reside on the process P0 and the second half can be found in the memory of the process P1. The goal is to calculate the dot product of these two vectors. Note that the names of pointers to global address space are preceded with `gas_` whereas pointers to private memory are preceded by `loc_`.

```
1  #include <GASPI.h>
2
3  int main(int argc, char *argv[]) {
4
5    const int vlength = 8;
6    int loc_a[10] = {1,2,3,4,5,6,7,8};
7    int loc_b[10] = {2,2,2,2,2,2,2,2};
8    gaspi_pointer_t seg_ptr;
9    int *gas_a, *gas_b, *gas_result;
10
```

```
11    // initiate GASPI
12    gaspi_proc_init(GASPI_BLOCK);
13    gaspi_proc_rank(&myrank);
14    gaspi_proc_num(&nprocs);
15    int ll = vlength/nprocs; // local vector length
16    gaspi_size_t seg_length = (2*ll+2) * sizeof(int);
17
18    // create segment for each process, get pointer
19    gaspi_segment_create(seg_id
20                        ,seg_length
21                        ,GASPI_GROUP_ALL
22                        ,GASPI_BLOCK
23                        ,GASPI_MEM_INITIALIZED
24                        );
25    gaspi_segment_ptr(seg_id, &seg_ptr);
26
27    // setup data in global memory
28    // P0:  1,2,3,4,2,2,2,2 (a_0,b_0)
29    // P1:  5,6,7,8,2,2,2,2 (a_1,b_1)
30    gas_a      = (int*) seg_ptr;
31    gas_b      = gas_a + ll;
32    gas_result = gas_b + ll;     //offset: 2 * ll * sizeof(int)
33
34    for(i=0;i<ll;i++) {
35      gas_a[i] = loc_a[i+myrank*ll];
36      gas_b[i] = loc_b[i+myrank*ll];
37    }
38
39    // calculate intermediate result, store in gas_result
40    for( i=0; i<ll;i++ )
41      *gas_result += gas_a[i] * gas_b[i];
42
43    // get queue data
44    gaspi_queue_size_max(&queue_max);
45    gaspi_queue_size(queue_id, &queue_size);
46
47    // check if queue full
48    if(queue_size > queue_max - 1)
49      gaspi_wait(queue_id, GASPI_BLOCK);
50
51    // set offsets to position of gas_result[0] and gas_result[1]
52    gaspi_offset_t loc_offset = 2 * ll * sizeof(int);
53    gaspi_offset_t rem_offset = loc_offset + sizeof(int);
54
55    // write local result to other process
56    gaspi_write_notify( seg_id, loc_offset, (myrank+1)%nprocs
57                      , seg_id, rem_offset, sizeof(int)
58                      , not_id, not_val
59                      , queue_id, GASPI_BLOCK);
60
61    // wait on intermediate result from other process
62    gaspi_notify_waitsome(seg_id, not_id, 1, &not_id, GASPI_BLOCK);
63
64    // calculate global result
65    dotproduct = gas_result[0] + gas_result[1];
66    printf("(%d) dotproduct = %d\n", myrank, dotproduct);
67
68    // wait to ensure data in gas_result[0] is free before exiting
69    gaspi_wait(queue_id, GASPI_BLOCK);
70
```

```
71    gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK);
72    gaspi_proc_term(GASPI_BLOCK);
73    return 0;
74  }
```

Code 4.7: Calculation of dot product with gaspi_write_notify for the exchange of intermediate results. Short version of Code 6.1.

Here, in l. 34, the vectors are written into global memory first, although this is not necessarily needed for this code. However it is a likely case in larger parallel applications. Each process then first calculates its local intermediate result (l. 40) and then writes the result via `gaspi_write_notify` to the other process (l.56). After the successful reception of the second intermediate result (l.62), the two intermediate results are summed up and so each process has the global result. To ensure that the local part of the `gaspi_write` operation has completed before exiting the program, the queue is waited on in l.69. Of course, this case only demonstrates the simple usage of these functions and does not make any real use of the one-sided and asynchronous communication pattern. However it gives the user a feeling for the complexity and high level of detailed information that the user has to provide: from the structure how the data is distributed, to the size of the segments and to the determination of the exact location of the data.

4.2.5 Collectives

As already mentioned when explaining GASPI's group concept, GASPI also provides collectives of which it is not determined whether they are synchronous or not. Collectives are operations which are called by several processes simultaneously (the process members of a specified group) and typically involve global cohesive computation and therefore communication. The simplest one, introduced in Section 4.2.3, is `gaspi_barrier`, the only collective not involving any concrete data exchange. Further collectives that GASPI provides are all kinds of reduction operations. GASPI pre-defines different reduction operations like the determination of a minimum or maximum over some data set or the summation of distributed data. This list of reduction operations can be extended, as GASPI offers the possibility to create a user-defined allreduce-operation.
The following Code 4.8 shows how a collective may be employed to solve the calculation of the dotproduct which was implemented with `gaspi_write` operations before in Code 4.7. Instead of only two processes, here four GASPI processes come into play. The vectors are extended to hold sixteen elements each, so each process holds four elements per vector. The code excludes parts already shown in Code 4.7. The complete and executable version can be found in the appendix: Code 6.2.

```
1   #include <GASPI.h>
2
3   int main(int argc, char *argv[]) {
4
5     // declaration of other variables
6
7     const int vlength = 16;
8     int loc_a[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
9     int loc_b[16] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
10    int *gas_a, *gas_b;
11    gaspi_pointer_t buff_send, buff_recv;
12
13    // initiate GASPI
```

```
14     // create segment for each process, get pointer
15
16     // setup data in global memory
17     // P0:  1, 2, 3, 4,2,2,2,2 (a_0,b_0)
18     // P1:  5, 6, 7, 8,2,2,2,2 (a_1,b_1)
19     // P2:  9,10,11,12,2,2,2,2 (a_2,b_2)
20     // P3: 13,14,15,16,2,2,2,2 (a_3,b_3)
21     ll = vlength/nprocs;
22     gas_a = (int*) seg_ptr;
23     gas_b = gas_a + ll;
24
25     for(i=0;i<ll;i++) {
26       gas_a[i] = loc_a[i+myrank*ll];
27       gas_b[i] = loc_b[i+myrank*ll];    //offset: 2 * ll * sizeof(int)
28   }
29
30   // setup of allreduce buffers in global memory
31   buff_send = seg_ptr + 2*ll*sizeof(int);
32   buff_recv = buff_send + sizeof(int);
33
34   // local calculations, store intermediate result in buff_send
35   for( i=0; i<ll;i++ )
36     *((int*) buff_send) += gas_a[i] * gas_b[i];
37
38   // calculate global result via allreduce, is stored in buff_recv
39   gaspi_allreduce( buff_send, buff_recv, 1, GASPI_OP_SUM
40                  , GASPI_TYPE_INT, GASPI_GROUP_ALL, GASPI_BLOCK);
41
42   dotproduct = *( (int*) buff_recv );
43   printf("(%d) dotproduct = %d\n", myrank, dotproduct);
44
45   // barrier and GASPI termination
46   return 0;
47 }
```

Code 4.8: Calculation of dotproduct with the help of GASPI's allreduce function. Short version of Code 6.2.

Here, while the intermediate result is again stored in globally accessible memory, the intermediate result does not occur by write operations as in Code 4.7. Instead, in l.39 the reduction function `gaspi_allreduce()` with the predefined operation `GASPI_OP_SUM` is called to sum up the intermediate results on all processes.
In general, the GASPI specification only provides reduction operations and no gather, scatter or broadcast operations. However, a broadcast algorithm is under development at T-Systems Research[17] but has not been officially published or included in GPI-2, yet.

4.2.6 Further GASPI Features

GASPI provides further features which will be touched only briefly.
For one thing, the GASPI specification also includes passive communication primitives. These are communication calls with a two-sided semantic similar to `gaspi_write` but without the receiver knowing beforehand who the sender will be. They may be useful for example, if many processes have to send data to one process and some kind of synchronization is needed alongside. The passiveness comes into play as both communication calls

[17] http://www.t-systems-sfr.com/e/downloads/2014/vortraege/6end.pdf

are to avoid busy-waiting and thereby spending almost no CPU time at allow giving time for computations.
In order to detect processes that have failed, GASPI includes a state vector. It is a local construct and contains the states of all other processes in terms of whether they are still healthy or not. The health of a process indicates whether communication with it is possible or not. The state vector is first set after `gaspi_init` and then after each non-local operation depending on who was in involved in the operation.
In case of a node failure, the state of that process will return as not healthy. In this case a new GASPI process can be started on a spare host which then take on the rank of the failed process. However, communication infrastructure and groups have to be rebuilt wherever the failed process was involved.
Furthermore the GASPI specification includes a profiling interface. On the one hand it offers the possibility to collect basic profiling data based on implementation-defined statistics counters. The GASPI profiling interface thereby provides functions for retrieving information on these counters. On the other hand it offers an interface so that an event tracing tool may communicate with the application, e. g. to inspect or even intercept GASPI functions. For the GPI-2 implementation, Vampir, the Performance Analysis Tool of ZIH, TU Dresden, can be used.
GASPI also offers the possibility to jointly use MPI and GASPI in one code. As a result of this interoperability, it is possible to not having to completely port the existing MPI implementation but to only rewrite compute-intensive code parts to work with GASPI. However it should be taken care that MPI and GASPI communication do not cross.
Finally, important to the Fortran community, GASPI also offers language bindings for Fortran 2003 code. A Python interface is existent, as well [BSH14].

4.2.7 Overlap of Communication and Computation

The main aspect not clearly stated so far, is the use of the one-sided and asynchronous communication to overlap the data transfer with computations independent of it. As soon as a DMA call has been started the program may turn to other computational tasks which do not involve the data involved in the communication process. Such a dummy example is shown in Code 4.9.

```
1  if( myrank == 0 )
2    gaspi_notify(seg_id, 1, not_id, not_val, queue_id, GASPI_BLOCK);
3
4  if( myrank == 1 ) {
5    ret = gaspi_notify_waitsome(seg_id, not_start, num, &first_id,
         GASPI_TEST);
6
7    while(work==TRUE && ret==GASPI_TIMEOUT) {
8
9      do_other_work();    // sets work to TRUE when all work is done
10     ret = gaspi_notify_waitsome(seg_id, not_start, num, &first_id,
           GASPI_TEST);
11   }
12
13   if (ret == GASPI_TIMEOUT)
14     gaspi_notify_waitsome(seg_id, not_start, num, &first_id,
           GASPI_BLOCK);
15 }
```

Code 4.9: Overlap of communication and computation in GASPI.

Here data may have been transferred (either before the `gaspi_notify` in line 2 or with a `gaspi_write_notify` instead of the `gaspi_notify`). The process P0 wants to notify P1 that this transfer has been completed. P1 first checks that the notification has not yet returned successfully or with an error and therefore needs further attention. It then continues on other work, computational tasks that do not involve the memory space which is used for the DMA request and in between periodically checks if the notification has arrived meanwhile. Of course the periodical checking on the notification in line 10 is not necessary and can be omitted.
This example is only a theoretical construct, but it may be of use in real algorithms, too.

4.3 Comparing Features: GASPI vs. CAF & UPC

The focus of this section is to better understand in what way GASPI distinguishes itself from other PGAS implementations and to point out what similarities exist. For this, the feature-set of GASPI [Con13] is compared to two well established PGAS realizations, namely Coarray Fortran (CAF) [Rei10] and Unified Parallel C (UPC) [UPC13] which already were briefly introduced in Chapter 3.2.2 and 3.2.1. In this comparison we continue to use the GASPI naming convention and e. g. call both CAF images and UPC threads *processes*, as well. This section is based on the author's previous work in the publication [BSH14].

Programming Model and Setup – First of all, both, CAF and UPC, use the Single Program Multiple Data (SPMD) model of computation, whilst GASPI also allows for Multiple Program Multiple Data (MPMD) style programs. All three follow the PGAS model, CAF and UPC as extensions to a high level languages and GASPI as a PGAS API available with two interfaces.

View of Global Memory – Based on the PGAS model, each process in all three programming paradigms has a partition of the global address space to which it has a logical affinity to and additionally a private memory space to which only the local process has access to. Note, that in GASPI the processes need not necessarily have such a partition, although without it no data exchange may occur with this process, reducing its usability. It is important to note, that the view of the global memory is quite differently with GASPI and CAF/UPC. In CAF/UPC, the user has no clear notion of the exact location or shape of the global memory. It is only important to know the way that global data is defined. In GASPI on the other hand, the view of the memory is much more clearly that of separated memory which is globally accessible but it is not given to the user as the notion of a shared memory with shared variables.

Allocation of Global Memory – The allocation of global memory in CAF occurs solely via the declaration of coarrays which may be allocated dynamically. These coarrays are thought of to be within the local portion of the global memory. However, as mentioned, beforehand no information is given as to where exactly the coarrays are put within that memory. Similarly, shared variables in UPC are created without any prior knowledge of their exact position. But additionally, UPC allows for dynamic allocation of shared memory space without a binding to a specific variable. This shared memory space can then be accessed via shared pointers. GASPI on the other hand completely abstains from the notion of shared variables and provides dynamically allocatable and globally addressable

memory spaces, only. Furthermore, in contrast to CAF, UPC and GASPI do not necessitate a process and its memory to be located on the same physical node. If provided, a GASPI implementation may even allow for the inclusion of non-standard memory, e. g. the allocation of memory on a GPU, which none of the others allow. However, the specification requires only a default allocation policy.

Declaration and Access to Global Data – Due to the different setup of the global memory and in addition because of the different view of global data, the way global data, i. e., variables, arrays or matrices, are set up and distributed is different.

First of all, UPC provides a shared variables namespace. It thereby provides the easiest way to access global data by simply requiring the global index of the global object which is to be accessed. Here, the global arrays are distributed in a cyclic or block-cyclic fashion among all processes using a round-robin pattern. The user can then control the block size of the distribution. However, it is not possible to distribute a matrix, being stored element-wise, by using a two-dimensional block-cyclic distribution. Additionally, global arrays are always shared amongst all processes, not allowing for a global object to be shared only by a subgroups of processes.

In CAF, global data exists in terms of coarrays, which are declared globally. They are not distributed the same way as in UPC, although all processes have their share. Instead the (standard) view is rather that each process holds the very same variable or an array of exactly the same size. This means that a distributed array will have to divide up evenly among all processes or is filled in with zeros in the end. To access a coarray on another process the local index and the process ID (or co-dimension in this case) have to be named explicitly. The mapping of the global index to these values is the task of the user. Thereby access to these local array parts is still relatively easy but more information is required than with UPC. In the special case that non-evenly distributed arrays are required, access is more complicated. This is because this only may be accomplished by setting up a coarray containing pointers as components which point to arrays of different sizes.

But for all that, GASPI requires even more information about the location of the required global data. In GASPI no notion of shared or global variables exists. Instead global memory is simply viewed as a memory space on each process into which the user may place whatever data necessary. Therefore a global array which is distributed amongst GASPI processes may be placed in very different locations on each process. Only the user has to know exactly where the data is located. For this, first of all, the owning process of the required data has to be named, just as in CAF. Additionally, in GASPI the user has to provide the exact location of the data in terms of the offset to the start of the memory segment. This means more complexity in programming for the user, but also opens up opportunities for a more direct data/memory management, including the usage of any data distribution, e. g. a two-dimensional block-cyclic distribution.

Communication Mechanisms – CAF and UPC do not require any explicit communication mechanisms as data access occurs directly via the global variables. In order to enable access to global but remote data, GASPI on the other hand provides explicit communication calls. Thereby data is transferred from the remote to the local memory location (both in the globally addressable section) or vice versa. This entails an extra communication call but also provides the possibility of overlapping this data transfer with other computation which is independent of it. If wanted these accesses can be extended to include leaving a notification on the remote process's side when the access has been accomplished. With

this mechanism of remote completion the remote process asynchronously gains knowledge of the completion. Additionally GASPI provides passive communication calls which have a two-sided semantic (send and receive), allowing for communication patterns in which the receiver must not know the identity of the sender.

Synchronization Mechanisms – A synchronization mechanism supported by all three languages is the barrier which forces all processes to synchronize. CAF includes a kind of point-to-point barrier which is only relevant for the processes named in the call. This barrier does not require all participating processes to wait for each other simultaneously. Instead the calling process synchronizes with each of the processes named one at a time. GASPI on the other hand also offers barriers for subgroups of processes, avoiding unnecessary waiting of processes not truly involved in the synchronization. In CAF and UPC the barrier also ensures a consistent view on memory, i. e., all previously issued writes are guaranteed to be completed and all local copies of global data are marked as invalid or updated. A barrier in GASPI has no influence on memory consistency.

Another specialty concerning the synchronization of memory ought to be mentioned. As described before, in UPC shared data may be accessed directly by naming the variable or the array and its index. In what way these accesses are accomplished can be controlled by the user by defining the shared object in two ways: either as strict or relaxed objects. Accesses to a strict object essentially follow sequential consistency known from shared memory programming, while relaxed accesses follow relaxed consistency and therefore can be reordered freely by the compiler or the run time system. Strict memory accesses also serve as a memory fence ensuring that all previous modifications of shared data is visible to other processes and modifications of other processes are visible to the calling processes. If not declared otherwise, access to a shared object is relaxed by default. UPC also features an explicit fence. In CAF, optimization by the compiler may occur, but only in-between explicit synchronizations. In GASPI, the wait on a queue takes over the roll of a memory fence, as it guarantees that after the wait all operations within that queue are completed. However, as GASPI allows for multiple queues, it is possible to apply a fence only on a set of memory regions. In general, memory consistency in GASPI is managed locally by waiting on a queue and remotely via the wait-on-notification functionality which can be used for an overlap of communication and computation, as previously stated. But these wait and notify procedures may also be used for basic point-to-point synchronization. UPC also features notification and wait calls but here their purpose is to frame a synchronization phase for all processes and not only point-to-point. More precisely, no process may exit from the wait call unless all processes have executed the notification call.

Further tools for the synchronization of processes or events are the locking and unlocking functions featured by CAF and UPC and GASPI's atomics. Other than that, both UPC and GASPI have further collectives aside from the barrier, namely reductions which may be synchronous or asynchronous. UPC provides these and further collective operations such as broadcast, scatter and gather, in an extra specification [Con03]. In GASPI the collectives also support subgroups of processes, in contrast to UPC where the collectives always apply to all processes together. CAF does not supply any collectives other than the barrier.

Fault Tolerance – Neither CAF nor UPC feature the fault tolerance of GASPI which, together with the flexibility in the setup of groups of processes for synchronization purposes or for applying collectives, enables a GASPI program to make up for failing processes and

re-organizing itself. In contrast, in CAF the number of processes, equal to the number of program replications, is fixed at startup and accordingly the number of memory partitions is fixed, too. UPC also supplies a dynamic process environment. Moreover, GASPI provides another feature allowing for fault tolerant code: procedures acting on remote data (non-local functions) include a timeout. Among other things, this enables the user to check on the correctness of such a function. Both UPC and CAF do not feature any timeouts in procedures as GASPI does.

Compatibility & Interoperability – Regarding more general language information for Fortran, according to the MPI-3 specification [Mes12], MPI is only compatible "with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008". The Coarray Fortran 2.0 version, again from Rice University, on the other hand is compatible with MPI, as shown in [YBMCB14]. UPC on the other side is of course compatible with C, but not C++ or other languages. However, it allows for calls to highly optimized serial code written in other languages. Then again the UPC implementation of Rice University allows mixing of C, C++, Fortran and MPI with some limitations [18]. Finally, GASPI may be used together with C, C++, Fortran and Python and is interoperable with MPI.

Summary – Altogether CAF's shared data notation seems to come in handy if one is already familiar with Fortran, however CAF does lack the fine-tuned synchronization mechanisms GASPI and UPC provide. Comparing UPC and GASPI one immediately notices the easier way of accessing shared data in UPC but also the greater freedom in the exact placement of data in GASPI. Additionally the fault tolerance of GASPI may provide better futures prospects especially regarding the application on large clusters. Furthermore, GASPI provides a higher degree of freedom with regard to the ordering of memory accesses or synchronization mechanisms, however further research is required to identify if this provides any real world benefits.

4.4 Related Work with GASPI

This section is concerned with research carried out in conjunction with GASPI, its reference implementation GPI-2 or its predecessor GPI (named GPI-1 for a better distinguishability). Note that GPI-1 does not implement all of GASPI's features, e. g. notifications are not provided. However, the main concept of one-sided and asynchronous communication is still present in GPI-1.
In [Pfr10], on the basis of microbenchmarks, Pfreundt compares the latency and bandwidth of GPI-1 to MPI. Furthermore implementations of a 2D fast Fourier transform are evaluated, showing GPI-1 to be faster than MPI.
In [MLAP11], Machado et al. study a dynamic load balancing problem of irregular applications, based on the traversing of a large tree. For the implementation GPI-1 together with a threading package is utilized leading to a factor of 2.5 more tree nodes being processed per second than with MPI.
Simmendinger et al. [SJML11] rewrote an unstructured solver for computational fluid dynamics, called TAU. An increase in performance of a hybrid OpenMP/GPI-1 implementation was shown compared to the original two-sided MPI implementation and a hybrid OpenMP/MPI version. Furthermore both hybrid versions were shown to have an almost linear weak scaling.

[18] `http://upc.lbl.gov/docs/user/interoperability.shtml`

The performance of a 4D nearest neighbor stencil operator implemented with GPI-1 is evaluated by Grünewald [Grü12], especially in comparison to an MPI implementation which uses two-sided, non-blocking communication calls. Additionally, the overlap efficiency for an algorithm is defined and discussed. For GPI-1 it is found to be perfect under certain assumptions, e. g. requiring the total time needed for communication to be smaller than the computational time. The GPI-1 implementations achieve speedups of up to 20-30 per cent compared to MPI.

Shahzad et al. examine the implementations of a spare matrix-vector multiplication and a fluid flow solver based on Lattice Boltzmann based on GPI-1 vs. non-blocking MPI with and without the support of the APSM[19] library for the latter [Sha13]. Implementations with GPI show no significant advantages compared to the MPI implementations for the former application and a weaker strong scaling of performance when increasing nodes. For the latter application the authors considered the weak scaling and found GPI to outperform MPI.

In [WJJ13] Grünewald and Simmendinger introduce most of GASPI's features and mention its open-source implementation GPI-2.0.

A formal model for SIMD programs with PGAS APIs is defined in [CDMM13]. However, the model only includes the main features of the APIs and is thereby reduced to asynchronous data transfers.

In [Ode13] Oden performs microbenchmarks on a 2 node system containing GPUs comparing GPI-2 implementations to MPI. For small messages the ping-pong benchmark with GPI-2 is found to be up to 3 times faster compared to MPI, whilst for large messages the timing is about the same. In terms of latency and bandwidth the benchmarks show about the same results, whilst in terms of smaller messages achieving better results.

Oden also performed the measurement of the power consumption resulting from performance bottlenecks such as the data transfer between GPUs [OKF14]. Hereby a hybrid CPU/GPU system, where the communication is controlled by the CPU using MPI or GPI-2, is compared to a version where the communication is GPU-controlled. It is shown that the performance per Watt increases up to 10 per cent although the overall performance decreases.

The scalability of a hybrid GPI-2 and C++11 threads implementation of a TSQR algorithm which solves the least squares problem is studied by Kumar in [Kum14].

In [BSH14], Breitbart et al. present microbenchmarks on which basis the performance of multithreaded communication for small messages with GPI-2 is found to be up to an order of magnitude faster than a comparable implementation with MPI. Furthermore the feature set of CAF, UPC and GASPI is compared and a theoretical analysis of an implementation basis for a matrix-matrix multiplication is given. The author of this thesis is co-author of this publication.

Not a specific application but a software development platform is presented by Rotaru in [RRP14] which is a MapReduce implementation dealing with data intensive computations.

Stoyanov and Pfreundt study different hybrid GPI-2 implementations of a sparse matrix-vector multiplication. They also compare hybrid GPI-2 implementations of the Conjugate Gradients method and a Jacobi-preconditioned Richardson method to the according PETSc solvers, finding the GPI-based versions to show good scaling and a better performance than the PETSc methods.

[19] Asynchronous Progress Support for MPI – a library allowing for asynchronous communication when using MPI.

In [SRG15], Simmendinger et al. give an extensive overview of the GASPI API specification.

4.5 Further One-sided Communication APIs

Looking further than purely PGAS-based programming models, GASPI is not the only parallel library interface that mainly relies on or includes one-sided communication. Other interfaces include the Global Arrays Toolkit [NPT+06, NT02], MPI since version 2.2 [Mes09] and OpenSHMEM [Ope15]. Just as GASPI, UPC, CAF and others these interface/libraries themselves rely on low-level libraries or networking layers, such as ARMCI, GASNet, OpenFabrics Verbs, Myricom's MX which provide the routines for the communication calls such as RMA operations or atomics and other relevant operations for diverse networks.
Global Arrays is developed by Pacific Northwest National Laboratory with its latest release from August 2016. It consists of many libraries, including the RMA communication library ARMCI and IO libraries originating from computational chemistry, and an interface for a shared memory programming style on distributed systems.
OpenSHMEM was created as an effort to standardize diverse existent, non-conform
SHMEM libraries which were developed by several companies such as SGI, Quadrics, HP, IBM, Mellanox, QLogic and the University of Florida [Ope15]. OpenSHMEM also is a PGAS library with a notion of shared data objects (called symmetric in OpenSHMEM), similar to the idea of coarrays in CAF. Hence it also requires each process to allocate a symmetric data object with the same name, type, size and offset. Further, it provides communication and synchronization operations on private and remote data objects and also supports collectives, atomics and locks. As of now there is an active OpenSHMEM community in the process of further development of OpenSHMEM.
MPI, as described in Chapter 3.1.2 is a major message passing library [Mes12]. Due to its high dissemination, more details will be given about the one-sided communication calls of MPI. As mentioned, its most popular communication methods are two-sided, which means that the data transfer occurs through cooperative operations on both the origin and the target process. These two-sided methods appear either as blocking or non-blocking call with an additional completion call. Additionally, MPI provides one-sided communication methods. These are based on the concept of windows which are individually sized shared memory segments allocated by a group of processes, either collectively or may be dynamically attached later on. For data inside these windows, simple get and put operations are provided for remote access as well as an accumulation call, which combines the data transferred with the target data. One-sided communication only may occur within communication epochs which are framed by matching synchronization calls. These synchronization calls can either be collective calls or can be restricted to groups of processes, requiring synchronization calls on both the caller's and the target's side. Also, there is a synchronization variant where windows may be locked, giving a process exclusive access or allowing for multiple accesses to a window part on a specific process. Here no synchronization calls are needed on the target's side.
Furthermore MPI includes collective operations, dynamic process creation, parallel I/O and much more.

Chapter 5

Dense Matrix Multiplication in GASPI

In Chapter 2 the new algorithm *poadSGD* was introduced for the training of large-scale feedforward networks. Key to this algorithm is the multiplication of two dense matrices for a certain parallel distribution of these matrices which is the result from the network distribution and the pipelining of poadSGD. Both of these matrices are distributed across a group of processes each. These process groups however, are disjoint so the processes have to work together in order to accomplish the multiplication. This chapter presents such an implementation, focusing on the matrix multiplication, not the complete poadSGD algorithm.

In addition to dealing with the disjoint process group setup, another challenge posed in Chapter 2.4.4 is taken on: using a parallel programming language which supports one-sided and asynchronous communication schemes. Here, the programming model GASPI (Chapter 4) comes into play. The implementation of the matrix multiplication presented in the following relies solely on GASPI for its parallelization.

First the data distribution for matrices used in this thesis is described: the *g-matrix*. It is based on the matrix descriptor `global_desc_t` which contains information about the exact layout, information about the subgroups, the distribution across the processes and the exact location of remote data and its local equivalent `local_desc_t`, which additionally contains information local to the processor such as the exact length of the local data and many more.

This g-matrix is then employed for the implementation of the matrix multiplication for dense matrices. The implementation itself is described focusing on the two main aspects of this algorithm: a simple way of enforcing parallelism by diagonally processing the matrix sub blocks and a more complex way to achieve an overlap of computation and communication in the processing of each sub block. In addition the time required for the operation is compared in a theoretical analysis, once for a parallel implementation without any RMA-capabilities and processing the matrix blocks sequentially and once for the author's implementation. Finally, the results of scaling tests of the code executed on up to 2048 cores are presented and discussed.

Part of Section 5.1 and the implementation section of 5.2 are strongly based on the author's own contribution to the publication [BSH14].

5.1 Matrix Representation

In parallel implementations the setup and distribution of data is one of the crucial points when designing a parallel application. In this case, it concerns matrices of various sizes. In order to achieve a scalable and balanced data distribution, the matrix is distributed in a blockwise fashion, aiming at a good work balance. In addition, it allows to break down the original algorithm into smaller matrix operations based on these data blocks. This also provides the opportunity to transfer data blockwise. Furthermore, it allows for the local usage of BLAS-3 (matrix-matrix) operations that are already highly optimized for most platforms.

Therefore, in this thesis matrices are distributed in a two-dimensional block cyclic fashion, as described by Dongarra et al. [DLP03], among a given group of processes. First the matrix of dimension $g_m \times g_n$ is divided into blocks of size $b_m \times b_n$, allowing for non-full blocks in the most right block column or the lowest block row. Figure 5.1 shows the division of an exemplary 24×24 matrix into blocks of size 3×3. The processes of the process

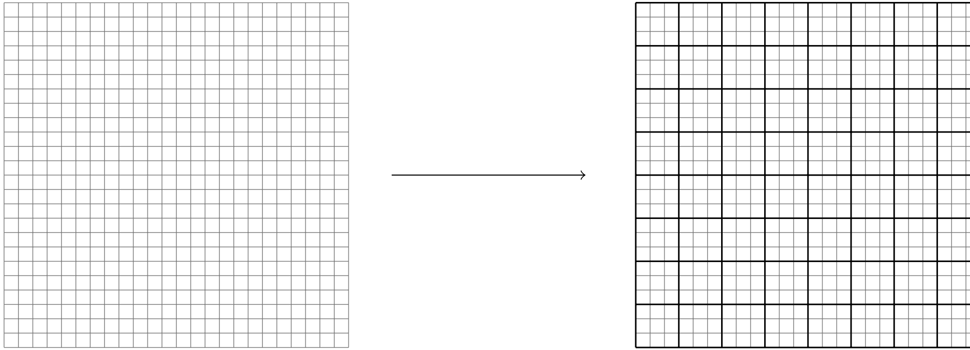


Figure 5.1: Dividing a $g_m \times g_n$ matrix (here 24×24) into blocks of size $b_m \times b_n = 3 \times 3$.

group holding the matrix are mapped to a process grid of dimension $\mathtt{P} \times \mathtt{Q}$. The blocks are then mapped in a two-dimensional block cyclic fashion to the processes corresponding to the process grid. Figure 5.2 depicts this distribution of the blocks amongst the processes $P0, P1, P2$ and $P3$ on a process grid of dimension 2×2. A matrix distributed this way is

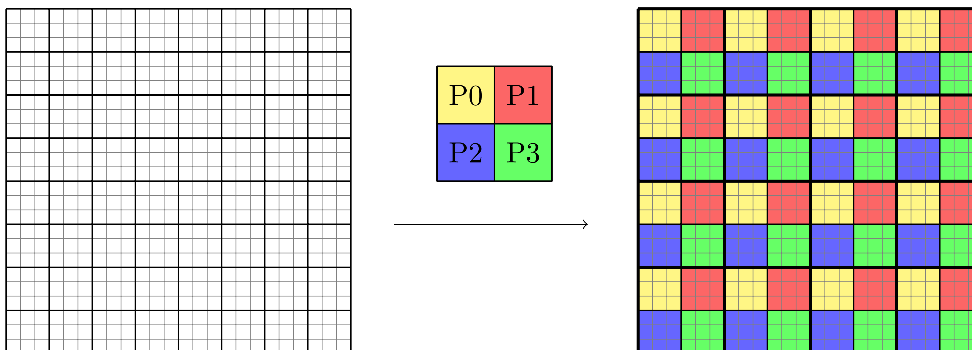


Figure 5.2: Mapping the matrix blocks in a two-dimensional block-cyclic fashion to processes on a 2×2 process grid, obtaining a *g-matrix*.

referred to as a *g-matrix* in this work.

Each process stores its local matrix data blockwise in a rows and continuously from a certain starting point in global memory. It is possible to access a specific element of the g-matrix by only knowing its global element index. Therefore, information is needed about the rank of the process holding the element, the segment ID on which the data resides and the exact offset of the element in the respective segment. So certain information has to be known by all processes that access matrix elements. This information is stored within a global descriptor of the g-matrix, called `global_desc_t` (the corresponding part of the code is given in Code 5.1). Its values have identical for all processes or at least for those working with the g-matrix. The global descriptor contains the following information

- the dimensions of the process grid ($\mathtt{P} \times \mathtt{Q}$),
- the dimensions of the matrix ($g_m \times g_n$) and its blocks ($b_m \times b_n$),
- an array `procs` containing an ordered list of the processes appendant to the matrix,
- an array `segment` containing the segment IDs on which the local matrix parts reside, respectively for each of the `procs` processes and
- an array `memloc` containing the associated offsets on the respective segment.

For each of the matrix processes the information within these arrays is stored corresponding to their position in the `procs` array.

```
1  typedef struct {
2
3    // process grid (PxQ)
4    gaspi_rank_t P;
5    gaspi_rank_t Q;
6
7    // array of procs in the group of length P*Q
8    gaspi_rank_t* procs;
9
10   // global size of matrix (mxn)
11   gaspi_number_t g_m;
12   gaspi_number_t g_n;
13
14   // size of matrix blocks (mxn)
15   gaspi_number_t b_m;
16   gaspi_number_t b_n;
17
18   // starting point in memory for each proc ordered by group rank
19   gaspi_offset_t* memloc;
20   gaspi_segment_id_t* segment;
21
22 } global_desc_t;
```

Code 5.1: The global descriptor of a *g-matrix*.

To complete this set of information, the local descriptor `local_desc_t` (Code 5.2) exists which also includes several other information containers next to the global descriptor. These containers are each only valid for the local process. The first is a GASPI group of which all processes noted in the `procs` array are part of. Further components are two arrays, `rowgroups` of length `P` and `colgroups` of length `Q`, which hold the group IDs of the groups containing the processes of each process grid row and column, respectively. These

subgroups are not always necessary but are useful in case of reduction operations which should only affect processes of the same row or column of the process grid. The number of blocks the local process owns in each dimension are stored in $\texttt{numblock}_m$ and $\texttt{numblock}_n$.

In case the blockwise data distribution does not come out evenly, the sizes of the last (most right) column of the matrix $\texttt{lastblock}_n$ and of the last row of the matrix $\texttt{lastblock}_m$ are stored as well.

```
typedef struct {

   // global information of the matrix
   global_desc_t global;

   // group which owns the matrix
   gaspi_group_t group;

   // my coordinates within the process grid
   gaspi_number_t myp;
   gaspi_number_t myq;

   // my position within the 'procs' array
   gaspi_rank_t mygrank;

   // subgroups of rows (array length global.P)
   gaspi_group_t* rowgroups;

   // subgroups of cols (array length global.Q)
   gaspi_group_t* colgroups;

   // #local blocks
   gaspi_number_t numblock_m;
   gaspi_number_t numblock_n;

   // #elements in last block
   gaspi_number_t lastblock_m;
   gaspi_number_t lastblock_n;

 } local_desc_t;
```

Code 5.2: The local descriptor of a *g-matrix*.

For an exemplary g-matrix that does not distribute evenly into blocks like a 14×13 matrix with block sizes of 3×3, the global descriptor would contain the following values:

$$
\begin{aligned}
(\texttt{P}, \texttt{Q}) &= (2, 2), \\
\texttt{procs} &= \{0, 1, 2, 3\}, \\
(g_m, g_n) &= (14, 13), \\
(b_m, b_n) &= (3, 3),
\end{aligned}
$$

with the `memloc` and the `segment` array depending on the user's setup. The local infor-

mation, e. g. for process $P2$ would be:

$$(\texttt{myp}, \texttt{myq}) = (1, 0)$$

$$\texttt{mygrank} = 2$$

$$(\texttt{numblock}_m, \texttt{numblock}_n) = (2, 3)$$

$$(\texttt{lastblock}_m, \texttt{lastblock}_n = (3, 1)$$

All information in the global descriptor and some information in the local descriptor such as the groups have to be set by the user. Getter functions are implemented for the rest of the components in the local descriptor. In case the global descriptor is set correctly, the getter functions also work for processes that are not members of the matrix group as they are only based on the global information.
Algorithms using this g-matrix structure benefit of a good data distribution which allows for a good work load balance, presuming the matrix size is not too small. It also enables the use of collectives on just one row or column of processors of the whole process grid.

5.2 A Matrix Multiplication Algorithm with GASPI

Assuming two matrices $\mathcal{A} \in \mathbb{R}^{m \times k}$ and $\mathcal{B} \in \mathbb{R}^{k \times n}$ are to be multiplied. The result is an $m \times n$ matrix $\mathcal{C}$:

$$\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$$

or written component-wise:

$$c_{i,j} = \sum_{l=0}^{m-1} a_{i,l} \cdot b_{l,j}$$

for $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$. When implemented sequentially, this algorithm needs $m \cdot n \cdot k$ operations, as the computation of each of the $m \cdot n$ elements of matrix $\mathcal{C}$ requires k operations.

Before turning to the actual implementation of the matrix multiplication of GASPI, the next section provides a review of previous matrix multiplication algorithms. After describing the author's implementation in the following section, a theoretical analysis of the algorithm is provided.

5.2.1 Related work

Several parallel algorithms for matrix multiplication exist. These are based on some kind of process grid to which the processes participating are mapped. They differ in various other aspects: they use different communication patterns, distribute their data differently, require the process mesh or the matrices to be squared or not, or also limit the sizes of the sub matrices. Of course, in times the underlying hardware also places additional constraints which push the algorithm being developed in a certain direction.
One of the best-known algorithms for the multiplication of two matrices was developed by Cannon in 1969 [Can69]. He assumes a distribution based on a joint two-dimensional process grid for all three matrices A, B and C. The algorithm targets an array-structured

computer and relies on a step-wise shifting of certain rows in the first and columns in the second matrix in order to generate intermediate elements of the sums giving the resulting element of the matrix C. It is also often referred to as the two-dimensional systolic approach.

The second classic algorithm is Fox's algorithm [FOH87], also known as the broadcast-multiply-roll approach. It also assumes at least a two-dimensional periodic mesh interconnect and the data is scattered across a squared process grid. Each process hereby receives one large continuous sub matrix. At each stage of the algorithm a block of the matrix A is broadcasted across its corresponding process row and one block of B is exchanged. Thereby each process can compute one component of the sum (5.2) per stage, i. e., the multiplication of two sub matrices.

Two algorithms were derived from the broadcast-multiply-roll approach: PUMMA by Choi et al. [CWD94] and BiMMeR by Huss-Lederman et al. [HLJT93]. With PUMMA Choi et al. developed a software package containing the standard matrix-matrix multiplication but also variants for one or two of the matrices being transposed. It is based on the idea of Fox's algorithm but also builds on the ideas of the ScaLAPACK implementation, hence also includes the two-dimensional block cyclic data distribution. Here however a non-square process grid and matrices of arbitrary dimensions are allowed. The algorithms presented in this paper achieve some parallelism in the outer block iteration by performing multiple instances of it together. In the inner loop blocks of B are broadcasted along a process column. For the computation and communication steps smaller blocks are conglomerated together in order to achieve a better performance. But as mentioned in [Cho97] this also limits the possibility for an overlap of computation and communication. The second algorithm building up on Fox's algorithm is BiMMeR [HLJT93]. A different data layout, the virtual 2D torus wrap, is utilized leading to a different algorithm. For simplicity Fox assumed the matrices to be squared, the process grid however may be non-square. Both algorithms were executed on the Intel Touchstone Delta, a supercomputer yet using its own message passing interface, NX.

Agarwal, Gustavson and Zubair also designed a matrix-matrix multiplication based on a two-dimensional processor grid for square matrices [AGZ94]. They claim to be the first to implement it with an overlap of communication and computation. In their algorithm, the data distribution is blockwise, each process receives one block of the corresponding size. The size of these sub matrices is chosen such that the local BLAS3 call runs at its peak performance and the actual matrix size is chosen based on a reasonable distribution on the given processor grid. The algorithm includes two steps in which an update for the resulting matrix C is computed whereas the data for the next update step is already transmitted and received. This is done in terms of broadcasts across processor columns or rows. The tests are run on an Intel iPSC System 860 and its successor Intel Touchstone Delta.

Independently Geijn and Watts [vdGW95] developed the same algorithm as Agarwal et al. and named it SUMMA. In addition, Geijn includes the algorithms for one or both of the matrices also being transposed before the multiplication.

In [Cho97] Choi also suggests a variant of SUMMA, named DIMMA, supplemented by two new ideas: a pipelined communication scheme for an effective overlap of communication and computation and further exploiting the block size determination concept in order to achieve a maximum performance when calling local BLAS procedures. Comparing DIMMA and PUMMA, Choi et al. found performance only to improve for smaller matrices.

Well known or rather widely used is the implementation of the matrix multiplication used in the PBLAS collection, the parallel BLAS collection [CDO+96].
Further algorithms concerning matrix multiplications are described in [FW97, GS94, KN04].
Another type of algorithm was developed by Li, Skjellum and Falgout [LSF95]. They developed a poly-algorithm which chooses between three algorithms: Cannon's, broadcast-multiply-roll and broadcast-broadcast, in order to always achieve the best possible performance.
In 2012 Georganas et al. [GGDS+12] reviewed Cannon's and the broadcast-multiply-roll algorithm and implemented both based on UPC (see Section 3.2.2). They also implemented two additional versions for each algorithm: one applying communication-overlap techniques and with the other aiming at avoiding communication, i. e., reducing the overall amount of communication.
Other recent work includes algorithms for matrix multiplications targeting heterogeneous clusters. The basic idea is for the algorithm carrying out the data partitioning to take into consideration the speed of each processor and accordingly balance the computational load in terms of differently sized rectangular sub matrices. KL [KL99] and BR [BBRR01] are examples hereof. Hereby the performance evaluation of the single processor depends on a static model returning a single value. Clarke et al. [CLR12] on the other hand use a functional performance model to determine the size of the rectangles and then maps these rectangles to the processors with the main goal of a minimizing the communication.
Furthermore dense matrix multiplication has not only been carried out on CPUs. For example in [CCZM10], the general matrix-matrix multiplication was implemented on the GPGPUs with cache on the Fermi architecture (Nvidia) using CUDA.

5.2.2 Implementation of Matrix Multiplication in GASPI

The basic ideas used in the GASPI implementation of the matrix-matrix multiplication, documented in this chapter, are based on the blockwise data distribution of the g-matrix as described in Section 5.1. In comparison, an important difference to other implementations (see Section 5.2.1), is that in this setting each of the matrices $\mathcal{A}$ and $\mathcal{B}$ has its own process grid. This means that a process may belong to only one of the process grids. This significantly distinguishes the algorithm from others, as considerations about data dependencies and data exchange are completely different. Before elaborating further on the details of the algorithm, information about the setup and assumptions that are made are described.

The simple notation of the dimensions in the section before is exchanged for one derived from the descriptor of the g-matrix, thereby achieving a better distinguishability when speaking about dimensions of different matrices. Therefore, the matrices and their dimensions are denoted as follows:
The matrices

$$\mathcal{A} \in \mathbb{R}^{g_m^{\mathcal{A}} \times g_n^{\mathcal{A}}} \text{ and } \mathcal{B} \in \mathbb{R}^{g_m^{\mathcal{B}} \times g_n^{\mathcal{B}}}$$

are divided into blocks of size

$$b_m^{\mathcal{A}} \times b_n^{\mathcal{A}} \text{ and } b_m^{\mathcal{B}} \times b_n^{\mathcal{B}},$$

respectively, further assuming $g_n^{\mathcal{A}} = g_m^{\mathcal{B}}$ and $b_n^{\mathcal{A}} = b_m^{\mathcal{B}}$. In general the implementation also is capable of an uneven apportionment but in order to ease notation and discussion, the matrices are required to divide evenly into blocks. That is, every block of the g-matrix is

of full size $b_m \cdot b_n$. The number of blocks a matrix then consists of is given by $bm^{\mathcal{A}} \cdot bn^{\mathcal{A}}$ for matrix $\mathcal{A}$ and $bm^{\mathcal{B}} \cdot bn^{\mathcal{B}}$ for matrix $\mathcal{B}$, whereas

$$bm^{\mathcal{A}} = g_m^{\mathcal{A}}/b_m^{\mathcal{A}},$$
$$bn^{\mathcal{A}} = g_n^{\mathcal{A}}/b_n^{\mathcal{A}} = g_m^{\mathcal{B}}/b_m^{\mathcal{B}} = bm^{\mathcal{B}},$$
$$bn^{\mathcal{B}} = g_n^{\mathcal{B}}/b_n^{\mathcal{B}}.$$

Furthermore, it is assumed that the process groups of $\mathcal{A}$ and $\mathcal{B}$ are distinct and that the resulting matrix $\mathcal{C}$ is stored with the exact same process grid as matrix $\mathcal{A}$, adopting the same order of process ranks. This data distribution of the matrices $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ to the two process grids is illustrated in Figure 5.3.

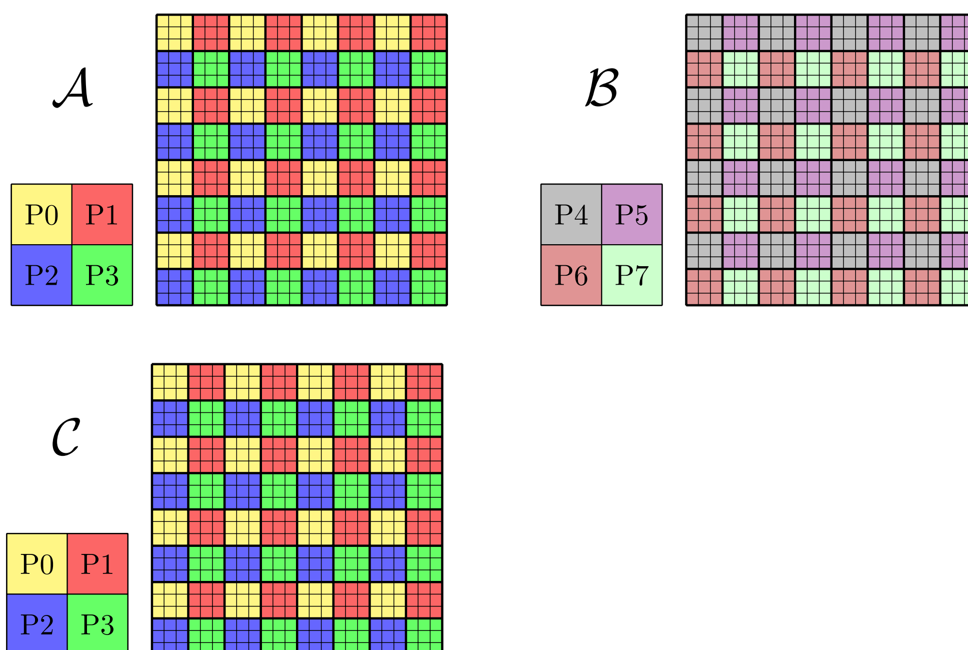


Figure 5.3: Mapping of the matrices $\mathcal{A}$ and $\mathcal{B}$ to their process grids and the multiplication result $\mathcal{C} = \mathcal{A}\mathcal{B}$ to $\mathcal{A}$'s process grid.

Additionally, the process grids of both matrices must match in one dimension just as the matrices themselves, i. e.,

$$\mathtt{Q}^{\mathcal{A}} = \mathtt{P}^{\mathcal{B}}. \tag{5.1}$$

In the following, a matrix block is referred to by its block coordinates within its matrix, e. g. $\mathcal{A}_{0,1}$ denotes the second block of $\mathcal{A}$ in its first block row. On the other hand,

$$(p, *)^{\mathcal{A}}$$

denotes the set of processors of $\mathcal{A}$ that are in the p-th row of the process grid, while

$$(*, q)^{\mathcal{B}}$$

refers to the set of processors in the q-th column of the process grid of $\mathcal{B}$ and the other combinations, analogously.

Computation of $\mathcal{C}$ – Diagonal Iteration

Due to the blockwise distribution of data, the calculation of a block of the resulting matrix consists of a summation of several smaller matrix multiplications. Enumerating all blocks of the matrix $\mathcal{C}$ with the indices $i \in \{0, \ldots, bm^{\mathcal{A}} - 1\}$ and $j \in \{0, \ldots, bn^{\mathcal{B}} - 1\}$, the block $\mathcal{C}_{i,j}$ is computed as

$$\mathcal{C}_{i,j} = \sum_{k=0}^{bm^{\mathcal{A}}-1} \mathcal{A}_{i,k} \cdot \mathcal{B}_{k,j}. \tag{5.2}$$

The goal of this implementation is to use a scheme that divides the computations into parts which each affect only one process or a subgroup of processes of each matrix group, respectively. By this means, the resulting computational parts are independent from each other and can be run in parallel.
Within the calculation of one block $\mathcal{C}_{i,j}$ only the processes of one process row $(p,*)^{\mathcal{A}}$ of $\mathcal{A}$ and the processes of the corresponding process column $(*,q)^{\mathcal{B}}$ of $\mathcal{B}$ are active, assuming that the blocks $\mathcal{A}_{i,*}$ are held by the processes of $(p,*)^{\mathcal{A}}$ and the processes $(*,q)^{\mathcal{B}}$ hold the blocks in $\mathcal{B}_{*,j}$. This is illustrated in Figure 5.4. So to compute block $\mathcal{C}_{2,5}$ data is needed

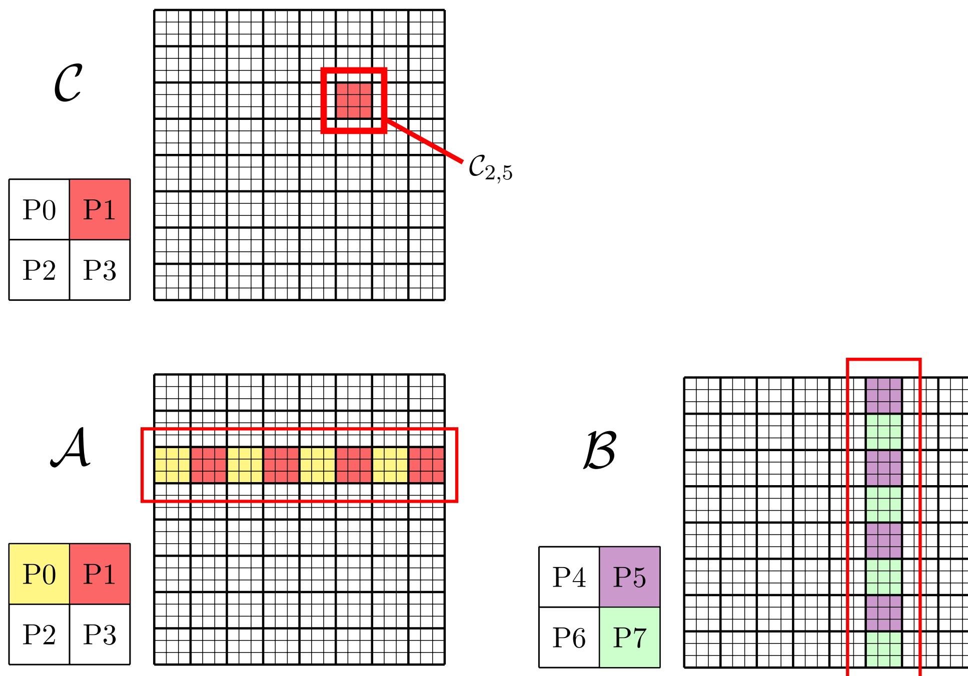


Figure 5.4: Illustration of the data needed for the computation of block $\mathcal{C}_{2,5}$ and the processes of $\mathcal{A}$ and $\mathcal{B}$ involved in the calculation.

from only one process row of $\mathcal{A}$ and one process column of $\mathcal{B}$. Hence the other process rows

of $\mathcal{A}$ and process columns of $\mathcal{B}$ are free to work on other process blocks of $\mathcal{C}$ in parallel, provided that they are not in the same process row or columns, e. g. block $\mathcal{C}_{1,4}$.
In order to make use of this natural parallelization, the procedure iterates over the blocks of the resulting g-matrix $\mathcal{C}$, computing one block of $\mathcal{C}$ in each step. Processes not belonging to the corresponding process row of $\mathcal{A}$ and process column of $\mathcal{B}$ of one iteration simply skip that calculation and continue with the next block and so on. A process not involved in a computation moves on until it reaches a block calculation in which it can participate. The idea is to accomplish this iteration in a way that subsequently computed blocks are held by groups of processes disjoint from that of the last block in which case subsequent blocks can be computed in parallel. Thus, the iteration starts with the upper left block and moves across the other blocks in a diagonal pattern, as shown in Algorithm 4. The diagonal iteration ensures the parallel computation of at least $\min\{\mathtt{P}^{\mathcal{A}}, \mathtt{Q}^{\mathcal{A}}, \mathtt{Q}^{\mathcal{B}}, bm^{\mathcal{C}}, bn^{\mathcal{C}}\}$ blocks at once.

```
1 for k = 0,...,bn^C − 1 do
2 |   for i = 0,...,bm^C − 1 do
3 |   |   j = (i + k) mod bn^C;
4 |   |   compute_block(C_{i,j});
5 |   end
6 end
```

Algorithm 4: Diagonal iteration scheme for iteration over blocks of $\mathcal{C}$.

Computation of block $\mathcal{C}_{ij}$

The function `compute_block`$(\mathcal{C}_{i,j})$ in Algorithm 4 computes the sub block $\mathcal{C}_{i,j}$ of the matrix $\mathcal{C}$. First the basics of the computation are explained, the content of body is then given in Algorithm 5. It is based on the fact that the calculation of $\mathcal{C}_{i,j}$ can be split into a sum of several smaller matrix multiplications, as given in Equation (5.2). So each block equals a summation of other blocks and each of these sums is calculated by a subgroup of processes of matrix $\mathcal{A}$ and a subgroup of processes of the matrix $\mathcal{B}$'s matrix group. Due to the assumption in Equation (5.1), these subgroups are of the same size. Viewing Equation (5.2) from a processes point of view, it can be written as follows:

$$\mathcal{C}_{i,j} = \sum_{q=0}^{\mathtt{Q}^{\mathcal{A}}-1} \underbrace{\sum_{k=0}^{N_q-1} \mathcal{A}_{i,k} \cdot \mathcal{B}_{k,j}}_{=:S_q(i,j)} = \sum_{q=0}^{\mathtt{Q}^{\mathcal{A}}-1} S_q(i,j) \tag{5.3}$$

where the block $\mathcal{C}_{i,j}$ is owned by the process $(\hat{p},\hat{q})^{\mathcal{C}}$ and

$$N_q = \mathtt{numblock}_n^{(\hat{p},q)^{\mathcal{A}}}$$

is the number of blocks a process has locally. $(p^{\mathcal{A}}, q^{\mathcal{B}}) = (\hat{p},\hat{q})$ denotes the grid coordinates of the process that owns block $\mathcal{C}_{i,j}$ in the process grid of $\mathcal{C}$ and $S_q(\cdot)$ the inner sum. In Figure 5.4 this would be process $P1$ which owns block $\mathcal{C}_{2,4}$ with coordinates $(\hat{p},\hat{q}) = (0,1)$ in the process grid of $\mathcal{C}$.

The outer sum of Equation (5.3) iterates over the processes within the grid row $(\hat{p}, *)^{\mathcal{A}}$ or grid column $(*, \hat{q})^{\mathcal{B}}$. In Figure 5.4 this corresponds to the process grid row 0 of $\mathcal{A}$ containing processes $P0$ and $P1$ and the process grid column $(*, 1) = \{P6, P7\}$ for the $\mathcal{B}$ matrix. For each q the inner sums $S_q(i, j)$ are only computed by a pair of processes, one from each matrix group: $a \in (\hat{p}, *)^{\mathcal{A}}$ and its counterpart $b \in (*, \hat{q})^{\mathcal{B}}$. No data has to be exchanged with any processes of the other matrix group. In Figure 5.4, for example, $P1$ only needs to exchange data with $P7$ and not communicate with $P6$.
The idea of the proposed algorithm is that as a first step each process $a = (\hat{p}, q)$ computes its contribution to block $C_{i,j}$, namely the inner sum $S_q(i, j)$ together with its counterpart $b = (q, \hat{q})$. Together they have to compute N_q matrix multiplications of their submatrices. To share the workload the number of blocks is divided in half. So basically, the inner sum from Equation (5.3) is split once more into two halves: the first $\alpha = \lceil N_q/2 \rceil$ summands and the latter $\beta = N_q - \alpha$ ones.

$$S_q(i,j) := \underbrace{\sum_{k=0}^{\alpha-1} \mathcal{A}_{i,k} \cdot \mathcal{B}_{k,j}}_{=:S_q^1(i,j)} + \underbrace{\sum_{k=\alpha}^{N_q-1} \mathcal{A}_{i,k} \cdot \mathcal{B}_{k,j}}_{=:S_q^2(i,j)}, \tag{5.4}$$

$$\mathcal{C}_{i,j} = \underbrace{\sum_{q=0}^{\mathtt{Q}^{\mathcal{A}}-1} S_q^1(i,j)}_{=:S^1(i,j)} + \underbrace{\sum_{q=0}^{\mathtt{Q}^{\mathcal{A}}-1} S_q^2(i,j)}_{=:S^2(i,j)}. \tag{5.5}$$

Algorithm 5 outlines the computation of these sums.

```
1 if myrank ∈ (p̂,*)^A or myrank ∈ (*,q̂)^B then
      // Computation involving process pair (a,b)
2     (S_q^1(i,j), S_q^2(i,j)) = compute_inner_sum_half(i,j,q);
      // A procs now hold S_q^1(i,j), B procs hold S_q^2(i,j).
      // reduction involving process row or column
3     S^1(i,j) = reduce_inner_sums( S_q^1(i,j) ) ;
4     S^2(i,j) = reduce_inner_sums( S_q^2(i,j) ) ;
5 end
6 if myrank == (p̂,q̂)^A or myrank ∈ (0,q̂)^B then
      // Summation involving process pair
7     C_{i,j} = final_sum(S^1(i,j), S^1(i,j));
8 end
```

Algorithm 5: Sketch of computation of block $\mathcal{C}_{i,j}$ in function compute_block($\mathcal{C}_{i,j}$) in Algorithm 4.

In function `compute_inner_sum_half()` in Line 2 of Algorithm 5 the inner sum parts $S_q^1(i, j)$ and $S_q^2(i, j)$ are computed by applying Equation (5.4). Thereafter, each process $a = (\hat{p}, q)$ holds the sum part $S_q^1(i, j)$ and each process $b = (q, \hat{q})$ holds the sum part $S_q^2(i, j)$.

The exact computation of these sum parts is explained in the "Overlap Algorithm" in the following section.
As a next step in the functions `reduce_inner_sums()` in Line 3 and 4 all processes of $(\hat{p}, *)^{\mathcal{A}}$ and analogously of $(*, \hat{q})^{\mathcal{B}}$ reduce their results within their row or column group, respectively. After this allreduce, all processes of $(\hat{p}, *)^{\mathcal{A}}$ contain the result of the first half of the inner sum $S^1(i, j)$ and all processes of the $(*, \hat{q})^{\mathcal{B}}$ column receive the result of the second half $S^2(i, j)$. An allreduce function is used as GASPI does not provide any other type of reduction operations. These allreduce-calls are possible as each descriptor contains the groups of each block row and block column. Following this step in the function `final_sum()` in Line 7 the first process in the process column of $\mathcal{B}$, $b = (0, \hat{q})^{\mathcal{B}}$, writes its allreduce result $S^1(i, j)$ to the $\mathcal{A}$ process $a = (\hat{p}, \hat{q})^{\mathcal{A}}$ which will hold the final result. The process $a = (\hat{p}, \hat{q})$ then calculates the final sum given in Equation (5.5) and stores it in the space it belongs to.
This concludes the computation of block $\mathcal{C}_{i,j}$ and therefore the function `compute_block(`$\mathcal{C}_{i,j}$`)` in Algorithm 4 returns and the diagonal iteration moves on with the next computation as described by that algorithm.

Computation of $S^1(i, j)$ and $S^2(i, j)$ – Overlap Algorithm

The details of function `compute_inner_sum_half()` in Line 2 of Algorithm 5 are described in the following paragraphs. Here, the inner sum parts $S^1_q(i, j)$ and $S^2_q(i, j)$ are computed as shown in Equation (5.4). The implementation is explained for two processes $a = (\hat{p}, q)$ from the corresponding row group of $\mathcal{A}$ and its counterpart $b = (q, \hat{q})$ from the corresponding column group of $\mathcal{B}$.
In order to share the workload, the inner sum is split into two halves as explained in Equation (5.4). The general idea is for a to compute $S^1_q(i, j)$ and b to compute $S^2_q(i, j)$. However for this task a needs the blocks $\mathcal{B}_{k,j}$ $(k = 0, \ldots, \alpha - 1)$ from b and b needs the blocks $\mathcal{A}_{i,k}$ $(k = \alpha, \ldots, N_q - 1)$ from a.
One possibility to implement this would be for b to first send its blocks $\mathcal{B}_{k,j}$ to a and for b to meanwhile send the blocks $\mathcal{A}_{i,k}$ to b. Then each part could compute its inner sum half as all sub blocks involved are available (assuming a large enough buffer for that). In order to avoid the overhead of the communication involved with this block of data exchange, the *overlap algorithm* (Algorithm 6) strives to mix these local matrix multiplications of $S^*_q(i, j)$ with the necessary exchange of blocks $\mathcal{A}_{i,k}$ or $\mathcal{B}_{k,j}$. The key feature of the overlap algorithm is that by mixing these operations, the communication resulting from the block exchange can be overlapped by the matrix multiplications.
This is accomplished by employing a pipeline. The idea is to overlap the sending of a block needed by the other process with the computation of the matrix multiplication of a different block that has already been sent. To initialize this pipeline, both processes a and b send the first block needed by the other process to each other (Line 6 and 25). Each data exchange is implemented by using the `gaspi_write_notify` so that not only the data is sent but additionally a notification is set on the receiver's side as soon as the block's data has arrived. In Algorithm 6 this function is split into two commands (`write block` and `notify`) for a better readability. The data is always stored in a designated location on a buffer within the global address space on the remote side. Same as the notification the ID of the buffer is circulated in order to not write into a memory space or set a notification that has not yet been seen by the remote side.
Then the pipeline itself starts, represented by the for loops in Line 8 and 24 of Algorithm 6. Both a and b processes write a block needed by the other process to the other process's

```
   // Processes a = (p̂, q)^A and b = (q, q̂)^B compute S_q^1(i, j) and S_q^2(i, j).
 1 α = ⌈N_q/2⌉;
 2 β = N_q − α;
 3 k = 0;
 4 if process a then
 5 │  determine rank of partner process b = (q, q̂)^B;
 6 │  write block A_{i,k+α} to buffer_{(k+1) mod 4} on b;
 7 │  notify b with ID k + 1;
 8 │  for k = 1 : β − 1 do
 9 │  │  write block A_{i,k+α} to buffer_{(k+1) mod 4} on b;
10 │  │  notify b with ID k + 1;
11 │  │  wait for notification k mod 4 from b;
12 │  │  compute A_{i,k−1} B_{k−1,j};
13 │  end
14 │  wait for notification k mod 4 from b;
15 │  compute A_{i,β−1} B_{β−1,j};
16 │  k++;
17 │  wait for notification k mod 4 from b;
18 │  compute A_{i,β} B_{β,j};
19 end
20 if process b then
21 │  determine rank of partner process a = (p̂, q)^A;
22 │  write block B_{k,j} to buffer_{(k+1) mod 4} on a;
23 │  notify a with ID k + 1;
24 │  for k = 1 : β − 1 do
25 │  │  write block B_{k,j} to buffer_{(k+1) mod 4} on a;
26 │  │  notify a with ID k + 1;
27 │  │  wait for notification k mod 4 from a;
28 │  │  compute A_{i,k+α−1} B_{k+α−1,j};
29 │  end
30 │  write next block to buffer_{(k+1) mod 4} on a;
31 │  notify a with ID k + 1;
32 │  wait for notification k mod 4 from a;
33 │  compute A_{i,α+β−1} B_{α+β−1,j};
34 end
```

Algorithm 6: The overlap algorithm: Overlapping the exchange of blocks and local matrix-matrix multiplications for the computation of the inner sum halves $S_q^1(i,j)$ and $S_q^2(i,j)$ (see Equation (5.4)).

buffer and notify it. Afterwards, it waits on the notification that a block from the other process has arrived as sketched in Line 11 and 27. In case a process receives a positive answer whilst waiting for a notification, it then knows that the expected data has arrived and can start the local matrix-matrix multiplication (Line 12 and 28). The result is added to a buffer designated to store the intermediate result, thereby step by step computing $S_q^1(i,j)$ and $S_q^2(i,j)$, respectively.
In order to obtain a better understanding of the algorithm, an example of this algorithm is illustrated in Table 5.1. The processes $a = (\hat{p}, q)^{\mathcal{A}}$ and $b = (q, \hat{q})^{\mathcal{B}}$ together compute $S_q^1(i,j)$ and $S_q^2(i,j)$. They use the simplified pseudo-code functions `write_notify`(B, x),

$a \in \mathcal{A}$		$b \in \mathcal{B}$
`write_notify`($\mathcal{A}_3$, 0)		`write_notify`($\mathcal{B}_0$, 0)
`write_notify`($\mathcal{A}_4$, 1)	(for-loop)	`write_notify`($\mathcal{B}_1$, 1)
`wait`(0) & `calc`($\mathcal{A}_0\mathcal{B}_0$)		`wait`(0) & `calc`($\mathcal{A}_3\mathcal{B}_3$)
`wait`(1) & `calc`($\mathcal{A}_1\mathcal{B}_1$)		`write_notify`($\mathcal{B}_2$, 2)
`wait`(2) & `calc`($\mathcal{A}_2\mathcal{B}_2$)		`wait`(1) & `calc`($\mathcal{A}_4\mathcal{B}_4$)

Table 5.1: Simplified version of the overlap algorithm described in Algorithm 6 for $N_q = 5$, $\alpha = 3$ and $\beta = 2$. The number in brackets denotes the notification ID.

`wait`(x) and `calc`(B). A call from process a to `write_notify`(B, x) means that the calling process is writing block B (assuming a sequential numbering) to b into buffer x and setting the notification x. The `wait`(x) call executed by process b designates that the process waits for notification ID x to be set by a. This means that block B sent by `write_notify`(B, x) has arrived in buffer x. Finally, function `calc`(B) denotes the computation of block B.

5.2.3 Theoretical Analysis of the Algorithm

The goal of this section is to evaluate the algorithm with respect to its potential for overlapping communication phases with other computational parts. This overlap reduces the total runtime and therefore enhances the efficiency of the implementation. Thereby, an important aspect is to detect necessary synchronization points in the algorithm. These divide the algorithm into sections that can only be executed sequentially. However, not all synchronization points affect all processes. Instead they may only apply to a subgroup of one matrix or of each process grid. Furthermore, it should be pointed out that within these sections between synchronization points computations still can be executed in parallel. In order to investigate if an overlap within these sections is possible, the time needed for the computational steps and the communication calls is evaluated in detail.
The implementation of the matrix multiplication is sketched in Algorithm 7. This pseudo-code builds up on Algorithm 4 but also points out the synchronization points. Synchronization occurs in two places between subgroups of both matrices. First, in Line 1 all processes are synchronized together, then in Line 5 only two subgroups are synchronized, whilst the other processes continue on.
Thereby the algorithm naturally consists of two main parts which are framed by barriers. In this implementation a barrier also conclude all previous memory operations to be completed. The first of these two parts is the *diagonal iteration*, that is the outer loop over the blocks of the matrix $\mathcal{C}$ starting in Line 2 after the global barrier. The second part is

```
1 gaspi_barrier (ALL);
2 for diagonal iteration(i, j) do
3   if myrank ∉ (p̂, ∗)^A and myrank ∉ (∗, q̂)^B then
4     return GASPI_SUCCESS ;
5   else
6     gaspi_barrier ((p̂, ∗)^A, (∗, q̂)^B);
7     compute_block(C_{i,j});
8   end
9 end
```

Algorithm 7: Sketch of matrix multiplication algorithm with focus on global synchronization.

the `compute_block()` procedure in Line 7 which is implemented in the *overlap algorithm* and starts after the synchronization of the row and column group.
Therefore, in a first step, only the outer loop of the algorithm which iterates over these sub-blocks is analyzed. The computations of such a sub-block (the *overlap algorithm*) is assumed to be fixed and is characterized by $T_{\text{block}}(b)$. It is compared to a naive version of its implementation. Then, in a second step, the *overlap* implementation of loop body is evaluated and again compared to a naively implemented version.
All in all, in the following the timings of the following sections are regarded:

- $T_{\text{naive_loop}}$: Outer loop implemented naively.
- $T_{\text{diag_loop}}$: Outer loop implemented with *diagonal iteration*.
- $T_{\text{naive_body}}$: Loop body implemented naively.
- $T_{\text{overlap_body}}$: Loop body implemented with *overlap algorithm*.

For the theoretical analysis, a simplified data distribution is assumed: the matrices are assumed to be square, i. e., $g \times g$, as well as its $b \times b$ sub-blocks. The process grids are assumed to be square as well, i. e., $\mathtt{P} \times \mathtt{P}$. Furthermore, each process shall receive the same number of blocks of a matrix ν_l^2. The total number of blocks in a block row of a matrix is then denoted as $\nu_g := \nu_l \mathtt{P}$.
Consequently the size of the matrices is given as

$$g^2 = (\nu_g b)^2 = (\nu_l \mathtt{P} b)^2.$$

Here ν_g represents the global number of blocks in one dimension, whilst ν_l denotes the local number of blocks in one dimension.

Analysis of the outer loop

Assuming a naive approach, the computation of the sub-blocks would proceed row by row, left to right. This naive approach is shown in Algorithm 8. So the order in which the blocks are processed is $\mathcal{C}_{0,0}, \mathcal{C}_{0,1}, ...\mathcal{C}_{0,\nu_g}, \mathcal{C}_{1,0}, ..., \mathcal{C}_{\nu_g,\nu_g}$. When moving through one row of $\mathcal{C}$, the calculation of two consecutive blocks involves two disjoint $\mathcal{B}$ (column) groups but the same $\mathcal{A}$ (row) group. For example, the calculation of the blocks

$$\begin{aligned}\mathcal{C}_{i,j} &: \text{ requires } \mathcal{A}(i,*) \text{ and } \mathcal{B}(*,j) \text{ and} \\ \mathcal{C}_{i,j+1} &: \text{ requires } \mathcal{A}(i,*) \text{ and } \mathcal{B}(*,j+1)\end{aligned}$$

```
1 for i = 0, ..., ν_g − 1 do
2 |  for i = 0, ..., ν_g − 1 do
3 |  |  compute_block(C_{i,j});
4 |  end
5 end
```

Algorithm 8: Naive iteration over the blocks to be calculated of $\mathcal{C}$.

where $i \in [0, \nu_g)$ and $j \in [0, \nu_g - 1)$.
Therefore blocks within a block row can only be computed sequentially one after the other as they all depend on the same process row group of $\mathcal{A}$. The only possibility where two consecutively called blocks do not depend on the same groups and therefore can be computed in parallel, is the calculation of the last block in one row and the first block in the next row:

$$\begin{aligned} \mathcal{C}_{i,\nu_g-1} &: \text{ requires } \mathcal{A}(i,*) \text{ and } \mathcal{B}(*,\nu_g-1) \text{ and} \\ \mathcal{C}_{i+1,0} &: \text{ requires } \mathcal{A}(i+1,*) \text{ and } \mathcal{B}(*,0) \end{aligned}$$

where $i \in [0, \nu_g - 1)$. For such a pair of blocks no data or processor dependencies exist. This is exemplarily illustrated by a matrix consisting of 6×6 sub-blocks in Table 5.2.

s	s	s	s	s	**p**
p	s	s	s	s	**p**
p	s	s	s	s	**p**
p	s	s	s	s	**p**
p	s	s	s	s	**p**
p	s	s	s	s	s

Table 5.2: Illustration of a row-wise iteration over matrix blocks in a naive implementation. ’s’ characterizes the blocks that can only be computed sequentially one after the other and ’**p**’ the blocks that can be computed in parallel.

Thus, in total the time required for the computation of $\mathcal{C}$ is as follows

$$\begin{aligned} T_{\text{naive_loop}}(b, P, \nu_l) &= \left(\nu_g^2 - (\nu_g - 1)\right) T_{\text{block}}(b) \\ &= \left((\mathtt{P}\nu_l)^2 - \mathtt{P}\nu_l + 1\right) T_{\text{block}}(b). \end{aligned} \tag{5.6}$$

Note that this formula assumes that the computation of two blocks may run in parallel ideally, i. e., ignoring any sharing of latency bandwidth or other resources between the different process groups of each block computation.

Now using the *diagonal* approach, the iteration proceeds as implemented in Algorithm 4. This approach leads to a different ordering of the block iteration which is exemplarily visualized in Table 5.3 with the same matrix example as in Table 5.2.
With this algorithm, the blocks are processed in a diagonal fashion, whereby the numbers in Table 5.3 indicate when a block is computed. If one block’s number is different from

1	4	7	10	13	16
16	1	4	7	10	13
14	17	2	5	8	11
11	14	17	2	5	8
9	12	15	18	3	6
6	9	12	15	18	3

Table 5.3: Illustration of a diagonal iteration over matrix blocks distributed across a 2×2 process grid. The numbers indicate the order when blocks are computed.

an other, they are computed at different times, with the lower number sooner than the higher one. Blocks labeled with the same number are computed simultaneously.
Looking at these blocks from a more general perspective, the computation of P consecutively computed blocks in the diagonal fashion involves the following groups:

$$\begin{aligned}
&\mathcal{C}_{i,j} &&: \text{ requires } \mathcal{A}(i,*) \text{ and } \mathcal{B}(*,j),\\
&\mathcal{C}_{(i+1)\%\nu_g,(j+1)\%\nu_g} &&: \text{ requires } \mathcal{A}((i+1)\%\nu_g,*) \text{ and } \mathcal{B}(*,(j+1)\%\nu_g),\\
&\mathcal{C}_{(i+2)\%\nu_g,(j+2)\%\nu_g} &&: \text{ requires } \mathcal{A}((i+2)\%\nu_g,*) \text{ and } \mathcal{B}(*,(j+2)\%\nu_g),\\
&\quad\vdots\\
&\mathcal{C}_{(i+\mathtt{P}-1)\%\nu_g,(j+\mathtt{P}-1)\%\nu_g} &&: \text{ requires } \mathcal{A}((i+\mathtt{P}-1)\%\nu_g,*) \text{ and } \mathcal{B}(*,(j+\mathtt{P}-1)\%\nu_g),
\end{aligned}$$

for $i = n\mathtt{P}$ with $n = 0, \ldots, \nu_l - 1$ and $j \in [0, \nu_g - 1)$. Due to the design of the process grid, in the case of $\mathtt{P} \geq 2$, the calculations of these $\mathtt{P}$ blocks involve $2\mathtt{P}$ different groups which do not interfere. Therefore, here the number of blocks that can be computed independently and in parallel is limited by $\mathtt{P}$. Note, that in a more general (non-square) case, the limitation would be $\mathcal{L} := \min\{\mathtt{P}^{\mathcal{A}}, \mathtt{P}^{\mathcal{B}}, (\nu_g)_m^{\mathcal{C}}, (\nu_g)_n^{\mathcal{C}}\}$. This gives the following estimation for the time needed for the diagonal outer iterations, again assuming a perfect architecture, an infinite bandwidth and a network worth buying:

$$\begin{aligned}
T_{\text{diag_loop}}(b, \mathtt{P}, \nu_l) &= \left\lceil \frac{\nu_g^2}{\mathcal{L}} \right\rceil T_{\text{block}}(b)\\
&= \left\lceil \frac{(\mathtt{P}\,\nu_l)^2}{\mathtt{P}} \right\rceil T_{\text{block}}(b)\\
&= \mathtt{P}\,\nu_l^2\, T_{\text{block}}(b) \qquad (5.7)
\end{aligned}$$

It is assumed that the diagonal iteration is superior to the naive implementation:

$$T_{\text{diag_loop}} \leq T_{\text{naive_loop}} \qquad (5.8)$$

$$\overset{(5.7),(5.6)}{\iff} \quad \mathtt{P} \cdot \nu_l^2 \cdot T_{\text{block}}(b) \leq \left((\mathtt{P} \cdot \nu_l)^2 - \mathtt{P} \cdot \nu_l + 1\right) T_{\text{block}}(b)$$

$$\overset{T_{\text{block}}(b) \neq 0}{\iff} \quad \mathtt{P} \cdot \nu_l^2 \leq \mathtt{P}^2 \cdot \nu_l^2 - \mathtt{P} \cdot \nu_l + 1$$

Looking at $T_{\text{naive_loop}}$ (Equation (5.6)) and $T_{\text{diag_loop}}$ (Equation (5.7)), it can be seen that for a fixed matrix and block size, $T_{\text{naive_loop}}$ roughly grows in $\mathcal{O}(\mathtt{P}^2)$ and $T_{\text{diag_loop}}$ in $\mathcal{O}(\mathtt{P})$. Figure 5.5 emphasizes this superiority of the *diagonal iteration* over the *naive iteration*. It shows the times needed for each iteration in dependency of both parameters ν_l and $2 \cdot \mathtt{P}^2$. Here

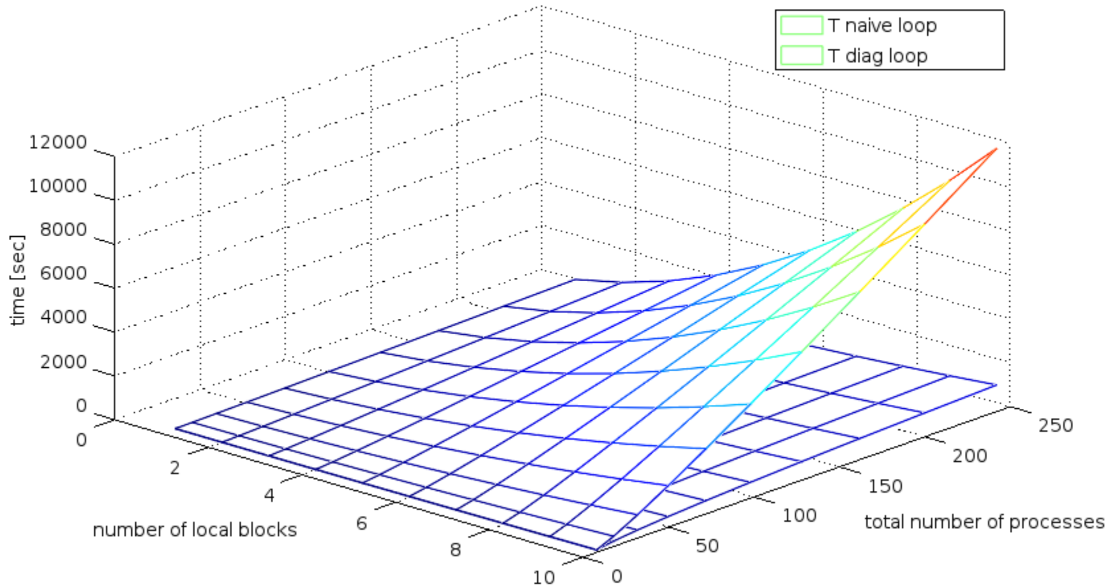


Figure 5.5: The time needed for the outer loop iteration of the matrix computation. The loop body is assumed to be of fixed size. The time (vertical z-axis) depends on two parameters given on the x and y-axis: ν_l and $2\mathtt{P}^2$.

Analysis of the loop body

Turning to the loop body of the outer loop described in the last sections, we turn to the computations of the sub-blocks of the matrix $\mathcal{C}$ which are computed in the *overlap algorithm*. The computation of one sub-block is assumed to be fixed and is characterized by $T_{\text{block}}(b)$. The *overlap algorithm* is based on the assumption of being able to overlap part or all of the transfer of one sub-block with a computational step.
In order to determine the time needed for the computation of a matrix multiplication of one sub-block of $\mathcal{A}$ times a sub-block of $\mathcal{B}$, first the time needed for one floating point operation needs to be known. The system's hardware (see Section 5.3) contains Intel Xeon E5-2630 v3 which manage 16 FLOPs per cycle. Running at a clock rate of 2.4 GHz, a single such processor is able to compute

$$2.4 \text{ GHz} \cdot 16 \text{ FLOP/cycle} = 2.4 \cdot 10^9 \text{ cycle/s} \cdot 16 \text{ FLOP/cycle} = 38.4 \cdot 10^9 \text{ FLOP/s}$$

This comes down to one FLOP taking about $2.6 \cdot 10^{-11}$ seconds.
Another way of computing the time needed for one FLOP, is by using the maximum peak performance achieved by benchmarks for the top500 list[20]. The maximum performance

[20] `http://www.top500.org/system/178541`

acquired by the computing cluster that was used, the 'bwForCluster MLS&WISO'[21], with the Linpack benchmark is 241.113 TFLOP/s = $241.113 \cdot 10^{12}$. Divided by 7,552 cores this gives $3.192704 \cdot 10^{10}$ FLOP/s per core.

$$\begin{aligned} T_{\text{FLOP}} &= \frac{1 \text{ FLOP}}{3.192704 \cdot 10^{10} \text{ FLOP}/s} \\ &= 3.13 \cdot 10^{-11} \text{ s} \end{aligned}$$

All in all, it can be safely assumed that the most time a floating point operation will take, is

$$T_{\text{FLOP}} = 3.13 \cdot 10^{-11} \text{ s}. \tag{5.9}$$

Setting T_{FLOP} to be the average time[22] a single processor needs for one floating-point operation, the time consumed by the computation of a single sub matrix can be estimated. If all data is locally available, a matrix multiplication of two matrices of size $b \times b$ requires b multiplications and $b - 1$ additions for each element of the resulting sub matrix $\mathcal{C}_{i,j}$. This occurs when one process already received the counter sub matrix from its counter rank. In total this gives $b^2(2b-1)$ floating point operations. Equation (5.10) presents the resulting time needed for this computation.

$$\begin{aligned} T_{\text{comp}}(b) &= b^2(2b-1)T_{\text{FLOP}} \\ &= (2b^3 - b^2)T_{\text{FLOP}} \end{aligned} \tag{5.10}$$

Of course, the loop body also contains a call to the allreduce function where even more floating-point operations occur and which is not covered so far. But due to the associated synchronization, for now the focus is only on the overlap algorithm.

As mentioned before, the time needed for the transfer of a sub block of the matrix is the other side of the equation. The time needed for a transfer of a block with b^2 elements, each of size β bits, is

$$T_{\text{tr}}(b) = \frac{b^2\beta}{B},$$

where B denotes the point to point bandwidth of two nodes of the cluster. So next to the block dimension, the time for a block transfer is highly dependent on the architecture and the network topology it runs on.

The bandwidth between two nodes on the bwForCluster MLS&WISO is 40 Gbit/s. However, in a general setting not only a single core is used on each node, but several cores. In this case the cores may be sending data at the same time and thus have to share the network, reducing it to B/ppn, where ppn is the number of cores on both nodes in use. Using the worst case scenario of all 16 cores using the network simultaneously and data

[21] See Section 5.3.1 for details on the cluster.

[22] On different platforms, floating-point operations may be implemented differently, thereby requiring a different number of instructions to complete and consuming a different amount of time. However multiplication and addition usually take up the same amount of time and else wise an average time is assumed.

of type double ($\beta = 64$ bits), the time needed for the transfer comes down to:

$$T_{\mathrm{tr}}(b) = \frac{b^2 \cdot 64 \text{ bit}}{\frac{40 \text{ Gbit/s}}{16}} = \frac{b^2 \cdot 64 \text{ bit} \cdot 16}{40 \cdot 1024^3 \text{ bit/s}} = \frac{b^2}{40 \cdot 1024^2} \text{s}. \tag{5.11}$$

In order to hide the amount of communication T_{tr} behind the computations achieved in T_{comp}, we need

$$T_{\mathrm{tr}} < T_{\mathrm{comp}} \tag{5.12}$$

to hold. Plotting these two functions as given in Equation (5.11) and (5.10), gives a first indication of the block dimension from which on Equation (5.12) holds. Figure 5.6

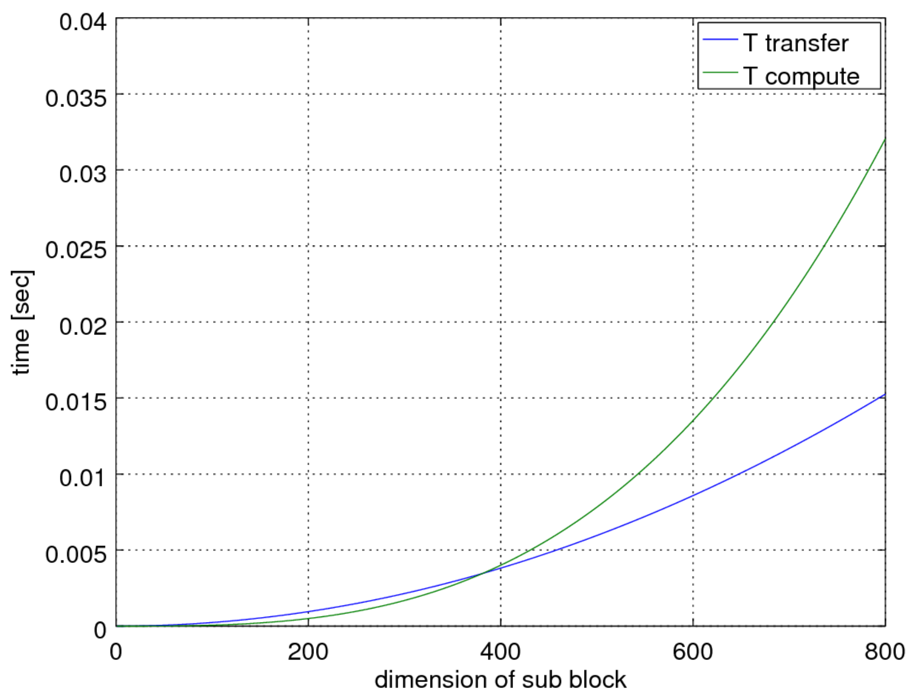


Figure 5.6: Comparing the time needed for the transfer of a sub-block T_{comp} and the time needed for a local matrix multiplication of two sub-blocks T_{tr} with block dimensions of up to 800.

shows the functions of T_{tr} (blue graph) and T_{comp} (green graph). The time needed for the transfer is longer than the time needed for the computational operations up till a block size between 370 and 390 from which on T_{comp} is faster.
Following this indication, we turn to the mathematical evaluation of the behavior of these two functions and determine when T_{tr} is smaller than T_{comp} and can therefore be over-

lapped efficiently:

$$\begin{aligned} & T_{\text{tr}} < T_{\text{comp}} \\ \overset{(5.11),(5.10)}{\Longleftrightarrow} \quad & \frac{b^2}{40 \cdot 1024^2}\text{s} < (2b^3 - b^2) T_{\text{FLOP}} \\ \overset{b \neq 0}{\Longleftrightarrow} \quad & \frac{1}{40 \cdot 1024^2}\text{ s} < 2bT_{\text{FLOP}} - T_{\text{FLOP}} \\ \overset{T_{\text{FLOP}} \neq 0}{\Longleftrightarrow} \quad & b > \frac{\frac{1}{40 \cdot 1024^2}\text{ s} + T_{\text{FLOP}}}{2T_{\text{FLOP}}} \\ \overset{T_{\text{FLOP}} \approx 3.13 \cdot 10^{-11}}{\Longleftrightarrow} \quad & b > 381.10 \end{aligned}$$

So a simultaneously performed data transfer can be completely overlapped by a local matrix multiplication, when $b \geq 382$. However, this only applies to the rather simple consideration of one sub block communication and computation.

Next, the time needed for the loop body is considered. Before, it was simply taken to be a fixed amount T_{block} occurring in each iteration step of the outer block loop. Again first a naive implementation of the loop body is considered which assumes the sub-computations and the necessary data transfers to occur strictly consecutively, resulting in the time $T_{\text{naive_body}}$. Then, the timing $T_{\text{overlap_body}}$ of the *overlap algorithm* as described in Algorithm 6 is addressed.

Again the pseudo-code notation of before is used but slightly adapted and extended:

- `write_notify`(b, n) denotes that the calling process writes the b-th block in the corresponding row or column to its counterpart and there also sets the notification n.
- `wait`(n) denotes that the calling process waits in a blocking way on the notification with ID n.
- `calc`(b) denotes that the calling process calculates the sub matrix multiplication of the b-th block.
- T_{write} denotes the time needed by a process to post a write call to its queue assuming the queue is available.
- T_{wait} denotes the time a processes is waiting on a notification to be posted by another process. It is no fixed time but varies from call to call depending on the current algorithmic step.

Next to the block sizes, the time needed for both loop bodies also depends on the number of blocks a processor locally has in a given row or column. Therefore, we will consider the different cases one by one. They are outlined in Table 5.4 through 5.8. Again α denotes the first half of a processes local blocks and β the second half, as introduced in Algorithm 6. Due to these different cases, processes from matrix $\mathcal{A}$ may have to accomplish different operations than processes from matrix $\mathcal{B}$. Timings are therefore additionally equipped with an exponent representing the process affiliation, e. g. $T^a_{\text{naive_body}}$ or $T^b_{\text{overlap_body}}$. The total time needed then results of the maximum of both process-bound times, i. e.,

$$T_{\text{naive_body}} = \max\{T^a_{\text{naive_body}}, T^b_{\text{naive_body}}\}, \tag{5.13}$$

$$T_{\text{overlap_body}} = \max\{T^a_{\text{overlap_body}}, T^b_{\text{overlap_body}}\}. \tag{5.14}$$

Case 1: $\nu_l = 1$, Table 5.4
In this case both a and b hold only one block in total ($\nu_g = \mathtt{P}$), so only one sub matrix

$\nu_l = 1, \quad \alpha = 1,$	$\beta = 0$
$a \in \mathcal{A}$	$b \in \mathcal{B}$
`wait`(1)	`write_notify`(0, 1)
`calc`(0)	

Table 5.4: $\nu_l = 1$

multiplication has to be calculated by a and b only has to write its sub matrix to a. This is visualized in Table 5.4. For the naive implementation these steps take place one after the other. Here, the total duration of time for process b is

$$T^b_{\text{naive_body}} = T^b_{\text{write}}.$$

The write call itself takes up close to no time at all. After posting the write call, the network takes on whilst the process b can continue on. In this case b moves on to the allreduce function which actually has no work to perform at all. For process a the time duration depends on the time spent in the wait and the time needed for the computation of the one small matrix block. At this, the time spent in the wait depends on the writing of process b and the time needed for the data transfer by the network. This results in:

$$\begin{aligned} T^a_{\text{naive_body}} &= \underbrace{T^a_{\text{wait}}}_{T^b_{\text{write}} + T_{\text{tr}}(b)} + T^a_{\text{comp}}(b) \\ &= T^b_{\text{write}} + T_{\text{tr}}(b) + T^a_{\text{comp}}(b). \end{aligned}$$

Hence, the total amount of time is given by:

$$T_{\text{naive_body}} \overset{(5.13)}{=} T^b_{\text{write}} + T_{\text{tr}}(b) + T^a_{\text{comp}}(b).$$

Regarding the implementation of Table 5.4 using the *overlap algorithm*, it can be seen, that for this small number of blocks no overlap is algorithmically possible. Therefore, the timings for the naive and the overlap implementation coincide:

$$T_{\text{overlap_body}} = T_{\text{naive_body}}.$$

Case 2: $\nu_l = 2$, Table 5.5
In this case each process has to process exactly one block, i. e., to write once and to compute once. Again, due to the number of blocks available no overlap is possible and $T_{\text{overlap_body}}$ equals $T_{\text{naive_body}}$. However the determination of T^a_{wait} is not as trivial as before. It now concludes as the maximum time spent by the process a before that call and the time needed by process b to reach the associated `write_notify` call plus the time needed by the network for the actual transfer.

$$\begin{aligned} T^a_{\text{naive_body}} &= T^a_{\text{write}} + T^a_{\text{wait}} + T^a_{\text{comp}}(b) \\ &= \max\{T^a_{\text{write}}, T^b_{\text{write}} + T_{\text{tr}}(b)\} + T^a_{\text{comp}}(b) \\ &= T^b_{\text{naive_body}}. \end{aligned}$$

$\nu_l = 2,$	$\alpha = 1, \quad \beta = 1$
$a \in \mathcal{A}$	$b \in \mathcal{B}$
`write_notify`$(1, 1)$	`write_notify`$(0, 1)$
`wait`(1)	`wait`(1)
`calc`(0)	`calc`(1)

Table 5.5: $\nu_l = 2$

Basically this comes down to a sequential flow again, as no data is yet present at the time the a starts its computation (i. e., the maximum only contains writes and transfers). So, here the following equation holds:

$$T_{\text{naive_body}} = T^a_{\text{naive_body}} = T_{\text{overlap_body}}.$$

Case 3: $\nu_l = 3$, Table 5.6
In the last "special" case, each process has three blocks in a block row or block column, respectively. It is similar to case 2 but now for the first time two `gaspi_write_notify`s occur right after another which are to start a pairing of the computation of one block and a data transfer of the next block happening at the same time. For process a the following

$\nu_l = 3,$	$\alpha = 2, \quad \beta = 1$
$a \in \mathcal{A}$	$b \in \mathcal{B}$
`write_notify`$(2, 1)$	`write_notify`$(0, 1)$
`wait`(1)	`write_notify`$(1, 2)$
`calc`(0)	`wait`(1)
`wait`(2)	`calc`(1)
`calc`(1)	

Table 5.6: $\nu_l = 3$

holds:

$$\begin{aligned} T^a_{\text{total}} &= T^a_{\text{write}} + T^a_{\text{wait}} + T^a_{\text{comp}}(b) + T^a_{\text{wait}} + T^a_{\text{comp}}(b) \\ &= \underbrace{\max\{T^a_{\text{write}}, T^b_{\text{write}} + T^b_{\text{tr}}(b)\}}_{=:M \text{ (constant)}} + T^a_{\text{comp}}(b) + T^a_{\text{wait}} + T^a_{\text{comp}}(b) \\ &= \max\{M + T^a_{\text{comp}}(b), 2T^b_{\text{write}} + 2T^b_{\text{tr}}(b)\} + T^a_{\text{comp}}(b). \end{aligned}$$

So in case the initial setup M does not take too long, the data transfer of the second block from b to a may be overlapped with the first block computation of a (so $2T^b_{\text{write}} + 2T^b_{\text{tr}}(b) \leq m + T^a_{\text{comp}}(b)$). But for process b clearly no opportunity for overlap is given.

$$\begin{aligned} T^b_{\text{total}} &= T^b_{\text{write}} + T^b_{\text{write}} + T^b_{\text{wait}} + T^b_{\text{comp}} \\ &= \max\{T^a_{\text{write}} + T^a_{\text{tr}}(b), 2T^b_{\text{write}}\} + T^b_{\text{comp}}. \end{aligned}$$

Case 4: $\nu_l > 3$, Table 5.7 and Table 5.8
All of the following cases can be divided into the two general cases: the case where the

number of local blocks in a block row or block column are even (Table 5.7) and the case where they are uneven (Table 5.8). Both tables are almost identical, except for the IDs of the blocks and the implementational part right after the for-loop. As they hardly differ elsewise, the runtime will only be evaluated for the even case.

$\nu_l = 2r$, $\alpha = r$, $\beta = r$	
$a \in \mathcal{A}$	$b \in \mathcal{B}$
write_notify$(r, 1)$	write_notify$(0, 1)$
for$(k = 1 : \beta - 1)$	
write_notify$(k + r, (k + 1)\%4)$	write_notify$(k, (k + 1)\%4)$
wait$(k\%4)$	wait$(k\%4)$
calc$(k - 1)$	calc$(k + r - 1)$
wait$(r\%4)$	wait$(r\%4)$
calc$(r - 1)$	calc$(2r - 1)$

Table 5.7: $\nu_l = 2r$, $r \geq 2$

$\nu_l = 2r + 1$, $\alpha = r + 1$, $\beta = r$	
$a \in \mathcal{A}$	$b \in \mathcal{B}$
write_notify$(r + 1, 1)$	write_notify$(0, 1)$
for$(k = 1 : \beta - 1)$	
write_notify$(k + r + 1, (k + 1)\%4)$	write_notify$(k, (k + 1)\%4)$
wait$(k\%4)$	wait$(k\%4)$
calc$(k - 1)$	calc$(k + r)$
wait$(r\%4)$	write_notify$(r, (r + 1)\%4)$
calc$(r - 1)$	wait$(r\%4)$
wait$((r + 1)\%4)$	calc$(2r)$
calc(r)	

Table 5.8: $\nu_l = 2r + 1$, $r \geq 2$

Thanks to the two write_notify calls at the beginning, in each iteration of the for-loop always the transfer of one block is initialized and right after that the computation of a block independent from the communication step (the block sent by the counterpart in the iteration before) is computed, starting as soon as the transfer has finished.
For this transfer to be completed, it has to take place simultaneously with the computation of the counter rank in the iteration before.
For the following equations, we keep in mind, that both processes start at the very same time and therefore although they do not synchronize in each step, due to the homogeneous hardware, they can be viewed as running simultaneously for each step. For reasons of overview we shorten the notation for the transfer time as follows:

$$T^b_{\text{tr}}(k) := T^b_{\text{tr}}(b),$$

for the k-th block in the sequence. Additionally note, that the superscript a or b does not

matter for the execution time itself but when rethinking the equations it helps to know "who" executed the action.
Determining the time needed for the wait in the k-th iteration, the time needed for the first wait is as follows:

$$\begin{aligned}
T^a_{\text{wait}}(1) &= \max\{2T^a_{\text{write}}, T^b_{\text{write}} + T^b_{\text{tr}}(1)\} \\
&= T^b_{\text{tr}}(1) \\
&= T^a_{\text{tr}}(1) \\
&= \max\{2T^b_{\text{write}}, T^a_{\text{write}} + T^a_{\text{tr}}(1)\} \;=\; T^b_{\texttt{wait(1)}}
\end{aligned}$$

where T_{write} is negligible as it takes only about one cycle to complete.

$$\begin{aligned}
T^a_{\text{wait}}(2) &= \max\{T^a_{\text{wait}}(1) + T^a_{\text{comp}}(1) + T^a_{\text{write}}, T^b_{\text{tr}}(1) + T^b_{\text{tr}}(2)\} && (5.15) \\
&= \max\{T^a_{\text{tr}}(1) + T^a_{\text{comp}}(1), T^b_{\text{tr}}(1) + T^b_{\text{tr}}(2)\} && (5.16) \\
&\overset{T^b_{\text{tr}}(1)=T^a_{\text{tr}}(1)}{=} T^a_{\text{tr}}(1) + \max\{T^a_{\text{comp}}(1), T^b_{\text{tr}}(2)\} && (5.17)
\end{aligned}$$

It is shown that while a computes its first block, the transfer of the second[23] block on the side of b occurs. Again, the same applies to $T^b_{\text{wait}}(2)$.

$$\begin{aligned}
T^a_{\text{wait}}(3) = &\max\{T^a_{\text{wait}}(2) + T^a_{\text{comp}}(2) + T^a_{\text{write}}, T^b_{\text{wait}}(1) + T^b_{\text{comp}}(1) + T^b_{\text{write}} + T^b_{\text{tr}}(3)\} \\
&[(5.15), \text{neglecting } T_{\text{write}}] \\
= &\max\{T^a_{\text{tr}}(1) + \max\{T^a_{\text{comp}}(1), T^b_{\text{tr}}(2)\} + T^a_{\text{comp}}(2), T^b_{\text{tr}}(1) + T^b_{\text{comp}}(1) + T^b_{\text{tr}}(3)\} \\
&[T^b_{\text{tr}}(1) = T^a_{\text{tr}}(1)] \\
= &\, T_{\text{tr}}(1) + \max\{\max\{T^a_{\text{comp}}(1), T^b_{\text{tr}}(2)\} + T^a_{\text{comp}}(2), T^b_{\text{comp}}(1) + T^b_{\text{tr}}(3)\} \\
&[\max\{T^a_{\text{comp}}(1), T^b_{\text{tr}}(2)\} \geq T^b_{\text{comp}}(1)] \\
= &\, T_{\text{tr}}(1) + \max\{T^a_{\text{comp}}(1), T^b_{\text{tr}}(2)\} + \max\{T^a_{\text{comp}}(2), T^b_{\text{tr}}(3)\} \qquad (5.18)
\end{aligned}$$

Through mathematical induction it can be shown, that the following equation holds for all $k \geq 3$:

$$T^a_{\text{wait}}(k) = T_{\text{tr}}(1) + \sum_{j=1}^{k-1} \underbrace{\max\{T^a_{\text{comp}}(j), T^b_{\text{tr}}(j+1)\}}_{=:m(j,j+1)} \qquad (5.19)$$

Basis: Show that the statement holds for $k = 3$. See (5.18).

[23]Note that "first" and "second" here does not refer to the actual IDs but only to the sequence in which they are called in the overlap algorithm.

Inductive step: Show that Equation (5.19) holds for $k+1$ if it holds for $k \geq 3$.

$$
\begin{aligned}
T^a_{\text{wait}}(k+1) = \; & \max\{T^a_{\text{wait}}(k) + T^a_{\text{comp}}(k) + T^a_{\text{write}}, \\
& \quad T^b_{\text{wait}}(k+1) + T^b_{\text{comp}}(k-2) + T^b_{\text{write}} + T^b_{\text{tr}}(k)\} \\
& \text{[Basis, neglecting } T_{\text{write}}\text{]} \\
= \; & \max\{T^a_{\text{tr}}(1) + \sum_{j=1}^{k-1} m(j,j+1) + T^a_{\text{comp}}(k-1), \\
& \quad T^b_{\text{tr}}(1) + \sum_{j=2}^{k-1} m(j,j+1) + T^b_{\text{comp}}(k-2) + T^b_{\text{tr}}(k)\} \\
= \; & T_{\text{tr}}(1) + \sum_{j=2}^{k-1} m(j,j+1) + \max\{m(1,2) + T^a_{\text{comp}}(k-1), T^b_{\text{comp}}(k-2) + T^b_{\text{tr}}(k)\} \\
& [m(1,2) \geq T^b_{\text{comp}}(k-2)] \\
= \; & T_{\text{tr}}(1) + \sum_{j=2}^{k-1} m(j,j+1) + m(1,2) + \underbrace{\max\{T^a_{\text{comp}}(k-1), T^b_{\text{tr}}(k)\}}_{=m(k-1,k)} \\
= \; & T_{\text{tr}}(1) + \sum_{j=1}^{k} m(j,j+1)
\end{aligned}
$$

As can be seen, each communication step occurs at the same time as an hereof independent computation step. In order to tell if the communication is really hidden perfectly by the computation, it is necessary to take a closer look at $T^a_{\text{comp}}(j)$ and $T^b_{\text{tr}}(j+1)$. In order to speed up this part, it is desirable that T^b_{comp} and T^{ppn}_{tr} overlap. This is the case when $T_{\text{tr}} \leq T_{\text{comp}}$ (see Equation (5.12) which for the bwForCluster was shown before to be the case when $b \geq 382$.
However this equation assumes a homogeneous architecture and does not take into consideration any cache effects (no memory hierarchies, cache misses or similar). Furthermore no difference is made between different floating-point operations, although here only additions and multiplications are considered, which most often take up the same number of instructions to complete.
All in all, it was shown that the overlap algorithm achieves the most overlap when each process holds at least a certain number of blocks in one row or column. To get the best performance from this, it is best when $T_{\text{tr}}(ppn, s)$ and T_{comp} are almost equally sized. If the block size is very small, the block transfer $T_{\text{tr}}(ppn, s)$ hardly takes up any time. The consequence is that all local computations do take place sequentially and the only benefit could be taken from the fact that the computational workload is shared. However the processes still have to go through the steps of reduction and last handshake between two processes of each group. This decreases the performance if two times the local computations plus these last steps take up more time than one processor executing the whole computational work in one step (which is the case if each process only has one

block). This is an overhead which does not exist when each process holds exactly one block.

5.3 Performance Tests

The matrix multiplication algorithm is to be used for the training of large-scale neural networks Thus, it is not only important to evaluate the performance of the algorithm, but also to know how it scales considering larger networks on large clusters. Hereby it is necessary to consider scaling the matrix sizes, the block sizes, and the number of processes employed, represented by the scaling of the process grids.
The exact configuration of the test parameters depends on the physical features of the cluster, algorithmic requirements such as memory for buffering and eventually the user's choice. In order to understand the final choice of the parameter settings, the influential circumstances are described in Section 5.3.1. Afterwards, in Section 5.3.2, the results based on these parameter settings are presented and discussed.

5.3.1 Test Setup

Before presenting how the test parameters came about, a description of the method is given that is used to validate the results of the matrix multiplication algorithm and a brief overview of the compute cluster employed.

Method Validation and Time measurement

The implementation of the matrix multiplication presented in this work is validated using Octave [E$^+$]. For this the matrices $\mathcal{A}$ and $\mathcal{B}$ are generated externally, the code reads them from file and the matrix $\mathcal{C}_{GASPI} = \mathcal{AB}$ is output to file again. Proof of correctness is achieved by entering these files into Octave and comparing the solution with the multiplication performed by Octave:

$$\texttt{norm}(\mathcal{C}_{\text{GASPI}} - (\mathcal{AB})_{\text{Octave}}) \leq 10^{-6}.$$

The check was performed for different grid sizes and matrices of up to approximately $2.1 \cdot 10^9$ elements (i. e., a matrix dimension of $\approx 46,340 \times 46,340$), as Octave cannot handle larger matrices. For the performance tests, the matrices are generated locally by each process itself in order to avoid time delays due to file input and output.
The amount of time needed for the matrix multiplication is measured via `gettimeofday()` from the start of the matrix multiplication to its end – in between barriers to ensure that all processes have completed their previous actions. It is assumed that when this function is called within a larger application, e. g. the poadSGD method introduced in Chapter 2, the matrices are already existent. Therefore the time measurement does not include the setup of the matrices, i. e., the input from file for $\mathcal{A}$ and $\mathcal{B}$, the output of the resulting matrix $\mathcal{C}$ or their local generation.

Computing and Software Requirements

Running an application using GASPI requires certain compute resources. First of all, GASPI is designed to carry out communication between cores not necessarily present on the same socket or sharing any memory. Actually in these cases other designs may even work better. Therefore, the main requirement for the compute resources to be used, is

having a minimum of two cores with some extra memory available each and connected by an InfiniBand network[24]. The memory is used in parts by the GASPI library itself. However, it is mainly needed for the share of each core in the partitioned global address space (the GASPI segments) when using asynchronous communication. The size of the memory needed depends on the application and the number of processes participating in the execution. Although the InfiniBand network is needed for the asynchronous communication calls, it is also possible to execute a GASPI application on a computing resource with an Ethernet network, thereby eliminating the effect of asynchronity. In addition, applications using the GASPI communication routines of course also need to install the GASPI library. In this case the GPI-2 v.1.0 [WJJ13] implementation was used. Furthermore, the matrix multiplication algorithm locally calls BLAS routines [CDO+96] for local matrix multiplications for which the MKL v.11.2.3[25] implementation was chosen.

Computing Resources

The test runs were executed on the *bwForCluster MLS&WISO Production*[26] in Heidelberg and Mannheim. It contains 666 nodes that are linked via an InfiniBand QDR [Inc] which provides an interconnect with 10Gb/s in each direction.The cluster contains different types of nodes. For the test runs in this work, the “standard” nodes of the cluster were used. These contain two octa-core 64-bit Intel Xeon CPUs with Haswell Bridge architecture running at 2.4 GHz and with 64 GB main memory. Point to point bandwidth between two standard nodes ought to be more than 5000 MB/s.
For each run the same amount of cores had to be requested for each node. The nodes were always used exclusively. In order to use the local batch system for execution, a trick had to be applied: in the code MPI is initiated at the beginning before starting GASPI and is finalized at the very end after GASPI has finished. That way `mpirun` could be used to run the application properly on the cluster. MPI in no other way influenced the execution or any communication.

Parameter Selection

The choice of the test settings includes various parameters depending on hardware settings, algorithmic requirements or simply user-preferences. For the sake of simplicity and comparability, the process mesh, the matrices and the sub matrices (the blocks) are all assumed to be squared. However, the implementation is more general and can cope with all non-square cases, only the testing was done based on squared blocks and meshes. Further it is assumed that each process part of a process grid also receives at least one block of its matrix. The test run itself then requires the following settings:

- `P` – dimension of process grid ($\mathtt{P} \times \mathtt{P}$)
- g – dimension of matrices ($\mathcal{A}, \mathcal{B}, \mathcal{C} \in \mathbb{R}^{g \times g}$)
- b – dimension of blocks in which the matrix is divided
- `segsize` – the size of the GASPI segments used

[24]Interconnect technology allowing for direct memory accesses [Pad11]
[25]`https://software.intel.com/en-us/intel-mkl/`
[26]`https://www.bwhpc-c5.de/wiki/index.php/Category:BwForCluster_MLS&WISO_Production`

These values are not completely freely chosen but are dependent on cluster-defined parameters such as

- `CoresPerNode` – number of cores used per node
- `GBperNode` – GB RAM available per node (and thus per core)

and the memory limitations induced by the matrix multiplication algorithm itself, including

- `maxsegsize` – maximum size of the GASPI segment
- `freespace` – amount of space needed for each core as temporary storage (e. g. for buffering).

Dimension of process grids $\mathtt{P}$

The matrices A and B are held by disjoint groups of processes and the process grids are of the same size and squared, i. e., $\mathtt{P}^{\mathcal{A}} = \mathtt{Q}^{\mathcal{A}} = \mathtt{P}^{\mathcal{B}} = \mathtt{Q}^{\mathcal{B}} := \mathtt{P}$. Therefore, the total number of processes `numproc` participating in the test runs is given by

$$\mathtt{numproc} = \mathtt{P}^{\mathcal{A}}\mathtt{Q}^{\mathcal{A}} + \mathtt{P}^{\mathcal{B}}\mathtt{Q}^{\mathcal{B}} = 2\mathtt{P}^2$$

Furthermore, GASPI does not run sequentially but requires at least two processes to run. Also, most of the algorithm is designed for being run by at least two processes per process grid, therefore the minimum number of processes is $2\mathtt{P} = 2 \cdot 4 = 8$. To allow for an even divisibility of the data, the process grid sizes are multiplied by two, starting with the smallest process grid of size 2×2. The resulting process grid sizes and the associated number of processes used are given in Table 5.9.

Process grid ($\mathtt{P} \times \mathtt{P}$)	2×2	4×4	8×8	16×16	32×32
# processes per grid	4	16	64	256	1024
# all processes (`numproc`)	8	32	128	512	2048

Table 5.9: Dimensions of the process grids used and the resulting number of processes used per grid and in total. Running the process grid 32×32 turned out to be not possible.

Further limitations apply to the maximum number of processes. The queuing policy of the "standard" nodes of the cluster only allows the usage of a maximum of 128 nodes (2048 cores). Thereby using process grids of size 64×64 or larger was not possible.

`CoresPerNode and GBperNode`

A "standard" node on the bwForCluster MLS&WISO contains 16 cores. When using several nodes, one can only use the same amount of cores for each node. Using less than 16 cores per node means that each process used has more memory available. Using more nodes on a core leads to less inter-node communication. In order to balance the larger memory size per core and the amount of communication in between nodes, 8 cores per node are used. Thereby only the 2×2 grids theoretically have an advantage as their communication infrastructure is located solely on one node. Note that for the process grid size of 32×32 less memory is available per core compared to all the others. Later results also showed that for the matrix multiplication algorithm, the 32×32 process grid failed

Process grid (P × P)	2 × 2	4 × 4	8 × 8	16 × 16	32 × 32
Nodes	1	4	16	64	128
Cores per node	8	8	8	8	16

Table 5.10: Listing of nodes and cores per node used for both process grids. Process grid of 32×32 had trouble running.

to run as the initialization of GASPI failed. The resulting distribution of cores and nodes is given in Table 5.10.
A node on the bwForCluster MLS&WISO has 64 GB of RAM. However, not all of it is freely available to the cores on the node, due to the operating system running on the node also requesting resources. The amount of memory running on the node can only be estimated. Following the experience gained with less memory available to the operating system and thereby crashing several nodes, a safety estimate of 4 GB is made which is not to be used by the application. The maximum amount of memory available hence is 60 GB per node. The resulting amount of memory available per core is 3.75 GB for the 32×32 grid and 7.5 GB for all the other process grids.

Maximum segment size `segsize`
The GPI2-library requires a certain amount of memory space in order to build up the communication topology of GPI2 and to store GASPI-elements such as notifications and queues. This quantity is application-dependent and varies with the segment size, the number of GASPI processes used and how they are connected. Unfortunately, there is no formulae which returns the total amount of memory needed by GPI2. Therefore, an application was created in which each participating process solely sets up the communication topology as required by the matrix multiplication algorithm and creates a segment of a given size. By trial and error it was then determined which the maximum usable segment size is. If this maximum size was exceeded, processes were terminated by the system. For the process grid 32×32 even for segment sizes as small als 0.5 GB the test application failed, indicating trouble whilst creating the segments or when setting up the communication topology during the initialization of GASPI. Because of that the maximum usable process grid that was used is 16×16. In the application the following segment sizes can be used:

Process grid (P × P)	2 × 2	4 × 4	8 × 8	16 × 16	32 × 32
Segment size [GB]	7.4	7.2	6.8	4.0	–

Table 5.11: Listing of segment sizes used depending on the process grid.

Once the segment size is known, the potential size of a matrix in connection with the given number of processes can be determined. For some cases, e. g. for larger matrices on smaller process grids, technical problems arose, presumably due to the unclear status of the memory.

Amount of Buffer in Global Address Space `freespace`
Freespace denotes the amount of memory space needed for each core as a temporary buffer for different one-sided communication operations not operating directly on matrix data.

This application-specific, user-defined memory space is required to be within the global address space, that is inside the allocated GASPI segments.
For the matrix multiplication in this work it is used as a buffer for the overlap algorithm, the allreduce and the final block computation. This additional buffer is used in favor of more synchronization points. All in all, the memory needed is seven matrix blocks plus a single element, all of the same data type:

$$\texttt{freespace} = (7b^2 + 1)\ \text{sizeof}(double) \tag{5.20}$$

Matrix dimension g
In order to achieve only full blocks when dividing the matrix into sub blocks, matrix sizes are powers of two. To start with a meaningful matrix size, the smallest matrix size is 64. The maximum matrix size is limited by the memory capacity available for the segment, the given block size b and the memory needed for the buffer `freespace`. Further, the size depends on the process that holds the most data. The processes of the process grid of $\mathcal{A}$ hold both matrices $\mathcal{A}$ and $\mathcal{C}$ and therefore need more memory than processes of $\mathcal{B}$. Hence, the maximum amount of matrix elements that are assigned to a core is the amount that is assigned to the process with grid coordinates $a_0 = (0, 0)$ of matrix $\mathcal{A}$.
In order to fit the maximum matrix into a GASPI segment, using Equation (5.20), the following requirement needs to hold:

$$\begin{aligned} \texttt{segsize} &\geq (2\,\texttt{maxsizematrix} + \texttt{freespace})\ \text{sizeof}(double) \qquad (5.21) \\ &= \left(2 \left\lceil \frac{\lceil g/b \rceil}{\texttt{P}} \right\rceil^2 b^2 + 7b^2 + 1 \right) \text{sizeof}(double) \end{aligned}$$

where $\lceil g/b \rceil$ is the number of blocks in a block row of a matrix. Divided by `P` accordingly denotes the number of blocks a process locally has in one dimension. Together, the first term states how many elements of the matrices $\mathcal{A}$ and $\mathcal{C}$ are stored in the memory of process a_0.

Process grid ($\texttt{P} \times \texttt{P}$)	2×2	4×4	8×8	16×16
Maximum matrix size g	32,768	65,536	131,072	262,144
Number of elements g^2	1.1 billion	4.3 billion	17.2 billion	68.7 billion
Total Memory [GB]	8	32	128	512
Memory per core [GB]	2	2	2	2

Table 5.12: Table of largest possible matrix sizes, the corresponding number of matrix elements and their overall size, as well as the resulting memory space needed on each core.

Equation (5.21) was applied to various combinations for matrix sizes for

$$g \in \{64, 128, 256, \ldots, 8589934592\}$$

and block sizes from

$$b \in \{2, 4, 8, 16, \ldots, 1048576\}$$

for all process grids mentioned in Table 5.9. In addition it was checked that each process receives at least one block ($g \geq \mathtt{P}b$). The maximum attained sizes in this process are given in Table 5.12.

Block dimension b

As explained before, first the triplets that are technically possible in terms of memory requirements and process numbers are determined. Then these combinations were run several times in order to identify the block dimension that achieves the best performance with respect to each process grid and matrix size. These best combinations are noted in Table 5.13.

$\mathtt{P} \times \mathtt{P} \setminus g$	**64**	**128**	**256**	**512**	**1024**	**2048**
2×2	32	64	128	128	512	512
4×4	16	32	64	128	256	512
8×8	2	16	32	8	16	256
16×16	4	8	8	32	32	128
$\mathtt{P} \times \mathtt{P} \setminus g$	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**
2×2	1024	2048	2048	2048		
4×4	512	1024	2048	4096	4096	
8×8	512	1024	1024	2048	4096	4096
16×16	256	512	1024	2048	2048	4096

Table 5.13: Block dimension b of fastest triplet of process grid $\mathtt{P}$, matrix dimension g and block dimension b.

$\mathtt{P} \times \mathtt{P} \setminus g$	**64**	**128**	**256**	**512**	**1024**	**2048**
2×2	1	1	1	2	1	2
4×4	1	1	1	1	1	1
8×8	4	1	1	8	8	1
16×16	1	1	2	1	2	1
$\mathtt{P} \times \mathtt{P} \setminus g$	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**
2×2	2	2	4	8		
4×4	2	2	2	2	4	
8×8	1	1	2	2	2	4
16×16	1	1	1	1	2	2

Table 5.14: Number of local blocks in a block row or column per process for fastest triplets ($\mathtt{P}$, g, b) in Table 5.13.

The combinations from this table show that for the smaller matrices the best possible block size almost always had the best timing performance. The biggest exceptions occur for the process grid 8×8 with the matrix sizes $64, 512$ and 1024. Especially for the latter two the best runs were achieved with 8 blocks per process (see Table 5.14). Other than

that the overlap algorithm seems to work better for larger block sizes as it is the case when the matrices themselves are larger. This is of course always delayed a bit for the smaller process grids gain bigger block sizes much quicker than larger process grids.

5.3.2 Results

The setup for the performance runs of the matrix multiplication algorithm is based on the parameter combinations from Table 5.13. Each $(\mathtt{P}, g, b)$ triplet from that table is run 25 times. The first 5 runs are cut off as the first ones often take inexplicably longer and the average is computed for each combination based on the following 20 runs. Figure 5.7 depicts these averaged runtimes for each process grid whilst varying the dimension of the matrices. In this graphic both axes are logarithmically scaled.

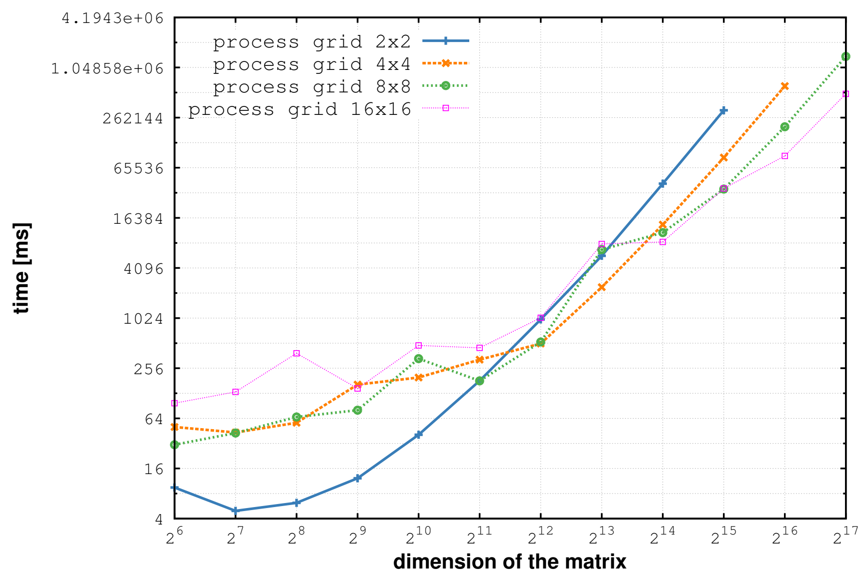


Figure 5.7: Runtime of matrix multiplication for different process grids and different matrix sizes. Note the logarithmic axes.

First, it is observed that all run times more or less increase with the computation of larger matrix sizes, except for the smallest process grid. Up to a matrix dimension of $2^{11} = 2,048$, the smaller the process grid is, the faster are the run times. This is possibly due to the fact that the larger process grids in their computational step only have very small blocks (see Table 5.13) which are no match for the communication overhead produced by the 16 processes per process subgroup. The smaller process grids have block sizes between 16-64 for the first two matrices and less communication overhead in the allreduce than the larger process grids.

For matrix sizes of $2^{12} = 4,096$ and $2^{13} = 8,192$ the run times of all process grids are similar. For larger matrices the previously observed order is reversed: the largest process grid also is the comparatively fastest. The block sizes mostly are larger for the smaller process grids but each process also holds more of these blocks and therefore has more locally sequential computations to complete. For the matrix size $2^{15} = 32,768$, all process grids have the same block size, except for the 4×4 grid. The difference here is that

each process of the 2×2 grid has to compute 8 of these blocks whilst the 8×8 only deals with 2 of these blocks and hence the 16×16 grid only has one block. So even if the overlap algorithm does hide away the necessary communication for the exchange still the performance of the smaller grids for this matrix size is worse as they each have to accomplish more computations each.

Next to the author's own observations of the runtimes, the scalability of the parallel implementation is analyzed based on its speedup and efficiency. The speedup $S_p(n)$ is defined as the ratio of the runtime of a sequential implementation $T^\star(n)$ and the parallel runtime $T_p(n)$ with p processes, both with a problem size of n [RR10]:

$$S_p(n) = \frac{T^\star(n)}{T_p(n)}.$$

The ideal speedup is achieved if $T_p(n) = \frac{1}{p}T^\star(n)$ and therefore $S_p \leq p$. However, as mentioned before, any algorithm implemented with GASPI cannot be run sequentially as GASPI requires at least two GASPI processes to run. In addition, the algorithm is set up to use all processes it was called with. So there is no chance of cheating the program by starting it with two processes but only using one in the implementation. In lack of a sequential implementation, the runtimes achieved by process grid 2×2 are used as reference. The speedup is then calculated as

$$\hat{S}_p(n) = \frac{T_4(n)}{T_p(n)}. \tag{5.22}$$

Hence, the ideal speedup needs to be scaled as well, becoming $\hat{S}_p(n) \leq \frac{p}{4}$.
The efficiency of an algorithm is computed by dividing the speedup by the number of processors used:

$$E_p(n) = \frac{S_p(n)}{p}. \tag{5.23}$$

The ideal efficiency in reference to a sequential implementation would be constantly 1. Again using the results from 2×2 as a basis, the Equation (5.23) can be rewritten as in Equation (5.24). The ideal efficiency then remains the same:

$$\hat{E}_p(n) = \frac{4\hat{S}_p(n)}{p}. \tag{5.24}$$

Speedup and efficiency could not be evaluated for matrix dimensions $65,536$ and $131,072$ as for these sizes no runtime reference was available from process grid 2×2. The results discussed for the total execution time of the matrix multiplication are also reflected in the speedup and efficiency charts, given in Figure 5.8 and 5.9. Again, the behavior very strongly depends on the matrix dimension. Matrices with a dimension smaller than 2^{10} at first have a very good speedup and efficiency, even scaling super-linearly. However, these speedups drop below the ideal line with increasing process numbers but still reaches about half of the ideal speedup. On the contrary, for matrices with a dimension between 2^{11} and 2^{13} the speedup first drops and then slightly increases again. The worst performance occurs for the largest matrices 2^{14} and 2^{15} where the speedup simply drops and is far off from the ideal speedup when increasing the number of processes. This also matters when considering the efficiency. Efficiency drops far below 0.004 for both the largest matrices.

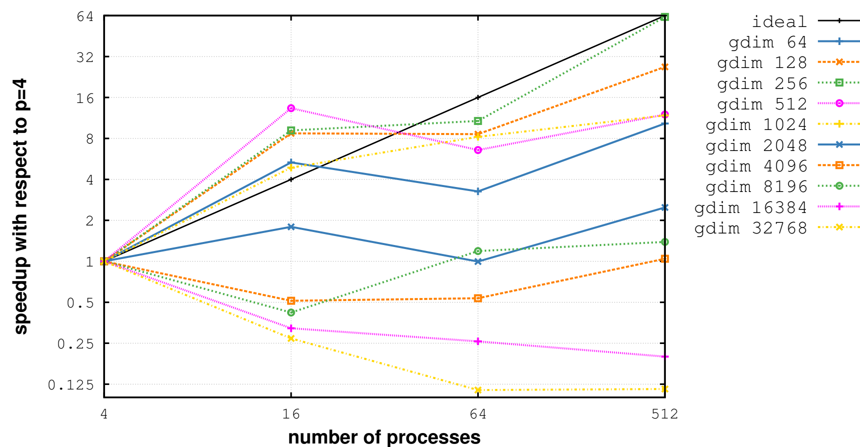


Figure 5.8: Speedup for matrix multiplication with timing from process grid 2×2 as basis. Note the logarithmic axes.

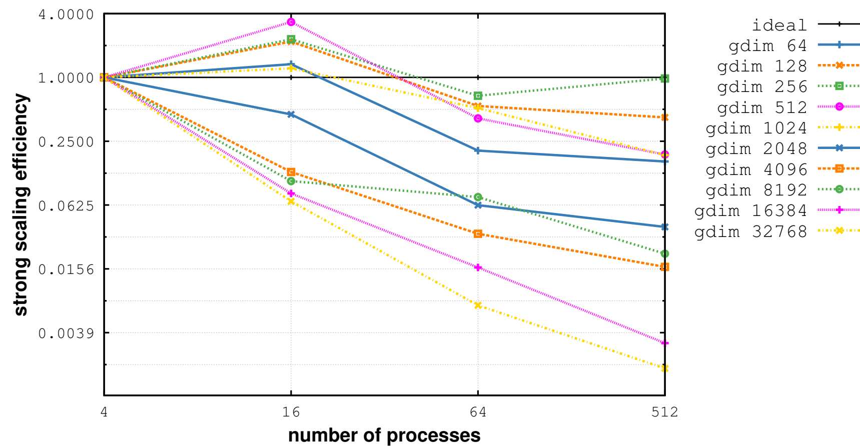


Figure 5.9: Strong scaling efficiency for matrix multiplication with timing from process grid 2×2 as basis. Note the logarithmic axes.

For the matrices smaller than 2^{10} the efficiency first is far better than 1 but then also decreases with a minimum for $g = 2^6$ at 0.16.

It turns out hard to interpret these results on a solid basis. On one hand, it seems clear that the scalability is better for smaller or medium matrices than for significantly larger matrices. On the other hand, the differences in the block sizes and the associated number of blocks per process have a high influence on the overlap algorithm and the allreduce operation and their performance. This is not represented in Figures 5.8 and 5.9. It is therefore difficult to reason about these results since the number of blocks held by each process of the reference process grid 2×2 is significantly different to the other process grids for larger matrices.

Another approach to evaluate the scalability is therefore undertaken by evaluating the behavior of the implementation when the workload for each process is kept constant while the number of processes is increased. This is referred to as the weak scalability and is

defined as

$$\mathcal{R} = \frac{T(\alpha p, \alpha n)}{T(p,n)}, \tag{5.25}$$

where $T(p,n)$ is the runtime for the algorithm run with p processes and the workload n. If the increase of both workload and number of processes by the factor α scales well, $\mathcal{R}$ is constant. Again, the efficiency is computed by referring to the runtime of $T(4,n)$ as a basis and given in Figure 5.10.

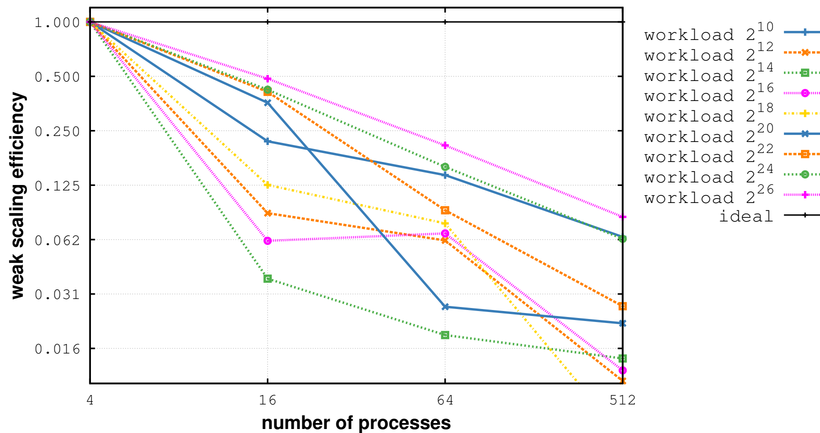


Figure 5.10: Weak scaling efficiency for matrix multiplication for different workloads and according process numbers. Note the logarithmic axes.

Again, the workload distribution does not take into account the number of blocks held by each process. Instead the best triplet is used once more. In general, in Figure 5.10 it can be deduced that the more processes have the same workload, the worse is their performance. However, the best behaviors are achieved for the largest workload per process, so a better result may be expected for larger workloads per process.

All in all, the performance did not withstand the expectations. However, the implementation of the algorithm is fully functional and although the test were carried out only for square matrices and process grids, the implementation is more general and can also be applied to non-square cases (under the condition that $g_n^{\mathcal{A}} = g_m^{\mathcal{B}}$, $b_n^{\mathcal{A}} = b_m^{\mathcal{B}}$ and $\mathtt{Q}^{\mathcal{A}} = \mathtt{P}^{\mathcal{B}}$). Therefore it can be used for the implementation of the poadSGD algorithm for the training of fully-connected FNNs which was presented in Chapter 2.4.2.
Furthermore, this algorithm gave first impression on how to use the different features of GASPI. For example by the usage of GASPI's groups it is possible to apply collectives only to a subgroup of processes, in this case to block rows or block columns featured by the process grid of the g-matrix. Without this feature the distributed and parallel calculation of blocks of the resulting g-matrix $\mathcal{C}$ would not have been possible. Furthermore, the groups allowed for a group-specific synchronization used for the implementation of the diagonal iteration over the $\mathcal{C}$ blocks. The usage of the notifications provided by GASPI also made the overlap algorithm possible: by using the notifications when sending data, an action on the remote process's side was avoided. Unfortunately, the memory and node limitations on the compute nodes restricted the dimensions of the matrices. The overlap

algorithm may have benefited from a larger number and size of the sub matrices. Finally, the algorithm is implemented to easily be integrated into a larger application such as the training of a neural network as proposed in the poadSGD method or whenever needed in other applications.

Chapter 6

Conclusion and Outlook

The major contribution of this thesis is the development of a new parallel pipelined training algorithm for fully-connected feedforward networks based on one-sided and asynchronous communication and the implementation of the key part of this algorithm, a matrix multiplication for matrices distributed amongst disjoint processes which takes advantage of the features GASPI's communication paradigm provides.

Summary of the Work

In this section for each of the chapters presented in this work, a short summary of the main findings is given.

An overview of machine learning and an introduction to the field of artificial neural networks was given in **Chapter 2**. Based on a simple fully-connected feedforward neural network (FNN) the feedforward and backward propagation were described and the stochastic gradient descent method was presented. After a review on various parallelization techniques for neural networks, a new parallelization approach was introduced which recombines concepts from previous parallelization methods with new ideas. The poadSGD algorithm was presented which relies on a distribution of its forward and backward copy of the network whilst pipelining the training samples batch-wise and uses a one-sided communication mechanism for the asynchronous update of the network. It was pointed out that the pipelining scheme induces a matrix multiplication with a special data distribution: the two matrices to be multiplied are distributed across different groups of processes.

Chapter 3 provided the necessary background for a basic understanding of the parallel programming interface GASPI which is to be used for the asynchronous weight update of the poadSGD algorithm and the implementation of the matrix multiplication. For this purpose different classifications of parallel architectures and parallel programming models were presented and a key distinguishing feature was identified: the view on memory that these models possess. Thereby one differentiates between programming models with the underlying communication model assuming a shared view of memory with common direct access for all processes, a distributed view with only local accessibility requiring explicit data exchange and a global view of memory where the memory is possibly distributed but either provides a global view or is even directly accessible without interaction on the host's side. As two well-known representatives of the third category, the PGAS-models, UPC and CAF were introduced and their communication mechanisms were presented.

The following **Chapter 4** then introduced a rather young development from the field of PGAS models: the *global address space programming interface* GASPI which was used for the parallel implementations of this work. GASPI was developed by a team consisting of members from science and industry in a three year project from 2011-2014 in which the author actively took part. The key features of GASPI were presented: being based on a globally accessible address space, regardless of the actual layout of the hardware architecture and the communication concept relying on one-sided and asynchronous communication calls. The usage of these communication routines, further mechanisms for synchronization such as the process barrier and collective communication requests were explained, accompanied by introductory programming examples. In addition, GASPI was compared to UPC and CAF whereby the focus was on the setup of globally shared data and its distribution type, remote data accesses and synchronization mechanisms. Eventually, an overview of further communication models was presented, all of which follow the idea of one-sided and asynchronous communication but differ in other aspects and are not categorized as PGAS models.

The final chapter, **Chapter 5** was dedicated to the development of a matrix multiplication for matrices distributed across disjoint process groups by making use of GASPI's communication mechanisms in order to improve their performance. The two-dimensional block-cyclic data distribution, relying on a distribution of the participating process to a process grid, was presented. Furthermore, the programming structure of the g-matrix which contains layout, memory and process groups information, were explained. For the implementation of the matrix multiplication method two sub algorithms were presented. Firstly, the "diagonal algorithm" was put forward which describes in what way the main loop iterates over the blocks of the resulting matrix. Secondly, the "overlap algorithm" was described which has the goal of, after an initial offset, overlapping needed data transfers with other computations based on a pipeline concept together with the asynchronous capabilities of GASPI. In order to evaluate the algorithm with respect to its potential for overlapping communication phases with other computational parts, the time needed for the computational steps and the communication calls was evaluated and compared to a theoretical analysis of the same algorithm implemented in parallel but without RMA-capabilities. This theoretical analysis provided good results for the proposed algorithms. Finally, the setup of the test runs was described. To this end, first for each combination of process grid dimensions and possible matrix sizes tests were run to determine the block sizes that achieve the best performance. With these best possible settings the actual test runs were performed. The runtimes on up to 512 cores were presented and discussed. The speedup and efficiency showed strong variances depending on the actual matrix size used for a test run with the efficiency varying between 0.10 and almost 1 for smaller and medium sizes matrices and falling below 0.004 for both the largest matrices of dimension $16,384$ and $32,768$.

Conclusions

After the summary we now turn to the discussion of the main findings in this work and draw conclusions. In addition, open questions or suggested extensions to this thesis are presented.

GASPI as a PGAS model
With the introduction of the parallel world and its multiple ways of categorizing languages,

models and architectures, the foundation is laid to be able to understand where GASPI is to be placed herein. GASPI categorizes itself as a "PGAS API for communication" [Con13]. It is described as a communication interface that is based on a partitioned global address space, a PGAS model. However, as the comparison to UPC and CAF shows, this does not fully classify GASPI, as representatives of this category already differ from each other. UPC and CAF, next to being set up as language extensions in contrast to the interface GASPI, differ from GASPI especially in terms of their ways of setting up and accessing remote data. Whilst in GASPI data in different memory segments is only connected to each other in the mind of the programmer, UPC and CAF use global data objects for the setup and distribution of data. Furthermore, with UPC and CAF data access occurs indirectly as the user only needs to state the index of the global data object to access a certain element, whereas GASPI therefore requires explicit data requests. From a user's point of view global data objects allow for an easier introduction into PGAS programming. However, the simplification also prevents a more direct control of the global data. With more direct control, the user has a greater freedom in the design of an implementation, providing better opportunities to include an overlap of data transfer with other computations.

Other interfaces such as OpenSHMEM or the one-sided mechanisms of MPI-3 are found to be very similar to GASPI in several aspects. However from the start, they have different objectives. The goal of OpenSHMEM is to provide a low-level layer with which PGAS languages can be implemented. MPI-3 aims at a broad audience, providing two-sided and one-sided communications and many more operations including broadcasts and gathers in different flavors. Reducing the view on MPI-3 to the one-sided communication mechanisms, they are found to be very similar to GASPI's, excluding the notification mechanism.

Programming with GASPI

In general, starting to program with GASPI requires the knowledge of only a small amount of GASPI functions. The communication calls are concentrated on a few basic procedures, thereby narrowing the hurdle for new users. However, programming a simple example with GASPI does not give the same impression as writing a more complex algorithm. A more complex algorithm entails an increased complexity of the memory management, the synchronization details and the tracking of which process is involved in which tasks, especially when tasks are generically distributed via subgroups.

Also truly understanding the basic functions of GASPI requires some experience. For example, forgetting to check the return value of an RDMA request whether the communication queue that is to be used is already full, quickly results in an undefined behavior of the program. In addition to the communication library, the open-source implementation of GASPI, GPI-2, also provides a debug library which incorporates several of these checks. Next, if generic notification IDs are used for a `gaspi_write_notify()`, the user has to take special care to have the remote process wait for the correct ID at the correct time. Otherwise the process may proceed working on code involving old data. After a successful wait for these notifications, they also need to be reset. When running the same application multiple times it was even observed that notifications set in one application run were still set in the next run causing the application to misbehave in an inexplicable way. Furthermore, simply checking the values of the notifications without resetting them, is not easy with the current GASPI calls where the notification values are always directly reset. Checking the notification values without resetting them is only possible by knowing the exact location of the notifications which is implementation-dependent and is usually

hidden from the user.
Another pitfall a user may encounter is the `gaspi_barrier()`. It has to be clearly understood that although all processes wait until the last process arrives at this point, still data transfers which were initiated before the barrier but not waited for, may not have completed yet. For this reason but also for convenience, an additional barrier which includes a memory fence would be a useful extension to GASPI's functionalities.
In order to develop a correct working implementation a concept of the memory management needs to be derived beforehand by the user: the data that is worked on, written to or read from another location needs to be tracked. This concerns all data in the GASPI segments, both the self-managed buffer space and the memory space where the matrix data is stored. The matrix data is tracked in terms of marking the data within the matrix itself and not as it is distributed in memory.
If a synchronization mechanism is not used correctly, for example, it may have concerned the wrong processes or a memory buffer may have accidentally been used twice, it is hard to debug the code. Tracking the change of a specified data location by employing a debugger is one method. However this does not help if the user monitors a part in memory whilst the data is transferred elsewhere. Furthermore, the debugger is only attached to one process at a time which is usually the process from which the program is started. To attach a debugger to another process the program needs to be started, then all processes need to be put to sleep for some seconds in order for the debugger to be started and attached to an other process.
All in all, programming with GASPI is not trivial and can be error-prone and hard to debug, especially when targeting complex data structures and algorithms.

Implementing a parallel matrix multiplication based on GASPI
After drawing conclusions on the programming with GASPI, the aspects of the development of the algorithm for the matrix multiplication are examined. At first thought, the matrix multiplication may appear to be straight-forward when considering the few lines of code needed for a sequential implementation. However, truly aiming for a highly parallel and scalable application many more characteristics have to be considered.
Firstly, it is important to bear in mind the computations that are to be accomplished and how the data distribution affects the work load for each process. Then, the amount and size of data exchange resulting from the data distribution need to be taken into consideration. The author chose the two-dimensional block-cyclic data distribution as it ensures quite a good work load balance for both applications. Furthermore, the author developed a helpful structure in order to store the global and local information about the distribution of the matrices which are essential for appropriate communication calls within the applications, ease the work with the matrices and provide a better overview on the distribution. Considering the work with the distributed data, it was found challenging to cope with the mapping of global indices for blocks or elements to the local indices to determine their actual location. For this reason auxiliary functions were implemented for all necessary conversions from global or local indices of blocks or elements as well as their coordinates and also to provide the possibility to map them to the rank that holds the queried data. In addition mappings from global ranks to the corresponding group ranks within a given group or their process grid coordinates and back were implemented.
On the basis of these, the algorithm was implemented as described in Chapter 5.2.2. The goal of the diagonal iteration is to achieve a maximum amount of parallelism by performing the computations for several blocks at the same time. After analyzing the dependencies on the processes involved in the computation of each block, the develop-

ment of the diagonal iteration was rather straightforward. In the diagonal algorithm, the parallelization is achieved as disjoint groups of processes are concerned with the computation of each block. However, the theoretical analysis of the diagonal iteration compared to a naive implementation shows just how much is achieved by such a seemingly “easy” implementation.

The other sub algorithm, the “overlap algorithm” was developed with the goal of overlapping needed data transfers with other computations, after an initial offset, based on the asynchronous capabilities of GASPI. The implementation was more complex, as the two processes working together in this sub algorithm and the synchronization of the individual block parts had to be precisely coordinated: it is important to determine exactly which block is written into which buffer at which time. It is also necessary to know which notification is used, so that, on the one hand, no data is overwritten in the buffer before it has been used and, on the other hand, processes do not idle whilst waiting for data. In doing so, data races and deadlocks were to be avoided and, at the same time, not to interfere with the local and global indices for elements, blocks and processes was challenging.

Performance of implementation

After the theoretical analysis of the matrix multiplication had predicted the overlap algorithm to work well for block sizes larger than 382, the test results were behind expectations. Already tests run to determine the block size that achieves the best performance for a given matrix size and process grid, showed that for these parameter combinations the overlap algorithm was best when it was basically not used. That is, the time needed mostly was the lowest when the block size was the largest possible, thereby reducing the overlap algorithm to just a single step.

Nevertheless the algorithm was scaled up to 512 cores, whereas both matrices were distributed onto 256 cores each. The matrix dimensions ranged from 64 to $131,072$ with the largest two matrices not being able to be run on all process grids due to memory limitations. A speedup of about 62.5 was achieved for matrices of dimension 256 ($65,536$ elements) for process grids with 256 processes each. As the speedup was determined with respect to the results achieved on the smallest process grids with 4 processes each, the ideal speedup in this case is 64. However, the speedup for matrices larger than 4096 was only slightly above one and even less for larger matrices with the same process grid size.

With the size of about seven matrix blocks, the amount of memory required for the buffer took up a lot of memory for some parameter settings. Due to this buffering used for the asynchronous communication to not overwrite needed data and the associated limitations of the memory, no larger matrices could be tested of which the overlap algorithm may have benefited with more and larger block sizes.

poadSGD – a distributed pipelined algorithm for FNNs

The poadSGD introduced in Chapter 2.4 has not been implemented yet, as the first implementational step in this thesis was to develop and implement the algorithm for the matrix multiplication based on GASPI. Still, already first thoughts are given to the proposal. In general, there are no convergence proofs for these algorithms, so there is no indication other than previous experience and trial-and-error to determine whether or not this training algorithm for neural networks works well. As to the implementation, some details yet have to be clarified which do not appear in the algorithmic description. For example the memory management should be well prepared: where to put the buffer for the matrix multiplication or where to store the matrices. Especially as each process grid will have to store at least two matrix parts if more than one layer is given to a process

group. Further, the matrix multiplication algorithm needs to be changed to store the result matrix on the second process grid instead of the first. In addition the number of layers that is mapped to each process group should be chosen carefully with respect to the frequency of updates being output at the end of the pipeline. Assuming a large-scale FNN, a large data set will be needed for the network pipelining to have an effect and for the network to not overfit the data. Based on the experience of the other researchers with similar ideas for improvement, the algorithm seems to be very promising.

Future Work

Several extensions to GASPI are taken into consideration. First, one could think of a `gaspi_setup()` function that initiates GASPI with a default set of configuration, creates one segment on each process of a default size and connects it to the other processes. In addition it could directly return the rank for each process, the total number of processes given by `gaspi_proc_rank()` and `gaspi_proc_num()` and the pointer to the start of the segment. This would allow new users an easier start with GASPI, allowing to start with the programming right away and not having to consider these configurations as their first step. For more experienced users, the setup of user-created queues was only recently added and is now included in the current specification of GASPI [GAS17]. Similarly, the possibility to set up user-specific notifications should be provided as well. This would be especially useful for the development of own libraries or when outsourcing parts of the application into external functions, to ensure that the notifications and queues there do not interfere with those in the main code. In the author's work this would have eased the usage and the separate handling of the notifications in the different sub functions. Further possible additions include the provision of a broadcast function for groups. The author developed a broadcast function that can be called for a specified group. It is implemented as passing the data from one process to the next in a line. This function could be extended to be used with different topologies, such as a ring, a tree or other topologies that the user could choose from. In the same way improvements for the reduction functions are conceivable.

Suggested improvements of the implementation of the matrix multiplication include increasing the initialization of the pipeline in the overlap algorithm and thereby extending the number of blocks that are computed in each for-loop step. In this way, the time needed for the computation of the blocks is increased whilst the bandwidth is better exploited by transferring more data in the same time. Another suggestion would be to design faster implementations of the allreduce function. Or the allreduce operation could be incorporated into the overlap algorithm as well, computing it step by step and avoiding the group barrier in each iteration. As a follow-up step a performance analysis tool could be employed in order to detect which points of the current algorithm could be optimized. Of interest would be to use the parallel analysis tool Vampir [Vam15], making use of GASPI's profiling interface. The final suggestion to be taken into account is to combine the usage of GASPI with the usage of threads in order to speed up the local computations.

The next natural step for the poadSGD algorithm is to implement it. As described before, a one-sided communication scheme such as GASPI should be used for this task. An pre-analysis should first be applied to determine certain parameters such as the number of layers to map to each process group, the block sizes in the distribution of the weight matrix or the size of the process grids. This algorithm is a first sketch of the basic ideas, it can be improved by applying some of the techniques presented in Chapter 2.2.4. One of these

possibilities could include applying the Parzen-Window optimization to the update step similar to Keuper and Pfreundt [KP15].

Outlook

GASPI proved to be an asynchronous communication model with great opportunities when targeting parallel applications with a high capability for overlap and communication. But in order to truly achieve a good performance both an experienced user and an application that can make use of GASPI's communication scheme are essential. Without a good potential for an overlap of communication phases with computation, a GASPI implementation will only be as good as an implementation based on a synchronized communication model. Assuming an algorithm provides this potential, implementations with GASPI are very promising.

As of today, the programming community has not yet agreed on a common parallel communication model. Of course, it is hard to imagine a perfect programming model that suits all types of applications. On the contrary, it might even be the case that there is a need for different models since each one targets different aspects. For one reason or the other, in the recent years, new programming languages and models are still being developed or "old" ones are enhanced, sometimes not only marginally but significantly just as Fortran did when the coarrays were included. Of course, much also depends on the development of the hardware. A single-core unit with a large amount of memory naturally necessitates a different programming model than a multi-core processor and a homogeneous cluster may call for other models than a cluster including specialized hardware such as GPUs or Xeon Phis. Although it only started out as a three year project, GASPI is still present at important HPC conferences and is employed by different research institutions. With the establishment of the GASPI forum, the forum members expressed an earnest interest to continue the development of GASPI. This thesis thereby represents an important milestone in the evaluation of the applicability of GASPI targeting dense linear algebra and the training of neural networks and is the basis for future work.

The topic of machine learning (ML) and especially of artificial neural networks remains a hot topic. Due to the increase in digitalization, more and more data is generated and collected than ever before. Thereby the amount of data that is to be analyzed and its complexity has increased. Therefore, ML is increasingly used in areas where classical algorithms reach their limits. ML algorithms are applied in research in the field of autonomous driving, for the generation of online recommendations based on previous search results, for the analysis of other data related to the internet, for medical applications and many more. ML algorithms can not only learn an application once, but they can always adapt and improve on new input. Companies also benefit from this, for example for the analysis of economical factors, the exchange or real estate prices. Research approaches are therefore pushed from many different directions in science and industry, which has a positive impact on the development. Continuously new developments and algorithms are tested. With the larger amount of data and the increased complexity of the applications, the approaches to parallelization are increasingly of interest and are promoted by the technological advances in hardware. It is the author's belief that advancements in machine learning will remain an important topic in the years to come and that the results will have a strong impact on everyone's future.

Appendix

```
1  /** ex-dotproduct-rw.c
2   *
3   * simple example to demonstrate the writing mechanism of GASPI
4   * setup: 2 processes, 2 vectors distributed across these two procs
5   * task : calc the dotproduct of the two
6   *
7   * Note: this is a demonstration program written for 4 processes, more
8   *       processes may be called, however only 4 will hold the data,
9   *       performance may decrease
10
11  */
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <GASPI.h>
15 #include <assert.h>
16
17 int main(int argc, char *argv[]) {
18
19   int i, dotproduct, ll;
20   const int vlength = 10;
21   int loc_a[10] = {1,2,3,4,5,6,7,8};
22   int loc_b[10] = {2,2,2,2,2,2,2,2};
23   int *gas_a, *gas_b, *gas_result;
24
25   gaspi_rank_t myrank, nprocs;
26   const gaspi_segment_id_t seg_id = 0;
27   gaspi_pointer_t seg_ptr;
28   gaspi_number_t queue_size, queue_max;
29   gaspi_queue_id_t queue_id = 0;
30   gaspi_notification_id_t not_id = 1;
31   gaspi_notification_t not_val = 1, val;
32
33   // initiate GASPI
34   gaspi_proc_init(GASPI_BLOCK);
35   gaspi_proc_rank(&myrank);
36   gaspi_proc_num(&nprocs);
37   ll = vlength/nprocs;
38   gaspi_segment_t seg_length = (2*ll+2) * sizeof(int);
39
40
41   if( nprocs > 2 ) {
42     printf("Program not written for %d processes, please use 2 processes
           only!\n", nprocs);
43     return -1;
44   }
45
46   // create segment for each process
```

```
47   gaspi_segment_create(seg_id
48                        ,seg_length
49                        ,GASPI_GROUP_ALL
50                        ,GASPI_BLOCK
51                        ,GASPI_MEM_INITIALIZED
52                        );
53
54   // reset notID to ensure no other initialization from program before
55   gaspi_notify_reset(seg_id, not_id, &val);
56
57   // barrier all procs, otherwise proc 0 may be quicker write_notifying
58   // than proc 1 with this first notification reset above
59   gaspi_barrier( GASPI_GROUP_ALL, GASPI_BLOCK );
60
61   // get segment pointer, set data pointers
62   gaspi_segment_ptr(seg_id, &seg_ptr);
63   int_ptr = (int*) seg_ptr;
64
65   // setup data in global memory
66   // P0:  1,2,3,4,2,2,2,2 (a_0,b_0)
67   // P1:  5,6,7,8,2,2,2,2 (a_1,b_1)
68   gas_a      = (int*) seg_ptr;
69   gas_b      = gas_a + ll;
70   gas_result = gas_b + ll;      //offset: 2 * ll * sizeof(int)
71
72   for(i=0;i<ll;i++) {
73     gas_a[i] = loc_a[i+myrank*ll];
74     gas_b[i] = loc_b[i+myrank*ll];
75   }
76
77   // calculate intermediate result, store in gas_result
78   for( i=0; i<ll;i++ )
79     *gas_result += gas_a[i] * gas_b[i];
80
81   // get queue data
82   gaspi_queue_size_max(&queue_max);
83   gaspi_queue_size(queue_id, &queue_size);
84
85   // check if queue full
86   if(queue_size > queue_max - 1)
87     gaspi_wait(queue_id, GASPI_BLOCK);
88
89   // set offsets to position of gas_result[0] and gas_result[1]
90   gaspi_offset_t loc_offset = 2 * ll * sizeof(int);
91   gaspi_offset_t rem_offset = loc_offset + sizeof(int);
92
93   // write local result to other process
94   gaspi_write_notify( seg_id, loc_offset, (myrank+1)%nprocs
95                     , seg_id, rem_offset, sizeof(int)
96                     , not_id, not_val
97                     , queue_id, GASPI_BLOCK);
98
99   // wait on intermediate result from other process
100  gaspi_notify_waitsome(seg_id, not_id, 1, &not_id, GASPI_BLOCK);
101
102  // calculate global result
103  dotproduct = gas_result[0] + gas_result[1];
104  printf("(%d) dotproduct = %d\n", myrank, dotproduct);
105
106  // wait to ensure data in gas_result[0] is free before exiting
```

```
107   gaspi_wait(queue_id, GASPI_BLOCK);
108
109   gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK);
110   gaspi_proc_term(GASPI_BLOCK);
111
112   return 0;
113 }
```

Code 6.1: Complete example demonstrating usage of GASPI functions for a simple dotproduct of two vectors. Extended version of Code 4.7.

```
1  /** ex-dotproduct-group.c
2   *
3   * simple example to demonstrate the writing mechanism of GASPI
4   * setup: 4 processes, 2 vectors distributed across these two procs
5   * task : calc the dotproduct of the two, use gaspi_allreduce
6   *
7   * Note: this is a demonstration program written for 4 processes, more
8   *       processes may be called, however only 4 will hold the data,
9   *       performance may decrease
10  */
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <GASPI.h>
14 #include <assert.h>
15
16 int main(int argc, char *argv[]) {
17
18   int i;
19   const int vlength = 16;
20   int loc_a[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
21   int loc_b[16] = {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
22   int dotproduct;
23   gaspi_rank_t myrank, nprocs;
24   const gaspi_segment_id_t seg_id = 0;
25   gaspi_pointer_t seg_ptr;
26   int *gas_a, *gas_b;
27   gaspi_pointer_t buff_send, buff_recv;
28
29   // initiate GASPI
30   gaspi_proc_init(GASPI_BLOCK);
31   gaspi_proc_rank(&myrank);
32   gaspi_proc_num(&nprocs);
33   int ll = vlength/nprocs;
34   gaspi_size_t seg_length = (2*ll+2) * sizeof(int)
35
36   // create segment for each process
37   gaspi_segment_create(seg_id
38                        ,seg_length
39                        ,GASPI_GROUP_ALL
40                        ,GASPI_BLOCK
41                        ,GASPI_MEM_INITIALIZED
42                        );
43
44   // get segment pointer, set data pointers
45   gaspi_segment_ptr(seg_id, &seg_ptr);
46   gas_a = (int*) seg_ptr ;
47   gas_b = gas_a + ll ;
48
```

```
49   // set pointers for allreduce op
50   buff_send = seg_ptr + 2*ll*sizeof(int);
51   buff_recv = buff_send + sizeof(int);
52 
53   // prog written for 4 procs only, rest may take a break
54   if( myrank <= 3 ) {
55 
56     // setup data in global memory
57     // P0:  1 , 2 , 3 , 4 ,2 ,2 ,2 ,2 ( a_0 , b_0 )
58     // P1:  5 , 6 , 7 , 8 ,2 ,2 ,2 ,2 ( a_1 , b_1 )
59     // P2:  9 ,10 ,11 ,12 ,2 ,2 ,2 ,2 ( a_2 , b_2 )
60     // P3: 13 ,14 ,15 ,16 ,2 ,2 ,2 ,2 ( a_3 , b_3 )
61     for(i=0;i<ll;i++) {
62       gas_a[i] = loc_a [ i + myrank * ll ];
63       gas_b[i] = loc_b [ i + myrank * ll ];
64     }
65 
66     // local calculations, store intermediate result in buff_send
67     for( i=0; i<ll;i++ )
68       *((int*) buff_send) += gas_a[i] * gas_b[i];
69   }
70 
71 
72   // calculate global result via allreduce, is stored in buff_recv
73   gaspi_allreduce( buff_send, buff_recv, 1, GASPI_OP_SUM
74                  , GASPI_TYPE_INT, GASPI_GROUP_ALL, GASPI_BLOCK);
75 
76   dotproduct = *( (int*) buff_recv );
77   printf("(%d) dotproduct = %d\n", myrank, dotproduct);
78 
79   gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK);
80   gaspi_proc_term(GASPI_BLOCK);
81 
82   return 0;
83 }
```

Code 6.2: Complete example demonstrating usage of GASPI's allreduce function for a simple dotproduct of two vectors. Extended version of Code 4.8.

	sync/async (s/a)	local/non-local (l/nl)	time-based blocking/blocking (tb/b)
Setup and management			
gaspi_config_get	s	l	b
gaspi_config_set	s	l	b
gaspi_proc_init	s	nl	tb
gaspi_proc_num	s	l	b
gaspi_proc_rank	s	l	b
gaspi_proc_term	s	nl	tb
gaspi_proc_kill	s	nl	tb
gaspi_connect	s	nl	tb
gaspi_disconnect	s	l	b
gaspi_state_vec_get	s	l	tb
Groups			
gaspi_group_create	s	l	b
gaspi_group_add	s	l	b
gaspi_group_commit	s	coll	tb
gaspi_group_delete	s	l	b
gaspi_group_num	s	l	b
gaspi_group_size	s	l	b
gaspi_group_ranks	s	l	b
Segments			
gaspi_segment_alloc	s	l	b
gaspi_segment_register	s	nl	tb
gaspi_segment_create	s	coll	tb
gaspi_segment_delete	s	l	b
gaspi_segment_num	s	l	b
gaspi_segment_list	s	l	b
gaspi_segment_ptr	s	l	b
One-sided communication			
gaspi_write	as	nl	tb
gaspi_read	as	nl	tb
gaspi_wait	as	nl	tb
Weak synchronization			
gaspi_notify	as	nl	tb
gaspi_notify_waitsome	s	nl	tb
gaspi_notify_reset	s	l	b
Extended communication and utilities			
gaspi_write_notify	as	nl	tb
gaspi_write_list	as	nl	tb
gaspi_write_list_notify	as	nl	tb
gaspi_read_list	as	nl	tb
gaspi_queue_size	s	l	b
gaspi_queue_purge	s	l	tb
Passive communication			
gaspi_passive_send	s	nl	tb
gaspi_passive_receive	s	nl	tb
gaspi_passive_queue_purge	s	l	tb

Table 6.1: Part I of list of all GASPI procedures; stating whether they are synchronous (s) or asynchronous (a), local (l) or non-local (nl) and if they are time-based blocking (tb) [Con13].

	sync/async (s/a)	local/non-local (l/nl)	time-based blocking/ blocking (tb/b)
Global atomics			
gaspi_atomic_fetch_add	s	nl	tb
gaspi_atomic_compare_swap	s	nl	tb
Global atomics			
gaspi_atomic_fetch_add	s	nl	tb
gaspi_atomic_compare_swap	s	nl	tb
Collective communication			
gaspi_barrier	s	nl	tb
gaspi_allreduce	s	nl	tb
gaspi_allreduce_user	s	nl	tb
Getter functions			
gaspi_group_max	s	nl	tb
gaspi_segment_max	s	nl	tb
gaspi_queue_num	s	nl	tb
gaspi_queue_size_max	s	nl	tb
gaspi_transfer_size_max	s	nl	tb
gaspi_notification_num	s	nl	tb
gaspi_passive_transfer_size_max	s	nl	tb
gaspi_atomic_max	s	nl	tb
gaspi_allreduce_buf_size	s	nl	tb
gaspi_allreduce_elem_max	s	nl	tb
gaspi_network_type	s	nl	tb
gaspi_build_infrastructure	s	nl	tb
Environmental Management			
gaspi_version	s	nl	tb
gaspi_time_get	s	nl	tb
gaspi_time_ticks	s	nl	tb
gaspi_error_message	s	nl	tb
Profiling interface			
gaspi_statistic_counter_max	s	nl	tb
gaspi_statistic_counter_info	s	nl	tb
gaspi_statistic_verbosity_level	s	nl	tb
gaspi_statistic_counter_get	s	nl	tb
gaspi_statistic_counter_reset	s	nl	tb
gaspi_pcontrol	s	nl	tb

Table 6.2: Part II of list of all GASPI procedures; stating whether they are synchronous (s) or asynchronous (a), local (l) or non-local (nl) and if they are time-based blocking (tb) [Con13]..

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0. Technical Report Version 1.0, Sun Microsystems, Inc. March 2008.

[AGZ94] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-performance Matrix-multiplication Algorithm on a Distributed-memory Parallel Computer, Using Overlapped Communication. *IBM J. Res. Dev.*, 38(6):673–681, November 1994.

[ALO02] Götz Alefeld, Ingrid Lenhardt, and Holger Obermaier. *Parallele Numerische Verfahren.* Springer-Lehrbuch Masterclass. Springer Berlin Heidelberg, 2002.

[BAdA+08] R. F. Barrett, S. R. Alam, V. F. d. Almeida, D. E. Bernholdt, W. R. Elwasif, J. A. Kuehn, S. W. Poole, and A. G. Shet. Exploring HPCS languages in scientific computing. *Journal of Physics: Conference Series*, 125(1):012034, 2008.

[Bar14] Blaise Barney. Introduction to Parallel Computing, 2014. UCRL-MI-133316 or Indira Gandhi National Open University (IGNOU).

[BBRR01] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1033–1051, Oct 2001.

[BLP+12] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *CoRR*, abs/1211.5590, 2012.

[Bon02] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, EECS Department, University of California, Berkeley, Oct 2002.

[BP95] M. Besch and H. W. Pohl. Flexible data parallel training of neural networks using mimd-computers. In *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, pages 27–32, Jan 1995.

[Bre12] J. Breitbart. Dataflow-like Synchronization in a PGAS Programming Model. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 762–769, May 2012.

[BRSS15] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435, 2015.

[BSH14] J. Breitbart, M. Schmidtobreick, and V. Heuveline. Evaluation of the Global Address Space Programming Interface (GASPI). In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 717–726, May 2014.

[BYEG06] Richard Barrett, Yiyi Yao, and Tarek El-Ghazawi. Evaluation of UPC on the Cray X1E, 2006.

[Can69] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[CCZ04] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60, 2004.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[CCZM10] Xiang Cui, Yifeng Chen, Changyou Zhang, and Hong Mei. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 237–242, Dec 2010.

[CDMM13] Georgel Calin, Egor Derevenetc, Rupak Majumdar, and Roland Meyer. A Theory of Partitioned Global Address Spaces. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, pages 127–139, 2013.

[CDO+96] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, PARA '95, pages 107–114, London, UK, UK, 1996. Springer-Verlag.

[CGBFRAB06] Enrique Castillo, Bertha Guijarro-Berdiñas, Oscar Fontenla-Romero, and Amparo Alonso-Betanzos. A very fast learning method for neural networks based on sensitivity analysis. *J. Mach. Learn. Res.*, 7:1159–1182, December 2006.

[CH08] Barbara Chapman and Lei Huang. Enhancing OpenMP and Its Implementation for Programming Multicore Systems. In *Parallel Computing: Architectures, Algorithms, and Applications*, Advances in parallel computing, pages 3 – 18. IOS Press, 2008.

[Cho97] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, pages 224–229, Apr 1997.

[CLR12] David Clarke, Alexey Lastovetsky, and Vladimir Rychkov. Column-Based Matrix Partitioning for Parallel Matrix Multiplication on Heterogeneous Processors Based on Functional Performance Models. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Di Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, StephenL. Scott, JesperLarsson Traff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 450–459. Springer Berlin Heidelberg, 2012.

[Con03] UPC Consortium. UPC Collective Operations Specifications V1.0. Technical report, George Washington University and IDA Center for Computing Sciences, 2003.

[Con13] GASPI Consortium. GASPI: Global Address Space Programming Interface, Specification of a PGAS API for communication, Version 1.01. Technical report, 11 2013.

[CS14] Ian D. Chivers and Jane Sleightholme. Compiler Support for the Fortran 2003 and 2008 Standards. Periodical ACM SIGPLAN Fortran Forum, 2014.

[CSC+05] Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, and Tarek El-Ghazawi. *UPC Manual.* The George Washington University, 801 22nd Street NW, Suite 607, Washington, DC 20052, v1.2 edition, 2005.

[CWD94] Jaeyoung Choi, David W. Walker, and Jack J. Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors,

Advances in Neural Information Processing Systems 25, pages 1223–1231. Curran Associates, Inc., 2012.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[DMN08] George Dahl, Alan McAvinney, and Tia Newhall. Parallelizing neural network training for cluster systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN '08, pages 220–225, Anaheim, CA, USA, 2008. ACTA Press.

[Dre05] Gérard Dreyfus. *Neural networks.* Springer, Berlin ; Heidelberg [u.a.], 2005. Orig.-Ausg.: Editions Eyerolles.

[Dun90] Ralph Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, February 1990.

[E+] John W. Eaton et al. GNU Octave.

[EGCS+06] Tarek El-Ghazawi, Franois Cantonnet, Proshanta Saha, Rajeev Thakur, Rob Ross, and Dan Bonachea. UPC-IO: A Parallel I/O API for UPC. Technical Report v1.02, The George Washington University, September 2006.

[FBC+14] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 4:1–4:11, New York, NY, USA, 2014. ACM.

[FFA92] Y. Fujimoto, N. Fukuda, and T. Akabane. Massively parallel architectures for large scale neural network simulations. *IEEE Transactions on Neural Networks*, 3(6):876–888, Nov 1992.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[FOH87] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube I: Matrix multiplication . *Parallel Computing*, 4(1):17 – 31, 1987.

[For15] MPI Forum. Status of MPI-3 Implementations — MPI Forum, 2015. [Online; accessed 23-January-2015].

[For16] MPI Forum. Status of MPI-3 Implementations — MPI Forum, 2016. [Online; accessed 28-February-2017].

[FV13] Gary Funck and Nenad Vukicevic. GNU UPC (GUPC) 4.9.0.1 User Manual. Online, 10 2013.

[FW97] Jeremy D. Frens and David S. Wise. Auto-blocking Matrix-multiplication or Tracking BLAS3 Performance from Source Code. *SIGPLAN Not.*, 32(7):206–216, June 1997.

[GAS17] GASPI: Global Address Space Programming Interface, Specification of a PGAS API for communication, Version 17.1. Technical report, 02 2017.

[GGDS+12] Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Touriño, and Katherine Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 100:1–100:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[Grü12] Daniel Grünewald. BQCD with GPI: A case study. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 388–394, July 2012.

[GS94] Himanshu Gupta and P. Sadayappan. Communication Efficient Matrix Multiplication on Hypercubes. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 320–329, New York, NY, USA, 1994. ACM.

[Hay09] Simon S. Haykin. *Neural Networks and Learning Machines.* Prentice Hall, McMaster University, Ontario Canada, 3rd edition, 2009.

[HLJT93] Steven Huss-Lederman, Elaine M. Jacobson, and Anna Tsao. Comparison of Scalable Parallel Matrix Multiplication Libraries. In *Proceedings of the Scalable Parallel Libraries Conference, Starksville, MS*, pages 142–149. Society Press, 1993.

[Hoc91] J. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[HPDC09] L.P. Hewlett-Packard Development Company. HP UPC/HP SHMEM User's Guide. Technical report, Hewlett-Packard Development Company, L.P., 10 2009.

[HW10] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[Inc] Mellanox Technologies Inc. Introduction to InfiniBand. Whitepaper 2003WP, Mellanox Technologies Inc, 2900 Stender Way, Santa Clara, CA 95054. Rev 1.90.

[ITW] Fraunhofer ITWM. Global Programming Interface (GPI) - User Manual. Version 1.0.

[JEG16] Prateek Joshi, David Millan Escriva, and Vinicius Godoy. *OpenCV By Example.* Packt Publishing Ltd, 2016.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[KCD+93] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM.

[KL99] Alexey Kolinov and Alexey Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1593 of *Lecture Notes in Computer Science*, pages 189–200. Springer Berlin Heidelberg, 1999.

[KN04] M. Krishnan and Jarek Nieplocha. SRUMMA: a matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 70–, April 2004.

[KP15] Janis Keuper and Franz-Josef Pfreundt. Asynchronous parallel stochastic gradient descent: A numeric core for scalable distributed machine learning algorithms. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, MLHPC '15, pages 1:1–1:11, New York, NY, USA, 2015. ACM.

[Kri07] David Kriesel. *A Brief Introduction to Neural Networks.* 2007.

[Kum14] P. Kumar. Communication Optimal Least Squares Solver. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, pages 316–319, Aug 2014.

[Lip88] Richard P. Lippmann. An introduction to computing with neural nets. *SIGARCH Comput. Archit. News*, 16(1):7–25, March 1988.

[LSF95] Jin Li, Anthony Skjellum, and Robert D. Falgout. A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies, 1995.

[MCASJ09] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[Mes94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[Mes09] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2. Specification, September 2009.

[Mes12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. Technical report, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[Mit97] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[ML09] Rui Machado and Carsten Lojewski. The Fraunhofer virtual machine: aÂ communication library and runtime system based on the RDMA model. *Computer Science - Research and Development*, 23(3-4):125–132, 2009.

[MLAP11] Rui Machado, Carsten Lojewski, Salvador Abreu, and Franz-Josef Pfreundt. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science - Research and Development*, 26(3-4):229–236, 2011.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[Ng16] Andrew Ng. Machine learning. Coursera Online Course, December 2016. Course offered by Stanford University.

[Nie15] Michael A. Nielsen. Neural networks and deep learning. online, 2015.

[Nis03] S. Nissen. Implementation of a Fast Artificial Neural Network Library (fann). *Report, Department of Computer Science University of Copenhagen (DIKU)*, 31, 2003.

[NPT+06] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[NR98] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[NS92] Tomas Nordström and Bertil Svensson. Using and designing massively parallel computers for artificial neural networks. *J. Parallel Distrib. Comput.*, 14(3):260–285, March 1992.

[NT02] Ju J. Krishnan M. Palmer B. Nieplocha, J. and Tipparaju. The Global Arrays User's Manual. Technical Report PNNL-13130, Pacific Northwest National Laboratory, 2002.

[Ode13] Lena Oden. GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 461–470, 2013.

[OKF14] Lena Oden, Benjamin Klenk, and Holger Fröning. Energy-efficient Stencil Computations on Distributed GPUs Using Dynamic Parallelism and GPU-controlled Communication. In *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 31–40, Piscataway, NJ, USA, 2014. IEEE Press.

[Ope13] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2013.

[Ope15] OpenSHMEM Application Programming Interface Version 1.2. online, March 2015.

[Pad11] David Padua, editor. *Encyclopedia of Parallel Computing*. Springer, 1 edition, October 2011.

[Pfr10] Dr. Franz-Josef Pfreundt. The Building Blocks for HPC: GPI and MCTP - Efficient Scalable Multicore. Technical report, Fraunhofer ITWM, 2010.

[PLWH03] Mark Pethick, Michael Liddle, Paul Werstein, and Zhiyi Huang. Parallelization of a backpropagation neural network on a cluster computer. In *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[pro14] Berkeley UPC project. Berkeley UPC User's Guide version 2.20.0, 12 2014.

[Rei10] John Reid. Coarrays in the Next Fortran Standard. *SIGPLAN Fortran Forum*, 29(2):10–27, July 2010.

[RHW88] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[Roj96] Raúl Rojas. *Neural networks*. Springer, Berlin ; Heidelberg [u.a.], 1996. Hier auch später erschienene, unveränderte Nachdrucke.

[RR10] T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.

[RRP14] Tiberiu Rotaru, Mirko Rahn, and Franz-Josef Pfreundt. MapReduce in GPI-Space. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 43–52. Springer Berlin Heidelberg, 2014.

[RS97] R O Rogers and D B Skillicorn. Strategies for parallelizing supervised and unsupervised learning in artificial neural networks using the BSP cost model. 1997.

[SAB+10] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. Technical report, Toronto, Canada, June 2010. APGAS model.

[Sam59] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, pages 71–105, 1959.

[SBG+10] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1947–1950, May 2010.

[Sch14] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[Sei04] Udo Seiffert. Artificial neural networks on massively parallel computer hardware. *Neurocomputing*, 57:135 – 150, 2004. New Aspects in Neurocomputing: 10th European Symposium on Artificial Neural Networks 2002.

[SGI13] SGI. Unified Parallel C (UPC) User Guide, 01 2013. 007-5604-005.

[Sha13] Wittmann M Kreutzer M Zeiser T Hager G Wellein G Shahzad, F. PGAS implementation of SpMVM and LBM using GPI. In Jackson A & Johnson N Weiland, M, editor, *Proceedings of the 7th International Conforence on PGAS Programming Models PGAS2013, 3./4. October 2013, Edinburgh, Scotland, UK.*, pages 172–184, Edinburgh, Scotland, UK, October 2013. The University of Edinburgh. ISBN 978-0-9926615-0-2.

[SJML11] Christian Simmendinger, Jens Jägersküpper, Rui Machado, and Carsten Lojewski. A PGAS-based Implementation for the Unstructured CFD Solver TAU. In *5th Conference on Partitioned Global Address Space Programming Models*, Conference Proceedings online, Tremont House, Galveston Island, Texas, USA, October 2011. Rice University, Houston, Texas, USA.

[SRG15] Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald. The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures. In Michael M. Resch, Wolfgang Bez, Erich Focht, Hiroaki Kobayashi, and Nisarg Patel, editors, *Sustained Simulation Performance 2014*, pages 17–32. Springer International Publishing, 2015.

[STG+14] Vijay A. Saraswat, Olivier Tardieu, David Grove, David Cunningham, Mikio Takeuchi, and Benjamin Herta. A Brief Introduction To X10 (For the High Performance Programmer), 2014. [Online; accessed 06-February-2015].

[STW+10] Dirk Schmidl, Christian Terboven, Andreas Wolf, Dieter an Mey, and Christian Bischof. How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 0:29–37, 2010.

[UPC13] UPC Consortium. UPC Language Specifications, v1.3. Tech Report LBNL-6623E, Lawrence Berkeley National Lab, 2013.

[Vam15] Vampir 9.0 user manual. Technical report, GWT-TUD GmbH, November 2015.

[vdGW95] Robert A. van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical report, Austin, TX, USA, 1995.

[Wei07] Michele Weiland. *Chapel , Fortress and X10 : novel languages for HPC*. UoE HPCx Ltd., 2007.

[WJJ13] M Weiland, A Jackson, and N Johnson, editors. *The GASPI API specification and its implementation GPI 2.0*, volume Proceedings of the 7th International Conference on PGAS Programming Model, At Edinburgh, UK, October 2013. 7th International Conference on PGAS Programming Models.

[YBMCB14] Chaoran Yang, Wesley Bland, John Mellor-Crummey, and Pavan Balaji. Portable, MPI-interoperable Coarray Fortran. *SIGPLAN Not.*, 49(8):81–92, February 2014.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM*, pages 10–11, 1998.

[ZSS06] Zhang Zhang, J. Savant, and S. Seidel. A UPC runtime system based on MPI and POSIX threads. In *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, pages 8 pp.–, Feb 2006.

[ZWLS10] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603. Curran Associates, Inc., 2010.