Gilbert: A distributed sparse linear algebra environment executed in massively parallel dataflow systems

by

Till Rohrmann

Matriculation Number: 343756

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Database Systems and Information Management

Master Thesis

18. August 2014

Supervised by:
Prof. Dr. Volker Markl
Prof. Dr. Odej Kao

Assistant supervisor:
Dipl. Inf. Sebastian Schelter

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, August 18, 2014 Till Rohrmann

Acknowledgments

This thesis arose in the context of the master's degree at the Database Systems and Information Management Group (DIMA) of the Technische Universität Berlin. I would like to express my deep and sincere gratitude to my adviser Dipl. Inf. Sebastian Schelter for his encouraging and personal guidance throughout this thesis. He was a steady source of thought-provoking impulses and provided me with untiring help, which led this thesis onto the right path. Moreover, I would like to thank him for reviewing my manuscript and giving me constructive criticism and excellent advice.

My earnest thanks goes as well to my supervisor Prof. Dr. Volker Markl for offering me the possibility to work on a fascinating topic and to gain insight into the current state of research on the field of scalable numerical data processing. I also want to thank Prof. Dr. Odej Kao for taking the time to review and evaluate this thesis. Moreover, I would like to thank the Database Systems and Information Management Group for providing me with the essential computational resources needed for the evaluation of Gilbert.

Last but not least, I would like to thank my family and friends for their continuous and sincere support during all these years.

Abstract

In recent years, the generated and gathered data has increased at an almost exponential rate. At the same time, people realized its usefulness in terms of insights it can provide. However, lifting the true treasures requires powerful analysis tools, since the insights are buried deep below a pile of meaningless data. Unfortunately, our analytic capacities did not scale well with the growing data. Most of our existing tools run only on a single computer and thus are limited by its memory. The most promising remedy to deal with large-scale data is to exploit parallelism.

We propose Gilbert, a distributed sparse linear algebra system, to solve the imminent lack of analytic capacities. Gilbert offers a MATLAB®-like programming language for linear algebra programs, which are automatically executed in parallel. Transparent parallelization is achieved by compiling the linear algebra operations first into an intermediate representation. This language-independent representation is amenable to high-level algebraic optimizations. Different optimization strategies are evaluated and the best one is chosen by a cost-based optimizer. The optimized intermediate representation is then transformed into a suitable format for parallel execution. Gilbert generates execution plans for Spark and Stratosphere, two massively parallel dataflow systems, and can easily be extended to support further parallel execution engines. Distributed matrices are represented by square blocks to guarantee a well-balanced trade-off between data parallelism and data granularity.

Exhaustive evaluation indicates that Gilbert is able to process varying amounts of data exceeding the memory of a single computer on clusters of different sizes. Three famous machine learning (ML) algorithms, namely PageRank, k-means and Gaussian non-negative matrix factorization (GNMF), were implemented with Gilbert. That emphasizes Gilbert's support for distributed iterations, an essential prerequisite for a broad set of ML algorithms. The performance of these algorithms is compared to optimized implementations based on Spark and Stratosphere. Even though Gilbert is not as fast as the optimized algorithms, it simplifies the development process significantly due to its high-level programming language.

Zusammenfassung

In den vergangenen Jahren ist die Menge der erzeugten und gesammelten Daten regelrecht explodiert. Fast konstante Wachstumsraten erzeugen ein exponentielles Ansteigen der aufgezeichneten Daten. Mit zunehmender Größe hat man schnell den Nutzen der Daten erkannt. Jedoch werden leistungsfähige Analysewerkzeuge benötigt, um gewinnbringende Erkenntnisse aus den Daten zu extrahieren. Unglücklicherweise haben sich unsere Analysefähigkeiten nicht im gleichen Maße wie das Datenvolumen entwickelt. Die am meisten verwendeten statistischen Programme können immer noch nur auf einem Computer ausgeführt werden. Dadurch kann man nur Datenmengen verarbeiten, die in den Hauptspeicher eines einzelnen Rechners passen. Die vielversprechendste Lösung, um der Datenflut Herr zu werden, stellt die parallele Ausführung unserer Analysewerkzeuge dar.

Wir stellen Gilbert, eine verteilte Umgebung für lineare Algebra dünnbesetzter Matrizen, als Lösung des oben genannten Problems vor. Gilbert bietet eine MATLAB®-artige Programmiersprache zum Erstellen von Programmen der linearen Algebra, die automatisch parallelisiert werden. Die Parallelisierung wird erreicht, in dem Gilbert das eingegebene Programm zuerst in ein Zwischenformat übersetzt. Diese Darstellung ermöglicht die sprachunabhängige Optimierung der Operatoren. Ein kostenbasierter Optimierer bewertet verschiedene Optimierungsstrategien und wählt das beste Resultat aus. Das optimierte Zwischenformat wird anschließend in ein zur parallelen Ausführung geeignetes Format übersetzt. Gilbert erstellt Ausführungspläne für Spark und Stratosphere, zwei massiv parallele Datenflusssysteme, und kann sehr einfach um weitere Ausführungssysteme erweitert werden. Verteilte Matrizen werden durch quadratische Blöcke dargestellt, was eine gute Balance zwischen Datenparallelität und Datengranularität garantieren.

Gründliche Untersuchungen haben ergeben, dass Gilbert gut geeignet ist, Daten verschiedener Größe, die die Speicherkapazität eines einzelnen Rechners übersteigen, auf einer variierenden Anzahl an Rechnern zu verarbeiten. Drei bekannte Algorithmen des maschinellen Lernens, PageRank, k-means und Gaussian non-negative matrix factorization (GNMF), wurden mittels Gilbert implementiert. Die erfolgreiche Implementierung belegt Gilberts Fähigkeit, verteilte Iterationen ausführen zu können. Dies ist eine essentielle Voraussetzung um eine Vielzahl an Algorithmen des maschinellen Lernens implementieren zu können. Die Laufzeit der Algorithmen, die mittels Gilbert implementiert wurden, wurde mit der Laufzeit von optimierten Algorithmen verglichen. Die Ergebnisse zeigen, dass Gilbert nicht so schnell wie die optimierten Algorithmen ist. Die Laufzeiteinbußen werden jedoch durch die Benutzerfreundlichkeit und die einfache Programmierung kompensiert.

Contents

Outline of Thesis

Part I: Introduction & Related Work

Chapter 1: Introduction

This chapter gives an introduction to the field of large-scale data processing. It points out the problems people are currently facing when trying to analyze big data. It further explains the necessity of Gilbert as a scalable data analysis tool to solve the presented challenges.

Chapter 2: Problem Statement

This chapter specifies the goals of this thesis. More precisely, it defines the problem Gilbert tries to solve, specifies the necessary features and describes the solution approach. Moreover, it determines the evaluation criteria to measure the overall success of the work.

Chapter 3: Related Work

This chapter presents the related work on the field of large-scale data processing. The field can be roughly distinguished into distributed data processing systems, distributed numerical computing systems, database systems, specialized distributed computing frameworks and explicit parallelization.

Part II: Theory & Concept

Chapter 4: Gilbert Language

This chapter defines Gilbert's programming language, which is strongly inspired by MATLAB®. Furthermore, it explains Gilbert's fixpoint abstraction, which serves as a loop mechanism.

Chapter 5: Gilbert Typing

This chapter explains the assets and drawbacks of static vs. dynamic typing. Furthermore, it emphasizes the importance of type information for the execution of Gilbert programs. It explains how type information can be automatically inferred without having to clutter the programming code. It also shows how type inference can be used to infer matrix sizes.

Chapter 6: Intermediate Representation

This chapter defines Gilbert's intermediate representation for linear algebra programs and explains its benefits. It also gives an example of how the compiling process proceeds.

Chapter 7: Gilbert Optimizer

This chapter describes Gilbert's optimizer and the employed optimization strategies. The matrix multiplication reordering and transpose pushdown optimization strategy are explained in detail.

Chapter 8: Gilbert Runtime

This chapter explains the internal data structures used to represent distributed matrices and vectors. Furthermore, it describes different blocking schemes for matrices and weighs the assets and drawbacks for each scheme up. It also explains how the different linear algebra operations are implemented within massively parallel dataflow systems, such as Spark and Stratosphere.

Part III: Architecture & Implementation

Chapter 9: Architecture

This chapter presents Gilbert's layered architecture. It further explains the design decisions, which have been made to make Gilbert easily extensible.

Chapter 10: Implementation

This chapter covers the implementation details. It explains which libraries are used for the linear algebra operations and what kind of execution engines are currently supported.

Part IV: Evaluation & Conclusion

Chapter 11: Evaluation

This chapter evaluates how well Gilbert meets the defined goal. It assesses Gilbert's scalability, optimizer and blocking scheme for matrices. Moreover, algorithms implemented with Gilbert are compared to their optimized versions to gauge Gilbert's performance. At last, the performance of Gilbert's math back ends, Breeze and Mahout, are evaluated.

Chapter 12: Conclusion

This chapter closes the thesis with drawing a conclusion of the obtained results and giving an outlook on future developments.

Appendices

Appendix 1: Gilbert's Mathematical Operations

This appendix lists all available mathematical operations of Gilbert. This includes the primitive linear algebra operators as well as the supported functions.

Part I

Introduction & Related Work

1 Introduction

"The important thing is not to stop questioning. Curiosity has its own reason for existing."

—Albert Einstein, (1879 - 1955)

The Sloan Digital Sky Survey (SDSS), an imaging and spectroscopic redshift survey, was initiated in the year 2000. Only a couple of weeks after its start, its telescope had already produced more data than in the entire history of astronomy before. Nowadays, its archive is filled with about 140 terabytes of sky images in different spectra. The successor telescope of SDSS, the Large Synoptic Survey Telescope in Chile, will collect this amount of data every five days, once put into operation [25].

The fact that we are gathering more and more data at an ever-increasing pace is not only observable in the astronomy but also in every other aspect of our daily lives. It is estimated that we are producing 2.3 EB of information each day, which is roughly 10000 times the data stored in the Library of Congress [2]. Actually, we have created 90% of mankind's data during the last two years [42]. Rightly so, we can claim to live in the information age, where data became a ubiquitous yet precious resource for companies and people alike.

Since the beginning of the information age, our environment became more and more infused with digital technology gathering, processing, storing and broadcasting information of various kinds. Nowadays, computing devices are almost omnipresent and many can hardly imagine a world without them anymore. We can find information technology in our cars making our ride safer, in our washing machines controlling the proper cleansing and in our mobile phones acting as our personal assistant, just to name a few. Apart from the personal use, digital devices found their way into almost all important branches of science and industry, such as finance, engineering, health and commerce.

The information technology's key to success and the reason for its quick adoption is its unprecedented amplification in computing, storage and telecommunication capacity over the last couple of decades. The growth has been mainly spurred by the technological progress on the fields of integrated circuits, broadcasting technology and algorithms. Since digital devices are strongly linked to the semiconductor technology, the aforementioned capacities exhibit a similar exponential growth rate [39] as Moore [55] predicted for the number of components on integrated circuits. It is predicted that mankind will have produced roughly 40 ZB of data by the year of 2020 [13]. Recently, the term "Big Data" was coined to describe the steadily growing accumulation of information.

In fact, there is an ongoing debate on what Big Data actually denotes. The sudden appear-

ance of the term, which resembles strongly a buzz word nowadays, might suggest that Big Data is something revolutionary. But since its beginnings, computer science actually occupied itself with the automatic processing of data. Even before the term Big Data came up, there were people, like the database community, who handled massive amounts of data. Thus, one could argue that Big Data is a mere marketing word, describing an evolutionary aspect of computer science. What people could agree on, is that Big Data solutions differ from other IT solutions in terms of *Volume*, *Velocity*, *Variety* and *Veracity*, also denoted as the four Vs.

Volume describes the aspect of storing and processing enormous amounts of data. Of course, the concrete amount of data always has to be seen in the temporal context. What today is considered to be Big Data is probably ridiculously small in 10 years. Velocity means that data is arriving more and more rapidly and that the system has to cope with that. It might even be necessary to support streaming and achieve real-time computing, as it is required for the financial sector. Variety alludes to the problem of integrating data of various kinds and from various sources. It is very common to work on structured as well as unstructured data. For example, a medical record not only consists of the diagnosis written by the doctor, but it might also include x-ray files from the last screening. Veracity is the last dimension and says that not all data can be taken as gospel. The system has to handle possibly incorrect and imprecise data and must be able to distinguish it from reliable data. The first three Vs were first introduced by Gartner [31].

In order to develop Big Data solutions, one has to solve multifaceted problems. The necessary technology stack comprises the recording of data, the data cleaning, the meta-data generation, the representation and integration with other data sources, the analysis and modeling of the actual problem as well as the interpretation. Each task for itself is highly complex and deserves an individual paper. For us, the data analysis part is the key aspect in solving Big Data problems. Henceforth, we will concentrate on how to gain valuable insights from a huge data set.

Once big data sets were amassed, people quickly recognized that these sets contain valuable information they only have to harness. Unfortunately, the search for useful patterns and peculiarities strongly resembled the search for a needle in a haystack. The quest requires sophisticated tools being able to analyze the vast amounts of data. Most of these tools are based on statistics to extract interesting features. It is beneficial to apply these statistical means to the complete data set instead of smaller chunks. Even if the chunks cover the complete data set, important correlations between data points might get lost in smaller parts, if treated individually. Moreover, the statistical tools improve their descriptive and predictive results by getting fed more data. The more data is available, the more likely it is that noise cancels out and that the significant patterns manifest. However, these benefits come at the price of an increased computing time, which requires fast computer systems to make computations feasible.

The insights gained from collected data already help to govern business decisions, improve life quality or simply create new industries from scratch. For example, retail stores analyze their sales, customer, pricing and weather data in order to decide which products to offer or when to do a discount sale. That not only increases the revenue of the shops but also boosts the satisfaction of the customers by getting better offers. Police departments try to detect probable crime sites by extracting patterns from previously recorded criminal acts and then reinforce the policemen in that region [46]. The intelligent deployment of policemen improves security for citizens without having to hire more officers. Hospitals analyze their patients' records and

scientific studies in order to find the cancer treatment best complying with the specifics of the patient. The individual therapy maximizes the chance of cure [41]. Another example showing the benefits of data analysis made it even into a Hollywood film. The film Moneyball is based on the true story of the Oakland Athletics baseball team, whose management built an elite team in an unfavorable financial situation. They employed sophisticated data analysis methods to spot underestimated players, who they could cheaply recruit. That marked the start of sports statistics, which are nowadays a common tool for professional teams. These examples emphasize the importance and utility of information gained with analytic tools from gathered data. Interestingly, the IDC, an international market research firm, estimates that only 0.5% of the globally collected data is harnessed [30]. They further state that about 23% of today's data is worth being analyzed. Thus, there is still potential left for improvements.

What are the reasons for the huge gap between actual and dormant exploitation? One of the reasons is that we are lacking the tools to keep up the pace of how fast data is created and collected. Our analyzing methods do not scale well for the vast data sets. The more data a system has to process, the longer it will take to finish. Since the data is growing at an exponential rate, our analytic capacities have to improve at a similar speed to keep computations feasible.

In order to decrease the runtime of our analytic tools there are two adjusting screws. First of all, there are the algorithms. By finding an algorithm with a lower runtime complexity than an existing algorithm, e.g., for clustering, we can drastically decrease the required time. However, it is strongly doubted that it is possible to develop better algorithms for certain problems. In some cases, it is even proven that there exists a lower bound for any algorithm solving the problem. For example, it can be shown that the complexity of a sorting algorithm, which is based on comparing two of its elements to obtain their ordering, is in $\Omega(n \log n)$ with n being the number of elements. Thus, certain problems have an inherent limitation of how fast they can be computed.

The other way to speed up the analytic tools is to make the computer faster by vertical or horizontal scaling. To scale vertically means that we add more resources to a single computer. For example, the main memory or the frequency of a computer could be increased, giving them more computational power. In contrast to that, horizontal scaling means to add more computer nodes to a system. By having more than one node, the work can be split up and distributed across the nodes. Since each split is smaller than the original problem, the computer can process it faster.

In recent years, we have seen that clock rates of CPUs stagnated. Before, there was a simple receipt to increase the computing power of micro-controllers; increase the clock rate, which demands more power, and shrink the channel widths to mitigate for the increased power consumption. However, the shrinkage induces the problem of leakage, which increases the power demand again. The consumed power is limited by the amount of energy that can be dissipated and thus there is a technological limit for the increase of clock rate. When it became clear that the micro-controller would hit this so-called "power wall", one duplicated the micro-controller's functionality to support simultaneous execution of multiple applications and to harness the inherent parallelism of programs.

The emerging multi-core and distributed systems pose new challenges for programmers, since now they have to know about locking, deadlocks, race-conditions and inter-process communication in order to make most of the available hardware. Since they have to be able to

reason about interwoven parallel control flows, parallel program development is highly cumbersome and error-prone. Therefore, new programming models are conceived, a development that relieves the programmer from the tedious low-level tasks related to parallelization such as load-balancing, scheduling of parallel tasks and fault recovery. The programmer can concentrate on the actual algorithm and the goal he wants to achieve by using these new paradigms.

These are the reasons why Google's MapReduce [26] framework and its open source reimplementation Hadoop [6] became so popular among scientists as well as engineers. MapReduce is a programming framework for concurrent computations on vast amounts of data running on a large cluster. Its ingenious idea was to separate the computation into a *map* and a *reduce* phase. In the map phase, the input data set is split into elements which are all processed independently. Afterwards, the results of the mapper are grouped together and each produced group is given to a reducer. The reducer knows about all elements in his group and produces the final result. The strengths of MapReduce are that it is expressive enough to implement a multitude of different algorithms while facilitating at the same time parallel execution. In fact, the map phase is embarrassingly parallel and the reduce phase only needs a shuffling step prior to its parallel execution.

But still, MapReduce and other frameworks force the user to express the program in a certain way, which is often not natural or intuitive to a user coming from a different domain. This implies that one has to overcome a particular entry-barrier to use the system and this barrier might already be too high for some users. Furthermore, the actual program might become lengthy and complicated expressed within the programming model. This circumstance makes developing and debugging the program difficult as well as time-consuming and eventually expensive. Especially, in the field of data analytics and machine learning programs are usually expressed in a mathematical form. Therefore, systems such as MATLAB® [52] and R [72] are widely used and recognized for their fast prototyping capabilities and their extensive mathematical libraries. However, these linear algebra systems lack proper support for automatic parallelization on large clusters and are thus restricting the user to a single workstation. Therefore, the amount of processible data is limited to the size of the main memory, which constitutes a serious drawback for real-world applications. Moreover, data scientists are usually no experts in the field of distributed computing and people of the distributed computing community often do not know much about data mining. The group of people being experts in both fields is negligible small. Consequently, it is very difficult, time-consuming and costly to implement machine learning and analytic algorithms on distributed systems.

This problem would be mitigated by having a distributed sparse linear algebra system supporting a MATLAB- and R-like language. Assuming that such a system is realizable, then existing MATLAB- and R-code could be directly run in parallel. Furthermore, new distributed algorithms could quickly be implemented benefiting from the expressiveness of linear algebra. This would drastically speed up the application of machine learning and data analysis algorithms to web-scale data. Due to these reasons, a distributed sparse linear algebra environment, called Gilbert, is developed in the context of this thesis. In the following, we will present the approach, the encountered problems and the evaluation of our implementation.

2 Problem Statement

"Science never solves a problem without creating ten more."

—George Bernard Shaw, (1856 - 1950)

We observed the lack of severely needed analyzing tools capable of scaling to web-scale data. As delineated in chapter 1, there does not exist any transparent solution to deal consistently with data sets of different sizes. As soon as the data size exceeds the main memory, it becomes necessary to distribute the work in order to process it efficiently. However, this circumstance also implies to leave ones accustomed toolbox behind, because most of the tools do not support distributed execution natively. Especially the popular tools MATLAB and R only provide limited support for parallel execution. This fact poses a serious problem for data scientists who are challenged to analyze ever-growing data sets.

Since the amount of gathered data increases almost exponentially, it can be assumed that the demand for examining large data sets becomes even greater. In fact, the industry of data management and analytics is growing by leaps and bounds. Its value, currently estimated to be 100 billion $, shoots up by an annual rate of 10%, which is twice the rate of the whole IT sector [25]. The quest for valuable business information extracted from collected data is spurring this industry on. However, that upgrowth might come to a sudden stop if we cannot develop analytic tools which scale well with the data size and are easy to use for people occupied with data analytics.

We believe that a linear algebra environment, which incorporates a familiar programming language and automatic parallelization to seamlessly scale from megabyte to petabyte and beyond, is the best solution. Relieving the user from the tedious task of writing parallel code while offering a well known programming interface are the key aspects of a successful adoption by the analytics community. Of course, it is indispensable to achieve a descent runtime performance compared to explicitly parallelized algorithms.

As a solution for the aforementioned problems, we conceived Gilbert, a distributed sparse linear algebra environment, which is executed in massively parallel dataflow systems. Gilbert will be programmed using a subset of the MATLAB language. Therefore, it will be easy for people experienced in MATLAB programming to use our system without any further preconditions. We intend to develop Gilbert so that it can be easily extended to other programming languages such as R. Gilbert takes the MATLAB code and enables a seamless transition from local to distributed execution on the nodes of a cluster. Since we do not want to reinvent the wheel, we intend to delegate the parallel execution to established parallel dataflow systems.

In order to develop Gilbert, we have to deal with certain problems. At first, we have to

build a compiler for a subset of the MATLAB language. The compiler stack comprises the lexer, parser and compiler. For the parser, it is crucial to define the formal grammar of MATLAB. Since MATLAB does not disclose the internal definition of its language, we can only try to approximate it as closely as possible. In case that type information is needed for the parallel execution, we would also have to integrate a typing system. Originally, MATLAB is a dynamically typed language, which does not contain explicit type annotations. For the sake of compatibility, it will not be possible to add explicit type information to the language. Therefore, the necessary type information has to be gathered using a type inference algorithm. We have to look into different type inference algorithms to decide which one fits our needs best.

Once this information is extracted, the program can be compiled into an executable format. However, we want to design Gilbert as flexible as possible, allowing it to run its programs on different parallel execution engines. Therefore, we have to introduce an intermediate representation into which the compiler translates the Gilbert programs. That representation is the starting point for a subsequent translation step into the execution engine's format.

The intermediate representation fulfills two tasks. First, it hides details of the actually used front end language. Consequently, it is easier to add support for other linear algebra languages such as R. Secondly, it makes the program amenable to high-level linear algebra optimizations independently of the programming language. That, however, leads to the question of how to design the intermediate representation. We have to find universal abstractions for the different linear algebra operations. These abstractions have to be general enough to support optimization and expressive enough to represent a multitude of programs. Furthermore, it will be important to look into different optimization strategies for linear algebra problems and to decide which strategies perform best.

After the Gilbert program has been optimized, it has to be translated into a representation which is understood by the execution engine. As mentioned before, Gilbert will run its programs on a parallel dataflow engine. We intend to add support for Stratosphere [3] and Spark [80]. Both systems, inspired by MapReduce, are programmed similarly by defining a dataflow plan, which is evaluated lazily by the system. They offer an equivalent set of operators to build data flows. The research question will be how linear algebra operations can be mapped to operations supported by these systems. This question is also accompanied by how different data structures, such as matrices and vectors, can be represented in a way best suited for efficient parallel execution.

Last but not least, we will evaluate the performance of Gilbert. One crucial property is the scalability of the system. Since our goal is to develop a system for scalable data analytics, it should work on data sets of different sizes. A good indicator might be the matrix multiplication because it is one of the most expensive linear algebra operations and appears often in programs. In order to test the usability and expressiveness of Gilbert, we intend to implement some ML algorithms. Promising candidates might be PageRank, k-means and Gaussian non-negative matrix factorization (GNMF). Additionally, we will compare the performance of Gilbert implemented algorithms with specialized algorithms for Stratosphere and Spark, respectively. The effect of the optimization strategies will be evaluated by comparing them to the non-optimized version.

3 Related Work

"If I have seen further than others, it is by standing upon the shoulders of giants."

—Isaac Newton, (1642 - 1727)

The challenges entailed by harnessing vast amounts of data has propelled much research on the field of distributed computing as well as databases to subdue the looming data flood. In recent years, a couple of different programming paradigms and frameworks emerged, each with the goal to tackle the aforementioned problems. The different trends can be subsumed into the following categories: Distributed data processing systems, distributed numerical computing systems, database management systems, specialized distributed computing frameworks and explicit parallelization.

3.1 Distributed Data Processing Systems

In the last decade, MapReduce [26], created by Google, has been one of the most influential developments in the domain of distributed data processing systems. Their novel approach to look at the problem of program parallelization was so intriguing and successful that many researchers and engineers jumped on the bandwagon and spurred a whole new area of research. Initially, MapReduce was created because of the need to process a constantly increasing amount of data. Before, the common way to do processing tasks, such as index generation, data-mining and web log parsing, was to write specialized programs. Often, these programs needed to be parallelized, fault-tolerant, support load-balancing and data-distribution to work properly. Not only does the implementation of these features require a high level of expertise but it also inflicts significant costs in terms of labor and time. Therefore, Dean and Ghemawat conceived MapReduce.

MapReduce is inspired by the higher-order functions *map* and *reduce*. Both functions can be found in many functional programming languages. In the MapReduce framework, the user only has to provide code for the map and the reduce function. The description for the map and reduce function as well as the input data constitute a MapReduce job. The semantics are the following: Given a set of key-value pairs, map is called for each pair independently and produces a set of intermediate key-value pairs. The generation of this set is defined by a user defined function (*UDF*). After the map call, all key-value pairs are grouped according to their key value. The reduce operation is then called for each of these groups and generates a new set of key-value pairs. Even though this abstraction appears to be quite simple it is surprisingly powerful and a lot of algorithms can be expressed this way.

The execution model works as follows. The input data is stored in the Google file system [33]

(*GFS*), a distributed file system. Internally the data is divided into input splits, which are stored in a replicated fashion across the worker nodes. Each input split is processed by a map task. The map tasks are assigned to worker nodes, which have free mapper slots. Since the worker nodes also contain input data, the MapReduce scheduler tries to assign the tasks to worker nodes storing the respective input split. If this assignment is not possible, the scheduler tries to select a node which is located as close as possible to the data with respect to the network topology. Thereby, the required network communication will be minimized.

Once the map task has finished, it has produced a set of intermediate results, which is partitioned according to a given hash function. Each partition is stored locally in a temporary file and serves as the input of one reduce task. A reduce task, which is spawned on a worker node, retrieves its partitions from all map tasks. This shuffle step inflicts considerable network communication overhead. Then the key-value pairs are sorted with respect to the key value. Finally, the reduce function will be called on each group of key-value pairs. The result of the reduce tasks will be written to the distributed file system from where it can be used for another MapReduce job.

Each map call is executed independently and thus the map phase is embarrassingly parallel. Once the shuffle phase is done, the reduce functions can also be executed in parallel. Therefore, it is possible to automatically parallelize the execution of a MapReduce job and thus freeing the user from that tedious and cumbersome task. By having more map and reduce tasks than worker nodes, the framework achieves good load balancing, because the system can spread them evenly across all nodes. Furthermore, one can even achieve a high flexibility by defining many but short lasting tasks. The data distribution is controlled by GFS and thus it no longer concerns the user. Yet, it can be influenced indirectly by specifying the replication factor, for example. MapReduce also implements fault-tolerance by re-executing the failed tasks and those tasks whose input data is needed again but not retrievable. The system detects task failures by maintaining the internal state for each task. Moreover, it recurrently pings the worker nodes to check their health. If a worker nodes does not respond in a certain amount of time, all tasks scheduled to this worker node will be set to failed and eventually rescheduled.

MapReduce is a renunciation from prevailing programming paradigms by confining the user to the map and reduce function. The restricted model allows Dean and Ghemawat to efficiently encapsulate automatic parallelization, load balancing, data distribution and fault-tolerance into the framework. Therefore, the user no longer has to struggle with these technicalities and can instead concentrate on writing productive code. The ease of use and the fact that MapReduce scales well to clusters of thousands of machines are the reasons why the concept is so appealing.

However, there are also some deficiencies. MapReduce only offers map and reduce as higher-order functions. Several operations, for example a join between two data sets, are complicated to express with only these two functions. Furthermore, writing all results between two distinct phases to disk causes a lot of I/O. This approach is slow compared to keeping the results in memory. This deficiency becomes explicitly observable if one operates on a data set in an iterative manner.

Due to these problems, a new class of systems has recently been developed. These systems can be considered a generalization of MapReduce. One of them is Stratosphere [11], a distributed computing framework which employs PACTs (parallel contracts) [3]. Stratosphere adds additional 2*nd*-order functions, which are called input contracts, in order to improve the

expressiveness and efficiency of the MapReduce paradigm. The new higher-order functions are *join*, *cross* and *coGroup*. Furthermore, Stratosphere adds the concept of output contracts. Output contracts annotate input contracts with certain properties, such as key uniqueness, record cardinality or constant fields of the output. These properties are exploited by a cost-based optimizer to select the most efficient execution plan. Recently, the framework has been extended to support bulk and incremental iterations [28]. These extensions make Stratosphere applicable to machine learning and complex data analysis tasks.

The new higher-order functions allow to express certain problems occurring in data analysis more elegantly and succinctly. Additionally, it gives the system a higher level of abstraction on which it can apply optimizations. Consider, for example, the join operator. In MapReduce the straightforward implementation is the following: First one has to unify both input sources in the map phase, adding a tag in order to distinguish them in the reduce phase. In the reduce UDF, the input sources are separated according to these auxiliary variables and then joined manually. For the join operation all entries with the same key have to be loaded into memory, because, a priori, the entries are not sorted according to their keys. This process can cause some serious thrashing. Since the join logic is hidden within the UDFs, there is no possibility for the system to automatically optimize this operation. Yet, there are ways to optimize join [15] and multi-way join [1] operations on MapReduce. But they either require extensive hand-tuning by the user or extensions to the MapReduce programming model to work. In contrast, the Stratosphere system is aware of the join operation and can choose the best execution plan for it. Depending on the size of the inputs, either the sort-merge join or the hash join algorithm performs better.

Stratosphere is programmed by using the native Java or Scala API to specify a dataflow. It is the representation of the computation at the operator-level of the system. A dataflow is an directed acyclic graph (DAG), whose nodes are the operators and edges denote the data flows between the operators. A transformed version of the DAG is then given to Nephele [75], the parallel execution engine.

An outstanding property of Stratosphere is that it uses database-inspired pipelining to reduce data materialization and thus costly I/O. First of all, it chains as many operators as possible. Chaining means that multiple operators are packed into one task so that intermediate results are directly fed to a subsequent operator on the same worker node. For example, multiple map operations can be smoothly chained. Furthermore, the system always tries to deploy succeeding tasks so that the intermediate results can directly be forwarded. Since Stratosphere runs in a JVM, excessive object creation can cause a significant performance loss due to the necessary garbage collection. Therefore, the system reuses existing objects wherever possible, so keeping the created number of objects low. In the event of memory shortage, intermediate results will be gently spilled to disk. Additionally, Stratosphere supports out-of-core algorithms such as external sort or hybrid-hash join to deal with massive amounts of data.

Stratosphere offers a powerful programming abstraction to easily implement parallel programs. The extension of the MapReduce paradigm and the integration of database features make it a promising candidate to supersede MapReduce as the prevalent distributed data processing system in the future. The iteration support helps to make Stratosphere a more general purpose processing system than MapReduce. However, there are still some aspects which have to be improved. First of all, Stratosphere currently lacks a proper fault tolerance. Secondly, it is not well geared towards supporting multi jobs and multi user scenarios. And last, the system

lacks the primitives, such as linear algebra data types and their operations, to easily implement data mining and machine learning algorithms based on linear algebra.

Another distributed data processing system is Spark [80]. Spark was developed to tackle the blind spot of MapReduce: Its inefficiency when it comes to iterative computations. The system is build around resilient distributed datasets (RDD) [79], a distributed memory abstraction. It allows programmers to execute in-memory computations on a large cluster with fault-tolerance. A Spark application is controlled by a driver program, which orchestrates the parallel operations on the cluster.

The programming API offers similar higher-level functions to manipulate the underlying RDD like Stratosphere. The operations can be distinguished into transformations and actions. Transformations take an existing RDD and create a new one. Its execution, though, is deferred until an action is called. Internally, transformations construct a directed acyclic graph (DAG), which is used to control the parallel execution. Amongst others *map*, *join* and *cartesian* belong to the group of transformations. Actions return a value to the driver program. Possible return values can be the cardinality of the RDD or the result of writing the RDD to disk, for example.

Spark also supports two restricted forms of shared variables. Broadcast variables are used to send and store read-only data to worker nodes so that it does not have to be shipped with the tasks. This mechanism reduces the communication overhead if the broadcast variable is used across several tasks. For example, the data points in a k-means clustering algorithm should be broadcasted because it stays constant throughout the execution of the algorithm. Another type of shared variables are accumulators. Accumulators can only be changed by applying an associative operation. Therefore, they can easily be implemented in parallel. A counter, which is incremented by all worker nodes processing a certain input item, would be a use case for accumulators.

Spark was constructed with an efficient iteration mechanism in mind. Therefore, the system offers a flexible execution pipeline, which can contain an arbitrary sequence of RDD operations. This property is unlike MapReduce where a single job can only contain a map and a reduce phase. All of Spark's distributed data is represented by RDDs, which are kept inherently in memory. This concept also applies to iterations. All the loop's computations are realized with RDDs and thus the loop's state is kept in memory between subsequent iterations. The usage of RDDs frees the loop from costly I/O as it would happen in MapReduce where each iteration would be an individual job.

Fault-tolerance is implemented by storing the lineage of each RDD. The lineage is the information how to build the RDD from its predecessors. In case of a lost RDD, e.g., due to a machine outage, the system can backtrack the RDDs which were used to construct the lost RDD. Once it found a retrievable state, the computation is reissued from this point on. Compared to checkpointing, which stores the complete state at a specific moment, lineage has far smaller memory footprint. But Spark also supports traditional checkpointing, which can be triggered by the user.

Spark does not only offer a native Scala API but also higher level languages, which are built on top of Spark. Apache Shark [77] is one of these languages. Similar to Apache Hive [7], Apache Shark allows to query data stored in the Hadoop distributed file system with a SQL-like language. Apache Shark allows people only acquainted with SQL to employ the Spark system.

The Spark project is revolutionary, because it adds efficient loop support by means of RDDs to the MapReduce world. This feature makes it well suited for a wide variety of domains where complex tasks, such as data mining, machine learning or statistics, have to be performed. For example, logistic regression is executed up to 100 times faster than on Hadoop. Additionally, Spark has a fully functional fault-tolerance, which can be geared towards the requirements of the user. It can either be lightweight storing only the lineage information or heavy in the sense that it copies the whole system state. However, Spark also have some weak points. Unlike Stratosphere, it does not have a cost-based optimizer, which can select execution plans depending on the applied strategies. Furthermore, the system lacks support for out-of-core operations and does not spill data to disk in case that it runs out of memory. This circumstance implies that the maximal size of data a job can process is limited by the amount of available main memory. Since the hard disk space is usually several magnitudes greater, set of solvable problems is unnecessarily limited. Finally, Spark lacks an intuitive MATLAB-like language and, thus, it requires a considerable expertise to code well performing algorithms.

There are also other notable projects trying to amend the deficiencies of MapReduce. The HaLoop [18] project adds loop support to Hadoop. In order to speed up the iterations, it allows to cache loop invariant data. Furthermore, it makes the task scheduler iteration-aware so that tasks of subsequent iterations are scheduled on the same worker node as their predecessors. This scheduling increases data locality and consequently decreases the communication overhead. All these features come with the same fault-tolerance support like the original Hadoop implementation.

A similar project is Twister [27], which also adds loop support to the MapReduce framework. In contrast to HaLoop, Twister stores intermediate data of iterations in-memory. Furthermore, it distinguishes between static and dynamic data. The static data does not have to be loaded from their producers for every iteration again, thus decreasing the communication overhead. Another extension is the additional reduction phase "combine". Combine can be called after a reducer to reduce the produced results into a single value, which is accessible by the user program. This collective output can be used to steer the termination of a loop, for example. In contrast to the distributed file system based communication scheme in Hadoop and MapReduce, Twister relies on a publish-subscribe system for data and communication transfer. This feature further decreases the I/O overhead of storing data to disk and allows to directly send produced results from a mapper to a reducer. However, Twister does not have a fully functional fault-tolerance mechanism yet.

3.2 Distributed Numerical Computing Systems

SystemML [34] has the aim to make machine learning algorithms run on massive datasets without burdening the user with low-level implementation details and tedious hand-tuning. In order to achieve this goal, it provides a declarative higher-level language, called Declarative Machine learning Language (*DML*). With DML, the user writes linear algebra programs whose operations are automatically executed in parallel. DML is inspired by the R [72] language, which is the quasi standard among data scientists and statisticians. The language supports matrices and scalars as basic types. DML can be used to implement a wide variety of supervised and unsupervised machine learning algorithms.

The linear algebra operations are translated into a directed acyclic graph of high-level operators (*HOP-DAG*). This abstract representation allows to apply several optimizations such as

algebraic rewrites, choice of internal matrix representation and cost-based selection of the best execution plan.

Matrices are separated into quadratic blocks. Each block is uniquely identified by its row and column index. The block representation has the advantage to reduce the overhead inflicted by a cell-wise representation where for each entry a pair of indices has to be stored additionally. Assuming an entry is stored as a double, requiring 8 bytes, and 2 integers for the indices, requiring 4 bytes each, this scheme would double the memory footprint of a matrix. That would cause serious memory shortages and is thus not feasible. Furthermore, the block representation allows to choose an memory-efficient internal representation depending on its sparsity. If a block has only few non-zero entries, then it is represented as a sparse matrix, otherwise a dense matrix is employed. Depending on the choice of the internal matrix representation, SystemML picks the right block-level operations for best performance.

A crucial operation is the matrix multiplication, which appears in many machine learning algorithms and often inflicts the highest runtime costs. Therefore, SystemML offers two different execution strategies. The replication based strategy broadcasts the smaller matrix to all worker nodes, which contain blocks of the bigger matrix. Then the result block can be computed with this information. The other strategy is based on the outer-product representation of a matrix multiplication. Here, the columns of the left matrix are joined with the rows of the right matrix. The outer-product computes a set of intermediate matrices, which have to be added up to deliver the final result. Both strategies exhibit different runtime characteristics with respect to the file system and network communication costs. File system costs are caused by reading and writing data to the distributed file system, which occurs at the beginning and ending of each map- and reduce-phase of a MapReduce job. Network communication costs are the dominant factor and thus the system chooses the best matrix multiplication strategy with respect to the estimated network costs.

Once the optimization is applied, the HOP-DAG is translated into a directed acyclic graph of low-level operators (*LOP-DAG*). These low-level operators can be directly mapped onto MapReduce jobs, which represent the final execution format. Between two MapReduce jobs the output of the former, which is often the input of the latter, is written to disk and then read again. This I/O causes a considerable performance loss and therefore SystemML tries to minimize the number of jobs. It uses piggybacking to aggregate the low-level operators into as few map- and reduce-phases as possible. Additionally, the system employs *local aggregators* which reduce data in the reducer by combining produced results. The idea is similar to the *combiner* concept [26] and also reduces the amount of I/O.

SystemML proves to scale well with an increasing number of worker nodes and an increasing amount of data. However, as it is not described in the paper, it lacks a proper iteration mechanism. It is not specified whether only loops with a static termination criterion, such as a maximum number of iterations, or also dynamic termination criteria are supported. In either case, the iterations have to be realized within distinct MapReduce jobs. This circumstance makes the loop very inefficient, because for each iteration the loop data has to be written to and read from disk. Furthermore, SystemML relies on the MapReduce framework with all of its deficiencies compared to the more powerful parallel dataflow systems Stratosphere and Spark. Stratosphere and Spark support in-memory storage of intermediate results. This approach significantly speeds up loop processing. Yet, SystemML is a very promising project heading in the right direction to make web-scale data analytics accessible to people not famil-

iar with distributed computing.

Ricardo [24] is part of the eXtreme Analytics Platform [10] (*XAP*) developed at IBM. XAP is developed with the intention to support deep analytics on large-scale data and comprises several modules. Ricardo uses existing technologies to implement a scalable system for statistical analysis. The statistical computations are done within the R ecosystem, because it has a rich library of analytic methods and a close-knit community of statisticians, who are not eager to adopt a completely new system. The problem, however, is that R works only on data which is stored in the computer's main memory. Hadoop [6], an open-source implementation of MapReduce, is a data processing system, which does not suffer from this problem. Yet it lacks the statistical functionality of R. The idea is to combine the strengths of the two systems: The data shipment is done by Hadoop and the actual analytic computation by R.

For the integration of both systems, Jaql (JavaScript Object Notation query language), which is part of XAP, is used. Jaql [12] constitutes a declarative high-level language for data-processing on Hadoop. Jaql provides a rich set of high-level operators such as *join*, *group by* and *transform* with which it is possible to easily and quickly define data-flows. Furthermore, it still gives you access to the underlying MapReduce code if needed. Therefore, it is flexible and powerful enough to be used as an interface between R and Hadoop.

The system is controlled by a R driver process. By using Jaql as a R-Hadoop bridge, the user can initiate the distribution, transformation and aggregation of data within Hadoop. Furthermore, the system supports to run R code on the worker nodes as data transformations. The calculated Hadoop results can also be collected at the R driver process once they have an appropriate size. This aspect makes it possible to apply statistical computations to large-scale data. Since all is done from within R, the user does not have to re-adapt and can profit from a huge code base of sophisticated statistical algorithms.

The combination approach enables the authors to quickly come up with a working system without reinventing the wheel. However, Hadoop is not integrated transparently into R. The user still has to specify explicitly the data-flow and the data distribution. This specification requires a substantial understanding of the working principles of MapReduce and of the used algorithm. Only then the user can efficiently distinguish between parallelizable and serial parts of the algorithm. Therefore, the system is not well suited for people only familiar with R.

RHIPE [36] is another project which tries to extend R to scale to large data sets. RHIPE follows the new statistical approach of divide and recombine (*D&R*). The idea is to split the examined data up into several chunks so that they fit in the memory of a single computer (*S-Step*). Then a collection of analytic methods is applied to each chunk without communicating with any other computer (*W-Step*). This approach makes the computation embarrassingly parallel. After each chunk is evaluated, the results will be recombined in an aggregation step to create the overall analytic result (*B-Step*).

This methodology strongly resembles the MapReduce principle and thus it is not surprising that the authors use the Hadoop system to execute RHIPE programs. In fact, the W- and B-Step can be directly mapped to map and reduce tasks, respectively. The actual statistical analysis is done by R. The user has to define R code for each step. First, the R code is used to split the data into chunks for the S-Step. This data is then distributed to the Hadoop worker nodes using Hadoop's distributed file system (HDFS). On the worker nodes, the R code for the W-Step is executed in the map tasks. This scheme places the whole analytic power of the R system with

all its proven libraries at the analyst's disposal. And finally, the recombination in the B-Step is carried out by applying the provided R code on the intermediate results of the W-Step.

The approach of RHIPE has the charm to be perfectly executable in parallel due to its D&R paradigm. However, this strict execution pipeline constrains the analysis process considerably. First of all, the statistical method, the analyst wants to apply, has to be suitable to be split up into a map phase where subsets of the data are processed independently and a reduce phase where the final result is processed from the intermediate results. For example, it works for the mean calculation of a data set if the chunks are of equal size. Yet, as soon as you need access to the whole data set to compute the exact result, as it is, e.g., the case for linear regression, one can in general only compute an approximation. This estimate might still be enough for most cases since statistics itself is by its definition an approximation. But first, the user has to conceive the data distribution, the data processing and the final recombination step. This approach strongly resembles programming a MapReduce job. Since not many data analysts are familiar with the concepts of MapReduce, it might hinder them to utilize the system.

A system, which integrates more seamlessly into the R ecosystem, is pR (parallel R) [61]. Again, the goal of pR is to let the statistician compute large-scale data without requiring him to learn a new system. pR achieves its goal by providing a set of specialized libraries which offer parallelized versions of different algorithms. In this way, the user only has to switch the used library and can directly benefit from the parallel processing power. Thus, the adoption requires only little code changes what makes it cost- and labor-efficient.

pR utilizes several methods to achieve parallelization. One approach is to integrate 3rd party libraries, containing specialized code, into the R session. Often, high performance computing (HPC) code is used, which is parallelized by using a message-passing system such as MPI [35, 47]. Using external libraries has the advantage to rely on well tested and efficient code. But MPI requires to have an HPC cluster with a high throughput network to run efficiently [69]. Furthermore, it assumes to have exclusive access to the computing resources. This exclusive access is usually not guaranteed in a multi-user environment. Therefore, this approach is unsuitable to be used on a shared commodity cluster.

pR also offers parallelization for the *lapply* method of R. R's lapply method takes a list of values and a user-defined function and applies this function on the list of values. Since this operation is embarrassingly parallel, it is a natural candidate for automatic parallelization. The parallelization is again achieved by using MPI to orchestrate the distributed execution.

pR impresses with its seamless integration into R so that the parallel execution of R code comes almost for free. The user neither has to learn a new system nor does he has to change a lot of code. But the downside is that not all operations in R are supported to be run in parallel. Moreover, MPI is used for the parallelization. MPI is well suited for HPC where a single large job is exclusively run on a set of machines. But nowadays, many data analysts share a cluster of commodity hardware and run several jobs concurrently, some of which are interactive and others take a long time to finish. This multi-job scenario interferes with the design of MPI. Furthermore, MPI inherently lacks often requested properties such as fault-tolerance, elasticity and reliability.

3.3 Database Systems

As the statistics community tries to enrich systems like R with massive data processing capabilities, the database community tries to extend database management systems (*DBMS*) with analytic functionality. It is already the case that the current SQL standard supports simple statistical computations such as linear regression, correlation coefficient and T-test functions. However, the system quickly reaches its limits, as soon as slightly more complex problem are addressed, such as classification. Cohen et al. [21] devised a set of properties modern DBMSs have to satisfy in order to meet the requirements of Big Data analysts. In order to further integrate statistical computations into the DBMS, they showed how to implement a matrix type and the corresponding matrix operations with SQL. Additionally, they implemented exemplarily the conjugate gradient method and Mann-Whitney U test. Due to the matrix operations the implementation is more elegant than with pure SQL. But still, the definition of statistical functions within this system is laborious and complex compared to R. Also, there is no easy way to migrate existing R code. A huge amount of work would be necessary to re-implement the algorithms statisticians are used to work with. For these reasons, it is doubtful if the statistics community will adapt this approach to implement large-scale data analytics.

Stonebraker et al. [66] have embarked on a similar path as Cohen et al. in the sense that they first attempted to specify the requirements for a science database, called *SciDB*. They identified support of multi-dimensional arrays to be essential for many scientific and industrial use cases. Therefore, they constructed SciDB around this concept as its integral data type. That makes it more natural to implement linear algebra operations within this system. SciDB tries to push computations to the data to improve its scalability. Stonebraker et al. stress similar to Cohen et al. the importance to operate on "in situ" data. That feature is very helpful for analysts to directly start working on the data instead of wasting time with tedious data preprocessing in order to load the data into a database.

The ideas proposed by Stonebraker et al. for a scientific database seem to be very promising. It will be interesting to see how the multi-dimensional array data model performs in real world scenarios. However, in the meantime it is important to further extend the statistical functionality. Only then, the system can get adopted by the statistics community.

The RIOT [81] (R with I/O Transparency) project is aimed to incorporate the advantages of a DBMS into R. The authors Zhang, Herodotou, and Yang have discovered that inefficient I/O of R causes a substantial performance loss when applied to large data sets. Since R loads all its data into main memory, the virtual memory mechanism has to swap data to and from the local disk as soon as the data amount exceeds the memory limit. This circumstance can cause thrashing of the system what ultimately leads to poor performance. The first approach to cope with this problem is to hand-tune the critical code sections. The hand-tuning, however, burdens the programmer. A second solution is to use I/O optimized libraries, which tackle the problem at the intra-operational level. This method, though, leaves out valuable chances for inter-operation optimization.

DBMS offer I/O-efficiency and have with their query optimizer a powerful tool at hand to optimize code on an inter-operation level. Therefore, the authors chose to solve the aforementioned problems by utilizing a DBMS, which they call RIOT-DB. To guarantee a quick and wide adoption among R users, they paid explicit attention to a seamless integration into the R environment. They implemented the RIOT-DB as an R package, which can be dynamically

plugged in. Additionally, it does not require any user code changes to benefit from its features. RIOT provides the basic set of R types: Matrices, vectors and arrays. They are implemented within the database as key-value pairs. Besides, the primitive matrix and vector operations are provided so that the R user never has to face any SQL code.

The authors identified the following I/O-inefficiencies in R, which can be mitigated by RIOT: R programs consist of a set of statements, which each produce a result. Usually these results are the input for latter statements and thus need to be stored as intermediate results. As long as the memory can hold all these values there is no problem. However, if not, the data has to be materialized on disk, which is expensive. By compiling the whole program into a database query, it can be efficiently pipelined. The inputs for each operator are computed when needed and thus the unnecessary materialization of intermediate results is avoided.

Unneeded computations are another inefficiency. These computations always appear if the calculation is continued with a subset of the former result. By deferring the computations until it is clear which elements have to be computed, the system can selectively evaluate expressions and save valuable resources. The postponing feature is almost naturally achieved by using database views during the query construction.

Data layout and consequently data access patterns constitute a further inefficiency. In R the access patterns are inherently sequential and data layouts are static. Yet it may be faster for certain operations, such as a matrix-matrix multiplication, to have, for example, a column- instead of a row-wise partition for the right matrix. This feature would significantly reduce the number of page faults. Unfortunately, the RIOT-DB implementation is not yet capable to dynamically adjust data layouts.

An extra improvement is the reordering of computations. Considering a threefold matrix multiplication, it is important to choose the right execution order. Depending on the size of the intermediate matrix, one or the other order might be infeasible to execute. An efficient system has to be able to make these kind of optimizations, which are similar to database query optimizations. But again, the current RIOT-DB is not capable of doing it yet.

Even though matrix representations in a relational database as a set of indices and a value are inherently inefficient as shown in [65], the pipelined execution and the inter-operation level optimization of the DBMS makes RIOT in many cases faster than R. Considering current developments of the statistical capabilities of DBMS as well as support for matrices and vectors, it is not far-fetched to believe that RIOT will significantly improve its performance in the future. But the system suffers currently from a non-negligible disadvantage when it comes to state of the art machine learning algorithms: Namely iteration support. There is no way described how to efficiently realize loops.

3.4 Specialized Distributed Computing Frameworks

Apart from these more general approaches to distributed data processing and numerical computing systems, quite a lot of specialized systems emerged. By restricting oneself to a more confined domain, it is often possible to find more efficient ways to solve a problem at the expense of generality.

Pegasus [43] is a programming model mainly intended for graph mining purposes. It is centered around the abstraction of a generalized iterative matrix vector multiplication (GIM-V), which can be found in many graph algorithms. The GIM-V operation can be efficiently repre-

sented by a map and a reduce task. Consequently, Pegasus implements it on top of Hadoop.

Bu et al. [19] remarked that there exist a multitude of more or less specialized programming models for the task of distributed machine learning. All of these systems exhibit a tight coupling of a solution's logical representation and physical representation. This coupling renders optimization difficult, because one is bound to the underlying runtime implementation and disregards alternative execution strategies. Moreover, the systems are mostly disjoint which implies that each framework has to be updated in order to profit from new optimization strategies. As a solution the authors propose to employ Datalog as a declarative language for the specification of higher level programming models such as iterative MapReduce or Pregel [49] within their system. Standard query optimization techniques are applied on this common intermediate representation and then it is transformed into a physical execution plan. This plan is executed by Hyracks [17], a data-parallel platform for data-intensive tasks. This approach has the advantage that a wider class of machine learning algorithms are efficiently supported within the same system, thus profiting from the same underlying infrastructure. However, at the moment the physical plans are still created by hand which makes the framework not applicable yet.

Apache Mahout [8] is a project offering a library of scalable machine learning algorithms. Many algorithms use the MapReduce paradigm to achieve scalability and are written for Hadoop. By using Hadoop, the actual execution plan of an algorithm has to be hand-tuned for the specific cluster and input size. Just recently, the Mahout project decided to steer away from developing isolated applications and instead follows now a more general approach to solve the problem of scalable machine learning. In the context of this re-orientation, Mahout developed a Scala DSL (domain specific language) to express linear algebra operations. The supported functionality is not identical but similar to the functionality of MATLAB and R. The Scala DSL can either be executed locally or in a distributed fashion using Spark. The basic idea of the Scala DSL, which is hiding the implementation details of the parallel logic behind a well understood linear algebra abstraction, is very similar to Gilbert's approach. However, Mahout forces the user to do their programming in Scala, which might be a serious obstacle for people coming from the MATLAB and R world.

3.5 Explicit Parallelization

A popular approach to develop parallel programs running on a large cluster of computers is to employ message passing. A message passing library offers primitives to send and receive messages to and from a remote host while hiding the used communication infrastructure. This transparency allows to interconnect a collection of possibly heterogeneous machines to a coherent computation unit, thus allowing to solve large computation problems in a cost-efficient manner. Message passing implementations usually support inter- as well as intra-node communication. The used transfer channel, for example, Ethernet, InfiniBand, Myrinet or shared memory, is chosen transparently by the library.

The message passing itself happens synchronously or asynchronously and the message can contain arbitrary data. However, the user has to specify explicitly what data shall be send at what time and to which host. Thus, he is fully in charge for the communication pattern. It implies that the user needs a thorough understanding of the algorithm in order to identify which code sections can be parallelized. Moreover, he has to make sure that there are not any deadlocks or race conditions between multiple parallel instances. In order to implement

desirable properties such as fault-tolerance, load balancing or elasticity, the programmer has to add even more code. Message passing only gives us the tools at hand to implement these features. These aspects make parallelization with message passing a very cumbersome and error-prone task, which requires a highly trained professional. In other words, message passing can be seen as the assembler language of parallel programming.

There are several implementations of a message passing system. One of them is Parallel Virtual Machine [32] (PVM), available for a wide variety of computer architectures. Another popular message passing system is Message Passing Interface [47] (MPI), which later supplanted PVM. Usually these libraries are used with languages such as C/C++ or Fortran. But there are also bindings for R, namely rmpi [59] and rpvm [60]. Since they are only wrappers for the respective MPI and PVM calls, they also suffer from the aforementioned problems.

R users are not trained to program parallel programs and therefore only few can take advantage of rmpi and rpvm. In order to alleviate this problem, Simple Network of Workstations [74] (SNOW) was developed. SNOW is built on top of message passing and achieves task- and data-parallelism. At the same time, it offers a more high-level interface for initiating parallel computations and thus is more user-friendly. The system is controlled from a R session which is executed on a master node. From there, the user can start parallel tasks on the worker nodes. The result of the computation is then transferred back to the master node. Even though the user does not have to program at the level of message passing, it still needs a considerable effort to parallelize code. At first, the user has to identify which code sections are task- or data-parallel. Then he has to add SNOW functions to implement the parallel execution. If he requires advanced features such as elasticity or fault-tolerance, then the user has to even further bloat his code.

For the other popular numerical computing environment out there, namely MATLAB, exists also a set of parallelization tools. The most popular are the MATLAB Parallel Computing Toolbox [51] and MATLAB Distributed Computing Server [50], which are developed by MathsWorks. The former permits to parallelize MATLAB code on a multi-core computer and the latter scales the parallelized code up to large cluster of computing machines. In combination, they offer a flexible way to specify parallelism in MATLAB code. They combine several paradigms for parallelization [64] with the intent to require only few changes to existing code.

Several MATLAB functions are extended to run in parallel on a multi-core or multi-processor system by supporting multi-threading. The degree of parallelism is controlled by the user through a MATLAB function. However, multi-threading is limited to a single machine. Since the prevailing trend is to employ large shared nothing clusters to solve computing intensive tasks, the authors implemented other parallelization means, too.

One of them is the exposure of MPI to the user. Because the MPI API gives users great power at hand, it also requires great responsibility of them. But MATLAB offers MPI's functionality through a simplified API, which assists the user in writing error free code. The MATLAB MPI functions encapsulate deadlock and mismatched communication detection. A mismatched communication occurs if there is a sender or a receiver whose communication counterpart (receiver or sender respectively) is missing. Yet, the MPI functions offer only a very low-level approach to code parallelization. Therefore, the MATLAB authors devised some higher-level constructs, too.

Distributed arrays are used to abstract the communication details of parallel computations

on arrays. Distributed arrays are conceptually equal to usual arrays, just with the slight difference that the applied operations are automatically executed in parallel. Most of the built-in functions are adapted for distributed arrays. Internally, they are built on top of MPI and the user only gets a coherent logical view of the complete array. The distributed arrays and their parallel executed operations are an example of how to exploit data parallelism.

Another type of parallelism is the task parallelism where independent tasks are exploited to be run in parallel. The for-drange mechanism allows to specify parallel for loops over distributed arrays. The for loop assigns iteration ranges to the individual workers. But the user has to make sure that the data required by the iteration is stored on the respective worker node. A requirement is that the iterations are independent and no communication between different workers occurs. Thus, the tasks to be executed have to be embarrassingly parallel.

A more sophisticated mechanism is the parallel for loop (parfor). The parfor is similar to the parallel for loop in OpenMP [22]. It transparently distributes work, iterations in the case of a for loop, among the available workers. Unlike OpenMP whose parallel for loop spawns multiple threads for each chunk of iterations, MATLAB spawns separate processes. This mechanism permits to let the different iterations run on different machines and thus supporting NUMA (non-uniform memory architecture) systems.

The MATLAB Distributed Computing Server provides the infrastructure to run MATLAB code in parallel on a large set of workers. For this purpose, a scheduler based on the JINI/RMI framework was developed. The scheduler takes a MATLAB job, comprising of a command set, and assigns it to an idling worker. The authors decided against the support of distributed file systems and instead chose a relational database to store and communicate task related data. Furthermore, the whole infrastructure was developed having small clusters in mind, with up to 128 nodes.

Hence, nothing was said about how well the MATLAB system scales to clusters with more than 128 nodes. Moreover, it was not stated whether the system supports a fault-tolerance mechanism or not. Supposedly, the fault tolerance can be achieved by using the relational database for checkpointing. But there might be scenarios for which this approach is not applicable. For example, if the memory footprint of the program is too huge, then the checkpointing time makes the computation infeasible. For long running jobs it is crucial to recover automatically from a machine failure without user intervention.

Besides MATLAB Parallel Computing Toolbox there are also other projects which try to parallelize MATLAB code. The most noteworthy candidates are pMatlab [16] and MatlabMPI [44] which shall be named here for the sake of completeness. Also, they are non-commercial and thus they stand out against MathsWorks product.

Part II

Theory & Concept

4 Gilbert Language

"The limits of my language means the limits of my world."

—Ludwig Wittgenstein, (1889 - 1951)

Prior to the actual start of development of Gilbert, we had to decide which higher-level language Gilbert should support. Since Gilbert aims to be a sparse linear algebra environment, R and MATLAB are the natural candidates. In fact, R and MATLAB are quite similar and are both widely used in academics as well as industry to develop mathematical programs. Due to their expressiveness and their rich library support with myriads of mathematical functions, they are just as well suited for quick prototyping as it is for full-fledged development. In combination with their ease of use, they quickly became the standard in industry for numerical processing. Our final choice fell on MATLAB, because of our higher familiarity with this language. Yet, we believe that there are no fundamental reasons which would prohibit the support of R. In fact, the translation from one language to the other should be straight-forward on the compiler level.

Another reason to imitate an existing language and not to devise a new language geared towards parallel execution is the laziness of people. Since its part of human nature to be sluggish, people are initially unwilling to learn new things or to re-adapt. Therefore, Gilbert was conceived to require as little re-adaption as possible of users familiar with MATLAB. We believe that this aspect is a crucial property of Gilbert in order to be successfully adopted by the statistical community. Furthermore, a neat side effect is that existing MATLAB code can almost seamlessly be ported to Gilbert and thus benefits instantaneously from the computational power of a large shared nothing cluster. This feature becomes particularly relevant considering the huge existing code base. Re-coding parts of it would be prohibitive, because of the required labor and provoked costs.

In the following sections, we will describe the key aspects of the Gilbert language and explain the design decisions taken. Subsequently, we will give a formal specification of the Gilbert language and describe the used parser.

4.1 Language Features

In order to not be overwhelmed by the complexity of MATLAB but still support a full functional subset of it, Gilbert was restricted to support the following language features and data types. We decided to keep the feature set as small as possible to concentrate on the parallelization instead of supporting hardly used language features. The elementary data type of linear algebra and therefore also of MATLAB is a matrix. Gilbert supports arbitrary 2-dimensional

matrices whose elements can be *double* or *boolean*. Vectors are not represented by a special type but instead are treated as a matrix. Additionally, scalar *double* and *boolean* values are supported. The reason to introduce a *boolean* type is the control flow of loops, being covered in a later paragraph. Since Gilbert has to interact with data stored on disk, strings are supported.

Gilbert also implements cell array types. A cell array consists of indexed data containers, called cells. Each cell can contain an individual data type. Therefore, cell arrays can be used to pass multiple data items combined as one argument to a function or to obtain multiple items from a function. The latter aspect is particularly important, because Gilbert only supports functions with a single return value. A cell array is defined using curly braces and commas to separate individual values. The cells can be accessed by an index appended in curly braces to the cell array variable. Listing 4.1 shows how to define and access them. The formal definition of the syntax of cell arrays can be found in section 4.2. In contrast to MATLAB's cell arrays, though, Gilbert only allows to have 1-dimensional arrays. However, this constraint does not impose a serious restriction, since all multi-dimensional arrays can be transformed into 1-dimensional arrays.

```
1 c = {true, 2*2, 'cell', 'array'};
2 b = c{1} & false; % = false
3 d = c{2} ^ 2; % = 16
4 s = {c{4}, c{3}}; % = {'array', 'cell'}
```

Listing 4.1: Cell array usage in Gilbert. Definition of a 4 element cell array which is accessed subsequently.

Gilbert supports the basic linear algebra operations defined on matrices and scalars. They include among others the common operations `+`, `-`, `/` and `*`, whereas `*` denotes the matrix-matrix multiplication and all other operations are interpreted cellwisely. The cellwise multiplication is indicated by a point preceding the operator as it is common for cellwise operations in MATLAB. Gilbert also supports comparisons operators such as `>`, `>=`, `==` and `~=`. These operators become handy for realizing dynamic termination criteria for loops. The full set of supported mathematical operations can be found in appendix 1.

Besides the basic arithmetic operations, the user can also define named functions and anonymous functions. This feature not only enables a better structure of the code but also allows to express 2nd-order functions. The syntax of anonymous functions adheres to the MATLAB syntax. An anonymous function which calculates the sum of its input squares could be defined as follows: `@(x,y) x*x + y*y`. A formal definition can be found in section 4.2.

An important aspect of MATLAB are loops. MATLAB permits the user to express `for` and `while` loops. However, these loops are quite powerful in the sense that they allow iterations with side effects. The problem of parallelization of iterations with side effects is that the referenced external state has to be maintained. This circumstance makes preprocessing and execution unnecessarily complex. For the sake of simplicity, Gilbert is limited to a different loop mechanism. Gilbert offers a fixpoint operator `fixpoint`, which iteratively applies a given update function f on the previous result of f, starting with an initial value x at iteration 0. Thus, the n^{th} iteration is equivalent to applying the function f n times to x:

$$n^{th} \text{ iteration} \equiv \underbrace{f(f(\ldots(f(x))\ldots))}_{n \text{ times}}$$

In order to terminate the fixpoint operation, the operator provides two mechanisms. First of all, the user has to specify a maximum number `m` of iterations after which the loop is terminated. This mechanism is henceforth denoted as the *static termination criterion*. Such a termination criterion is often not sufficient for machine learning algorithms, because they usually continue their computations until they have reached a certain convergence criterion. Since the number of iterations necessary to reach this state is not always known a priori, it is called the *dynamic termination criterion*. Gilbert offers support for the dynamic termination criterion. The user can provide a convergence function `c` to the fixpoint operator. The convergence function is called with the previous and current fixpoint value and returns a boolean value, indicating whether the termination criterion has been fulfilled or not. Thus, the fixpoint operator terminates either if convergence was detected or if the maximum number of iterations is exceeded. Consequently, the fixpoint operator is defined as follows:

$$fixpoint : \underbrace{T}_{\mathtt{x}} \times \left(\underbrace{T \to T}_{\mathtt{f}} \right) \times \underbrace{\mathbb{N}}_{\mathtt{m}} \times \left(\underbrace{T \times T \to \mathbb{B}}_{\mathtt{c}} \right) \to T \tag{4.1}$$

with T being a generic type variable.

In fact, the fixpoint operator replaces iterations by recursions whereas the update function f is pure. At this point Gilbert breaks with existing MATLAB code. To make Gilbert a real subset of MATLAB, the fixpoint operator would have to be integrated into MATLAB. The integration could easily be done by providing an external library with the definition of the fixpoint function.

Even though the iteration operator restricts the set of valid programs in Gilbert, it is still expressive enough to support a wide variety of programs. Moreover, all MATLAB programs can be transformed so that the for and while loops are replaced by the fixpoint operator. This replacement is simply achieved by passing all data that is read or written to as parameters to the update function. The update function returns the same set of variables, just with updated values. Such an exemplified transformation can be seen in listing 4.2. Here we can see that all data that is operated on is passed to the fixpoint operator as a cell array. Lines $1-2$ of listing 4.2(b) define the update function `f`. The parameter `x` is a cell array whose first entry contains the accumulator `A` and second entry is the loop counter `i`. In line 2 we see the returned cell array value of the anonymous function. The first entry is the sum of the current accumulator value and the loop counter. The second entry is the incremented loop counter for the next iteration. Line 3 calls the fixpoint operator with the initial cell array value, the update function and the maximum number of iterations. The final result is retrieved from the final cell array value in line 4.

4.2 Language Grammar

We could not find an official specification of the MATLAB language. Therefore, we defined our own grammar with the goal to imitate MATLAB as closely as possible. Even though we extensively tested the conformity of the language, we cannot guarantee that it is completely equivalent. In the following, we give the Backus-Naur form (BNF) of the Gilbert language.

⟨*program*⟩ ::= ⟨*stmtOrFuncList*⟩

⟨*stmtOrFuncList*⟩ ::= (⟨*stmt*⟩ | ⟨*funcDef*⟩)*

```
A = 0;
for i = 1:10
  A = A + i;
end
```

(a) For loop

```
f = @(x) ...
  {x{1} + x{2}, x{2} + 1};
r = fixpoint({0,1}, f, 10);
A = r{1};
```

(b) Fixpoint

Listing 4.2: Transformation from Matlab for loop (a) to Gilbert fixpoint (b) formulation. Essentially, all iteration data is combined and passed as a cell array value to the update function.

⟨*stmt*⟩ ::= ⟨*assignment*⟩ | ⟨*expression*⟩

⟨*assignment*⟩ ::= ⟨*lhs*⟩ '=' ⟨*rhs*⟩

⟨*lhs*⟩ ::= ⟨*identifier*⟩

⟨*rhs*⟩ ::= ⟨*expression*⟩

⟨*funcDef*⟩ ::= 'function' ⟨*funcValues*⟩? ⟨*identifier*⟩ ⟨*funcParams*⟩ ⟨*funcBody*⟩ 'end'

⟨*funcValues*⟩ ::= ⟨*identifier*⟩ '='
| '[' ⟨*identifier*⟩ (',' ⟨*identifier*⟩)* ']' '='

⟨*funcParams*⟩ ::= '(' ')'
| '(' ⟨*identifier*⟩ (',' ⟨*identifier*⟩)* ')'

⟨*funcBody*⟩ ::= ⟨*stmtOrFuncList*⟩

⟨*expression*⟩ ::= ⟨*aexp1*⟩

⟨*aexp1*⟩ ::= ⟨*aexp2*⟩ ('||' ⟨*aexp2*⟩)*

⟨*aexp2*⟩ ::= ⟨*aexp3*⟩ ('&&' ⟨*aexp3*⟩)*

⟨*aexp3*⟩ ::= ⟨*aexp4*⟩ ('|' ⟨*aexp4*⟩)*

⟨*aexp4*⟩ ::= ⟨*aexp5*⟩ ('&' ⟨*aexp5*⟩)*

⟨*aexp5*⟩ ::= ⟨*aexp6*⟩ (⟨*compOp*⟩ ⟨*aexp6*⟩)*

⟨*aexp6*⟩ ::= ⟨*aexp7*⟩ (':' ⟨*aexp7*⟩)*

⟨*aexp7*⟩ ::= ⟨*aexp8*⟩ (⟨*addOp*⟩ ⟨*aexp8*⟩)*

⟨*aexp8*⟩ ::= ⟨*aexp9*⟩ (⟨*multOp*⟩ ⟨*aexp9*⟩)*

⟨*aexp9*⟩ ::= ⟨*prefixOp*⟩ ⟨*aexp9*⟩ | ⟨*aexp10*⟩

⟨*aexp10*⟩ ::= ⟨*aexp11*⟩ (⟨*expOp*⟩ ⟨*aexp11*⟩)*

⟨*aexp11*⟩ ::= ⟨*unaryExpression*⟩ ⟨*postfixOp*⟩?

⟨*unaryExpression*⟩ ::= ⟨*elemExpression*⟩ | '(' ⟨*expression*⟩ ')'

⟨*elemExpression*⟩ ::= ⟨*funcApplication*⟩
| ⟨*cellExpression*⟩
| ⟨*identifier*⟩

	⟨*scalar*⟩
	⟨*booleanLiteral*⟩
	⟨*matrix*⟩
	⟨*stringLiteral*⟩
	⟨*anonymousFunc*⟩
	⟨*funcReference*⟩
⟨*funcApplication*⟩	::= ⟨*identifier*⟩ '(' ')'
	| ⟨*identifier*⟩ '(' ⟨*expression*⟩ (',' ⟨*expression*⟩)* ')'
⟨*funcReference*⟩	::= '@' ⟨*identifier*⟩
⟨*cellExpression*⟩	::= ⟨*cellArray*⟩ | ⟨*cellArrayIndexing*⟩
⟨*cellArray*⟩	::= '{' ⟨*expression*⟩ (',' ⟨*expression*⟩)* '}'
⟨*cellArrayIndexing*⟩	::= ⟨*identifier*⟩ ('{' ⟨*numericLiteral*⟩ '}')+
⟨*scalar*⟩	::= ⟨*numericLiteral*⟩
⟨*matrix*⟩	::= '[' ⟨*matrixRow*⟩ (⟨*newlineOrSemicolon*⟩ ⟨*matrixRow*⟩)* ']'
⟨*matrixRow*⟩	::= ⟨*expression*⟩ (',' ⟨*expression*⟩)*
⟨*prefixOp*⟩	::= '+' | '-'
⟨*multOp*⟩	::= '*' | '/' | '.*' | './'
⟨*addOp*⟩	::= '+' | '-'
⟨*compOp*⟩	::= '<' | '<=' | '>' | '>=' | '==' | '~='
⟨*postfixOp*⟩	::= '.'' | '''
⟨*expOp*⟩	::= '.^'
⟨*newlineOrSemicolon*⟩	::= NL | ';'

For the sake of simplicity, we have left out most of the whitespace and line break handling. Furthermore, we used notations known from regular expressions to shorten the grammar specification. We use parentheses to group subexpressions and apply the following multiplicity specifiers: `*`, `+` and `?`. The semantics of these specifiers can be found in table 4.1.

Multiplicity specifier	Meaning
⟨*exp*⟩*	⟨*exp*⟩ can appear multiple times
⟨*exp*⟩+	⟨*exp*⟩ can appear multiple times but at least once
⟨*exp*⟩?	⟨*exp*⟩ can appear exactly once or not at all

Table 4.1: Multiplicity specifier used to specify the Gilbert language. They are borrowed from regular expressions.

The different *aexp* non-terminals are used to model the precedence order of the mathematical operators. An overview of the precedence can be found in table 4.2. The ′ operator denotes the transpose of the operand.

Operator class	Operators
Short circuit logical or	||
Short circuit logical and	&&
Logical or	|
Logical and	&
Comparators	>, >=, <, <=, ==, ~=
Range operator	:
Addition and Subtraction	+, −
Multiplication and Division	∗, /
Prefix operator	+, −
Exponentiation	^
Postfix operator	'

Table 4.2: Precedence order of mathematical operators in ascending order.

The non-terminals *identifier*, *numericLiteral*, *stringLiteral* and *booleanLiteral* are tokens generated by the lexer. The corresponding tokens are defined by regular expressions and can be found in table 4.3.

Token name	Regular expression
identifier	[a-zA-Z] [a-zA-Z0-9_]*
numericLiteral	[+-]? [0-9]+ (.[0-9]*)? ([eE][+-]?[0-9]+)? | [+-]? .[0-9]+ ([eE][+-]?[0-9]+)?
stringLiteral	"[^"]*" | '[^']*'
booleanLiteral	true | false

Table 4.3: Definition of generated tokens by regular expressions.

4.3 Parser

In order to implement Gilbert, a parser had to be selected which is powerful enough to parse the Gilbert language. Luckily, the language is rather simple and since none of the production rules is left recursive, we can use a simple LL(n) parser. This aspect becomes important when we justify our implementation choices in chapter 10.

The attentive reader will have noticed that our language specification contains an ambiguity. Consider, for example, the following valid MATLAB code `A' + B'` where `A` and `B` are matrices. The code should produce the sum of two transposed matrices. However, on the lexer level the two transpose operators will be recognized as the enclosing apostrophes of a string literal. Therefore, the lexer will produce an identifier token followed by a string literal token. Eventually, these tokens will cause a parser error, because there is no production rule consuming such a combination of tokens.

In order to solve this problem, we have to differentiate a transpose operator from an apostrophe belonging to a string literal. Since the transpose operator always follows after an expression, we can check that the previously detected token is an identifier, closing parentheses, closing bracket or closing brace. If one of these cases is true, then the apostrophe is considered to be a transpose operator. Otherwise, it belongs to a string literal.

5 Gilbert Typing

"Experience without theory is blind, but theory without experience is mere intellectual play."

—Immanuel Kant, (1724 - 1804)

Program development in general is an error-prone and time-consuming task. Besides the actual development phase, it usually involves several iterations of bug fixing. In order to reduce the number of pitfalls a programmer can fall into, typing systems were developed. Typing systems assign a *type* to language constructs such as variables, functions and expressions. That way, the system gives meaning to an otherwise vacuous program. In the memory of a computer, everything is represented as a sequence of bits, no matter whether it is an instruction code, memory address, character, boolean or floating-point number. For a computer there is no way to intrinsically differentiate between the different meanings without an additional hint. This hint comes in the form of types. Knowing that a bit sequence represents a floating-point number, the computer is aware of the valid values and operations and can check for its correct usage.

The pursued goal in typing theory is to develop a system which detects erroneous program behavior and is at the same time *sound* and *complete*. Soundness means that if a program passes the typer, then it behaves correctly. Completeness means that if a program behaves correctly, then it will pass the typer. However, it turned out that typing systems need to be extremely sophisticated in order to detect non-trivial errors and as a result they are often undecidable for non toy example languages. For example, consider the division operation. The code `1/0` would be well-typed by almost all common type checkers, because integers are divisible. However, the code will cause a runtime error because of division by zero. In order to detect this type of error, the type checking would have to be far more detailed. Therefore, the constraints are usually relaxed. In general, current type systems can already detect many different errors but they are still incomplete and only partially sound.

Type checking can be distinguished into two categories: *static* and *dynamic* type checking. Static type checkers work on the source code and assign a type to each expression at compile-time. If it detects any typing incompatibilities, such as providing the wrong arguments to a function call, assigning conflicting data types or to apply not supported operations, the type checker will alert the programmer. Most type checkers are designed to act conservatively, meaning that they relax the constraints of soundness and completeness for the sake of decidability. It is easy to see that the typing problem can be reduced to the halting problem, if soundness and completeness are assumed. For this purpose consider listing 5.1. In order to decide whether this code is well- or ill-typed, the typer has to decide whether `f` halts. Since

the halting problem is undecidable, also the set of programs producing a runtime type error is undecidable.

```
1 if(willHalt(f)){
2   code_with_type_error;
3 }else{
4   code_without_type_error;
5 }
```

Listing 5.1: In order to type this code fragment, the typer has to solve the halting problem.

In contrast to static typing, dynamic type checkers enrich each object with some kind of type tag which is used to check type compatibility at runtime. However, possible errors are only recognized after the corresponding code has been executed. A more thorough discussion about the advantages and disadvantages of both paradigms follows in section 5.1. In practice, there is hardly any static typing system which does not rely at least partially on dynamic typing as well. Dynamic downcasts, for instance, as they are common in `C++`, can only be implemented by checking whether the underlying type is the target type or a subtype of it.

5.1 Static Typing vs. Dynamic Typing

Static type checking analyzes the program prior to execution to detect errors. This approach has the advantage that possible programming mistakes are caught early in the development process. An assignment of a string to a double would be an example for such a mistake. As indicated by Westland [76], who investigated the influence of errors during the software development process, unfixed errors become exponentially more costly with each phase. Therefore, it is important to detect and correct errors as soon as possible. Static typing usually requires the user to specify types explicitly in the source code, because the language lacks type inference or has ambiguities which prevent the type inference from inferring the correct types. Java, for example, does not have type inference and therefore the user has to specify types redundantly. In line 1 of listing 5.2, it is obvious that we have to specify twice the type `Object`, even though this type can easily be deduced from the right side of the assignment. Furthermore, in line 4 we see an addition of two integers. It is clear that the result is an integer and thus the `int` type specifier is dispensable.

```
1 Object obj = new Object();
2 int a = 1;
3 int b = 0;
4 int c = a + b;
```

Listing 5.2: Type annotations in Java.

Proponents of static typing emphasize that explicit type information, as they occur in Java and many other programming languages, documents the code. It is easier for a programmer to use existing code if he can identify function arguments and their types with one glance, for instance. Additionally, it is possible for the compiler to apply sophisticated optimization techniques, if it knows the types. By operating on these values, the compiler could, for example, use more efficient machine instructions for floating-point calculations. Furthermore, it is possible

to substitute virtual function calls for direct calls, if the actual object type is known. Another benefit of static typing is the increased type safety. After passing the type checker, the program is guaranteed to fulfill for all inputs some set of type safety properties. This guarantee frees the runtime from checking the safety properties and thus the program can be executed more efficiently. Type information additionally helps to provide a better programming experience in an IDE by offering type dependent context help. If the IDE knows the type of a variable, then it can tell the programmer which methods are supported by this type, for instance. And last but not least, explicit typing permits a better abstraction of functionality and, thus, increases modularity. Interfaces can be defined to orchestrate the interaction between several software components allowing them to be developed independently from each other.

In contrast to these arguments, advocates of dynamic typing argue that their approach is more vivid and better suited for prototyping, because the static typing approach is too rigid. These aspects come especially into effect for programs in a highly dynamic environment with unknown or quickly changing requirements such as data integration. Another advantage is that the compile time is reduced because of fewer passes the compiler has to go through. However, the avoided type checks will then be realized within the runtime which adds overhead. But the dynamic nature allows interpreters to dynamically load code more quickly, because all type checkings are deferred until its actual execution. Moreover, dynamic languages support duck typing as a powerful tool to write reusable code. The term duck typing originates from the poet James Whitcomb Riley, who stated: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." [37]. Essentially, duck typing means that not the actual type of an object decides whether a program is well- or ill-typed but the set of supported methods and its properties. For example, consider a function which calls a method `count` on its given parameter. Then all function calls are valid which are called with an argument having a method `count`, independently of its actual type. That way, the programmer can write code which is applicable for a wide variety of types without having to specify them explicitly. Ousterhout [57] even claims that static typed languages do not guarantee a higher type safety than dynamic typed languages. Furthermore, he states that they are more verbose and it is difficult to write reusable code. However, it is unclear in which way reducing language features leads to a more powerful and expressive language.

The debate of whether statically or dynamically typed languages are superior has almost reached religious character. It is likely that none of the approaches alone solves all the problems. Instead, a combination of the strengths of both paradigms promises the best results [53].

We are in the unfortunate situation that MATLAB belongs to the class of dynamically typed languages and Gilbert requires type information for its parallel execution. The parallel data processing systems, used to run Gilbert programs, have to know which data types are passed from one worker node to another. Therefore, the MATLAB language has to be enriched with type information.

It exists research of how to add explicit type information to MATLAB. For example, Hendren [38] introduced the special keyword *atype* to MATLAB which is understood and interpreted by an extended compiler. The *atype* keyword basically acts as a type annotation and can be implemented within a special library or as a weaver. Even though this approach seems quite promising, we decided to opt for a more transparent mechanism, namely type inference. Type inference has the advantage that the MATLAB user is not bothered by having to add explicit type information and thus can continue writing his code in the usual fashion. In case that the

type inference algorithm cannot properly infer the types, there has to be a way to resolve this problem. We decided to pursue a similar approach as Furr et al. [29]. Furr et al. added type information to Ruby by adding special comments to the respective code sections. Thereby, the code does not break with the Ruby standard and still contains type information. As typing system, we use the Hindley-Milner (HM) type system [40, 54] and a slightly derived form of algorithm W [23] for type inference. Algorithm W will be described in the next section in detail. Even though there exist more powerful type systems than the HM type system, it has the appealing charm that the algorithm W is sound, complete and decidable with respect to the type system. Furthermore, it has proven to type several algorithms implemented within Gilbert correctly.

5.2 Hindley-Milner Type Inference

The Hindley-Milner (HM) type inference assigns types to expressions. It was initially developed to type functional languages. In fact, HM type inference was first implemented as the typing part of the programming language ML. HM types the expressions of the lambda calculus enriched with the `let`-expression. For a detailed review of the lambda calculus, the interested reader is referred to the article of Cardone and Hindley [20].

The set of expressions e contains variables, function applications, function abstractions and `let`-expressions. The formal definition of e is

$$\begin{aligned} e &= x && \text{(Variable)} \\ &\mid e\,e && \text{(Application)} \\ &\mid \lambda x.e && \text{(Abstraction)} \\ &\mid \texttt{let}\ x = e\ \texttt{in}\ e \end{aligned}$$

For those unconversant with lambda expressions, we will quickly revise them. The function application is written as $e_1\ e_2$ whereas e_1 denotes a function and e_2 the function argument which is applied to the function body. The function abstraction $\lambda x.e$ is the equivalent of an anonymous function, as it is known from many program languages. It is initiated with a λ followed by its function parameter x and the function body e. The let-expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ introduces a new variable x, having the value e_1, into the context Γ. This variable can be used within the expression e_2. The semantic of the assignment is that every occurrence of x in e_2 is replaced by e_1. The context Γ contains all type information so far known and is a mapping from variables to types. A more precise definition is given in a later paragraph.

One might think that the lambda calculus is only a toy language and that the set of expressions is not expressive enough to represent any common programming problem. However, it turned out that the lambda calculus is Turing complete and thus any programming language can be reduced to it. Therefore, it is enough to reason about the typing aspects of these expressions.

Even though Gilbert is not a functional language, we can use the HM type inference to compute the types at compile-time. The only difference to the above defined set of expressions is that Gilbert does not have a `let`-expression. Instead, it has ordinary assignments of the form $x = e$. The notion of the assignment is similar to the `let`-expression. An assignment adds a new variable x with the value e to the context.

HM type inference distinguishes between two sorts of types, *mono-* and *polytypes*. Monotypes τ are defined as follows:

$$\begin{aligned} \tau &= \alpha \quad \text{(Variable)} \\ &\mid D\ \tau\ \ldots \tau \quad \text{(Application)} \end{aligned}$$

A monotype always denotes a concrete type. If the type is known, it can be a type constant, such as `int`, `float` or `string`, or a parametric type. A parametric type takes itself type arguments to form a concrete type. An example of a parametric type is the `Set` or the `List` which can be instantiated with arbitrary element types. Type constants are a special case of the application rule with an empty list of arguments. If the type is not yet known but can only be a single type, then a type variable α is used. Often, it can be the case that these type variables are replaced with known types at a later stage of the type inference.

In contrast to monotypes, polytypes σ denote multiple types. They are defined by

$$\begin{aligned} \sigma &= \tau \\ &\mid \forall\alpha.\sigma \end{aligned}$$

A type of the form $\forall\alpha.\sigma$ denotes the set of all types where α is substituted by a concrete type τ within σ. Consider, for example, the tuple access function `fst` which takes a tuple and returns the first entry. The function `fst` should be applicable to all different types of tuples, for example $(\texttt{int}, \texttt{int})$, $(\texttt{string}, \texttt{float})$, etc. If `fst` has a monotype, then it will only work for one concrete tuple type. Therefore, `fst` has a polytype. The actual type of `fst` is $\forall\alpha, \beta.(\alpha, \beta) \rightarrow \alpha$. Depending on the applied tuple argument the quantified type variables are instantiated respectively. Polytypes permit to implement generic functions in a type safe manner and that is called parametric polymorphism. It is important to note that polytypes introduce ambiguities with respect to the calculated type. An expression having the type $\forall\alpha.\alpha$ also has the type `int`, for example. Thus, the latter type would be a valid inferred type, even though the first one is more general. In order to dissolve this ambiguities, the typing system always looks for the most general type.

Usually, expressions depend on other expressions and thus their types. Therefore, a type dictionary is maintained where types of expressions are stored, which already have been seen while parsing the code. And that is exactly what the context Γ is used for. Γ contains a list of pairs $x : \sigma$, which are called assumptions. The formal definition of Γ is

$$\begin{aligned} \Gamma &= \Gamma, x : \sigma \\ &\mid \epsilon \text{ (empty)} \end{aligned}$$

Having defined the context, expressions and types, the typing judgment $\Gamma \vdash x : \sigma$ can be explained. The typing judgment can be interpreted that given the context Γ the variable x has the type σ. The way how to come up with these judgments is defined by deduction rules. Before taking a closer look at them, some auxiliary functions are first introduced.

An important distinction in an HM type expression is the state of the type variables. In general, type variables can appear as *free* or as *bound* variables. A type variable α is called bound if it occurs in an expression of the form $\forall\alpha.\tau$. $\forall$ binds the subsequent variable in the

context of the expression. All variables which are not bound are free. We can define a function *free* which calculates the set of free type variables in a type expression and in the type context.

$$\begin{aligned} free(\alpha) &= \{\alpha\} \\ free(D\ \alpha_1\ \ldots\ \alpha_n) &= \bigcup_{i=1}^{n} free(\alpha_i) \\ free(\forall\alpha.\tau) &= free(\tau) \setminus \{\alpha\} \\ free(\Gamma) &= \bigcup_{x:\sigma\in\Gamma} free(\sigma) \end{aligned}$$

5.2.1 Algorithm W

The algorithm W was initially developed by Damas and Milner [23] and allows to solve the HM type inference in almost linear time with respect to the size of the source code. Thus, it is also applicable to large programs. A modified version of the original algorithm which incorporates side effects into the deduction rules is presented. Even though these side effects contradict the purity of logical deduction rules, it allows to express the algorithm concisely. The algorithm W is defined by the following deduction rules:

$$\text{Variable:}\quad \frac{x:\sigma\in\Gamma \qquad \tau = inst(\sigma)}{\Gamma \vdash x:\tau}$$

$$\text{Application:}\quad \frac{\Gamma\vdash e_0:\tau_0 \qquad \Gamma\vdash e_1:\tau_1 \qquad \tau' = newvar \qquad unify(\tau_0, \tau_1\to\tau')}{\Gamma\vdash e_0e_1:\tau'}$$

$$\text{Abstraction:}\quad \frac{\tau = newvar \qquad \Gamma, x:\tau\vdash e:\tau'}{\Gamma\vdash \lambda x.e:\tau\to\tau'}$$

$$\text{Assignment:}\quad \frac{\Gamma\vdash e_0:\tau \qquad \Gamma, x:\overline{\Gamma}(\tau)\vdash e_1:\tau'}{\Gamma\vdash x = e_0; e_1:\tau'}$$

Each rule has the form $\frac{\text{Premise}}{\text{Conclusion}}$, which means that if the premise can be proven, then the conclusion is true.

Let us go step by step through the different deduction rules of the algorithm. The first rule says that a variable x has the type τ, if an assumption of x in Γ whose type σ can be specialized to τ is found. The function *inst* does the specialization of the type. A type is specialized by removing all quantifiers and replacing the bound variables by fresh monotype variables.

The second rule deals with function applications. In order to show that a function application has the type τ', we first have to look at the function type τ_0 and the argument type τ_1. In order to fulfill the conclusion, we know that the type of e_0 has to be a function with an argument of type τ_1 and some result type τ'. If the type τ_0 is equivalent to this function type $\tau_1 \to \tau'$, we

can deduce that the resulting type is τ'. The equivalence is tested by the $unify$ function. This function takes two types and tries to find the most general type unifying both arguments.

The $unify$ function is an essential part of the typing system, since it incorporates all constraints imposed by the different expressions in order to find the most general type validating the program. The working principle can be best understood by looking at its pseudocode in listing 5.3.

```
1 def unify(typeA: Type, typeB: Type): Type = {
2   val tA = resolve(typeA)
3   val tB = resolve(typeB)
4
5   if( at least one term is a type variable){
6     return union(tA, tB)
7   }
8     else if( both type expressions are of the form D p1 ... pn with
9     the same D and the number of arguments){
10    return D(unify(tA.p1, tB.p1), ..., unify(tA.pn, tb.pn))
11  } else{
12    return typeMismatch
13  }
14 }
```

Listing 5.3: $Unify$ function.

First of all, both type expressions are resolved. This means that type variables, which are contained as sub expressions, are looked up in the type variable assignment dictionary. This dictionary maintains the current assignments between variables and their types. If a type variable has an assigned type, then this type is substituted for the corresponding type variable. If at least one of the resulting type expressions is still a type variable, then we can simply construct the union of both types. The union function returns the more concrete type of both arguments and updates the dictionary correspondingly. Thus, if both arguments are type variables, then one of them is returned. If only one argument is a type variable, then the other argument is returned.

If none of the resolved type expressions is a type variable, then they have to be a type application. Applications can only be unified if they are constructed by the same function symbol and if they have the same number of arguments. If this condition is fulfilled, the arguments can be unified. Depending on the unifiability of the arguments, the overall type application is unifiable or not.

If both type expressions are constructed by different type applications, then the types are impossible to unify. Consequently, a type mismatch error will be returned.

The third rule allows to type lambda abstractions or in Gilbert's case concrete and anonymous functions. In order to show that a lambda abstraction $\lambda x.e$ has the type $\tau \to \tau'$, a new type variable for the unknown parameter type is introduced. After adding this assumption to the context, the type of the function body e has to be calculated. If the body e has the type τ', then it can be concluded that the lambda abstraction has the type $\tau \to \tau'$.

The last deduction rule covers assignments. An assignment simply binds a variable name x to an expression value. Since this expression value has a type τ, a new assumption $x : \tau$ can be added to the context. The type inference for the subsequent expressions is continued with the updated type context. That is expressed by the deduction rule. In fact, there is a slight difference. Instead of adding the assignment $x : \tau$, the $x : \overline{\Gamma}(\tau)$ is added to the context. $\overline{\Gamma}(\tau)$ is the generalization of τ. The generalization quantifies all free monotypes of τ not bound in Γ. That way, the most general type is found.

$$\begin{aligned} \overline{\Gamma}(\tau) &= \forall \overline{\alpha}.\tau \\ \text{with} \quad \overline{\alpha} &= free(\tau) \setminus free(\Gamma) \end{aligned}$$

5.2.2 Function Overloading

MATLAB's basic operators, such as $+$, $-$, $/$ and $\star$, for example, are overloaded. They can be applied to matrices, scalars as well as mixture of both data types. That makes it very convenient to express mathematical problems, but from a programmer's point of view it causes some hardships. Originally, HM cannot deal with overloaded functions properly, because it assumes that each function has an unique type. In order to extend HM's capabilities, we allowed each function symbol to have a list of signatures. In the case of $+$, the list of signatures would consist of

$$\begin{aligned} matrix[double] \times matrix[double] &\rightarrow matrix[double] \\ matrix[double] \times double &\rightarrow matrix[double] \\ double \times matrix[double] &\rightarrow matrix[double] \\ double \times double &\rightarrow double \end{aligned}$$

In order to solve the typing problem, the inference algorithm has to resolve this ambiguity. Having complete knowledge of the argument types is enough to select the appropriate signature. Sometimes even partial knowledge is sufficient. However, if this information is missing, Gilbert has to apply a heuristic to make the typing expression well-formed. The heuristic selects the first entry in the list of signatures.

Due to this heuristic, some well-typed programs will be rejected, though. Consider, for example, the code snippet in listing 5.4. In line 1, an anonymous function f is defined, which doubles its parameter. Within the body of f, the overloaded $+$ operator is used. Since Gilbert has no additional typing information of f's parameter, it has to apply the aforementioned heuristic to assign f a valid type expression. The heuristic picks the first entry which is $matrix[double] \times matrix[double] \rightarrow matrix[double]$ and , thus, f has the type $matrix[double] \rightarrow matrix[double]$. The subsequent function call `f(1.0)` in line 2 will throw a typing exception, because a scalar argument cannot be applied to a function having a matrix parameter. That is odd, because f should be applicable to all data types supporting the $+$ operator and scalar values definitely support this operation. It has to be stated that this peculiarity is clearly a restriction of Gilbert, which limits the space of accepted programs.

```
1    f = @(x) x + x;
2    f(1.0)
```

Listing 5.4: Wrongly rejected Gilbert program due to function overloading.

5.3 Matrix Dimension Inference

Matrices constitute the elementary data type in our linear algebra environment. Besides its element type, a matrix is also defined by its size. In the context of program execution, knowledge about matrix sizes can help a lot to optimize the evaluation. For instance, consider a threefold matrix multiplication $A \times B \times C$. The multiplication can be evaluated in two different ways: $(A \times B) \times C$ and $A \times (B \times C)$. For certain matrix sizes one way might be infeasible whereas the other way can be calculated efficiently due to the matrix size of the intermediate result $(A \times B)$ or $(B \times C)$.

To illustrate this point, assume that $A \in \mathbb{R}^{1000\times10}, B \in \mathbb{R}^{10\times1000}$ and $C \in \mathbb{R}^{1000\times10}$. Thus, the intermediate products are:

$$\begin{aligned} A \times B &\in \mathbb{R}^{1000\times1000} \\ B \times C &\in \mathbb{R}^{10\times10} \end{aligned}$$

In the first case we have to calculate a 1000×1000 matrix and in the second only a 10×10 matrix.

By knowing the matrix sizes, we can choose the most efficient strategy to calculate the requested result. Another advantage is that we can decide whether to carry out the computation in-core or in parallel depending on the matrix sizes. Sometimes the benefit of parallel execution is smaller than the initial communication overhead and thus it would be wiser to execute the calculation locally. Furthermore, it can be helpful for data partitioning on a large cluster and to decide on a blocking strategy with respect to the algebraic operations. Therefore, we extended the HM type inference to also infer matrix sizes where possible.

Gilbert's matrix type is defined as

$$MatrixType(\underbrace{\tau}_{\text{Element type}}, \underbrace{\nu}_{\text{Number of rows}}, \underbrace{\nu}_{\text{Number of columns}})$$

with ν being the value type:

$$\begin{aligned} \nu &= \gamma \quad \text{(Variable)} \\ &\mid \delta \quad \text{(Value)} \end{aligned}$$

The value type can either be a value variable or a concrete value.

Value variables always appear if we know that a certain relationship holds, but do not know the concrete values yet. For example, when we multiply two matrices $A \in \mathbb{R}^{a\times b}$ and $B \in \mathbb{R}^{b\times c}$ we know that the result is $A \times B = C \in \mathbb{R}^{a\times c}$, without having knowledge about the actual values of a and c. Once we have deduced the actual values for a variable we can simply replace them.

One of these occasions is the definition of a matrix. Gilbert implements several functions, known from Matlab: `eye`, `zeros`, `ones` or the `load` function, reading a matrix from disk. All of these functions require to specify the number of rows and columns for the resulting matrix. These values often serve as the source of deduction. Another possibility are special operations which modify the size of the matrix in a deterministic way. The sum operation, which calculates row or column sums, respectively, will reduce the dimension which is used for the summation to 1.

The matrix size inference is incorporated into the HM type inference by adding some logic to the `unify` function. Whenever we encounter a matrix type during the unification process, we call a `unifyValue` function on the two row and column values. The `unifyValue` function works similarly to the `unify` function. First, the function resolves the value expression, thus substituting value variables with their assigned values. Then, if at least one of the resolved value expressions is still a value variable, then the union is constructed and the corresponding value variable dictionary entry is updated. If both resolved expressions are equal, then this value is returned and otherwise a value mismatch error is thrown.

6 Intermediate Representation

"Abstraction is real, probably more real than nature."

—Josef Albers, (1888 - 1976)

After parsing and typing of the source code is done, we have all the information needed to execute the program in a distributed fashion. However, we do not compile the code directly to the respective format to let it run on a parallel data processing system. Instead, we compile it to an intermediate representation. Even though, the intermediate code adds an additional representation and thus more complexity to maintain, the advantages clearly outweigh the disadvantages. Moreover, this approach is similar to the approach the JVM and the .Net frameworks pursue. The Java and .Net code are compiled to a format which can be executed platform independently.

The additional abstraction layer allows Gilbert to apply language independent optimization in a coherent manner and independently of the actually used front end language. This implementation aspect gives the flexibility to easily extend Gilbert to support other linear algebra languages as well. The next natural candidate for a supported front end language would surely be R. Moreover, a higher-level abstraction of the mathematical operations holds more potential for algebraic optimization. One class of important transformations is the reordering of operations and the determination of the execution order. Often, they are based on the commutative and associative property of the corresponding operations. An obvious optimization would be the execution order of multiple matrix multiplications as indicated in section 5.3. Furthermore, the propagation and elimination of transpose operations offers further optimization potential.

The intermediate representation is designed with two goals in mind: First of all, it has to be expressive enough to represent in a general way a wide variety of MATLAB programs. Therefore, the intermediate format has to include the operational primitives of linear algebra as well as an iteration abstraction. With these building blocks at hand, it is already possible to express a multitude of iterative machine learning algorithms. As pointed out in section 4.1, Gilbert does not provide a direct `for` or `while` loop, but instead the `fixpoint` operator. The second design goal was simplicity. It is of particular interest to keep the set of intermediate operators as small as possible, because it would alleviate a possible optimization step prior to execution.

6.1 Specification

The intermediate format consists of a set of operators to represent the different linear algebra operations. Every operator has a distinct result type and a set of parameters which are

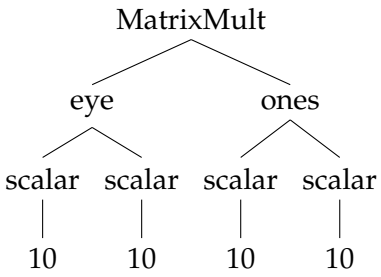


Figure 6.1: Dependency tree of a matrix multiplication between an identity matrix and a matrix filled with 1s.

required as inputs. Figure 6.1 shows how a matrix multiplication between an identity matrix $Id \in \mathbb{R}^{10\times10}$ and a matrix $Ones \in \mathbb{R}^{10\times10}$ filled with 1s would be represented. On the level of the intermediate format we distinguish between the following types: `MatrixType`, `ScalarType`, `CellArrayType`, `StringType` and `FunctionType`.

The matrix type is the elementary type and represents all sorts of matrices. A matrix contains information about its size, namely the number of rows and columns, and about its element type. The element type can be any supported type but currently only operations for double or boolean matrices are implemented.

The scalar type is a superset containing the `DoubleType` and `BooleanType`. Usually, the double type is used for all linear algebra operations. Boolean scalars are mainly used for convergence criteria of iterations.

The cell array type contains a list of types specifying the types of its cell entries. Cell arrays are usually used to return multiple values from a function. Thus, they can conceptually be seen as tuples. In contrast to the MATLAB implementation, Gilbert only supports 1-dimensional cell arrays.

The string type was included to be able to load matrices from HDFS or local disk. In order to load a matrix, the user has to define the location of the data, given as a string. Besides this usage, strings are usually not necessary for linear algebra operations.

Last but not least, Gilbert also supports function types. Function types are necessary to implement 2nd-order functions such as the fixpoint operator. The user can pass defined functions around like any other value. Thus, functions are considered first-class citizens in Gilbert.

The set of intermediate operators can be distinguished into three categories: *Creation operators*, *transformation operators* and *Output operators*. All of these categories will be explained in the following subsections.

6.1.1 Creation Operators

Creation operators generate or load some data depending on the used function. A list of all supported operators with its type definition and a short explanation can be found in table 6.1.

Operator	Type	Explanation
`load`	$string \times double \times double \rightarrow matrix[double]$	Loading a matrix from disk
`eye`	$double \times double \rightarrow matrix[double]$	Creating an identity matrix
`zeros`	$double \times double \rightarrow matrix[double]$	Creating a zero matrix
`randn`	$double \times double \rightarrow matrix[double]$	Creating a random matrix

Table 6.1: Creating operators of Gilbert, their types and a short explanation.

Each type definition consists of the input types and the result type of an operator:

$$\underbrace{type1 \times \ldots \times typeN}_{\text{Types of input parameters}} \rightarrow \underbrace{resultType}_{\text{Type of result value}}$$

Even though the input parameters for the matrix dimensions are denoted by doubles, the user should provide integer values. Internally, the double parameters are converted to integers. The type in brackets after the matrix type indicates the element type.

The `load` operator takes a path to a stored matrix on disk and the corresponding number of rows and columns as input parameters. It then reads the stored data and loads the matrix into the program context. The `eye` operator, known from Matlab, generates an identity matrix of the requested size, number of rows and columns. The operator also supports non quadratic result shapes. Another well-known operator is `zeros`. It generates a matrix of the given size which is initialized with zeros. And the last creation operator is `randn` which generates a random matrix. `Randn` takes the number of rows and columns of the resulting matrix and the mean and the standard deviation of a Gauss distribution. The specified Gauss distribution is used to generate random values for the resulting matrix.

6.1.2 Transformation Operators

The transformation operators constitute the main abstraction of the linear algebra operations. They group operations with similar properties and thus allow an easier reasoning and optimization of the underlying program. The list of all transformation operators and their types can be found in table 6.2.

The `UnaryScalarTransformation` takes a single scalar value and applies an unary operation on it. The list of all supported operations can be found in table 6.3. The operation *binarize* has the following semantics:

$$binarize(x) : x \mapsto \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{else} \end{cases}$$

The applicable operations to the `ScalarScalarTransformation` depend on the input parameter types. The operators $+, -, *, /, pow(\cdot,\cdot)$ take two double values and produce a new double value. The comparison operators, on the contrary, produce a boolean result value. The logical operations require two boolean values and produce a boolean value as the result. Gilbert supports short-circuit logical operators && and || as well as & and | evaluating all of their inputs prior to computing the result.

Operator	Type
`UnaryScalarTransformation`	$scalar \times unaryScalarOp \rightarrow scalar$
`ScalarScalarTransformation`	$scalar \times scalar \times scalarOp \rightarrow scalar$
`Transpose`	$matrix \rightarrow matrix$
`ScalarMatrixTransformation`	$scalar \times matrix \times smOp \rightarrow matrix$
`MatrixScalarTransformation`	$matrix \times scalar \times smOp \rightarrow matrix$
`CellwiseMatrixTransformation`	$matrix \times unaryScalarOp \rightarrow matrix$
`CellwiseMatrixMatrixTransformation`	$matrix \times matrix \times scalarOp \rightarrow matrix$
`MatrixMult`	$matrix \times matrix \rightarrow matrix$
`VectorwiseMatrixTransformation`	$matrix \times vectorwiseOp \rightarrow matrix$
`AggregateMatrixTransformation`	$matrix \times aggregateOp \rightarrow scalar$
`FixpointIterationMatrix`	$matrix \times (matrix \rightarrow matrix) \times double \times (matrix \times matrix \rightarrow boolean) \rightarrow matrix$
`FixpointIterationCellArray`	$cellarray \times (cellarray \rightarrow cellarray) \times double \times (cellarray \times cellarray \rightarrow boolean) \rightarrow cellarray$

Table 6.2: Transforming operators of Gilbert and their types.

The `ScalarMatrixTransformation` and `MatrixScalarTransformation` have an almost identical support of operations. The only difference is that the former transformation does not allow to calculate the power of a scalar where the exponent is a matrix.

The `VectorwiseMatrixTransformation` applies an operation on each row vector of the given matrix. A vectorwise operation produces a scalar value for each row vector. Thus, the resulting type of such an operation is an one-column matrix. Gilbert can currently retrieve the maximum and minimum of each row vector. Furthermore, one can calculate the 2-norm of each row vector:

$$\left\| (a_{i1}, \ldots, a_{in})^T \right\|_2 = \sqrt{\sum_{j=1}^{n} a_{ij}^2}$$

Similar to the vectorwise operations, the `AggregateMatrixTransformation` applies an operation to all matrix entries producing a single scalar result value. The aggregation operation can compute the maximum, minimum and the sum of all matrix entries. Furthermore, it supports the Frobenius norm:

$$\|A\|_2 = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}$$

The iteration mechanism is represented by the `FixpointIterationMatrix` and `FixpointIterationCellArray` operators. The former operator is used for a fixpoint operation on matrices and the latter on cell arrays. Both operators follow the semantics and have the same parameters as the fixpoint abstraction presented in eq. (4.1). A fixpoint operator receives an initial value, an update function, the maximum number of iterations and a convergence function. The update function is called with the current value and returns the value for the next iteration. The convergence function is called with the current and the previous value.

Operation type	Operations
$unaryScalarOp$	$-, \lvert\cdot\rvert, binarize(\cdot)$
$scalarOp : double \times double \rightarrow double$	$+, -, *, /, pow(\cdot,\cdot)$
$scalarOp : double \times double \rightarrow boolean$	$>, >=, <, <=, ==, \sim=$
$scalarOp : boolean \times boolean \rightarrow boolean$	$\&, \lvert, \&\&, \lvert\lvert$
$smOp : matrix[double] \times double \rightarrow matrix[double]$	$+, -, *, /, pow(\cdot,\cdot)$
$smOp : matrix[double] \times double \rightarrow matrix[boolean]$	$>, >=, <, <=, ==, \sim=$
$smOp : matrix[boolean] \times boolean \rightarrow matrix[boolean]$	$\&, \lvert, \&\&, \lvert\lvert$
$smOp : double \times matrix[double] \rightarrow matrix[double]$	$+, -, *, /$
$smOp : double \times matrix[double] \rightarrow matrix[boolean]$	$>, >=, <, <=, ==, \sim=$
$smOp : boolean \times matrix[boolean] \rightarrow matrix[boolean]$	$\&, \lvert, \&\&, \lvert\lvert$
$vectorwiseOp$	$min, max, norm2$
$aggregateOp$	$min, max, sumAll, norm2$

Table 6.3: Supported transformation operations.

Based on these values it can check for convergence. For further details see section 4.1.

6.1.3 Writing Operators

The writing operators are used to make the computed results accessible to the user by writing them back to disk. There exists a writing operation for each supported type. The list of all writing operations can be seen in table 6.4.

Operator	Explanation
`WriteMatrix`	Writes the matrix to disk
`WriteScalar`	Writes the scalar value to disk
`WriteString`	Writes the string value to disk
`WriteCellArray`	Writes the contents of the cell array to disk
`WriteFunction`	Writes the intermediate execution plan to disk

Table 6.4: Supported writing operators.

6.1.4 Compilation Example

The intermediate representation is the target format of the compilation process after it has been parsed and typed. In order to illustrate the compilation process and to get a feeling for the intermediate representation, we will compile an example program. For this purpose, we have chosen the well-known PageRank algorithm [58]. The Matlab implementation can be seen in listing 6.1.

We start at the fixpoint operation in line 20 and build our intermediate representation bottom up. Since the fixpoint operation is given an initial matrix, it will be compiled to a `FixpointIterationMatrix` operator. The intermediate code can be seen in fig. 6.2.

The translation from line 20 to the intermediate code is straight forward. The actual compilation process would replace the variables `r_0`, `f` and `c` with their actual definition. However,

```
1 numVertices = 10;
2 % load network matrix
3 N = load("network.csv", numVertices, numVertices);
4 % create the adjacency matrix
5 A = spones(N);
6 % outdegree per vertex
7 d = sum(A, 2);
8 % create the column-stochastic transition matrix
9 T = (diag(1 ./ d) * A)';
10 % initialize the ranks
11 r_0 = ones(numVertices, 1) / numVertices;
12 e = ones(numVertices, 1) / numVertices;
13 % update function
14 f = @(r) (.85 * T * r + .15 * e)
15 eps = 0.1;
16 c =  @(prev, cur) norm(prev-cur,2) <= eps
17 % PageRank calculation
18 fixpoint(r_0, f, 20, c);
```

Listing 6.1: Matlab PageRank implementation.

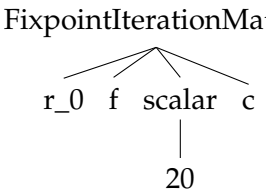


Figure 6.2: Intermediate representation of PageRank's fixpoint operation.

for the sake of simplicity, we have left them in place and will gradually refine them.

The variable `r_0` denotes the initial column vector of the PageRank iteration. Its resulting code from the compilation process can be seen in fig. 6.3.

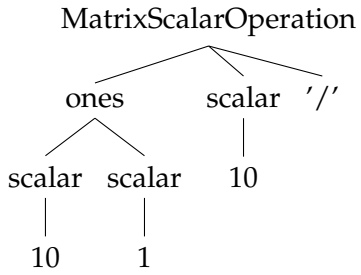


Figure 6.3: Intermediate representation of PageRank's `r_0`.

The update function `f` is defined in line 15 as an anonymous function. In order to represent functions in the intermediate format, we have to introduce a new operator `function` which takes the number of parameters and the body of the function. Additionally, all parameters of the function are represented by special intermediate code operators: `MatrixParameter`, `ScalarParameter`, `StringParameter` and `FunctionParameter`. Every parameter oper-

ator gets an index assigned which identifies the corresponding argument with which it has to be replaced upon instantiation of the function. The compiled code of `f` can be found in fig. 6.4.

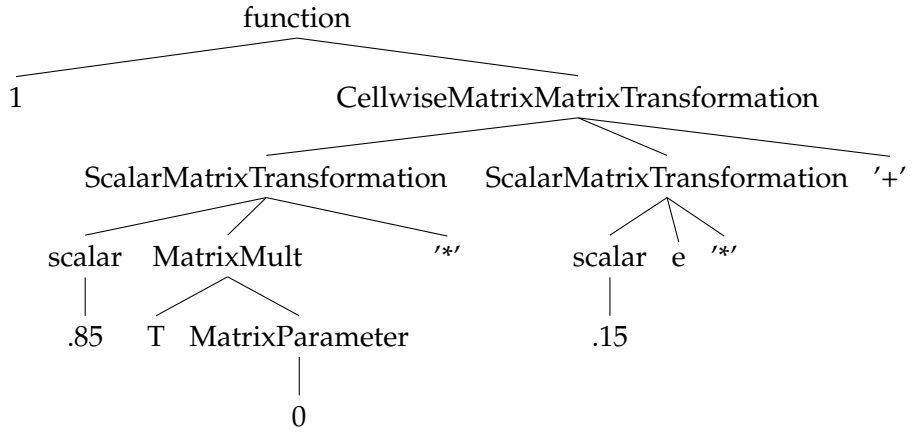


Figure 6.4: Intermediate representation of PageRank's `f`.

The matrix `T` is the transition matrix $T = (t)_{ij} \in \mathbb{R}^{10\times 10}$. The entry t_{ij} indicates the probability of going from site j to site i if the user is currently on site j. The dependency tree of the intermediate code of `T` is shown in fig. 6.5.

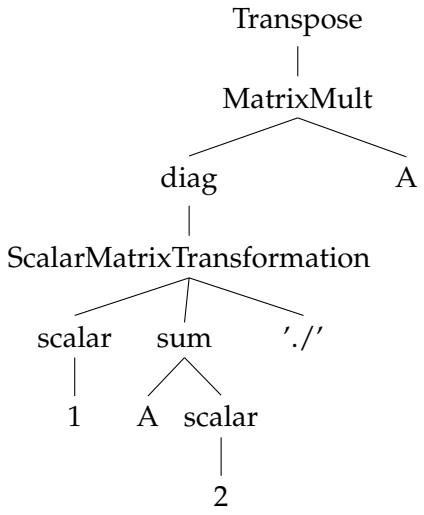


Figure 6.5: Intermediate representation of PageRank's `T`.

Here `diag` denotes a special operator which has a different behavior depending on the given argument. If one provides a one-column matrix d to `diag`, then a zero matrix with d on its diagonal is created. If one provides a matrix m to `diag`, then the diagonal of m is returned. The operator `sum` takes a matrix and a dimension and computes the sums along the specified dimension. Thus, the row sums will be computed in this case.

What is left to be compiled for `f` is the matrix `A`. The compiled code can be seen in fig. 6.6.

The Matlab function `spones`, which replaces nonzero sparse elements with ones while preserving the sparsity structure, is compiled into a CellwiseMatrixTransformation.

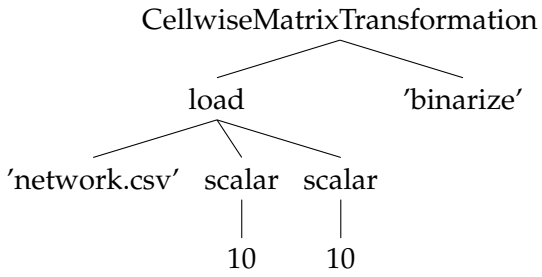


Figure 6.6: Intermediate representation of PageRank's `A`.

The `e` vector has the same representation as the initial PageRank vector `r_0`.

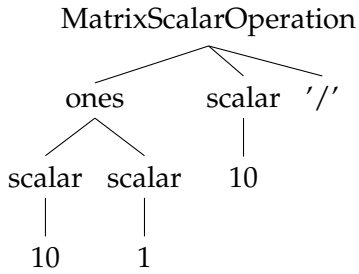


Figure 6.7: Intermediate representation of PageRank's `e`.

The convergence function `c` is an anonymous function taking two parameters. Its compiled version is shown in fig. 6.8

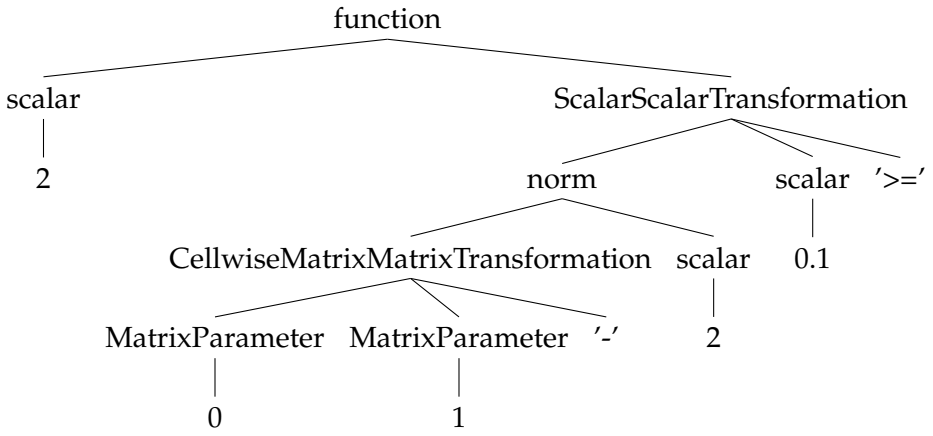


Figure 6.8: Intermediate representation of PageRank's `c`.

The `norm` operator, taking as first parameter the matrix m and as second the norm parameter p, calculates the p-norm of m over all matrix elements. Internally this operator is compiled to the intermediate code shown in fig. 6.9.

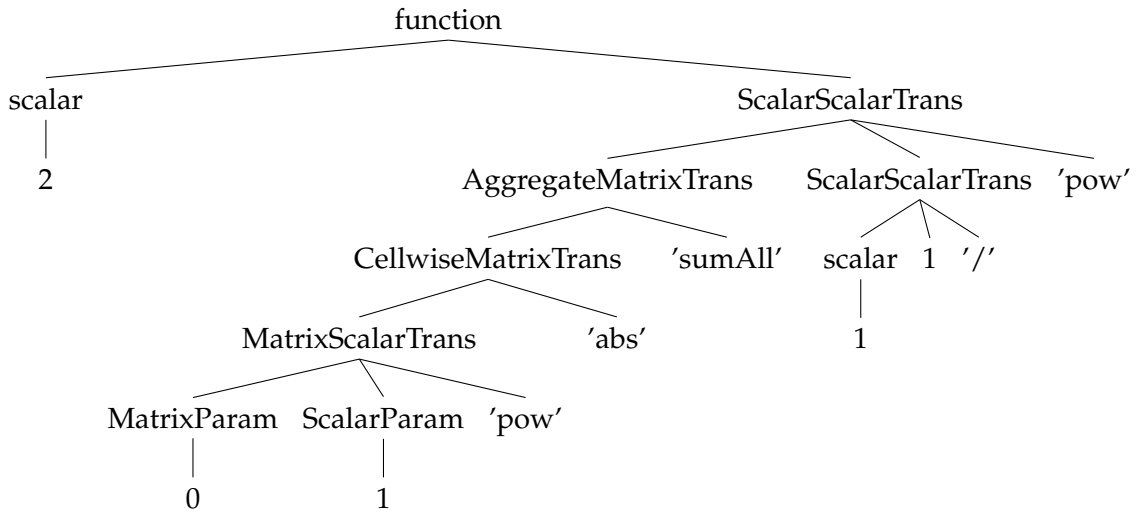


Figure 6.9: Intermediate representation of `norm`.

7 Gilbert Optimizer

"Premature optimization is the root of all evil."

—Donald E. Knuth, (born 1938)

The Gilbert optimizer applies algebraic optimizations to a Gilbert program. The optimizer exploits equivalence transformations which result from the commutative and associative properties of linear algebra operations. Therefore, it works on the intermediate format of a program, which provides an appropriate high-level abstraction.

By transforming the intermediate representation, it is often possible to obtain an equivalent algebraic expression with better runtime properties. Two distinct algebraic expressions, which are semantically equivalent, can differ significantly in terms of memory consumption and required floating-point operations. Consider, for example, the product of n sums as shown in eq. (7.1). The result could be computed with n additions and $n-1$ multiplications.

$$p = (a_1 + b_1)(a_2 + b_2)\dots(a_n + b_n) \tag{7.1}$$

By expanding the right-hand side, we would obtain an equivalent expression. The expression would be a sum of 2^n products, where each product contains n factors. Thus, we would have to compute $2^n - 1$ additions and $2^n(n-1)$ multiplications. Obviously, the latter expression is far more expensive to compute than the original formulation.

Therefore, it is important for Gilbert to choose the best execution plan. Currently, the optimizer can apply two optimization strategies: Matrix multiplication reordering and transpose pushdown. Both transformations will be described in the following sections.

7.1 Matrix Multiplication Reordering

Matrix multiplications belong to the most expensive operations in linear algebra programs. The naive multiplication of $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ requires $O(n^3)$ operations. Even though there are algorithms, such as the Strassen algorithm [67], which have an asymptotic complexity of $O(n^{\log_2 8})$, a matrix multiplication still remains a costly operation. In addition to the high computational complexity, the space complexity must also not be forgotten.

In the worst case, the result of a matrix multiplication has quadratic size of the input operands. This case occurs, for example, when the outer product of two vectors is computed. Given $A \in \mathbb{R}^n$, the result $C = A \times A^T \in \mathbb{R}^{n \times n}$ is a complete $n \times n$ matrix. In practice, this characteristic can render a matrix multiplication infeasible, because there is not enough memory space available. Or, in case of virtual memory, it will slow down the computation extensively, due to

hard disk accesses.

In general, there is not much one can do, if the final result of a matrix multiplication does not fit into memory. However, if an intermediate result of successive matrix multiplications becomes too huge, there are ways to alleviate the problem. Usually, by changing the multiplication order, huge intermediate results can be avoided and, thus, also memory shortages.

In order to illustrate the problem, consider the following example: Assume we want to multiply $A \times B^T \times C$ with $A, B, C \in \mathbb{R}^n$. If we calculated it naively, we would start with $D = A \times B^T \in \mathbb{R}^{n \times n}$. The result would be $D \times C \in \mathbb{R}^n$. Thus, as an intermediate result, we had to store a $n \times n$ matrix. Using parentheses to calculate $B^T \times C$ first, we could circumvent the problem.

$$\underbrace{A \times \left(\underbrace{B^T \times C}_{\in \mathbb{R}} \right)}_{\in \mathbb{R}^n}$$

In this case, the biggest matrix to store would be a vector of length n.

The best execution order of successive multiplications is the one that minimizes the maximum size of intermediate results. In order to determine the best execution order, the optimizer first extracts all matrix multiplications with more than 2 operands. Then, it will calculate for each evaluation order the maximum size of all occurring intermediate results. In order to do this calculation, the optimizer relies on the operands' automatically inferred matrix sizes, as described in section 5.3. At last, it will pick the execution order with the minimal maximum intermediate matrix size.

7.2 Transpose Pushdown

The idea of transpose pushdown is to move the transpose operations as close to the matrix input as possible. Thereby, consecutive transpose operations accumulate at the inputs and unnecessary operations erase themselves. An example is given in listing 7.1.

```
1 C = A'*B;
2 E = (C * D')';
```

Listing 7.1: Transpose pushdown can eliminate unnecessary transpose operations occurring in linear algebra programs.

By inserting C into E, the expression $E = (A^T B D^T)^T$ is obtained. This term is equivalent to $DB^T A$. The latter formulation contains only one transpose operation. Depending on the cost of such an operation the optimization might be worthwhile.

Usually multiple transpose operations occur because they are written for convenience reasons at various positions in the code. Moreover, in complex programs it is possible that the programmer loses track of them or simply is unaware of the optimization potential. Therefore, transpose pushdown might be a beneficial optimization. The performance improvement still needs to be verified and properly quantified. This evaluation will be done in chapter 11.

8 Gilbert Runtime

"A really great talent finds its happiness in execution."

—Johann Wolfgang von Goethe, (1749 - 1832)

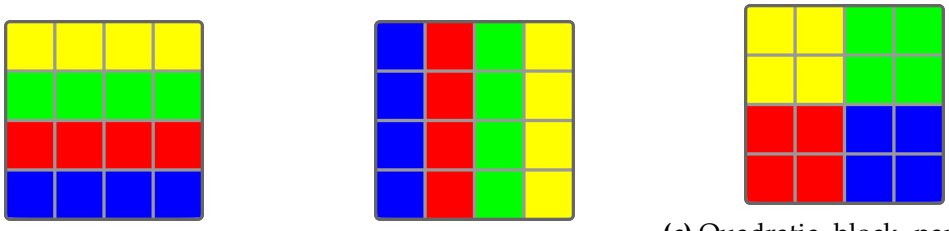
The Gilbert runtime is responsible for executing the compiled MATLAB code on a particular platform. For this purpose, it receives the intermediate representation of the code and translates it into the execution engine's specific format. Currently, Gilbert supports three different execution engines: *ReferenceExecutor*, *StratosphereExecutor* and *SparkExecutor*. The *ReferenceExecutor* executes the compiled MATLAB code locally. For the distributed execution Gilbert supports the Spark and the Stratosphere system as backends.

The *ReferenceExecutor* is an interpreter for the intermediate code. It takes the dependency tree of a MATLAB program and executes it by evaluating the operators bottom-up. In order to evaluate an operator, the system first evaluates all inputs of an operator and then executes the actual operator logic. Since the program is executed locally, the complete data of each operator is always accessible and, thus, no communication logic is required. All input and output files are directly written to the local hard drive.

In contrast to the *ReferenceExecutor*, the *StratosphereExecutor* executes the program in parallel. It takes the dependency tree of a MATLAB program and translates it into a PACT plan. After the plan is generated, it is issued to the Stratosphere system for parallel execution. This approach implies that the program is not directly executed by the executor. Instead, the executor represents just another translation step.

The PACT plans are executed in parallel. Consequently, data structures are needed which can be distributed across several nodes and represent the commonly used linear algebra abstractions, such as vectors and matrices. Moreover, the linear algebra operations have to be adjusted so that they keep working in a distributed environment. Fortunately, Stratosphere offers a rich and expressive API to easily realize distributed computations. The details of the distributed data structures and operations are explained in section 8.1 and section 8.2.

The *SparkExecutor* is the second executor for distributed computations. In contrast to the *StratosphereExecutor*, it executes the MATLAB programs on top of the Spark system. Since Spark and Stratosphere offer a similar set of programming primitives, they can both operate on the same data structures. Furthermore, even most of the linear algebra operations can be implemented similarly. The only programming difference is the incremental plan roll out feature of Spark. By emitting Spark transformations and actions, the user can trigger computations in the cluster and retrieve intermediate results on the driver node. This feature allows a more interactive way of programming, manifesting in a more natural way loops are defined, for example.



(a) Row partitioning **(b)** Column partitioning **(c)** Quadratic block partitioning

Figure 8.1: Row-wise, column-wise and quadratic block-wise matrix partitioning.

However, these differences are only subtle and do not limit or extend the expressiveness of either system.

8.1 Distributed Matrix Representation

One aspect of writing distributed algorithms is how the relevant data is distributed across several worker nodes. Since the distribution pattern directly influences the algorithms, one cannot consider them independently from one another. In Gilbert's use case, the main data structure are matrices. Thus, a partitioning for matrices has to be conceived which allows efficient algorithms working on the distributed data. Looking at a matrix, one easily finds a multitude of different partition schemes.

A first idea could be to partition a matrix according to their rows or columns, as it is depicted in fig. 8.1(a) and fig. 8.1(b). This scheme allows to represent a matrix as a set of vectors which is stored in a distributed fashion. Furthermore, it allows an efficient realization of cellwise operations, such as $+, -, /$ or $.*$. In order to calculate the result of such an operation, we only have to join the corresponding rows of both operands and execute the operation locally for each pair of rows.

However, this approach unveils its drawbacks when multiplying two equally partitioned matrices A and B as illustrated in fig. 8.2. In such a case, the row r of A and the complete matrix B is needed to calculate the resulting row with index r. This circumstance implies a complete repartitioning of B. The repartitioning is especially grave, since B has to be broadcasted to every row of A.

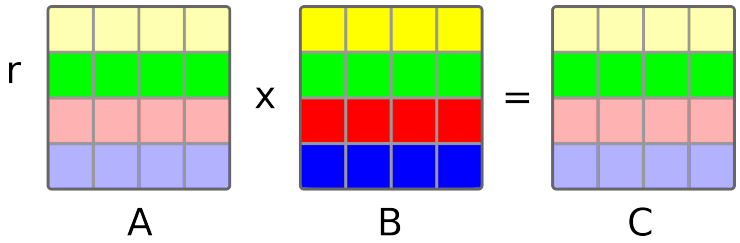


Figure 8.2: Matrix multiplication of A and B. The required data to calculate row r are highlighted.

In order to quantify the repartitioning costs of such a matrix multiplication, a simple cost model is developed. First of all, it is limited to modeling the communication costs, since network I/O usually constitutes the bottleneck of distributed systems and is thus the dominating factor. For the sake of simplicity, the multiplication of two quadratic matrices $A \in \mathbb{R}^{n \times n}$ and

$B \in \mathbb{R}^{n \times n}$ giving the matrix $C \in \mathbb{R}^{n \times n}$ is considered. Moreover, we assume that there are n worker nodes available, each of which receiving a single row of A and B, respectively. Thus, A and B are row-wise partitioned. We further assume that rows with the same index are kept on the same worker node.

Each row a_r requires the complete knowledge of matrix B in order to produce the row c_r. Therefore, every row b_r has to be sent to all other worker nodes. Thus, the number of rows sent by each worker node is $n-1$. All of the n worker nodes have to do the same. Consequently, the total number of sent messages is $n(n-1)$ and each message has a size of $n\alpha$ where α is the size of a matrix entry. Usually, each sending operation causes some constant overhead inflicted by resource allocation. Before sending the actual data over the network, memory to transfer the data to the network interface has to be allocated, network resources have to be reserved and a network connection with the remote peer has to be established. This overhead is denoted by Δ. Since all sending operations occur in parallel, the costs caused by constant overhead are $(n-1)\Delta$. The total amount of data, which has to sent over the network, is $n^2(n-1)\alpha$. The network interconnection is assumed to guarantee every node a bandwidth ν. Therefore, the time needed for sending the data is $\frac{n^2(n-1)\alpha}{\nu}$. These considerations lead to the following repartitioning cost model:

$$cost_{row} = \frac{n^2(n-1)\alpha}{\nu} + (n-1)\Delta$$

Row and column partitioning are extreme forms of blocking. A less extreme form would be to split the matrix into equally sized quadratic blocks as shown in fig. 8.1(c). In order to identify the individual blocks, each of them will be assigned a block row and block column index assigned. Thus, blocking adds some memory overhead in the form of index information. An example of a 4×4 matrix partitioned into 2×2 blocks can be seen in fig. 8.3.

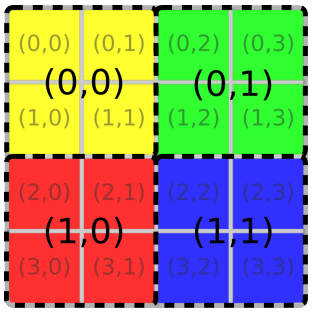


Figure 8.3: Detailed quadratic block partitioning with the added block row and block column indices.

The blocks are distributed across the worker nodes. The block size directly controls the granularity of the partitioning. Increasing the block size will reduce the memory overhead of distributed matrices while reducing the degree of parallelism. Thus, the user has to adjust the block size value depending on the matrix sizes and the number of available worker nodes in order to obtain best performance.

The quadratic block partitioning has similar properties like the row- and column-wise partitioning scheme when it comes to cellwise operations. We simply have to join corresponding blocks with respect to the pair of block row and column index and execute the operation on

matching blocks locally. But how does this pattern performs for matrix multiplications?

The assumptions are the same as before and additionally it is assumed that n is a square number. Since the matrices A, B and C are equally partitioned into square blocks, indices will henceforth reference the block and not the matrix entry. In order to calculate the block c_{ij}, we have to know the block row a_i and the block column b_j. The resulting block will be stored on the node n_{ij} which already contains the blocks a_{ij} and b_{ij}. Thus, each node n_{ij} has to receive the missing $2\left(\sqrt{n}-1\right)$ blocks from the block row a_i and block column b_j. In total, all worker nodes have to send $2n\left(\sqrt{n}-1\right)$ blocks. Each block has the size $n\alpha$. The total communication costs comprises the transmission costs and the network overhead:

$$cost_{squareBlock} = \frac{2n^2\left(\sqrt{n}-1\right)\alpha}{\nu} + 2\left(\sqrt{n}-1\right)\Delta$$

We see that the term $(n-1)$ is replaced by $2\left(\sqrt{n}-1\right)$ in the square block cost model. For $n > 2$, the square block partitioning scheme is thus superior to the row- and column-wise partitioning pattern with respect to the cost model. The reason for this outcome is that the square blocks promote more localized computations compared to the other partitionings. Instead of having to know the complete matrix B, we only have to know one block row of A and one block column of B to compute the final result.

Due to these advantages and also considering its simplicity, we decide to implement the square block partitioning scheme in Gilbert. It would also be possible to combine different partitionings and select them dynamically based on the data sizes and input partitionings.

Besides the partitioning, Gilbert also has to represent the individual matrix blocks. There exist several storing schemes for matrices depending on the sparsity of the matrix. For example, if a matrix is dense, meaning that it does not contain many zero elements, the elements are best stored in a continuous array. If a matrix is sparse, then a hash map or some form of compressed representation are best suited. Gilbert chooses the representation of each block dynamically. Depending on the non-zero elements to total elements ratio, a sparse or dense representation is selected.

8.2 Linear Algebra Operations

The other aspect of writing distributed algorithms is to think about how the algorithms work with the locally available data and which data has to be communicated in order to compute the final result. Stratosphere and Spark both offer a highly expressive programming API, which follows and extends the well-known MapReduce paradigm [26]. Conceptually, a set of distributed data items, called *DataSet* or *RDD* (resilient distributed dataset) in the context of Stratosphere and Spark, respectively, forms the basis of both systems. There are several ways to initially create a data set, such as reading from a file or distribution of a local collection. Once a distributed data set is created, it can be modified by applying one of the transforming functions, namely *map*, *reduce*, *join*, *cross* or *coGroup*. These functions are borrowed from the world of functional programming, since they happen to be expressive enough and can serve as a utile abstraction for automatic parallelization. Each of these transforming functions takes one or more data sets as inputs and produces a new distributed data set. The distributed data sets are not computed directly but instead they generate a dataflow plan which is lazily executed later on. Each node of the dataflow plan constitutes a function application and consequently a new data set. The

edges connect input data sets to output data sets and thus represent the data dependencies.

In order to understand how the different linear algebra operations are implemented, we will quickly revise the semantics of the transforming functions. For a more detailed explanation, the reader is referred to [79, 3, 11]. A distributed data set is a multiset of items. An item can have a key value, which is retrieved by the *key* function. Each of the transforming functions is called with a user defined function (UDF), which specifies how each item is processed and what kind of item is emitted as the result. In other words, it defines the program specific semantics of the transformation.

map The map operator is called with one input data set A and a UDF. The UDF is called for each item $a \in A$ independently, producing one result item.

reduce The reduce operator partitions the input data set A into groups of items with the same key. All items of each group are handed together to a call of the UDF. In other words, the UDF is called for each submultiset A' with $\forall a, b \in A' : key(a) = key(b)$ and $\forall a \in A', b \in A \setminus A' : key(a) \neq key(b)$.

join The join operator joins two data sets A and B with respect to a key value. This means that a UDF is called for each pair (a, b) with $a \in A$ and $b \in B$ with $key(a) = key(b)$.

cross The cross operator can be understood as the Cartesian product. Given two data sets A and B, cross calls the UDF for each pair (a, b) with $a \in A$ and $b \in B$.

coGroup The coGroup operator also takes 2 input data sets A and B. It groups the elements of A and B according to their keys and joins the grouped submultisets. In other words, the UDF is called for each pair of multisets (A', B') with $A' \subseteq A \wedge B' \subseteq B \wedge |keyset(A' \cup B')| = 1$ and $keyset(A' \cup B') \cap keyset((A \cup B) \setminus (A' \cup B')) = \emptyset$. The function *keyset* is the set of appearing keys in a set: $keyset(X) := \{key(x) \mid x \in X\}$.

Stratosphere as well as Spark offer a similar set of 2*nd*-order functions comprising the above-mentioned functions. Consequently, we can develop the linear algebra operations in a general fashion and do not have to adhere to a specific framework. In the following, we will outline the implementation behind the different intermediate operators with respect to the transforming functions. We assume that the matrices are partitioned using the square block scheme.

The `ScalarMatrixTransformation` and `MatrixScalarTransformation` work on a matrix input and a scalar input. The scalar input is a distributed data set with one item. The scalar value is required at every matrix block to perform the `smOp` operation. Thus, we use the cross function to pair the scalar value with every matrix block. The resulting dataflow plan is shown in fig. 8.4.

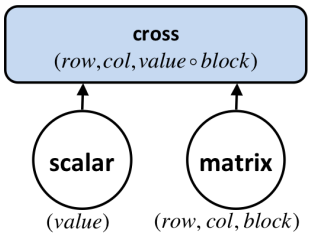


Figure 8.4: Dataflow plan of the `ScalarMatrixTransformation`.

The `CellwiseMatrixTransformation` operates locally on the cells of a single matrix.

Thus, no communication is needed and the `unaryScalarOp` operation f can be executed embarrassingly parallel. The transformation is implemented using the map function. The resulting dataflow plan is shown in fig. 8.5.

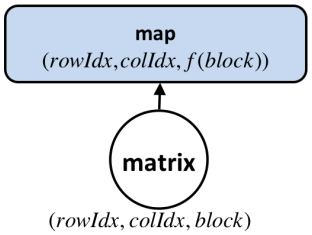


Figure 8.5: Dataflow plan of the `CellwiseMatrixTransformation`.

The `CellwiseMatrixMatrixTransformation` operates locally on the cells of two matrices. It applies an operation of type `scalarOp` to each pair of corresponding cell entries. Therefore, we have to pair all matrix blocks which have the same block row and block column index. Assuming, f is the function performing the cellwise operation on the given matrix blocks, the resulting dataflow plan can be seen in fig. 8.6. The join function is represented as a join and a map node in the dataflow plan. The map function is called for each of the matching pairs created by the join node.

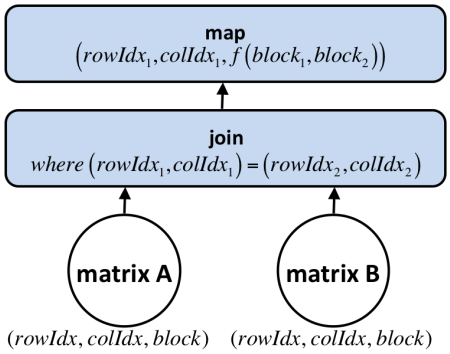


Figure 8.6: Dataflow plan of the `CellwiseMatrixMatrixTransformation`.

Matrix multiplications are probably the most critical operation for performance in linear algebra programs. Therefore, we have to pay attention to a thorough implementation within Gilbert. In a MapReduce-like system there exist two distinct matrix multiplication implementations for square blocking. The first approach is based on replicating rows and columns of the operands and is called replication based matrix multiplication (RMM). The other method is derived from the outer product formulation of matrix multiplications. It is called cross product based matrix multiplication (CPMM). The RMM and CPMM methods have been implemented within SystemML [34].

Let us assume that we want to calculate $A \times B = C$ with A, B and C being matrices. The block size has been chosen such that A is partitioned into $m \times l$ blocks, B is partitioned into $l \times n$ blocks and the result matrix C will be partitioned into $m \times n$ blocks. In order to reference the block in the ith block row and jth block column of A, we will write A_{ij}. A block row will be denoted by a single subscript index and a block column by a single superscript index. For

example, A_i marks the ith block row of A and A^j the jth block column of A.

The replication-based strategy will copy for each C_{ij} the ith block row of A and the jth block column of B. The replicated blocks of A_i and B^j will be grouped together. These steps can be achieved by a single mapper. Once this grouping is done, the final result C_{ij} can be calculated with a single reducer. The reduce function simply has to calculate the scalar product of A_i and B^j. It is important to stress that A_i is replicated for each $C_{ik}, \forall k$. The whole process is illustrated in fig. 8.7.

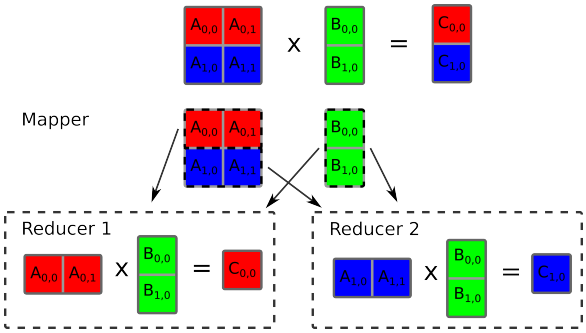


Figure 8.7: Replication based matrix multiplication with MapReduce.

In contrast to RMM, CPMM calculates the outer products between A^k and B_k for all k. A mapper can group the A^k and B_k together so that a reducer can compute the outer products. Consequently, this method does not replicate any data. The outer product produces intermediate result matrices C^k which have to be added up to produce the final result $C = \sum_{k=1}^{l} C^k$. This summation can be achieved by a subsequent reducer. The whole process is illustrated in fig. 8.8.

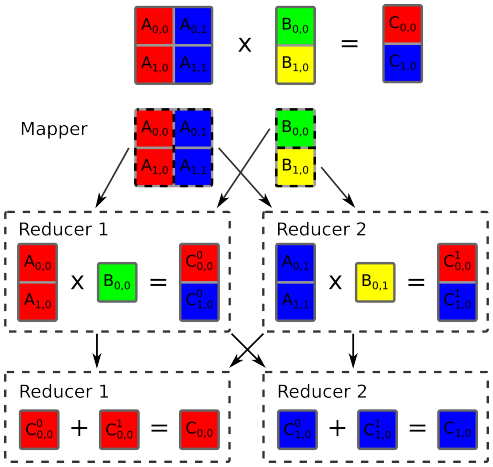


Figure 8.8: Cross product matrix multiplication with MapReduce.

The methods RMM and CPMM differ in terms of network communication. The former method can be realized within a single MapReduce job whereas the latter requires two. Neither RMM nor CPMM is always superior. The optimal matrix multiplication strategy depends on the matrix size of its operands A and B.

Fortunately, Stratosphere and Spark exhibit a little bit more flexibility in terms of higher order functions. Freed from the tight corset of the MapReduce world, matrix multiplications can be expressed more subtly. Looking at the definition of the matrix multiplication for $C_{ij} = \sum_{k=1}^{l} A_{ik} \times B_{kj}$, it can be seen that every A_{ik} has to be joined with its corresponding $B_{kj}, \forall k$. This pairwise mapping can be easily achieved by using the join function. The join-key is the column index of A and the row index of B. The joiner calculates for each matching pair A_{ik} and B_{kj} an intermediate result C_{ij}^{k}. Grouping the intermediate results with respect to the index pair (i, j) allows us to compute the final result in a subsequent reduce step. The overall algorithm strongly resembles the CPMM. The corresponding dataflow plan is shown in fig. 8.9.

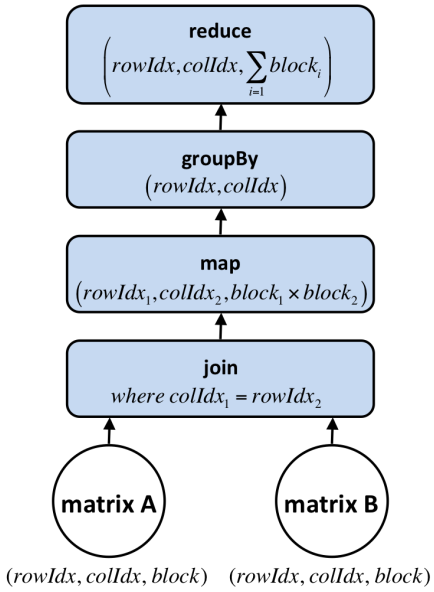


Figure 8.9: Dataflow plan of the `MatrixMult` operator.

Stratosphere supports several execution strategies for the higher-order functions. A cost-based optimizer selects the best strategies prior to execution. One possible optimization concerns the join function. The join can either be realized using a hybrid-hash join or a sort-merge join algorithm depending on the current partitioning and the input data sizes.

If one input data is relatively small compared to the other input, it is usually more efficient to use the hybrid-hash join algorithm. Without loss of generality, we assume that the matrix B constitutes such a small input. If we further assume that the block rows of A are kept on the same worker node, then the last reduce operation can be executed locally and without any shuffling. The resulting execution plan under these assumptions is equivalent to the RMM.

If the system chooses the sort-merge join algorithm instead, then the columns of A will be distributed across the worker nodes. Consequently, the last reduce step causes a mandatory repartitioning. Then, the resulting execution plan is equivalent to the CPMM.

Even though Gilbert has only one dataflow plan specified to implement the matrix multiplication, the Stratosphere system can choose internally between the RMM and CPMM strategy. The strategy is selected by the optimizer which bases its decision on the data size and the partitioning, if available. Consequently, Stratosphere frees the programmer from this decision.

The `VectorwiseMatrixTransformation` operator cannot be generalized independently of the `vectorwiseOp` operation. For example, consider the maximum and the norm operation. The maximum is calculated by taking the maximum of each row in a block. Afterwards, the maximum per block is grouped with respect to the block row index and the grouped blocks are reduced by taking the maximum over all group elements. Grouping is based on the key of each item. The corresponding dataflow plan is shown in fig. 8.10.

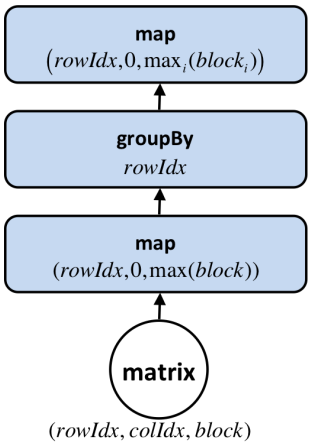


Figure 8.10: Dataflow plan of the `VectorwiseMatrixTransformation` with the `max` operation.

In contrast to that, the 2-norm operation requires a more sophisticated implementation. First, the cellwise square of all matrix entries is calculated with the map function. Then, the partial sums of every row are computed by summing the columns of each block. This calculation constitutes another map operation. The final row sums are calculated by grouping the partial sums with respect to their block row index and summing the items of each group up. After this reduce operation is done, the final result is computed by taking the cellwise square root.

The `AggregateMatrixTransformation` operator computes an aggregate over all elements of the matrix. Given that the aggregate operation is combinable, meaning that the aggregation can be expressed as a fold operation in terms of functional programming, we can implement it straightforwardly. We only need a reducer with the aggregation function f as UDF. The respective dataflow plan can be found in fig. 8.11. The $default$ variable holds the initial value for the fold operation. In case that the aggregate is not combinable, it has to be implemented individually.

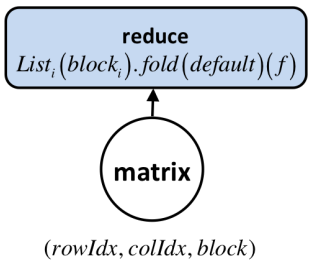


Figure 8.11: Dataflow plan of the `AggregateMatrixTransformation`.

Part III

Architecture & Implementation

9 Architecture

"We shape our buildings; thereafter they shape us."

—Winston Churchill, (1874 - 1965)

Gilbert provides a Matlab-like language for distributed sparse linear algebra operations. Being such a system, it comprises the complete stack of functionalities necessary to implement a programming language. At first, the system has to divide the given source code into tokens. These tokens are parsed and an abstract syntax tree (AST) is created. In order to generate the intermediate representation (IR), it is necessary to assign types to the occurring expressions. This assignment is done by the typing system. Afterwards, the compiler can translate the Gilbert program into its IR. The IR is well suited to apply high-level transformations to the program. At last, the execution plan generator translates the IR into an execution plan which can be executed in parallel.

Since we had to implement the whole stack of functionalities of a programming language, Gilbert became quite complex. In order to control this complexity, Gilbert's functionalities were separated into different layers with distinct tasks. The layered system architecture is shown in fig. 9.1. The separation of concerns limits the complexity of each layer so that each one of them becomes manageable. In cooperation, though, they implement a highly elaborate system.

The first layer is the language layer. It contains all functionality to parse the given Gilbert code and to compile it into the IR. The second layer is the intermediate layer and it receives the IR of the Gilbert code. The intermediate format is the ideal representation to apply language independent high-level transformations. Consequently, the second layer hosts the optimizer. The Gilbert optimizer applies several algebraic optimizations prior to the generation of the execution plan. The execution plan generator translates the optimized IR into a specific execution plan, depending on the selected execution engine. Once the program has been translated into an execution engine's specific plan, it is executed on the respective back end.

9.1 Language Layer

The language layer contains all the logic for translating Gilbert source code into the intermediate representation, see chapter 6 for more details. The layer itself is subdivided into three layers. The first of these sublayers is the lexer and parser of MATLAB code. Gilbert uses a recursive descent parser with backtracking, which is capable of parsing any LL(*) grammar. Consequently, it can also parse Gilbert's front end language as specified in section 4.2.

Once the MATLAB code is parsed, it is given to the next layer, namely the typing layer. Here, the produced AST is attributed with type information. This information is inferred automati-

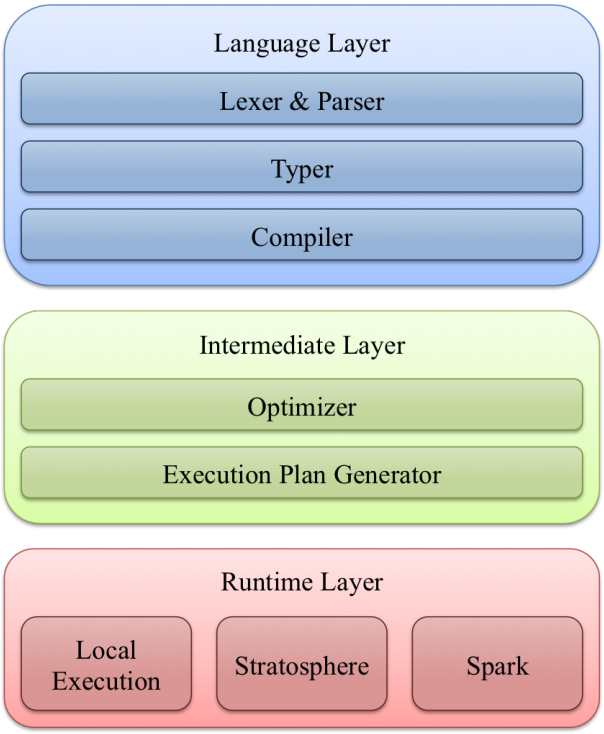


Figure 9.1: The layered system architecture of Gilbert. The language layer is responsible for parsing, typing and compiling the given Matlab code. The intermediate layer facilitates high-level optimization strategies. The runtime layer is responsible for executing the specified program in parallel.

cally from the source code without any need for explicit type annotations. The typer makes use of the Hindley-Milner type inference system, as it is described in section 5.2. After the types have been inferred, a new AST with additional type information is generated.

The type information enriched AST is given to the last sublayer. The compiler is responsible for translating the AST into a front end independent representation of linear algebra operations, namely the IR. The format Gilbert uses to represent linear algebra operations in a generalized format is described in chapter 6. In fact, it is also some form of AST just without the peculiarities of MATLAB.

9.2 Intermediate Layer

The intermediate layer contains the optimizer and the execution plan generator. The optimizer works on the intermediate representation of the Gilbert program. It applies the optimizations described in chapter 7. The execution plan generator's task is to translate the intermediate format into the execution engine's specific format. Currently, the system supports local execution and two engines for distributed execution, namely Stratosphere and Spark.

For the local execution, Gilbert employs an interpreter. The interpreter directly executes the intermediate representation. Beginning at the root of the dependency tree, the interpreter utilizes a recursive-descent strategy to evaluate each node. In order to evaluate a given node of the tree, the interpreter first descends to its children. After having retrieved the results of the

children nodes, the parent node is evaluated.

The distributed execution engines do not support immediate execution of the individual nodes. Instead, the plan generator has to create an execution plan for Stratosphere and Spark, which is lazily executed upon submission or when Spark's actions are called. Fortunately, Stratosphere and Spark offer a similar programming API. Thus, Gilbert can translate the distinct linear algebra operations using the same general building blocks as described in section 8.2. The execution plan is generated using a recursive-descent approach similar to the local interpreter. Once the intermediate representation has been transformed, the plan is sent to the respective system. The system is responsible for the parallel execution.

9.3 Runtime Layer

The runtime layer marks the transition from Gilbert to one of the supported execution engines. It is more of a conceptual abstraction of the Gilbert system. Gilbert supports 3 different engines at the moment, but it should be easy to add support for further execution systems.

10 Implementation

"In any moment of decision, the best thing you can do is the right thing, the next best thing is the wrong thing, and the worst thing you can do is nothing."

—Theodore Roosevelt, (1858 - 1919)

The development of Gilbert involved the implementation of the complete technology stack of a programming language. For each technology, we had to decide how to implement it. The lexer and parser, for example, can be automatically generated using tools such as ANTLR [5], Bison [14] or Yacc [78]. However, using these tools would have required to manage the complexity of several frameworks and probably a lot of boilerplate code to fuse them together. Therefore, we looked for a tool with the potential to implement the complete stack of Gilbert.

Fortunately, the Scala language [73, 56] unifies all required technologies under one umbrella. Scala is a highly scalable language, well suited for script sized programs as well as enterprise applications. The language combines object oriented and functional programming, giving a maximum of flexibility to the programmer. The Scala code is compiled to Java bytecode and thus executed on a JVM. This feature makes the language platform independent to the greatest possible extent. Furthermore, it can integrate existing Java code and thus benefit from the rich set of Java libraries. Last but not least, it is like the new cool kid on the block of programming languages, which everyone likes.

An important aspect of our choice was how easily a parser and a compiler can be implemented with Scala. If one decides to implement these programs within a popular language such as Java, C/C++ or C# and without any additional tools, then it quickly becomes a tedious and error-prone task. That's the reason why tools like Yacc and ANTLR have emerged. Scala circumvents this problem by providing an internal domain specific language (DSL) for an easy and quick development of parsers and lexers. This DSL is also known as Scala Parser Combinators. Once an abstract syntax tree has been generated using these parser combinators, it is very easy to develop a compiler using Scala's pattern matching functionality. The pattern matching capabilities of Scala are similar to functional languages, such as Haskell, ML or OCaml. It can even be applied on object hierarchies, making it a powerful tool for OOP and functional programming likewise. The Scala Parser Combinators generate recursive descent parsers which are capable of parsing LL(*) grammars.

We decide to develop Gilbert as an open source project so that everyone can benefit from the project. The complete source code is hosted on Github and is publicly accessible under `https://github.com/gilbert-lang/gilbert`.

10.1 Math-Backend

In chapter 8 we have explained the representation of distributed matrices and how the intermediate operators are mapped to dataflow plans. What was left out, though, is how the local operations on the block-level are implemented. Consider, for example, the matrix multiplication of two distributed matrices. The result is obtained by joining the blocks of both operands, performing a local matrix multiplication on a matching block pair and reducing the intermediate results. The local matrix multiplication is self-contained and has to be executed by some algorithm. Since there already exist highly optimized algorithms for different matrix representations, dense or sparse, we do not have to reinvent the wheel.

One of the numerous linear algebra libraries for the Java ecosystem is Mahout [8]. The math library of Mahout is mainly written in Java and thus offers a Java binding. Initially, Mahout was used to implement the local linear algebra operations of Gilbert. However, it quickly turned out that Mahout lacked reliable support for sparse matrices and suffered from poor performance. Since Gilbert is geared towards being a linear algebra environment for sparse matrices with descent performance, we were forced to look for alternatives. Next we came across Breeze [62], a library for numerical processing written in Scala.

Breeze offers data structures for all linear algebra primitives, such as matrices and vectors. Additionally, it supports sparse and dense variants. For each version, Breeze is shipped with elaborate algorithms implementing the linear algebra operations. In case that the host has a BLAS or LAPACK library installed, which is compiled and optimized for the underlying architecture, Breeze automatically detects and uses it. That way, Breeze can achieve near optimal performance when multiplying dense matrices compared to compiled languages such as C/C++ and Fortran. Under the hood, Breeze relies on the netlib-java library for this feature. The fact that it offers a Scala binding allowed it to be integrated seamlessly into Gilbert. We have chosen this library as our principal math back end because it is clean, very reliable and extremely powerful.

Additionally, Gilbert also supports the aforementioned Mahout library as a secondary math back end. The user can select on demand which system he wants to use depending on his personal preference. The inclusion of Mahout also emphasizes the extensible architecture of Gilbert. It is very easy to add further math libraries to Gilbert.

10.2 Execution Engines

From the very first moment, Gilbert was intended as a general purpose linear algebra front end for different runtime systems. Therefore, Gilbert includes an abstraction, called execution engine, which encapsulates the logic for running a Gilbert program. Currently, Gilbert comes with three different execution engines. The local executor serves as a reference implementation of the linear algebra operations and can be used for computing little data on a single machine. But the purpose of Gilbert was to enable linear algebra programs to handle big data and thus a distributed execution is strictly necessary. The Stratosphere and Spark executor fulfill that condition. However, Gilbert is not restricted to these systems as backends. In fact, it should be easy to add new execution engines to Gilbert. They only have to be able to map the intermediate operators to their runtime specific implementations. For example, an executor using a message passing system for the parallel execution is easily conceivable. We refrained from doing it, though, because of its error-prone implementation. We will describe the Stratosphere

and Spark parallel executors and the problems we encountered while implementing Gilbert in the following subsections.

10.2.1 Stratosphere

The Stratosphere project is still in an early stage and thus it was not expected that the development would proceed completely smoothly. The individual intermediate operators are implemented as described in section 8.2. However, the iteration mechanism of Stratosphere was slightly flawed. It was not possible to reuse expressions which were used outside of the loop. The system would generate a completely new instance of the expression in that case. This behavior not only degraded the overall performance but also corrupted the consistency of Gilbert programs. A random matrix, for example, would have had two different values: One outside of the loop and another one inside of the loop. Therefore, we had to fix this problem within Stratosphere to guarantee a correct functioning of Gilbert.

Another problem was that Stratosphere did not support the transmission of implicit values such as a context bound. These would have been necessary to implement a generic matrix being parameterized on the type of its elements. The Breeze library would have required the additional information for a proper functioning. Therefore, we had to instantiate a concrete matrix implementation for each element type we needed. This clumsy solution leads to code duplication at some points.

10.2.2 Spark

The development of the Spark executor was comparable to the Stratosphere executor. The intermediate operators could be implemented almost identically. Furthermore, we could reuse the distributed matrices defined for Stratosphere and the math back end. In conclusion, the development process proceeded without major problems.

Part IV

Evaluation & Conclusion

11 Evaluation

"An ounce of performance is worth pounds of promises."

—Mae West, (1893 - 1980)

In this chapter, we will investigate the scalability of Gilbert and its performance compared to famous hand-tuned ML algorithms. We show that Gilbert is not only easily usable in terms of programming but also produces results with decent performance. Furthermore, we compare the different execution engines and math-backends to see which system gives the best results. Based on these outcomes, we want to come up with a recommendation for the best configuration of Gilbert.

11.1 Experimental Setup

For our evaluation, we use the 400-core cluster provided by the DIMA faculty of the Technical University of Berlin. The cluster comprises 25 local machines with 32 GB of main memory per computer. Each machine is equipped with 2 AMD Opterons 6128 CPUs, each of them having 8 cores. The CPUs run at a speed of 2 GHz.

We employ Apache Spark-1.0.0 [9] for our test runs with the Spark execution engine. For the Stratosphere execution engine, we use a slightly extended version of Stratosphere-0.6-SNAPSHOT [68]. At the time of evaluation, Stratosphere-0.6 was still under development but we needed the latest features. Therefore, we use the current snapshot version. All extensions made to the current snapshot version are also pending pull requests and hopes are high that the final release will contain them all. Thus, executing Gilbert with the stable release Stratosphere-0.6 should work perfectly fine. As the underlying distributed file system both systems use Apache Hadoop-1.2.1 [6].

The measured execution times on the cluster can slightly differ from measurement to measurement because of non-deterministic factors, such as cache misses, load caused by the operating system and network communication. In order to cancel out the noise inflicted by these factors, we measure the execution time for each experimental setting five times and report the mean of the measurements. That way, we reduce the overall variance of the measurements.

11.2 Scalability

The scalability evaluation investigates how Gilbert behaves under increasing work loads and how well it can exploit varying cluster sizes. As we have implemented Gilbert to provide a scalable linear algebra environment, it is important that it can process data sizes exceeding the main memory of a single machine. We want to evaluate the scalability performance for

Stratosphere's and Spark's execution engine.

11.2.1 Matrix Multiplication

As a first benchmark, we choose the matrix multiplication $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ with n being the dimensionality. The matrix multiplication operation is demanding, both in CPU load as well as network I/O. The implementation of the matrix multiplication, shown in section 8.2, first joins the column blocks of the left matrix with the row blocks of the right matrix. This operation replicates the two operands partially and sends them across the network to their respective worker nodes. For each matching pair of blocks, a local matrix multiplication is executed. The local matrix multiplication has a complexity of $\mathcal{O}(n_{block}^3)$ with n_{block} being the block size. The produced intermediate results are grouped according to their left row and right column index. Finally, the grouped result blocks are added up to give the final result.

The matrices A and B are sparse matrices with uniformly distributed non-zero cells. They are randomly generated prior to the matrix multiplication, thereby avoiding costly file system I/O. The sparsity of both matrices is set to 0.001. Therefore, both matrices are represented by sparse matrices. As a baseline, we run the same matrix multiplication on a single machine of the cluster using Gilbert's local execution engine. The local execution engine uses the same math-backend as the distributed engines and thus serves as a good reference value to see the additional costs of parallel execution. Breeze is chosen as the math back end for the matrix multiplication. The Stratosphere and Spark execution engines are both started with 20 GB of main memory for their task managers. Furthermore, they are both configured to use a similar scheduling strategy, which distributes the work equally among the available computer nodes. This aspect is especially important in order to compare the results, because some parts of Breeze are multi-threaded and, consequently, take advantage of idling cores.

11.2.1.1 Increasing Problem Size

In the first experiment, we fixed the block sizes to 500×500 and set the number of cores to 50. We then increased the dimensionality n of A and B to observe the runtime behavior. The resulting execution times for the local, Stratosphere and Spark execution engines are shown in fig. 11.1(a).

On a single machine, we are able to execute the matrix multiplication for dimensionalities up to $n = 10000$ before the system ran out of memory. We can see that the local execution performs better for dimensionalities $n \leq 5000$. That is expected since the matrix still fits completely in the main memory of a single machine and the distributed execution adds some significant communication overhead. Furthermore, Spark and Stratosphere both exhibit some noticeable job start up latency which is dominating the execution time for $n \leq 10000$. Those are the reasons why the local executor performs better than the distributed implementations for small matrix sizes.

For matrices with $n > 5000$, Spark starts to calculate the matrix multiplication faster than the local executor. The Stratosphere execution engine does not beat the local computation for sizes which are manageable by a single machine, though. However, we can observe a steeper ascent of the local execution time compared to Stratosphere.

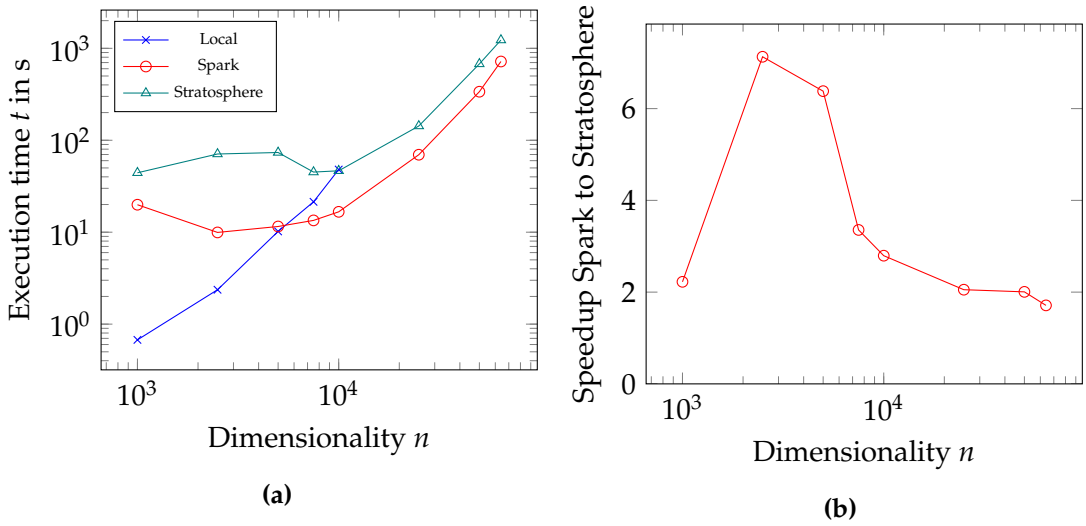


Figure 11.1: Scalability of matrix multiplication on a 50-core cluster. (a) Execution time of matrix multiplication depending on the data size. (b) Speedup of Spark's execution engine compared to Stratosphere's.

We also see that Gilbert can handle matrix sizes which scale far beyond the memory capacity of a single machine. The matrix multiplication for $n = 64000$ is finished in 719 s on Spark and in 1230 s on Stratosphere. We can observe that the Spark execution engine runs consistently faster than Stratosphere. Since the work load is similar for both systems, the difference has to be caused by the internal functioning of both systems. For smaller dimensionalities, the execution time of both systems stays almost the same. Therefore, it has to be mainly caused by the job start up times. We can consequently infer that Spark is capable of starting its jobs faster than Stratosphere.

Not only does Spark starts its jobs faster, but it also performs better than Stratosphere without exception. The speedup of Spark's execution engine compared to Stratosphere's execution engine calculating the matrix multiplication is given in fig. 11.1(b). We can observe that there is a peak speedup of 7 for $n = 2500$. However, we attribute this value to errors of measurement, because for $n > 10000$ the speedup evens out at about 2 with a slight decrease for $n = 64000$.

11.2.1.2 Increasing Cluster Size

In the second experiment, we investigate the scaling behavior of Gilbert with respect to the cluster size. As a benchmark, we calculate again $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ and n being the dimensionality. In order to observe the inflicted communication costs, we keep the work load per core constant while increasing the number of cores. For this experiment we vary the number of cores from 1 to 100 and scale n such that $n^3/\#cores$ is constant. We started with $n = 7500$ for a single core and reached $n = 35000$ on 100 cores. As block size we chose 500×500. The results of the experiment are shown in fig. 11.2.

The optimal scale-out behavior would be a horizontal line. However, it is impossible to achieve this scale-out behavior due to communication overhead inflicted by parallel execution. Nonetheless, the results depicted in fig. 11.2 indicate for both execution engines decent scale-

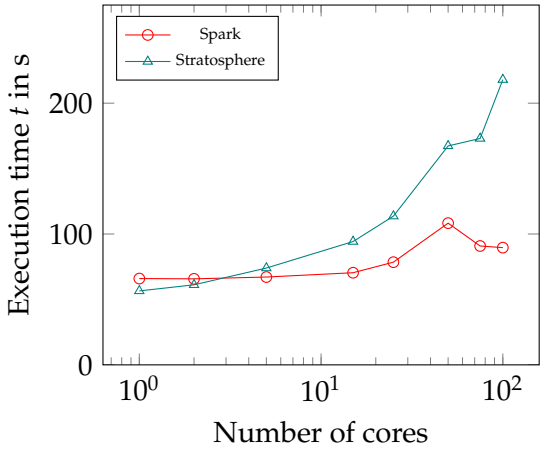


Figure 11.2: Execution time of matrix multiplication depending on the cluster size with constant work load per core.

out behavior. Especially for the number of cores ≤ 15, we observe for Stratosphere and Spark an almost horizontal line. From this point onwards, the scaling of Stratosphere degrades faster than Spark's scaling. Spark exhibits an outstanding scale-out behavior. The Spark execution engine requires 90 s to calculate the matrix multiplication with $n = 35000$ on 100 cores. That is only a slowdown by a factor of 1.5, if compared to the matrix multiplication with $n = 7500$ on a single core, which takes 65 s to finish. Interestingly, Spark's graph is not monotonic, as one would expect it. For #cores $= 50$, we can observe a slight peak in the execution time. This behavior is counterintuitive. We suppose that this peculiarity is linked to the internal functioning of the Spark system.

11.2.2 Gaussian Non-negative Matrix Factorization

As second benchmark for evaluating the scalability properties, we choose the Gaussian non-negative matrix factorization (GNMF) algorithm [63]. GNMF finds for a given matrix V a factorization W and H such that $V \approx WH$ holds. The algorithm is a popular ML algorithm which finds its application in computer vision, document clustering and topic modeling. In the context of topic modeling, we would have d documents and a set of w words which are contained in the documents. The goal of topic modeling is to identify the different topics and the words supporting a particular topic. For this purpose, the matrix $V = (v_{i,j})_{i=1\ldots d, j=1\ldots w}$ is defined, with $v_{i,j}$ containing the frequency of a word w_j appearing in document d_i. By specifying the number of topics t, the GNMF algorithm computes $W \in \mathbb{R}^{d \times t}$ and $H \in \mathbb{R}^{t \times w}$ such that $V \approx WH$. The row w_i of W indicates the topics the document d_i contains and the row h_i of H says which words correlate with topic t_i. The GNMF algorithm alternately updates the matrices H and W until the result converges. The algorithm is given in listing 11.1. The operator $.*$, $./$ and $*$ denote the cell wise multiplication, the cell wise division and the matrix multiplication, respectively.

We choose GNMF, because it is a ML algorithm which recently had been implemented for MapReduce systems [45]. As far as we know, the proposed MapReduce algorithm is one of the best distributed implementations of GNMF. Thus, it is well suited to assess the performance of

```
1    V = load(); % load matrix to factorize
2    W = load(); % load initial values of W
3    H = load(); % load initial value of H
4
5    while i < maxIterations
6      H = H.*(W'*V ./ W'*W*H); % udpate H
7      W = W.*(V*H' ./ W*H*H'); % update W
8      i = i + 1;
9    end
```

Listing 11.1: Non-negative matrix factorization algorithm.

Gilbert's implementation of GNMF.

For our evaluation, we calculate one step of the GNMF algorithm. We set $t = 10$, $w = 100000$ and vary the number of documents d. The matrix $V \in \mathbb{R}^{d \times 100000}$ is a sparse matrix whose sparsity is set to 0.001. The non-zero cells of V are uniformly distributed. Thus, each line of V contains roughly 100 non-zero entries, which are drawn from a Gaussian distribution. The matrices W and H are dense and initialized with random values drawn from a Gaussian distribution. As a baseline, we run the GNMF on a single machine of the cluster using the local execution engine. As math back end we choose the Breeze library, which is also used for the distributed execution engines. Like for the matrix multiplication benchmark, the task manager of Spark and Stratosphere are started with 20 GB of memory and both systems use the same scheduling strategy keeping the work load on all nodes equally distributed.

11.2.2.1 Increasing Problem Size

In the first experiment we fix the number of cores to 50 and investigate the runtime behavior for increasing values of d. We start with $d = 500$ and increase the number of rows of V to 150000. The block size of Gilbert is set to 500×500. In order to fairly compare the results with the optimized NMF MapReduce implementation proposed in [45], we re-implemented the algorithm using the Spark and Stratosphere runtime system. This hand-tuned implementation, henceforth denoted as HT-GNMF, is also executed on 50 cores. Additionally, the data is generated having the respective partitioning required by the algorithm. The execution times of HT-GNMF and Gilbert's GNMF are shown in fig. 11.3(a).

The local executor can be applied to sizes of d ranging from 500 to 1500 rows, before the data exceeded the available main memory of a single computer. As expected, the local execution performs far better for these data sizes than the distributed execution with Stratosphere. However, the GNMF and the HT-GNMF executed on Spark outperformed the local execution. That is rather surprisingly, since the distributed execution should inflict some noticeable communication overhead. But apparently the parallelism compensates for the additional communication costs.

The distributed systems can also be used for data sizes which exceed the memory of a single computer. Both distributed execution engines scale well up to the point where Spark and Stratosphere can no longer keep the data in memory. The two data flow systems have to perform internal sorting, partitioning and shuffling steps to implement the high level operations, such as join, reduce, cogroup and cross. If the memory size is not sufficient to execute these

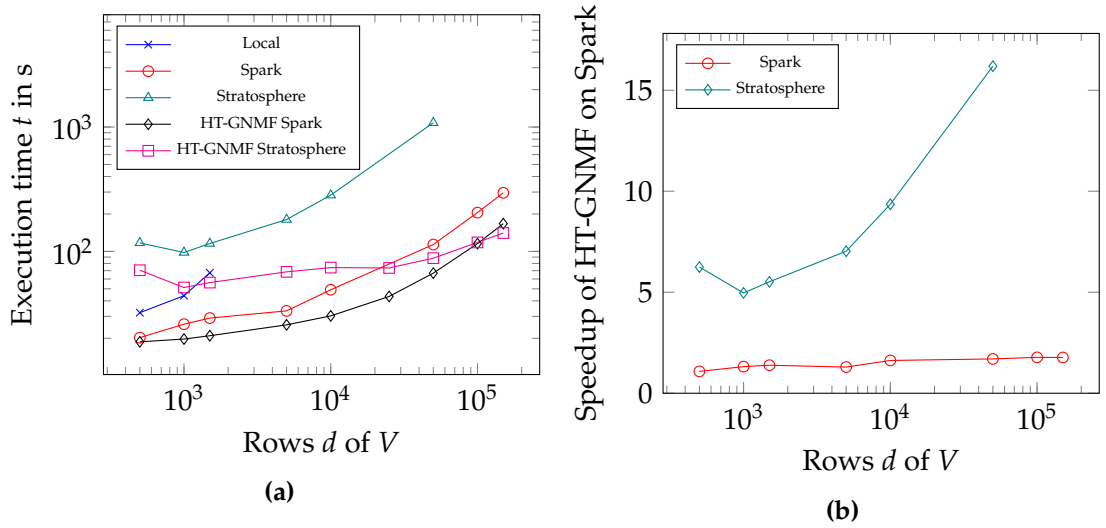


Figure 11.3: Scalability of GNMF on a 50-core cluster. (a) Execution time of one GNMF step depending on the data size. (b) Speedup of HT-GNMF running on Spark compared to Gilbert's GNMF using the Spark and Stratosphere execution engine, respectively.

steps, then the data will be gracefully spilled to disk. On the one hand, this behavior makes the systems more robust and applicable to data sizes which largely exceed the total amount of memory. But on the other hand, the performance abruptly deteriorates massively once the data has to be spilled. For the Stratosphere and Spark executor, we experienced this behavior for $d > 50000$ and $d > 150000$, respectively. For data sizes bigger than these thresholds, the benchmark runs became so slow that we could not finish them and discarded these runs as infeasible.

For Stratosphere, this behavior ensues earlier due to its specific memory management. Stratosphere assigns each cluster node a specific number of slots. The default value is the number of cores. The memory is then split evenly among the slots. Once the task managers have started and assigned the memory to each slot, it is not possible to dynamically transfer memory portions between slots. Therefore, the effectively available memory for each task is considerably smaller than the initial 20 GB. Additionally, Stratosphere's ability to support streaming further decreases the per task memory. Currently, it is assumed that all tasks belonging to one pipeline can be deployed simultaneously to one slot. In order to execute all pipeline tasks, the slot memory will be further divided by the number of tasks which can be concurrently run.

In contrast to that, Spark separates the execution of different operations into distinct stages. A stage is only submitted for execution, after all preceding stages have been completed. That way, the memory does not have to be splitted between succeeding tasks and thus spilling occurs later.

The runtime of HT-GNMF running on Spark, the Stratosphere and the Spark executor differ for varying data sizes almost by a constant factor. The fastest implementation for $d \leq 100000$ is the HT-GNMF algorithm running on Spark. The speedup of HT-GNMF running on Spark compared to GNMF on Spark's and Stratosphere's executor is shown in fig. 11.3(b). It can be seen that HT-GNMF runs approximately 1.7 times faster than Gilbert's GNMF using the Spark

executor. Compared to the Stratosphere execution engine, we can observe that HT-GNMF runs faster and faster for increasing d. HT-GNMF achieves a speedup of approximately 16 for $d = 50000$. The different runtime behaviors of Spark's and Stratosphere's execution engines are most likely caused by the Spark and Stratosphere system, since the linear algebra operations are implemented identically. The runtime behavior of HT-GNMF running on Stratosphere is interesting. At first, it performs rather poorly compared to its Spark counterpart. However, for $d > 100000$ it suddenly outperforms the Spark implementation. Thus, GNMF scales better if it is executed on Stratosphere.

Even though, Gilbert can not even reach the performance of HT-GNMF, the development using Gilbert was considerably easier. One GNMF step can be programmed in five lines of Gilbert code, whereas we needed 28 lines of Scala code for Spark's HT-GNMF and 70 lines of Scala code for Stratosphere's HT-GNMF. Not only did we have to know how to program Spark and Stratosphere, but it also took us quite some time to verify the correct functioning of both implementations. The verification was made significantly more difficult and time-consuming due to a programming bug we introduced. The debugging process showed us quite plainly how tedious the development process even with systems like Spark and Stratosphere can be. Thus, the productivity increase gained by using a high-level declarative programming language for linear algebra must not be neglected and compensates for the performance loss. Alvaro et al. [4] made a similar observations while developing a declarative programming language for distributed systems programming.

11.2.2.2 Increasing Cluster Size

In the second experiment of the GNMF benchmark, we analyze how Gilbert scales-out when increasing the cluster size while keeping the work load for each core constant. We vary the cluster size from 1 core to 100 cores and scale the number of documents d accordingly. Initially we start with 1000 documents and, consequently, calculate the matrix factorization for 100000 documents on 100 cores. The ideal behavior would be a horizontal line. However, this outcome cannot be expected, since the GNMF computation requires communication between the cluster nodes. The results of this experiment are shown in fig. 11.4.

The scale-out behavior of the Stratosphere and Spark execution engines both show good results for $\#cores \leq 5$. For higher degrees of parallelism, Stratosphere's performance quickly deteriorates. On 50 cores, Stratosphere needs 1082 s, whereas Spark needs only 113 s. We could not finish Stratosphere's computations for higher degrees of parallelism than 50, because the system simply became too slow. That vast performance decline is most likely caused by data spilling at some internal operation. In contrast to Stratosphere, Spark does not suffer from these limitations for the number of cores we tested. In fact, it almost exhibits an extraordinary scale-out behavior with an almost constant runtime. The runtime on 100 cores is only two times slower than the runtime on a single core with the same work load per core.

11.3 Block Size

The block size has a significant influence on the overall performance of Gilbert since it directly controls the data parallelism and data granularity of the local operations. The bigger the block size is, the fewer blocks are available for parallel computations. But the bigger the block size is, the more work can be done on a single computer without having to communicate with other

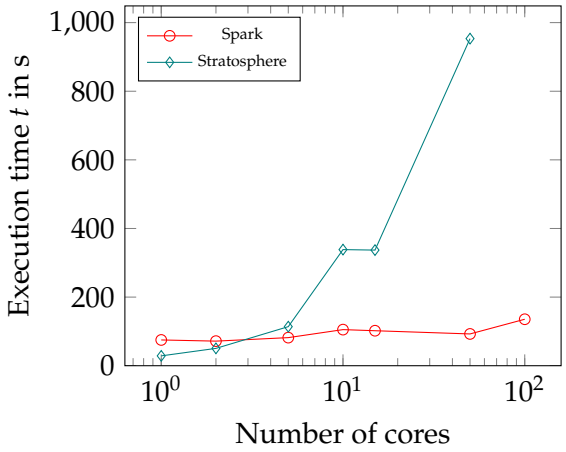


Figure 11.4: Execution time of one GNMF step depending on the cluster size with constant work load per core.

nodes. To measure the effect of the block size on the performance, we calculate a single GNMF step with varying block sizes using the Spark execution engine. For this benchmark we set $d = 100000$, $w = 100000$ and $t = 10$. The remaining parameters are set like in section 11.2.2. The execution times for different block sizes are shown in fig. 11.5(a).

The graph shows that we have a minimal runtime for a block size of 500×500. Apparently, this block size constitutes the best trade-off between data parallelism and data granularity. For lower block sizes, the additional overhead introduced by indexing information and the low data granularity increases the runtime. For higher block sizes, the decreased parallelism devours the benefits of a high data granularity. That evaluation justifies our block size choice of 500×500.

The effect of different block sizes is also compared in fig. 11.5(b) where we computed one GNMF step with the Spark execution engine for varying sizes of V. The GNMF algorithm is executed on 50 cores of the cluster using the Breeze math back end. We can observe that we obtain better results with an increased block size. Clearly, the block size 25×25 is inferior to the block sizes 500×500 and 1000×1000. However, the results between the last two block sizes do not differ in a significant way.

It is important to stress that there is no single block size which is optimal for all problems and all computer architectures. The best degree of parallelism and the best data granularity strongly depends on the problem, its inputs and the used computing machine. Therefore, the ideal block size should be determined for each problem individually.

11.4 Optimizations

In this section, we want to evaluate the effects Gilbert's optimizer has on the runtime of programs. For this purpose, we execute one GNMF step on Gilbert's Stratosphere and Spark execution engine using the same settings as in section 11.2.2. However, this time we disable the transpose pushdown and matrix-multiplication reordering optimization. The results are shown in fig. 11.6(a).

We can observe that the Gilbert optimizer has a significant effect on the runtime of GNMF.

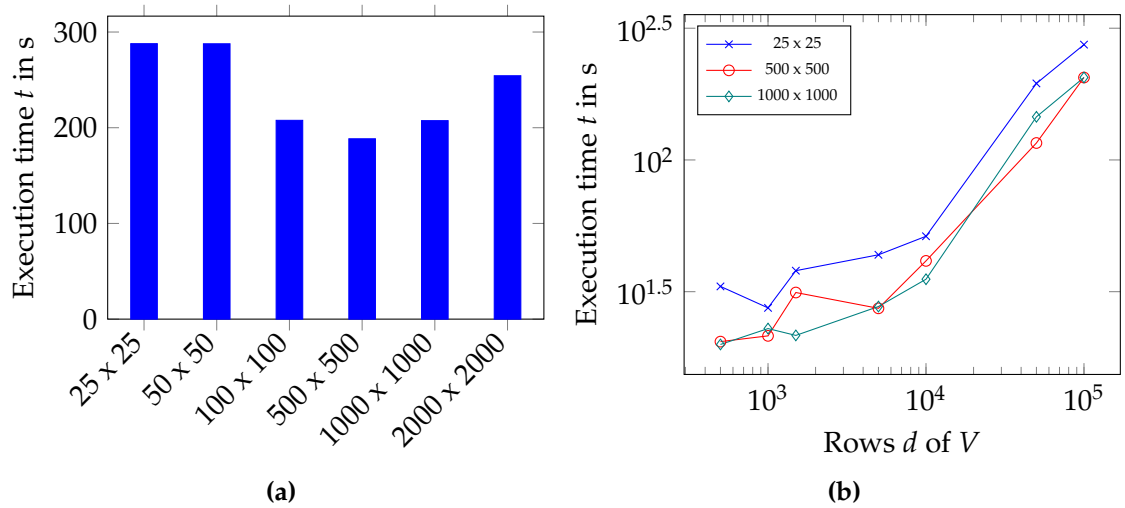


Figure 11.5: Execution time of a single GNMF step for different block and input sizes using Spark's execution engine on a 50-core cluster. (a) Execution time of a single GNMF step with constant input depending on the block size. (b) Execution time of a single GNMF step depending on the number of rows for different block sizes.

The optimized Gilbert program runs on both distributed engines faster than it's non-optimized counterparts. Interestingly, the Spark execution engine seems to benefit more from the optimizer than the Stratosphere executor. The speedups of the optimized GNMF step with respect to its non-optimized counterparts are shown in fig. 11.6(b) For the optimized program running on Spark's execution engine, a steadily ascending speedup, except for $d = 500$, can be observed. The optimized Spark version runs about 18 times faster than the non-optimized program for $d = 10000$. In contrast to that, Stratosphere shows a constant speedup of roughly 1.7 between the optimized and non-optimized version.

It is not yet known what causes these different speedups. However, looking at the GNMF formula in listing 11.1, it becomes obvious why the optimization works. There are two matrix multiplications that can be optimized: $W^T WH$ in line 6 and WHH^T in line 7. Keeping in mind that $W \in \mathbb{R}^{d \times 10}$ and $H \in \mathbb{R}^{10 \times 100000}$, we can assess the different strategies. The non-optimized version would execute the matrix multiplications due to left-associativity from left to right:

$$\overbrace{\underbrace{\left(W^T W\right)}_{\in \mathbb{R}^{10\times 10}} H}^{\in \mathbb{R}^{10\times 100000}}$$

and

$$\overbrace{\underbrace{(WH)}_{\in \mathbb{R}^{d\times 100000}} H^T}^{\in \mathbb{R}^{d\times 10}}$$

For the first matrix multiplication, the execution order is optimal, since the intermediate re-

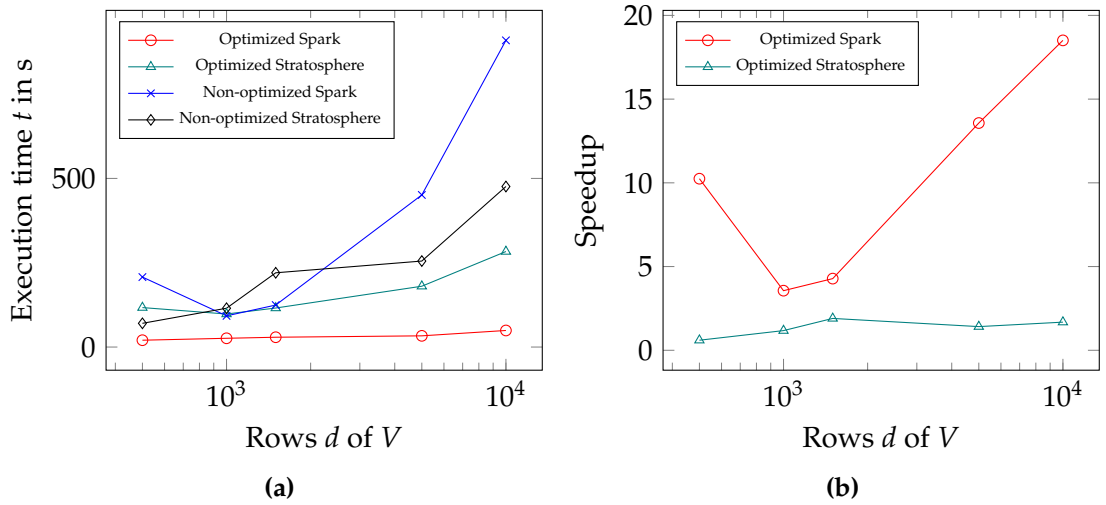


Figure 11.6: Effect of Gilbert's optimizations using the example of a single GNMF step executed on a 50-core cluster. (a) Execution time of the optimized and non-optimized GNMF step depending on the input size. (b) Speedup of the optimized GNMF step with respect to the non-optimized GNMF step depending on the input size.

sult of W^TW is much smaller than WH. However, for the second matrix multiplication, the execution order is far from optimal. The left-associativity produces an intermediate matrix result of size $d \times 100000$. For large d this matrix becomes really huge. Additionally, the intermediate result is dense, because the operands W and H are dense, as well. By changing the execution order, we can decrease the size of the intermediate result significantly.

$$\overbrace{W \underbrace{\left(HH^T\right)}_{\in \mathbb{R}^{10\times 10}}}^{\in \mathbb{R}^{d\times 10}}$$

The Gilbert optimizer first detects the matrix multiplication reordering and then changes it so that the maximum intermediate result is minimized. That is the reason why the optimized GNMF program runs clearly faster.

11.5 Gilbert Algorithms vs. Specialized Algorithms

In this section, we want to investigate how well algorithms implemented in Gilbert perform compared to specialized algorithms. We expect that the Gilbert runtime adds some overhead as trade-off for their easy-to-use programming interface. Furthermore, the high-level linear algebra abstraction of Gilbert might make it difficult to exploit certain properties to speed up the processing. Therefore, we believe that the hand-tuned algorithms will get the upper hand.

For our comparison, we chose two famous ML algorithms that can be expressed in terms of linear algebra: PageRank and k-means . Since both algorithms are iterative, we can demonstrate Gilbert's loop support. We first execute them directly in Gilbert, given the Gilbert code, and then run them directly on Stratosphere and Spark. For the direct execution, we have implemented both algorithms using the Stratosphere and Spark API. In contrast to Gilbert, the

direct implementation requires a deep understanding of the underlying runtime system. Furthermore, the distributed implementation is far from trivial compared to the linear algebra representation of the original problem.

11.5.1 PageRank

The PageRank algorithm [58] is the famous algorithm developed by Larry Page and Sergey Brin to compute a ranking between any kind of entities with reciprocal quotations and references. Initially, it was developed to rank the web sites of the world wide web. The algorithm did the ranking so well that Google grew quickly into a multi-billion dollar company. PageRank is only one algorithm comprising Google's web search, but it is probably one of the best known.

11.5.1.1 Algorithm

The idea of PageRank is to estimate the rank of a web site based on the ranks of other web sites, which link to the former site. It is assumed that high quality web sites are more likely to link to other high quality web sites. Thus, these sites get a high rank with which they can "vote" for other sites.

The PageRank can also be explained by the model of a "random" surfer. Assume there is a surfer who randomly follows an outgoing link from a web site. Occasionally, or if he ends up in a dead end, the surfer enters a random URL in his address bar. That way, he will eventually visit the whole web. By tracking the time he spends on each web site, an importance measure for each site is obtained. Web sites, which are linked by pages on which the random surfer spends more time, will receive more clicks by the random surfer and thus a higher importance. That importance measure is in fact the PageRank.

Even though the problem seems to be self-referencing, the PageRank vector turns out to be the solution to an eigenvalue problem. Thus, the PageRank vector can be easily computed using a power iteration. PageRank's MATLAB code is given in listing 11.2.

```
1 % load adjacency matrix
2 A = load();
3 maxIterations = 10;
4 d = sum(A, 2); % outdegree per vertex
5 % create the column-stochastic transition matrix
6 T = (diag(1 ./ d) * A)';
7 r = ones(numVertices, 1) / numVertices; % initialize the ranks
8 e = ones(numVertices, 1) / numVertices;
9 % PageRank calculation
10 while i < maxIterations
11   r = .85 * T * r + .15 * e
12   i = i + 1
13 end
```

Listing 11.2: MATLAB PageRank implementation.

The power iteration happens in the while loop, where the PageRank vector r is iteratively multiplied with the transition matrix T. In order to execute this code with Gilbert, the while

loop has to be replaced with a fixpoint operation. As it can be seen in listing 11.3, the replacement is only of syntactic nature.

```
1 % load adjacency matrix
2 A = load();
3 maxIterations = 10;
4 d = sum(A, 2); % outdegree per vertex
5 % create the column-stochastic transition matrix
6 T = (diag(1 ./ d) * A)';
7 r_0 = ones(numVertices, 1) / numVertices; % initialize the ranks
8 e = ones(numVertices, 1) / numVertices;
9 % PageRank calculation
10 fixpoint(r_0, @(r) .85 * T * r + .15 * e, maxIterations)
```

Listing 11.3: Gilbert PageRank implementation.

A common implementation of PageRank for Spark and Stratosphere works as follows. The PageRank vector is represented by a set of tuples (w_i, r_i) with w_i denoting the web site i and r_i being its rank. The adjacency matrix is stored row-wise as a set of tuples (w_i, A_i) with w_i denoting the web site i and A_i being its adjacency list. For the adjacency list A_i, it holds that $w_j \in A_i$ if and only if there exists a link from web site i to j.

Given the PageRank vector and the adjacency matrix representation, the next PageRank vector can be computed the following way. At first, the PageRank r_i of web site i is joined with its adjacency list A_i. For each outgoing link $w_j \in A_i$ a new tuple $(w_j, 0.85 r_i / |A_i|)$ with the rank of i being divided by the number of outgoing links is created. In order to incorporate the random jump behavior to any available web site, a tuple $(w_i, 0.15/|w|)$, with $|w|$ being the number of all web sites, is generated for each web site i. In order to compute the next PageRank vector, all newly generated tuples are grouped according to their ID. The resulting groups are reduced by adding up all partial rank values. This calculation produces the new PageRank vector. By executing these steps iteratively, the PageRank algorithm is obtained. One step of the PageRank algorithm is depicted as a data flow plan in fig. 11.7.

11.5.1.2 Experiment

For comparison, 10 steps of the PageRank algorithm for varying sizes of the adjacency matrix A are calculated. The adjacency matrix A is a sparse matrix of size $n \times n$ with a sparsity of 0.001. For each test run, the matrix is randomly generated so that their non-zero cell entries are uniformly distributed. The computation is executed on 50 cores of the DIMA cluster. The block size is set to 500×500 and Breeze is chosen as math back end for Gilbert's distributed execution engines. The execution times are depicted in fig. 11.8(a).

The graph shows that the directly implemented PageRank algorithm runs clearly faster than Gilbert's versions of PageRank. For Stratosphere, we were only able to compute Gilbert's PageRank for 50000 web sites, before the computation became too slow to execute. Even though Spark's execution engine showed a similar runtime behavior for $25000 \leq n \leq 50000$, Spark was able to scale up to $n = 100000$.

The specialized algorithms show a better scalability. For both systems, Spark and Stratosphere, we could compute the PageRank for up to 150000 web sites. Interestingly, the Strato-

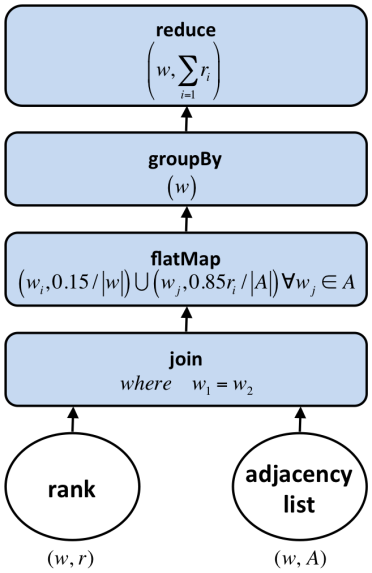


Figure 11.7: Data flow of one iteration of the PageRank algorithm for Spark and Stratosphere.

sphere system scales better for the specialized algorithm than Spark. For $n \geq 25000$, Stratosphere shows significant faster execution times. It is not yet fully clear what causes this difference. We assume that the fault tolerance mechanism of Spark is responsible for the performance loss. With increasing number of iterations and data size, Spark also has to increase the lineage information stored for a possible data recovery. This lineage information costs CPU time and consumes memory. Since it is not possible to disable the fault-tolerance mechanism, one has to keep this aspect in mind when comparing Stratosphere with Spark.

The speedup of the specialized algorithms with respect to Gilbert's implementations for this experiment is shown in fig. 11.8(b). For $n \leq 25000$, the hand-tuned algorithms running on Spark and Stratosphere show a similarly increasing speedup. The Stratosphere and Spark version achieve a speedup of approximately 14 and 10 for $n = 50000$, respectively. For $n \geq 50000$ Spark seems to exhibit a constant speedup.

The performance difference between the specialized algorithm and Gilbert's version can be explained by considering the different execution plans. As shown in fig. 11.7, each iteration of the specialized PageRank algorithm comprises one join, one group reduce and one flat map operation. The flat map operation can be executed without communication between the nodes. The join and reduce operation requires the data to be sent over the network. Thus, most of the time should be spent executing the join and reduce operation.

Looking at line 10 of listing 11.3, it can be seen that one matrix-vector product, two vector-scalar products and one vector sum have to be computed for each iteration of Gilbert's PageRank algorithm. Remembering section 8.2, we can deduce that all these operations require two cross, two join and one reduce operation. The two cross operations, where one of the operands is a scalar value, can be realized efficiently by simply broadcasting this value. However, the two join operations and the reduce operation require to shuffle the available data and thus inflict some serious network communication costs.

Comparing the specialized algorithm with Gilbert's implementation, it can be clearly seen

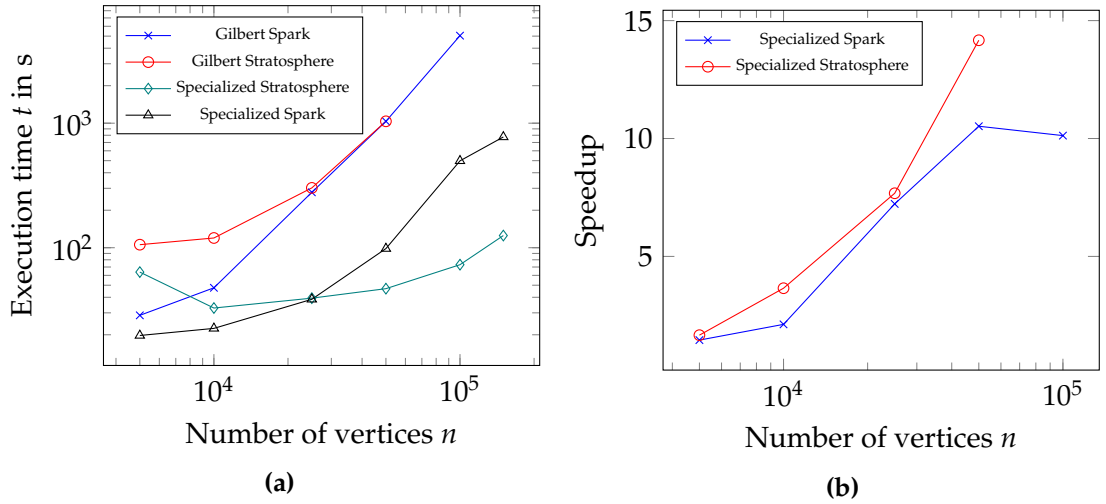


Figure 11.8: Comparison of Gilbert's PageRank implementation with specialized algorithms on Spark and Stratosphere running on 50-core cluster. (a) Execution time of 10 steps of the PageRank algorithm depending on the adjacency matrix's size. (b) Speedup of specialized algorithms with respect to Gilbert's implementations.

that the high-level linear algebra representation adds three additional operations, with one of them being highly expensive. Therefore, it is not surprising that the specialized PageRank algorithm performs better.

11.5.2 *k*-means

The *k*-means clustering algorithm [48] is very popular for cluster analysis in data mining. The goal of *k*-means is to partition the available n data points into k clusters, where each data point is assigned to the cluster with the nearest mean. Even though the optimal solution is NP-hard, heuristic algorithms exist to compute approximations.

11.5.2.1 Algorithm

The standard algorithm alternately assigns the data points to its nearest intermediate cluster and calculates new intermediate cluster centers as the mean of all assigned data points. The initial cluster centers are usually randomly selected points from the set of data points or random points. The MATLAB code for the *k*-means algorithm, shown in listing 11.4, is a little bit cumbersome if one wants to write it down in matrix notation.

The data points and centers are stored row-wise in the matrices `datapoints` and `centers`. The first step, assigning each data point to its nearest center, is done in the lines 5 and 6. The `pdist2` function calculates the pairwise euclidean distance between the rows of its two operands, in this case, the current centers and the data points. Once all pairwise distances have been calculated, the nearest cluster center is selected for each data point. This computation is done in line 6. The `assignments` column vector contains the index of the nearest cluster center for each data point. In order to calculate the new cluster centers as the mean of all assigned data points, some repmat magic has to be introduced. The idea is to construct for each center

```
1 datapoints = load();
2 centers = load();
3 mask = repmat((1:numCenters), numDatapoints, 1)';
4 for i = 1:maxIterations
5     distances = pdist2(datapoints, centers);
6     assignments = min(distances, [], 2);
7     repIdx = repmat(assignments', numCenters, 1);
8     multiplier = repIdx == mask;
9     divisor = repmat(sum(multiplier, 2), 1, dimension);
10    centers = (multiplier*datapoints)./divisor;
11 end
```

Listing 11.4: MATLAB *k*-means implementation.

a logical row vector which indicates all data points assigned to this cluster. The computation is achieved by generating a matrix which contains in each row the `assignments` vector. The matrix has as many rows as there are clusters. The generated matrix is compared with `mask` in line 8, whose cells are set to the row index of the cell. That gives the aforementioned indicator matrix. By adding the entries of each row, the number of data points assigned to a cluster center can be computed. The new cluster centers and thus the second step of the algorithm are obtained as the result of a matrix multiplication and cellwise matrix division in line 10.

Compared to the MATLAB representation, the direct implementation for Spark and Stratosphere seems rather simple. The first difference is that the data points are represented as tuples of the form (id_i, p_i) with id_i being the identifier of data point i and p_i being its coordinates. The clusters are represented likewise as tuples (id_j, c_j) with c_j being the coordinates of the cluster center. In order to calculate the pairwise distances between the data points p_i and the clusters c_j, the Cartesian product of the two datasets is constructed. This operation produces all combinations of data point cluster pairs. Having the coordinates of the data point and the cluster center, the distance can easily be computed. The output of the distance computation is a tuple of the form $(id_i, p_i, id_j, \|p_i - c_j\|_2)$, the data point ID, the data point coordinates, the cluster center ID and the distance. The cluster center with the minimal distance can be selected by grouping on the data point id id_i and selecting the cluster center ID with minimal distance. The result of the reduce operation has the form (id_j, p_i) with id_j being the nearest cluster center ID. In order to calculate the new cluster centers, the current data set has to be grouped on the cluster center ID. The generated groups contain the data points, which are assigned to the respective cluster center. The new cluster centers are computed by adding all data points p_i of each group up and dividing the result by the number of points contained in the groups. These operations constitute a single iteration step. The *k*-means algorithm is obtained by repeating the iteration step until the cluster centers converge. The dataflow plan of a single iteration step is depicted in fig. 11.9.

The execution time of Gilbert's *k*-means implementation is compared with the execution time of the hand-tuned algorithms. The Gilbert code is given in listing 11.5. As it can be seen, the only differences between the MATLAB and Gilbert version are the `fixpoint`, the `linspace` and the `minWithIndex` function. The `linspace` function generates a linearly spaced row vector, as it is known from MATLAB. The `minWithIndex` function returns the minimum value of the specified dimension and its index as column vectors in a cell array.

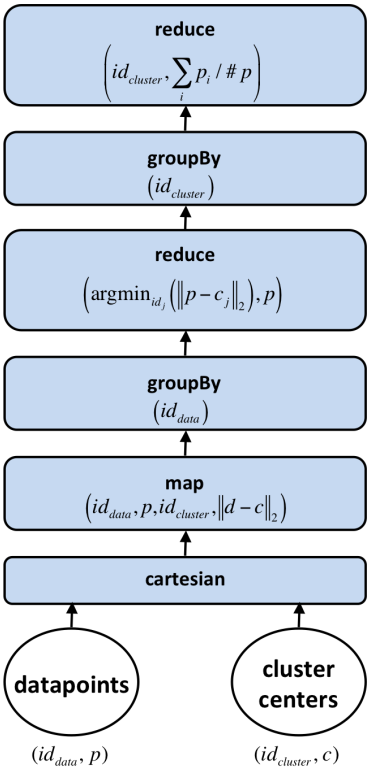


Figure 11.9: Dataflow plan of a single *k*-means step.

11.5.2.2 Experiment

For the experiments, 10 steps of the *k*-means algorithms on the 50-core DIMA cluster are calculated. The algorithms compute 100 cluster centers from n data points. The points are 2-dimensional. The number of data points is varied from 1000 to 1000000. The data points and the initial cluster centers are drawn from a uniform distribution. The block size is set to 500×500. The results of the experiment are depicted in fig. 11.10(a).

We observe that Gilbert's *k*-means implementations perform more slowly than the specialized implementation. We could only calculate 100000 points with Gilbert, because the computation took too long to finish for more points. The speedup of the specialized algorithm with respect to Stratosphere's execution engine is shown in fig. 11.10(b). With an increasing number of data points n, the speedup increases monotonically. For $n = 100000$ the specialized implementation achieves a speedup of 8 and 10 compared to the Spark executor and Stratosphere executor, respectively. Thus, we can conclude that the specialized algorithm of *k*-means executing on Stratosphere not only runs faster but also scales better than Gilbert's implementations. Interestingly, we were not able to run the specialized algorithm on Spark because the two reduce operations caused the system to become unbearably slow.

If we take a closer look at the execution plans of both algorithms, it becomes clear why Gilbert performs so poorly. For each step of the specialized algorithm, the dataflow system has to execute one Cartesian, one map and two group reduce operations. The two group reduce and the Cartesian operation are responsible for most of the work load, because they cause

```
1 datapoints = load();
2 centers = load();
3 mask = repmat(linspace(1, numCenters, numCenters), ...
4   numDatapoints, 1)';
5 function newCenters = kmeansStep(centers)
6     distances = pdist2(datapoints, centers);
7     assignments = minWithIndex(distances, 2);
8     repIdx = repmat(assignments{2}', numCenters, 1);
9     multiplier = repIdx == mask;
10    divisor = repmat(sum(multiplier, 2), 1, dimension);
11    newCenters = (multiplier*datapoints)./divisor;
12 end
13 fixpoint(centers,@kmeansStep, maxIterations)
```

Listing 11.5: Gilbert *k*-means implementation.

significant network I/O.

For Gilbert, the execution plan is more complex. Looking at the iteration step in listing 11.5, the following summation is obtained: The `pdist2` function is implemented using one join, one map and one group reduce operation. The `minWithIndex` function uses three Cartesian, one group reduce and two cogroup operations. The `repmat` function comprises three Cartesian and one cogroup operation. The comparison operator in line 9 requires one join operation. The matrix multiplication and cellwise matrix division in line 11 is implemented using two join and one reduce operation. In total, one step of Gilbert's *k*-means algorithm inflicts four join, nine Cartesian, four cogroup and three group reduce operations. Consequently, it can be concluded that the linear algebra abstraction of Gilbert comes at the price of a more complex and thus longer running execution plan.

11.6 Breeze vs. Mahout Math-Backend

As we have explained in section 10.1, Gilbert supports two math back ends for the computation of local tasks. The principal math back end uses the Breeze library for high performance linear algebra operations. As an alternative, Gilbert also supports the popular Mahout math library.

As first benchmark, the matrix multiplication from section 11.2.1 is repeated with the Mahout library. The benchmark calculates the matrix multiplication of two sparse matrices $A, B \in \mathbb{R}^{n \times n}$ with n being their dimensionality. The matrices have a sparsity of 0.001. The computation is executed on 50 cores and we use a block size of 500×500. The results are given in fig. 11.11(a).

The results of the matrix multiplication demonstrate impressively that Breeze is far superior to Mahout in multiplying two sparse matrices. Even for relatively small matrix sizes, the execution times differ strikingly. For $n = 2500$, Spark's and Stratosphere's execution engine using the Mahout library take 723 s and 745 s to finish the computation, respectively. By contrast, the same computation using the Breeze library finishes after 10 s and 71 s on Spark and Stratosphere, respectively. The performance gap even widens for an increasing dimensionality n. For $n = 5000$, Breeze reaches a speedup of 275 compared to Mahout on Spark. It can be concluded that even for small matrix sizes the multiplication using Mahout is almost infeasible.

The mystery of this bad performance can be unraveled by taking a look into the source code of Mahout. Mahout offers a sparse matrix implementation, based on hash maps. However,

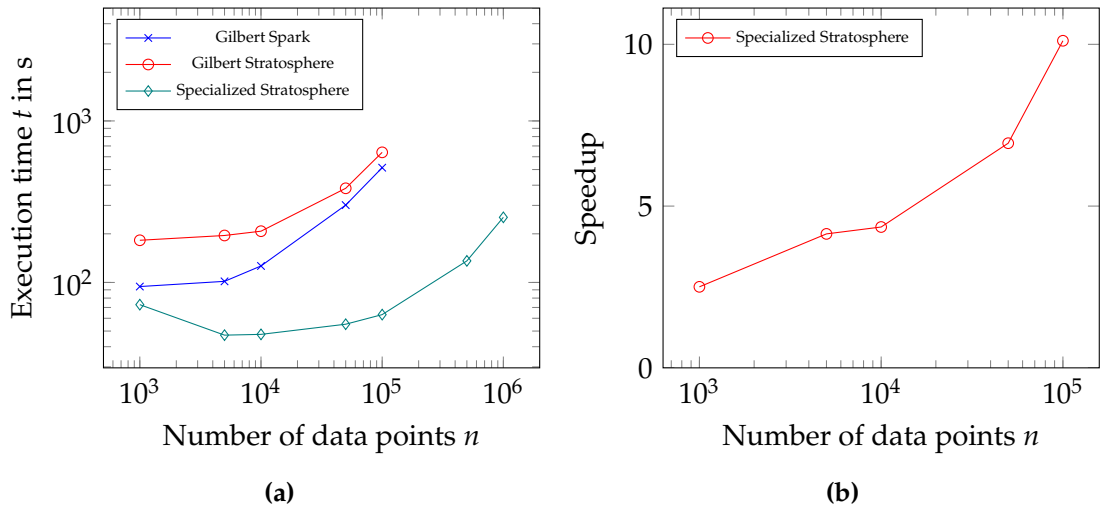


Figure 11.10: Comparison of Gilbert's k-means implementation with specialized algorithm on Stratosphere. (a) Execution time of 10 steps of the k-means algorithm on 50-cores depending on the adjacency matrix's size. (b) Speedup of specialized algorithm with respect to Gilbert's implementation on Stratosphere.

it does not provide an efficient algorithm for this data structure. Instead, Mahout uses the standard matrix multiplication algorithm. Thus, for each resulting element $c_{i,j}$, Mahout iterates over the complete row a_i of A and column b^j of B, even though most of the row and column entries will be zero. Even worse, each access triggers an hash map look up. In total, the matrix multiplication has to perform $2n^3$ hash table look ups and n^3 hash table insertions.

In contrast to Mahout, Breeze implements the sparse matrix using the compressed sparse column (CSC) storing scheme, which stores the data in a continuous array. This approach not only avoids costly hash map look ups, but Breeze also implements an optimized matrix multiplication algorithm for CSC. The algorithm is aware of the sparse nature of its operands and accesses only elements which are non-zero in both matrices.

As second benchmark, we repeated the GNMF computation from section 11.2.2. We calculated a single GNMF step for varying number of rows d of V on 50-cores of our cluster. The number of rows d reaches from 500 to 10000. As before, the sparsity of V is 0.001 and its non-zero cells are distributed uniformly. The sizes of W and H are $d \times 10$ and 10×100000, respectively. For the benchmark, we set the block size to 500×500. The runtimes of the Breeze and Mahout math back end are given in fig. 11.11(b).

It can be clearly seen that GNMF runs faster with Breeze as math back end on the Stratosphere as well as on the Spark execution engine. On Spark, Breeze reaches a speedup of roughly 4 for $d = 10000$. On Stratosphere, Breeze reaches a speedup of about 1.75 for the same number of rows.

We can conclude that Breeze has a better support for sparse matrix operations. Breeze also performs better for operations on mixed sparse and dense matrices as it is the case for GNMF. However, the gap is not as striking as it is the case for sparse matrix operations only. In conclusion, we highly recommend using Breeze as the math back end in order to experience best performance with Gilbert programs.

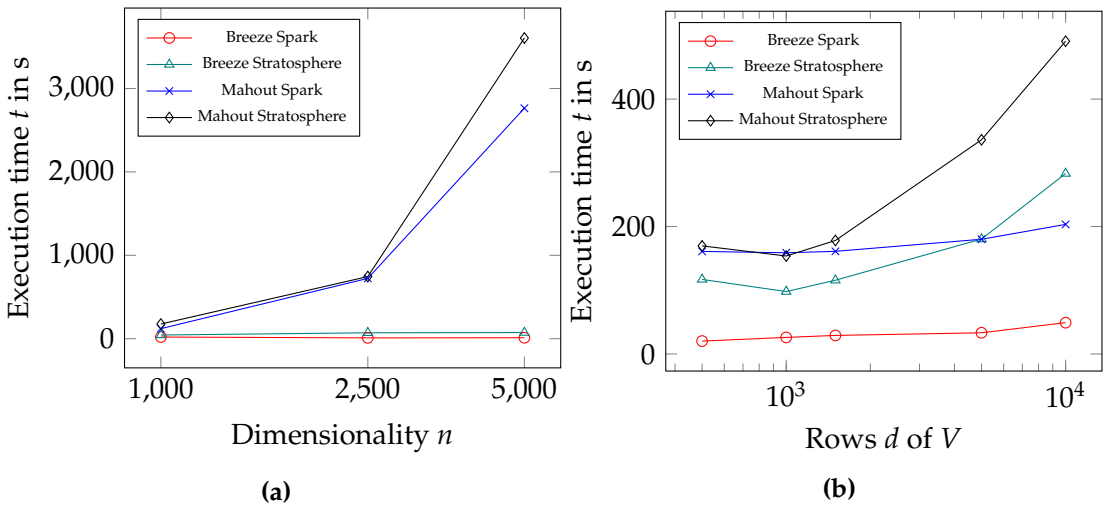


Figure 11.11: Comparison of Breeze and Mahout math back end. (a) Runtime of matrix multiplication on 50-core cluster depending on the dimensionality n using Breeze and Mahout. (b) Runtime of single GNMF step on 50-core cluster depending on the number of rows d of V using Breeze and Mahout.

12 Conclusion

"Finishing a good book is like leaving a good friend."

—William Feather, (1889 - 1981)

In the context of this thesis, we addressed the problem of managing and exploiting the ever increasing data flood. In contrast to ever growing gathering and storage capacities, the tools to harness the collected information did not scale accordingly. Our current analytic tools are mostly limited to data sizes which can be kept in the memory of a single machine. Often analysts have to reduce data to make them processable, thereby missing valuable insights. In order to analyze petabytes of data, the only viable solution is to split the work up and execute it in parallel. However, parallel data processing is a highly complex and error-prone task. Not only does it distract the analyst from its actual task but it also requires as skill set hardly anyone possesses. Thus, having analytic tools, which scale to vast amounts of data while hiding the low-level implementation details, becomes an imperative.

Gilbert tackles the aforementioned problems by fusing the assets of high-level linear algebra abstractions with the power of massively parallel dataflow systems. Gilbert is a fully functional linear algebra environment, which is programmed by the widespread MATLAB language. Gilbert programs are executed on massively parallel dataflow systems, which allow to process data exceeding the capacity of a single computer's memory. Gilbert only requires the user to program MATLAB code in order to use the system. Consequently, it is usable by a wide audience of data scientists, who can easily switch to Gilbert without having to re-adapt. These properties make Gilbert a suitable candidate for the data processing challenges of tomorrow.

Gilbert itself comprises the technology stack to parse, type and compile MATLAB code into a format which can be executed in parallel. In order to apply high-level linear algebra optimizations, we conceived an intermediate representation for linear algebra programs. Gilbert's optimizer exploits this representation to remove redundant transpose operations and to determine an optimal matrix multiplication order. The optimized program is translated into an highly optimized execution plan suitable for the execution on a supported engine. Gilbert allows the distributed execution on Spark and Stratosphere. Additionally, it exists a local execution mode. Gilbert was developed to be easily extensible with new execution engines. For the local linear algebra operations the Gilbert user can choose between the Breeze and Mahout library.

Our systematical evaluation has shown that Gilbert supports all fundamental linear algebra operations, making it fully operational. Additionally, its loop support allows to implement a wide multitude of machine learning algorithms within Gilbert. Exemplary, we have implemented the PageRank, k-means and GNMF algorithm. The code changes required to make

these algorithms run in Gilbert are minimal and only concern Gilbert's loop abstraction. Our evaluation has demonstrated the effectiveness of Gilbert's matrix multiplication reordering optimization. Furthermore, we could observe that the matrix blocking size has strong implications on the overall performance. The best performance is achieved with block sizes which offer a good compromise between data parallelism and data granularity. Our benchmarks have proved that Gilbert can handle data sizes which no longer can be efficiently processed on a single computer. Moreover, Gilbert showed a promising scale out behavior, making it a suitable candidate for large-scale data processing.

The key contributions of this thesis include the development of a scalable data analysis tool which can be programmed using the well-known MATLAB language. That way, we made the world of distributed computing accessible for data scientists and people concerned with machine learning. Furthermore, we researched how to implement linear algebra operations efficiently in modern parallel data flow systems, such as Spark and Stratosphere. In line with that research was the investigation of suitable distributed data structures for the representation of matrices and vectors. As part of the implementation of Gilbert, we also investigated how to add automatically type information to MATLAB code using a type inference mechanism. Type inference ensures a minimally invasive approach, since the user does not have to add additional statements to his code. Last but not least, we showed the effectiveness of a cost-based optimizer for linear algebra programs.

We also noted some limitations of Gilbert. The proposed Hindley-Milner type inference algorithm turned out to have problems resolving polymorphic types. As a consequence, Gilbert will incorrectly reject some well-typed programs. However, these programs constitute only corner cases. Even though, Gilbert can process vast amounts of data, it turned out that it cannot beat the performance of hand-tuned algorithms using Spark or Stratosphere. Considering the overhead the linear algebra abstraction entails, this fact is not really surprising, though. The overhead is also the reason for the higher memory foot-print, causing Spark and Stratosphere to spill faster to disk than the hand-tuned algorithms. Consequently, the processable problem sizes are smaller. Gilbert trades some performance off for better usability, which manifests itself in shorter and more expressive programming code. The fact that Gilbert only supports one hard-wired partitioning scheme, namely square blocks, omits possible optimization potential. Another limitation are the error messages of the parsing, typing and compilation phase. It is a well known problem of the HM type inference algorithm that in case of typing errors the original error source is hard to locate. The same holds true for issued error messages by Scala's Parser Combinators.

Interesting aspects for further research and improvements of Gilbert include the extension of the Gilbert optimizer by new optimization strategies. The investigation of different matrix partitioning schemes and its integration into the optimizer which selects the best overall partitioning seems to be very promising. Furthermore, a tighter coupling of Gilbert's optimizer with Stratosphere's optimizer could result in beneficial synergies. Forwarding the inferred matrix sizes to the underlying execution system might help to improve the execution plans. The numerical stability of Gilbert's computations is another important aspect, which has been completely left out in the context of this thesis. However, any linear algebra environment has to guarantee numerical stability which might be influenced by Gilbert's parallel execution. Therefore, the numerical stability of Gilbert deserves its own separate evaluation. As stated above, the HM inference algorithm has problems with polymorphic types. In our opinion,

it is worthwhile to look into other type inference algorithms, such as the type inference system of Haskell [70], to assess whether they are better suited to automatically annotate Matlab with type information. We also strongly advise to add support for further execution engines to Gilbert. An appropriate candidate could be the H2O [71] data processing engine. Additionally, it is always beneficial to extend the set of implemented algorithms and to probe them at a large data scale.

In conclusion, we have developed Gilbert, a sparse linear algebra environment executed in massively parallel dataflow systems. Gilbert transparently parallelizes sequential linear algebra code. Thus, Gilbert makes it blindingly easy to develop parallel analytic tools which are capable of processing vast amounts of data. With Gilbert at hand, we are well prepared for the looming challenges of today and tomorrow. We believe that our system helps mankind to benefit a little bit more from the world of data.

"What the caterpillar calls the end, the rest of the world calls a butterfly."

—Laozi, (6th century BC - 5th century BC)

List of Tables

List of Figures

Listings

Bibliography

Printed References

[1] Foto N. Afrati and Jeffrey D. Ullman. "Optimizing joins in a map-reduce environment". In: *Proceedings of the 13th International Conference on Extending Database Technology*. ACM. 2010, pp. 99–110.

[2] Divyakant Agrawal et al. *Challenges and Opportunities with Big Data 2011-1*. Tech. rep. 2011.

[3] Alexander Alexandrov et al. "MapReduce and PACT-Comparing Data Parallel Programming Models." In: *BTW*. 2011, pp. 25–44.

[4] Peter Alvaro et al. "Boom analytics: exploring data-centric, declarative programming for the cloud". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 223–236.

[10] Andrey Balmin et al. "A platform for eXtreme Analytics". In: *IBM Journal of Research and Development* 57.3/4 (2013), pp. 4–1.

[11] Dominic Battré et al. "Nephele/PACTs: a programming model and execution framework for web-scale analytical processing". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 119–130.

[12] Kevin S. Beyer et al. "Jaql: A scripting language for large scale semistructured data analysis". In: *Proceedings of VLDB Conference*. 2011.

[15] Spyros Blanas et al. "A comparison of join algorithms for log processing in mapreduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 975–986.

[16] Nadya Travinin Bliss and Jeremy Kepner. "'pMATLAB Parallel MATLAB Library'". In: *International Journal of High Performance Computing Applications* 21.3 (2007), pp. 336–359.

[17] Vinayak Borkar et al. "Hyracks: A flexible and extensible foundation for data-intensive computing". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 1151–1162.

[18] Yingyi Bu et al. "HaLoop: Efficient iterative data processing on large clusters". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 285–296.

[19] Yingyi Bu et al. "Scaling datalog for machine learning on big data". In: *CoRR* abs/1203.0160 (2012).

[20] Felice Cardone and J Roger Hindley. “History of lambda-calculus and combinatory logic”. In: *Handbook of the History of Logic* 5 (2006), pp. 723–817.

[21] Jeffrey Cohen et al. “MAD skills: new analysis practices for big data”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1481–1492.

[22] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[23] Luis Damas and Robin Milner. “Principal type-schemes for functional programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1982, pp. 207–212.

[24] Sudipto Das et al. “Ricardo: integrating R and Hadoop”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 987–998.

[26] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[27] Jaliya Ekanayake et al. “Twister: a runtime for iterative mapreduce”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010, pp. 810–818.

[28] Stephan Ewen et al. “Spinning fast iterative data flows”. In: *Proceedings of the VLDB Endowment* 5.11 (2012), pp. 1268–1279.

[29] Michael Furr et al. “Static type inference for Ruby”. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, pp. 1859–1866.

[30] John Gantz and David Reinsel. “The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east”. In: *IDC iView: IDC Analyze the Future* (2012).

[32] Al Geist. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system”. In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[34] Amol Ghoting et al. “SystemML: Declarative machine learning on MapReduce”. In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 231–242.

[35] William Gropp et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[36] Saptarshi Guha et al. “Large complex data: divide and recombine (D&R) with RHIPE”. In: *Stat* 1.1 (2012), pp. 53–67.

[37] Michael Heim. *Exploring Indiana Highways: Trip Trivia*. Exploring America’s Highway, 2007. ISBN: 097443583X.

[38] Laurie Hendren. “Typing aspects for MATLAB”. In: *Proceedings of the sixth annual workshop on Domain-specific aspect languages*. ACM. 2011, pp. 13–18.

[39] Martin Hilbert and Priscila López. “The worlds technological capacity to store, communicate, and compute information”. In: *Science* 332.6025 (2011), pp. 60–65.

[40] Roger Hindley. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* (1969), pp. 29–60.

[42] Dave Jewell et al. *Performance and Capacity Implications for Big Data*. IBM Redbooks, 2014.

[43] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. "Pegasus: A peta-scale graph mining system implementation and observations". In: *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE. 2009, pp. 229–238.

[44] Jeremy Kepner and Stan Ahalt. "MatlabMPI". In: *Journal of Parallel and Distributed Computing* 64.8 (2004), pp. 997–1005.

[45] Chao Liu et al. "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce". In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 681–690.

[46] Steve Lohr. "The age of big data". In: *New York Times* 11 (2012).

[47] Ewing Lusk et al. *MPI: A message-passing interface standard*. 2009.

[48] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 281-297. California, USA. 1967, p. 14.

[49] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[53] Erik Meijer and Peter Drayton. "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages". In: *OOPSLA'04 Workshop on Revival of Dynamic Languages*. 2004.

[54] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[55] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117.

[56] Martin Odersky. *Programming in Scala, 2nd edition*. Artima Press, 2010.

[57] John K Ousterhout. "Scripting: Higher level programming for the 21st century". In: *Computer* 31.3 (1998), pp. 23–30.

[58] Lawrence Page et al. *The PageRank citation ranking: bringing order to the web.* Technical Report 1999-66. Stanford InfoLab, 1999.

[61] Nagiza F. Samatova. "pR: Introduction to Parallel R for Statistical Computing". In: *CScADS Scientific Data and Analytics for Petascale Computing Workshop*. 2009, pp. 505–509.

[63] D. Seung and L. Lee. "Algorithms for non-negative matrix factorization". In: *Advances in neural information processing systems* 13 (2001), pp. 556–562.

[64] Gaurav Sharma and Jos Martin. "MATLAB®: a language for parallel computing". In: *International Journal of Parallel Programming* 37.1 (2009), pp. 3–36.

[65] Michael Stonebraker et al. "One size fits all? Part 2: Benchmarking results". In: *Proceedings of the Third International Conference on Innovative Data Systems Research (CIDR)*. Jan. 2007.

[66] Michael Stonebraker et al. "Requirements for Science Data Bases and SciDB." In: *CIDR*. Vol. 7. 2009, pp. 173–184.

[67] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356.

[69] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. "High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 105.

[74] Luke Tierney et al. "Snow: simple network of workstations". 2008.

[75] Daniel Warneke and Odej Kao. "Nephele: efficient parallel data processing in the cloud". In: *Proceedings of the 2nd workshop on many-task computing on grids and supercomputers*. ACM. 2009, p. 8.

[76] J Christopher Westland. "The cost of errors in software development: evidence from industry". In: *Journal of Systems and Software* 62.1 (2002), pp. 1–9.

[77] Reynold S Xin et al. "Shark: SQL and rich analytics at scale". In: *Proceedings of the 2013 international conference on Management of data*. ACM. 2013, pp. 13–24.

[79] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[80] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010, pp. 10–10.

[81] Yi Zhang, Herodotos Herodotou, and Jun Yang. "RIOT: I/O-efficient numerical computing without SQL". In: *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR)*. Jan. 2009.

Online References

[5] *ANother Tool for Language Recognition*. 1992. URL: `http://www.antlr.org/` (visited on 06/17/2014).

[6] *Apache Hadoop*. 2008. URL: `http://hadoop.apache.org/` (visited on 01/08/2014).

[7] *Apache Hive*. 2013. URL: `http://hive.apache.org/` (visited on 05/11/2014).

[8] *Apache Mahout*. 2011. URL: `http://mahout.apache.org/` (visited on 01/09/2014).

[9] *Apache Spark*. 2014. URL: `http://spark.apache.org/` (visited on 07/09/2014).

[13] *Big data within IBM*. 2014. URL: `http://www.ibm.com/big-data/` (visited on 01/08/2014).

[14] *Bison - GNU parser generator*. 1988. URL: `http://www.gnu.org/software/bison/` (visited on 06/17/2014).

[25] *Data, data everywhere*. 2010. URL: `http://www.economist.com/node/15557443` (visited on 06/25/2014).

[31] *Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data*. 2011. URL: `http://www.gartner.com/newsroom/id/1731916` (visited on 06/25/2014).

[41] *IBM Watson Hard At Work: New Breakthroughs Transform Quality Care for Patients*. 2013. URL: `http://www.mskcc.org/pressroom/press/ibm-watson-hard-work-new-breakthroughs-transform-quality-care-patients` (visited on 01/08/2014).

[50] *Matlab Distributed Computing Server*. URL: http://www.mathworks.com/products/distriben/ (visited on 05/12/2014).

[51] *Matlab Parallel Computing Toolbox*. URL: http://www.mathworks.de/products/parallel-computing/ (visited on 05/12/2014).

[52] *Matlab - The Language for Technical Computing*. 1984. URL: http://www.mathworks.com/products/matlab/ (visited on 06/26/2014).

[59] *Rmpi - MPI for R*. 2002. URL: http://cran.r-project.org/web/packages/Rmpi/index.html (visited on 05/12/2014).

[60] *Rpvm - PVM for R*. 2001. URL: http://cran.r-project.org/src/contrib/Archive/rpvm/ (visited on 05/12/2014).

[62] *Scala Breeze*. 2009. URL: https://github.com/scalanlp/breeze (visited on 06/17/2014).

[68] *Stratosphere - Big Data looks tiny from here*. 2014. URL: http://stratosphere.eu/ (visited on 07/09/2014).

[70] *The Haskell Programming Language*. 1990. URL: http://www.haskell.org/haskellwiki/Haskell (visited on 07/16/2014).

[71] *The Open Source In-Memory Prediction Engine for Big Data Science*. 2014. URL: http://0xdata.com/h2o-2/ (visited on 07/16/2014).

[72] *The R Project for Statistical Computing*. 1993. URL: http://www.r-project.org/ (visited on 05/05/2014).

[73] *The Scala programming language*. 2003. URL: http://www.scala-lang.org/ (visited on 06/17/2014).

[78] *Yacc: Yet Another Compiler-Compiler*. 1970. URL: http://dinosaur.compilertools.net/#yacc (visited on 06/17/2014).

Appendices

Appendix 1: Gilbert's Mathematical Operations

Gilbert is a fully functional linear algebra environment. As such a system, it provides a rich set of primitive operations. People familiar with MATLAB should quickly recognize the operations, because Gilbert's syntax is identical to the syntax of MATLAB. Additionally, Gilbert implemented several common MATLAB functions.

Gilbert's operations are used to modify scalar and matrix values. The full set of supported arithmetic operations is given in table 1. The full set of supported logical operations is given in table 2. The full set of supported comparison operations is given in table 3.

Certain operations can not be expressed with primitive operations and thus require special functions. Gilbert comes with a set of pre-implemented functions, which are popular in MATLAB. The full set of supported functions is given in table 4.

Operation	Syntax	Signature
Addition	`a + b`	$double \times double \rightarrow double$ $matrix[double] \times double \rightarrow matrix[double]$ $double \times matrix[double] \rightarrow matrix[double]$ $matrix[double] \times matrix[double] \rightarrow matrix[double]$
Subtraction	`a - b`	$double \times double \rightarrow double$ $matrix[double] \times double \rightarrow matrix[double]$ $double \times matrix[double] \rightarrow matrix[double]$ $matrix[double] \times matrix[double] \rightarrow matrix[double]$
Multiplication	`a * b`	$double \times double \rightarrow double$ $matrix[double] \times double \rightarrow matrix[double]$ $double \times matrix[double] \rightarrow matrix[double]$
Division	`a / b`	$double \times double \rightarrow double$ $matrix[double] \times double \rightarrow matrix[double]$
Exponentiation	`a ^b`	$double \times double \rightarrow double$ $matrix[double] \times double \rightarrow matrix[double]$
Matrix multiplication	`a * b`	$matrix[double] \times matrix[double] \rightarrow matrix[double]$
Cellwise matrix multiplication	`a .* b`	$matrix[double] \times matrix[double] \rightarrow matrix[double]$
Cellwise matrix division	`a ./ b`	$double \times matrix[double] \rightarrow matrix[double]$ $matrix[double] \times matrix[double] \rightarrow matrix[double]$
Cellwise exponentiation	`a .^b`	$matrix[double] \times matrix[double] \rightarrow matrix[double]$

Table 1: Set of Gilbert's arithmetic operations.

Operation	Syntax	Signature
Logical and	`a & b`	$boolean \times boolean \rightarrow boolean$ $matrix[boolean] \times matrix[boolean] \rightarrow matrix[boolean]$
Logical or	`a | b`	$boolean \times boolean \rightarrow boolean$ $matrix[boolean] \times matrix[boolean] \rightarrow matrix[boolean]$
Short-circuit logical and	`a && b`	$boolean \times boolean \rightarrow boolean$ $matrix[boolean] \times matrix[boolean] \rightarrow matrix[boolean]$
Short-circuit logical or	`a || b`	$boolean \times boolean \rightarrow boolean$ $matrix[boolean] \times matrix[boolean] \rightarrow matrix[boolean]$

Table 2: Set of Gilbert's logical operations.

Operation	Syntax	Signature
Greater	`a > b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$
Greater equals	`a >= b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$
Less	`a < b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$
Less equals	`a <= b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$
Equals	`a == b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$
Not equals	`a ~= b`	$double \times double \rightarrow boolean$ $matrix[double] \times double \rightarrow matrix[boolean]$ $double \times matrix[double] \rightarrow matrix[boolean]$ $matrix[double] \times matrix[double] \rightarrow matrix[boolean]$

Table 3: Set of Gilbert's comparison operations.

Function	Explanation	Signature
`ones(r,c)`	Creates a matrix of size (r, c) initialized with 1.0.	$double \times double \rightarrow matrix[double]$
`eye(r,c)`	Creates an identity matrix of size (r, c).	$double \times double \rightarrow matrix[double]$
`zeros(r,c)`	Creates a zero matrix of size (r, c).	$double \times double \rightarrow matrix[double]$
`rand(r,c,m,s)`	Creates matrix of size (r, c) whose elements are drawn from a Gaussian distribution with mean m and standard deviation s.	$double \times double \times double \times double \rightarrow matrix[double]$
`rand(r,c,m,s,l)`	Creates matrix of size (r, c) with a portion of l non-zero elements, distributed uniformly. The non-zero cells are drawn from Gaussian distribution with mean m and standard deviation s.	$double \times double \times double \times double \times double \rightarrow matrix[double]$
`pdist2(a,b)`	Calculates the pairwise euclidean distance between the rows of matrix a and b.	$matrix[double] \times matrix[double] \rightarrow matrix[double]$
`repmat(a, rm, cm)`	Repeats the matrix a rm times row-wise and cm times column-wise.	$matrix[double] \times double \times double \rightarrow matrix[double]$
`minWithIndex(a, d)`	Finds the minimum value and its index in dimension d of matrix a.	$matrix[double] \times double \rightarrow \{matrix[double], matrix[double]\}$
`linspace(s,e,n)`	Creates a matrix of size $(1, n)$ with n evenly spaced points between s and e.	$double \times double \times double \times matrix[double]$
`sum(a,d)`	Sums the dimension d of matrix a.	$matrix[double] \times double \rightarrow matrix[double]$
`norm(a,n)`	Calculates the n-Frobenius norm of matrix a.	$matrix[double] \times double \rightarrow double$
`spones(a)`	Creates a sparse matrix from matrix a where each cell is 1.0 if the corresponding cell in a is non-zero, else it is 0.0.	$matrix[double] \rightarrow matrix[double]$
`diag(a)`	If a is a matrix, then it extracts the diagonal. If a is a row or column vector, then it creates a matrix with a on its diagonal.	$matrix[double] \rightarrow matrix[double]$

Table 4: Set of Gilbert's functions.