Gilbert: Declarative Sparse Linear Algebra on Massively Parallel Dataflow Systems

Till Rohrmann[1] Sebastian Schelter[2] Tilmann Rabl[3] Volker Markl[4]

Abstract: In recent years, the generated and collected data is increasing at an almost exponential rate. At the same time, the data's value has been identified in terms of insights that can be provided. However, retrieving the value requires powerful analysis tools, since valuable insights are buried deep in large amounts of noise. Unfortunately, analytic capacities did not scale well with the growing data. Many existing tools run only on a single computer and are limited in terms of data size by its memory. A very promising solution to deal with large-scale data is scaling systems and exploiting parallelism.

In this paper, we propose Gilbert, a distributed sparse linear algebra system, to decrease 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 form enables high-level algebraic optimizations. Different optimization strategies are evaluated and the best one is chosen by a cost-based optimizer. The optimized result is then transformed into a suitable format for parallel execution. Gilbert generates execution plans for Apache Spark® and Apache Flink®, two massively parallel dataflow systems. Distributed matrices are represented by square blocks to guarantee a well-balanced trade-off between data parallelism and data granularity.

An 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. Two well known machine learning (ML) algorithms, namely PageRank and Gaussian non-negative matrix factorization (GNMF), are implemented with Gilbert. The performance of these algorithms is compared to optimized implementations based on Spark and Flink. Even though Gilbert is not as fast as the optimized algorithms, it simplifies the development process significantly due to its high-level programming abstraction.

Keywords: Dataflow Optimization, Linear Algebra, Distributed Dataflow Systems

1 Introduction

Key component of modern big data solutions are sophisticated analysis algorithms. Unfortunately, the search for useful patterns and peculiarities in large data sets resembles the search for a needle in a haystack. This challenge requires tools that are able to analyze vast amounts of data quickly. Many 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

[1] Apache Software Foundation, trohrmann@apache.org
[2] Technische Universität Berlin, sebastian.schelter@tu-berlin.de
[3] Technische Universität Berlin / DFKI, rabl@tu-berlin.de
[4] Technische Universität Berlin / DFKI, volker.markl@tu-berlin.de

data points might get lost if chunks are 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 significant patterns manifest. However, these benefits come at the price of an increased computing time, which requires fast computer systems to make computations feasible.

Analysis tools traditionally do not scale well for vast data sets. Because of the exponential data growth, analytic capacities have to improve at a similar speed. This problem can be tackled by vertical and horizontal scaling of computers. To scale vertically means that we add more resources to a single computer. For example, the main memory size or the CPU frequency of a computer can be increased to improve computational power. In contrast to that, horizontal scaling refers to adding more computer nodes to a system. By having more than one node, computational work can be split up and distributed across multiple computers.

The emerging field of multi-core and distributed systems poses new challenges for programmers. 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 tedious low-level tasks related to parallelization such as load-balancing, scheduling, and fault tolerance. With these new models, programmers can concentrate on the actual algorithm and use case rather than reasoning about distributed systems. These are the reasons why Google's MapReduce [DG08] framework and its open source re-implementation Hadoop [08] became so popular among scientists as well as engineers.

MapReduce and other frameworks, however, force users to express programs in a certain way which is often not natural or intuitive to users from different domains. Especially, in the field of data analytics and machine learning programs are usually expressed in a mathematical form. Therefore, systems such as MATLAB [Ma84] and R [93] 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 processable data is limited to the size of a single machine's memory, which constitutes a serious drawback for real-world applications.

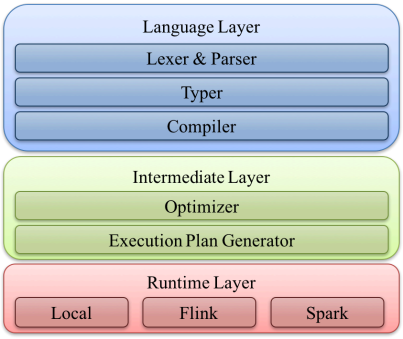


Fig. 1: Gilbert's system architecture consisting of the language, intermediate and runtime layer.

As a solution we propose Gilbert, a distributed sparse linear algebra environment whose name is a homage to William Gilbert Strang. Gilbert provides a MATLAB-like language for distributed sparse linear algebra operations. It has a layered architecture, which is shown in Figure 1. The first layer is the language layer. This layer parses Gilbert code and compiles it into an intermediate representation (IR). The second layer is the intermediate layer, which is given the IR. The intermediate format is the ideal representation to apply language independent high-level transformations. The Gilbert optimizer applies several algebraic optimizations prior to the generation of the execution plan. The execution plan generator translates the optimized IR into an execution engine specific plan. This execution plan is then executed on the respective back end.

Our main contributions are:

- Gilbert allows to write MATLAB-like code for sparse linear algebra and execute it on massively parallel dataflow systems.
- Its expressiveness allows to quickly develop scalable algorithms to analyze web-scale data.
- We introduce a novel fixed-point operator which replaces loops by recursion and can be efficiently parallelized.
- Furthermore, we demonstrate that dataflow optimizers can be used to automatically select a good matrix multiplication strategy.

The rest of this paper is structured as follows. In Sect. 2, Gilbert's language and its features are described. Sect. 3 presents how linear algebra operations are mapped to a dataflow system. Gilbert's performance and scalability is evaluated in Sect. 4. Related work is covered in Sect. 5 before the work is concluded in Sect. 6.

2 Gilbert Features

Gilbert's language has been strongly inspired by MATLAB, which should ease its adoption. The language is a fully functional subset of MATLAB and its grammar is specified in [Ro14]. The elementary data type is the 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 single column/row matrix. Additionally, scalar *double*, *boolean*, and *string* values are supported. Gilbert also implements cell array types. 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. List. 1 shows how to define and access them.

Gilbert supports the basic linear algebra operations defined on matrices and scalars. They include among others +, -, /, and *, whereas * denotes the matrix-matrix multiplication and all other operations are interpreted cell-wise. The cell-wise multiplication is indicated by a point preceding the operator. Gilbert also supports comparisons operators such as >,

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

Listing 1: Cell array usage in Gilbert. Definition of a 4 element cell array, which is accessed subsequently.

>=, ==, and ~=. Besides the basic arithmetic operations, the user can also define named functions and anonymous functions. The syntax of anonymous functions adheres to the MATLAB syntax: `@(x,y) x*x + y*y`.

An important aspect of MATLAB are loops. MATLAB permits the user to express `for` and `while` loops, which can have side effects. Parallelizing iterations with side effects is difficult because the referenced external state has to be maintained. This circumstance makes preprocessing and execution unnecessarily complex. 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.

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

In order to terminate the fixpoint operation, the operator provides two mechanisms. The user has to specify a maximum number `m` of iterations. Additionally, 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. Thus, the fixpoint operator terminates either if convergence was detected or if the maximum number of iterations is exceeded.

$$fixpoint : \underbrace{T}_{\mathtt{x}} \times \left(\underbrace{T \to T}_{\mathtt{f}}\right) \times \underbrace{\mathbb{N}}_{\mathtt{m}} \times \left(\underbrace{T \times T \to \mathbb{B}}_{\mathtt{c}}\right) \to T \tag{1}$$

with T being a generic type variable.

In fact, the fixpoint operator replaces iterations by recursions with a pure update function f. At this point Gilbert breaks with existing MATLAB code. However, all MATLAB loops can be expressed via the fixpoint operator by passing the loop's closure to the update function, see List. 2.

```
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 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.

2.1 Gilbert Typing System

MATLAB belongs to the class of dynamically typed languages. However, the parallel data processing systems used to run Gilbert programs need to know the occurring data types. Therefore, the MATLAB language has to be enriched with type information. We infer this type information using the Hindley-Milner (HM) type system [Hi69; Mi78] and a slightly derived form of algorithm W [DM82] for type inference. In case that the type inference algorithm cannot properly infer the types, there has to be a way to resolve this problem. Similar to [Fu09], we allow to add type information via special comments.

Function Overloading. MATLAB's basic operators, such as +, -, / and *, 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, because it assumes that each function has a 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

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

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.

Matrix Size 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 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 $(B \times C)$.

By knowing the matrix sizes, Gilbert 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 v being the value type. The value type can either be a value variable or a concrete value.

The matrix size inference is incorporated into the HM type inference by adapting 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. Otherwise an error is thrown.

2.2 Intermediate Representation

After parsing and typing of the source code is done, it is translated into an intermediate representation. The additional abstraction layer allows Gilbert to apply language independent optimization. Moreover, the higher-level abstraction of mathematical operations holds more potential for algebraic optimization. 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 required as inputs. The set of intermediate operators can be distinguished into three categories: *Creation operators*, *transformation operators* and *output operators*.

Creation Operators. Creation operators generate or load some data. The `load` operator loads a matrix from disk with the given number of rows and columns. The `eye` operator generates an identity matrix of the requested size. `zeros` generates a matrix of the given size which is initialized with zeros. `Randn` takes the size of the resulting matrix and the mean and the standard deviation of a Gauss distribution. The Gauss distribution is then used to initialize the matrix randomly.

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 `UnaryScalar-Transformation` takes a single scalar value and applies an unary operation on it. The `ScalarScalarTransformation` constitutes a binary operation on scalar values whereas `ScalarMatrixTransformation` and `MatrixScalarTransformation` represent a binary operation between a scalar and a matrix value. The `VectorwiseMatrixTransformation` applies an operation on each row vector of the given matrix. A vectorwise operation produces a scalar value for each row vector. The `AggregateMatrixTransformation` applies an operation to all matrix entries producing a single scalar result value. The iteration mechanism is represented by the `FixpointIterationMatrix` and `FixpointIterationCellArray` operators.

Writing Operators. The writing operators are used to write the computed results back to disk. There exists a writing operation for each supported data type: `WriteMatrix`, `WriteScalar`, `WriteString`, `WriteCellArray` and `WriteFunction`.

2.3 Gilbert Optimizer

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. It works on the intermediate format of a program, which provides an appropriate high-level abstraction.

Matrix Multiplication Reordering. Matrix multiplications belong to the most expensive linear algebra operations in terms of computational as well as space complexity. Intermediate results of successive matrix multiplications can easily exceed the memory capacity and thus rendering its computation infeasible. However, a smart execution order can sometimes avoid the materialization of excessively large matrices. 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 calculates the maximum size of all occurring intermediate results for each evaluation order. In order to do this calculation, the optimizer relies on the operands' automatically inferred matrix sizes. At last, it picks the execution order with the minimal maximum intermediate matrix size.

Transpose Pushdown. Transpose pushdown tries 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. Consider, for example, $C = A^T B$ and $E = (CD^T)^T$. By inserting C into E, the expression $E = (A^T BD^T)^T$ is obtained which is equivalent to $DB^T A$. The latter formulation contains only one transpose operation. Usually multiple transpose operations occur because they are written for convenience reasons at various places 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 can be a beneficial optimization.

3 Gilbert Runtime

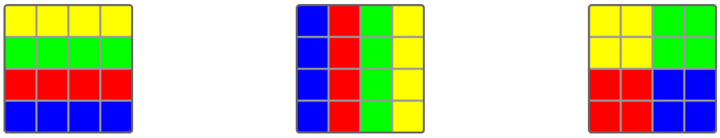
The Gilbert runtime is responsible for executing the compiled MATLAB code on a particular platform. For this purpose, it receives the intermediate representation of the code and translates it into the execution engine's specific format. Currently, Gilbert supports three different execution engines: *LocalExecutor*,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

The *LocalExecutor* 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 *LocalExecutor*, the *FlinkExecutor* and *SparkExecutor* execute the program 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. Since both systems offer a similar set of programming primitives, they can operate on the same data structures. Furthermore, most of the linear algebra operations are implemented in a similar fashion. The details of the distributed data structures and operations are explained in Sect. 3.1 and Sect. 3.2.

3.1 Distributed Matrix Representation

An important aspect of distributed algorithms is their data partitioning. 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, the matrix entries have to be partitioned. A first idea could be to partition a matrix according to their rows or columns, as it is depicted in Fig. 2a and Fig. 2b. This scheme allows to represent a matrix as a distributed set of vectors. 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.



a Row partitioning　b Column partitioning　c Square block partitioning

Fig. 2: Row-wise, column-wise and square block-wise matrix partitioning.

This approach, however, unveils its drawbacks when multiplying two equally partitioned matrices A and B. 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. 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. 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. 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 be 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 is to split the matrix into equally sized quadratic blocks as shown in Fig. 2c. In order to identify the individual blocks, each of them will get a block row and block column index assigned. Thus, blocking adds some memory overhead in the form of index information. 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:

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

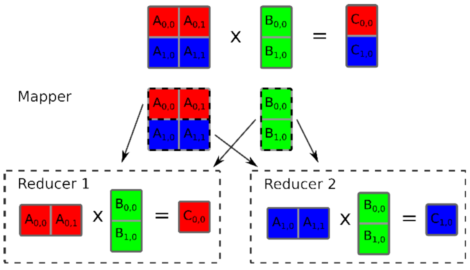
We see that the term $(n-1)$ is replaced by $2\left(\sqrt{n}-1\right)$ in the square block cost model. For $n > 2$, the square block partitioning scheme is thus superior to the row- and column-wise partitioning pattern with respect to the cost model. The reason for this outcome is that the square blocks 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 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 compressed representation is best suited. Gilbert chooses the representation for each block dynamically. Depending on the non-zero elements to total elements ratio, a sparse or dense representation is selected.

3.2 Linear Algebra Operations

In the following, we will outline the implementation of the matrix multiplication operator, which is most critical for performance in linear algebra programs. For a reference of the implementation of the remaining operators, we refer the interested reader to [Ro14]. Note that we use the Breeze [09] library for conducting local linear algebraic operations. 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).

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. 3a.


A0,0 A0,1 A1,0 A1,1 x B0,0 B1,0 = C0,0 C1,0
Mapper
Reducer 1
A0,0 A0,1 x B0,0 B1,0 = C0,0
Reducer 2
A1,0 A1,1 x B0,0 B1,0 = C1,0
a Replication based matrix multiplication.

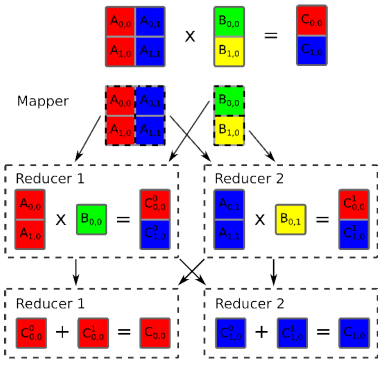

A0,0 A0,1 A1,0 A1,1 x B0,0 B1,0 = C0,0 C1,0
Mapper
Reducer 1
A0,0 A1,0 x B0,0 = C⁰0,0 C⁰1,0
Reducer 2
A0,1 A1,1 x B0,1 = C¹0,0 C¹1,0
Reducer 1
C⁰0,0 + C¹0,0 = C0,0
Reducer 2
C⁰1,0 + C¹1,0 = C1,0
b Cross product matrix multiplication.

Fig. 3: Execution strategies for matrix multiplication in 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. 3b.

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, Flink and Spark exhibit a little bit more flexibility in terms of higher order functions. 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.

Flink 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 broadcast hash join, a repartitioning hash join or a repartitioning 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 broadcast 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 any of the repartitioning join algorithms 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 Flink 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.

4 Evaluation

In this chapter, we will investigate the scalability of Gilbert and its performance compared to well known hand-tuned ML algorithms. We show that Gilbert is not only easily usable in terms of programming but also produces results with decent performance.

Experimental Setup. For our evaluation, we use a Google compute engine cluster with 8 machines of type `n1-standard-16`. Each machine is equipped with 60 gigabytes of main memory and has 16 virtual CPUs. The Spark execution engine runs on Apache Spark-2.0.0 [14]. For the Flink execution engine, we use Apache Flink-1.1.2 [16]. As underlying distributed file system we use Apache Hadoop-2.7.3 [08].

4.1 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.

Matrix Multiplication. As a first benchmark, we choose the matrix multiplication $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ and n being the dimensionality. The matrix multiplication operation is demanding, both in CPU load as well as network I/O as it requires two network shuffles. The matrices A and B are sparse matrices with uniformly distributed non-zero cells. They are randomly generated prior to the matrix multiplication with a sparsity of 0.001. As baseline, we run the same matrix multiplication on a single machine of the cluster using Gilbert's local execution engine. The Flink and Spark execution engines are both started with 48 gigabytes of main memory for their task managers and they distribute the work equally among all worker nodes. In the first experiment, we fixed the block sizes to 500×500 and set the number of cores to 64. We then increased the dimensionality n of A and B to observe the runtime behavior. The resulting execution times for the local, Flink and Spark execution engines are shown in Fig. 4a. In the second experiment, we investigate the scaling behavior of Gilbert with respect to the cluster size. In order to observe the 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 128 and scale n such that $n^3/\#cores$ is constant. We started with $n = 5000$ for a single core and reached $n = 25000$ on 128 cores. The results of the experiment are shown in Fig. 4b.

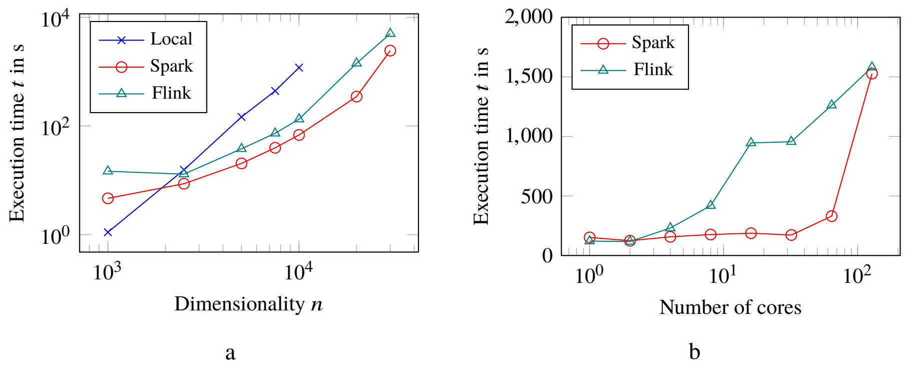


Fig. 4: Scalability of matrix multiplication. a Execution time of matrix multiplication depending on the data size. b Execution time of matrix multiplication depending on the cluster size with constant work load per core.

On a single machine, we are able to execute the matrix multiplication for dimensionalities up to $n = 10000$ in a reasonable time. We can see that the local execution performs better for dimensionalities $n \leq 2500$. That is expected since the matrix still fits completely in the main memory of a single machine and the distributed execution adds communication overhead and job start up latency. For matrices with $n > 2500$, Spark and Flink start to calculate the matrix multiplication faster than the local executor. We also see that Gilbert can handle matrix sizes which scale far beyond what a single machine can handle.

The results depicted in Fig. 4b indicate for both execution engines decent scale-out behavior. For Spark we observe an almost horizontal line for number of cores ≤ 32. For larger number of cores, the scaling degrades which is probably caused by spilling. The same applies to Flink. However, we can observe that the system has to start spilling data earlier. Thus, Flink's scale-out behavior is not as good as Spark's with respect to matrix multiplication.

Gaussian Non-negative Matrix Factorization. As second benchmark, we choose the Gaussian non-negative matrix factorization (GNMF) algorithm [SL01]. GNMF finds for a given matrix $V \in \mathbb{R}^{d \times w}$ a factorization $W \in \mathbb{R}^{d \times t}$ and $H \in \mathbb{R}^{t \times w}$ such that $V \approx WH$ holds true. For our evaluation, we calculate one step of the GNMF algorithm. We set $t = 10$, $w = 100000$ and vary the number of rows d of V. 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. As baseline, we run the GNMF on a single machine of the cluster using the local execution engine. Like for the matrix multiplication benchmark, the task manager of Spark and Flink are started with 48 gigabytes of memory. In the first experiment we fix the number of cores to 64. 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 compare the results with the optimized GNMF MapReduce implementation proposed in [Li10], we re-implemented the algorithm using Spark and Flink. This hand-tuned implementation, henceforth denoted as HT-GNMF (HT in the graphs), is also executed on 64 cores. The execution times of HT-GNMF and Gilbert's GNMF are shown in Fig. 5a. 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 64 cores and scale the number of documents d accordingly. Initially we start with 1000 documents and, consequently, calculate the matrix factorization for 64000 documents on 64 cores. The results of this experiment are shown in Fig. 5c.

The local executor is applied to sizes of d ranging from 500 to 3000 rows. Surprisingly, the local execution is outperformed by the distributed engines. This fact indicates that the network communication costs are not decisive. The distributed systems can also be used for data sizes which can no longer be handled by a single machine. Both distributed execution engines scale well and achieve almost identical results as the hand tuned implementations. The speedup of the HT-GNMF variants compared to GNMF on Spark's and Flink's executor is shown in Fig. 5b. In the case of Flink, the HT-GNMF variant runs at most 1.78 times faster than Gilbert's GNMF implementation for large data sets. For Spark, we can only observe a maximum speedup of 1.35. This result underlines Gilbert's good performance.

Furthermore, the development with 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 Flink's HT-GNMF. Not only did we have to know how to program Spark and Flink, 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

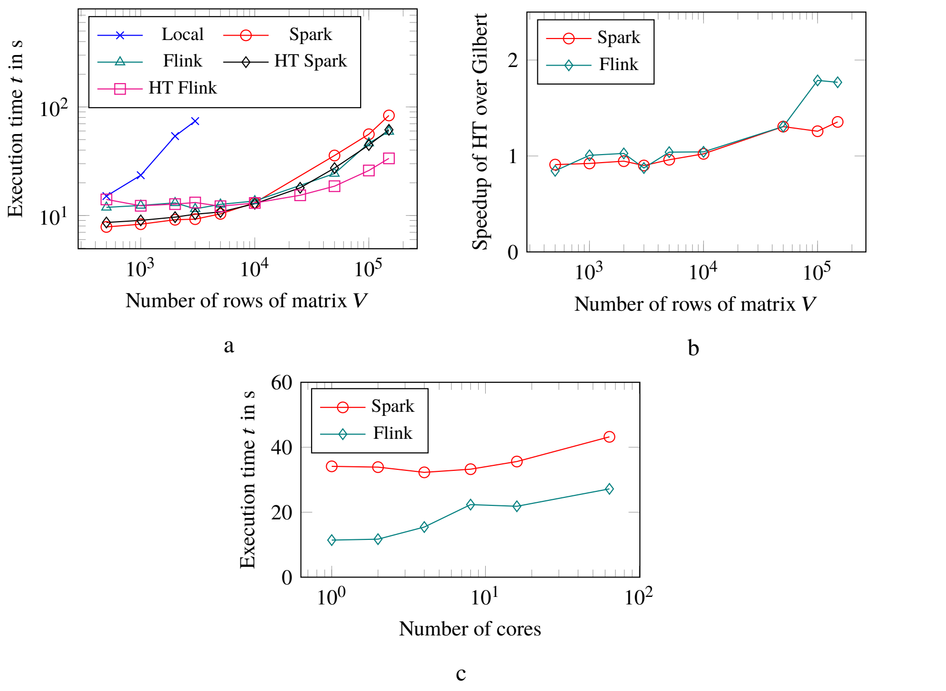


Fig. 5: Scalability of GNMF. a Execution time of one GNMF step depending on the data size. b Speedup of HT-GNMF over Gilbert's GNMF using Spark and Flink. c Execution time of one GNMF step depending on the cluster size with constant work load per core.

with systems like Spark and Flink 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 slight performance loss. [Al10] made similar observations while developing a declarative programming language for distributed systems programming. The scale-out behavior of the Flink and Spark execution engines Fig. 5c both show good results for #*cores* up to 64. The runtime on Spark with 64 cores is only 1.26 times slower than the runtime on a single core with the same work load per core. For Flink we observe a slowdown of 2.38 when running on 64 cores.

4.2 PageRank

In this section, we want to investigate how well the PageRank algorithm [Pa98] implemented in Gilbert performs compared to a specialized implementation (denoted as SP in the figures). We expect that the Gilbert runtime adds some overhead. Furthermore, the high-level linear algebra abstraction of Gilbert might make it difficult to exploit certain properties to speed up the processing. We first execute PageRank directly in Gilbert, given the Gilbert code in List. 3, and then run them directly on Flink and Spark. 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.

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

Listing 3: Gilbert PageRank implementation.

The specialized Spark and Flink implementation of PageRank 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. The next PageRank vector is calculated as follows. 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 $(w_j, 0.85r_i/|A_i|)$ 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 and summed up. These steps constitute a single PageRank iteration whose data flow plan is depicted in Fig. 6.

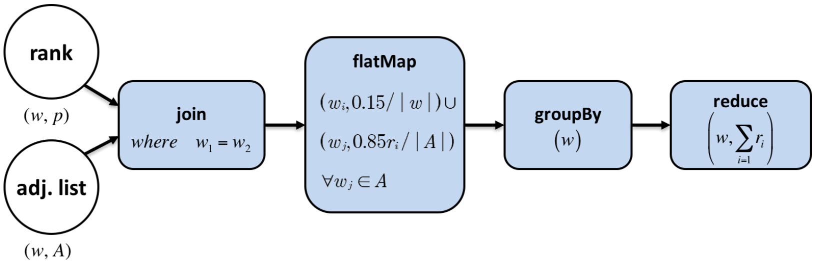
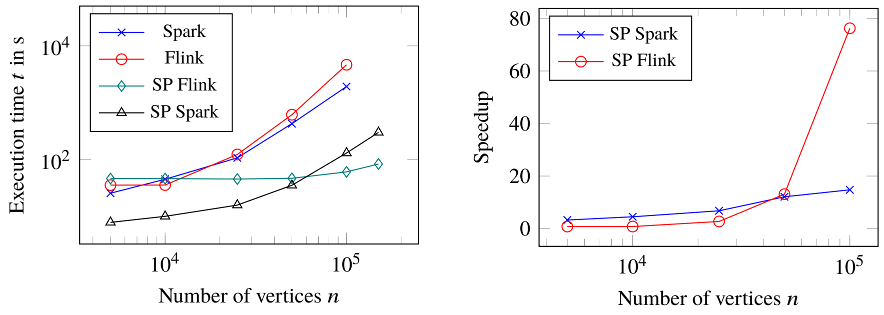


Fig. 6: Data flow of one iteration of the PageRank algorithm for Spark and Flink.

Experiment. For comparison, 10 steps of the PageRank algorithm for varying sizes of the adjacency matrix A are calculated. The randomly generated adjacency matrix A is a sparse matrix of size $n \times n$ with a sparsity of 0.001. The computation is executed on 64 cores with a block size of 500×500. The execution times are depicted in Fig. 7a.

The graphs in Fig. 7a show that the specialized PageRank algorithm runs faster than Gilbert's versions of PageRank. Furthermore, the specialized algorithms show a better scalability.



a b

Fig. 7: Comparison of Gilbert's PageRank implementation with specialized algorithms on Spark and Flink. 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.

The speedup of the specialized algorithms with respect to Gilbert's implementations for this experiment is shown in Fig. 7b. For $n \leq 50000$, the hand-tuned algorithms running on Spark and Flink show a similar speedup. The Flink and Spark version achieve a speedup of approximately 13 and 12 for $n = 50000$, respectively. However, for $n = 100000$ we can observe a speedup of 76 for the Flink specialized implementation whereas Spark reaches a speedup of 14. Gilbert's performance degradation with Flink is likely caused by earlier data spilling.

The performance difference between the specialized algorithm and Gilbert's version can be explained by considering the different execution plans. As shown in Fig. 6, each iteration of the specialized PageRank algorithm comprises one join, one group reduce and one flat map operation. In contrast, each iteration of Gilbert's PageRank implementation requires two cross, two join and one reduce operation. Thus, it can be clearly seen that Gilbert's 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.

5 Related Work

SystemML [Bo14a; Bo14b; El16; Gh11; Sc15] aims to make machine learning algorithms run on massive datasets without burdening the user with low-level implementation details and tedious hand-tuning. Therefore, it provides an R-like declarative high-level language, called Declarative Machine learning Language (*DML*), and compiles and optimizes the resulting programs to distributed dataflow systems. The linear algebra operations are translated into a directed acyclic graph of high-level operators. 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. Afterwards these plans are translated to low-level operators and executed either in the driver memory or in parallel. SystemML is one of the main inspirations for Gilbert. While it originally only supported MapReduce as runtime, it has recently also moved to supporting more advanced dataflow systems. Gilbert differs from SystemML by its fixpoint operator and by leveraging the general optimizers of the underlying system (e.g., the Flink optimizer for matrix multiplication optimization, see Section 3.2).

Samsara [LP16; Sc16] is a domain-specific language for declarative machine learning in cluster environments. Samsara allows its users to specify programs using a set of common matrix abstractions and linear algebraic operations, similar to R or MATLAB. Samsara automatically compiles, optimizes and executes these programs on distributed dataflow systems. With Samsara mathematicians and data scientists can leverage the scalability of distributed dataflow systems via common declarative abstractions, without the need for deep knowledge about the programming and execution models of such systems. Samsara is part of the Apache Mahout library [11] and supports a variety of backends, such as Apache Spark or Apache Flink.

Ricardo [Da10] uses existing technologies to implement a scalable system for statistical analysis: it executes the data shipment via Hadoop and runs the actual analytic computation by R. It integrates these systems via Jaql [Be11], a declarative high-level language for data-processing on Hadoop. Thereby, 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. However, in this setting the user still has to explicitly specify the data-flow and the data distribution, which requires a substantial understanding of MapReduce's principles. *RHIPE* [Gu12] also tries to extend R to scale to large data sets using Hadoop. RHIPE follows an approach called divide and recombine. The idea is to split the examined data up into several chunks so that they fit in the memory of a single computer. Next a collection of analytic methods is applied to each chunk without communicating with any other computer. After all chunks are evaluated, the results will be recombined in an aggregation step to create the overall analytic result. However, this strict execution pipeline constrains the analysis process considerably. A system which integrates more seamlessly into the R ecosystem is *pR* (parallel R) [Sa09]. The goal is to let statisticians compute large-scale data without having to learn a new system. pR achieves its goal by providing a set of specialized libraries which offer parallelized versions of different algorithms using MPI. However, MPI does not integrate well with an environment where clusters are shared and usually execute several jobs concurrently. Furthermore, it lacks important properties such as fault-tolerance, elasticity and reliability. Another related system for R is *rmpi* [02]. As this is only a wrapper for the respective MPI calls, it also suffers from the aforementioned problems. Furthermore, there is the SparkR [Ve16] project which aims to integrate Spark's API and SQL processing capabilities into R.

For the other popular numerical computing environment out there, namely MATLAB, also a set of parallelization tools exists. The most popular are the MATLAB Parallel Computing Toolbox [Mab] and MATLAB Distributed Computing Server [Maa], which are developed by MathWorks. 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. Besides these tools there are also other projects which try to parallelize MATLAB code. The most noteworthy candidates are pMatlab [BK07] and MatlabMPI [KA04] which shall be named here for the sake of completeness. Unfortunately, none of these approaches is known to integrate well with the current JVM-based Hadoop ecosystem, which is becoming the main source of data in production deployments. Another promising research direction is the deep embedding of the APIs of dataflow systems in their host language [Al15], where the potential to extend this embedding and the resulting optimizations to linear algebraic operations is currently explored [Ku16].

6 Conclusion

Gilbert addresses the problem of scalable analytics 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. The system 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 a highly optimized execution plan suitable for the execution on a supported engine. Gilbert allows the distributed execution on Spark and Flink.

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. Exemplary, we have implemented the PageRank and GNMF algorithm. The code changes required to make these algorithms run in Gilbert are minimal and only concern Gilbert's loop abstraction. Our benchmarks have proven 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 work include the development of a scalable data analysis tool which can be programmed using the well-known MATLAB language. We have introduced the fixpoint operator which allows to express loops in a recursive fashion. The key characteristic of this abstraction is that it allows an efficient concurrent execution unlike the `for` and `while` loops. Furthermore, we researched how to implement linear algebra operations efficiently in modern parallel data flow systems, such as Spark and Flink. Last but not least, we have shown that Flink's optimizer is able to automatically choose the best execution strategy for matrix-matrix multiplications.

Even though, Gilbert can process vast amounts of data, it turned out that it cannot beat the tested hand-tuned algorithms. This is caused by the overhead entailed by the linear algebra abstraction. The overhead is also the reason for the larger memory foot-print, causing Spark and Flink to spill faster to disk. 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. Interesting aspects for further research and improvements include adding 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 Flink's optimizer could result in beneficial synergies. Forwarding the inferred matrix sizes to the underlying execution system might help to improve the execution plans.

Acknowledgments. This work has been partially supported through grants by the German Federal Ministry for Education and Research for the project Berlin Big Data Center (funding mark 01IS14013A) and by the German Federal Ministry of Economic Affairs and Energy for the project Smart Data Web (funding mark 1MD15010A).

References

[02] Rmpi - MPI for R, 2002, URL: http://cran.r-project.org/web/packages/Rmpi/index.html, visited on: 10/06/2016.

[08] Apache Hadoop, 2008, URL: http://hadoop.apache.org/, visited on: 10/06/2016.

[09] Scala Breeze, 2009, URL: https://github.com/scalanlp/breeze, visited on: 12/11/2016.

[11] Apache Mahout, 2011, URL: http://mahout.apache.org/, visited on: 10/06/2016.

[14] Apache Spark, 2014, URL: http://spark.apache.org/, visited on: 10/06/2016.

[16] Apache Flink, 2016, URL: http://flink.apache.org/, visited on: 09/16/2016.

[93] The R Project for Statistical Computing, 1993, URL: http://www.r-project.org/, visited on: 10/06/2016.

[Al10] Alvaro, P.; Condie, T.; Conway, N.; Elmeleegy, K.; Hellerstein, J. M.; Sears, R.: Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In: Proceedings of the 5th European conference on Computer systems. ACM, pp. 223–236, 2010.

[Al15] Alexandrov, A.; Kunft, A.; Katsifodimos, A.; Schüler, F.; Thamsen, L.; Kao, O.; Herb, T.; Markl, V.: Implicit Parallelism through Deep Language Embedding. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 47–61, 2015.

[Be11] Beyer, K. S.; Ercegovac, V.; Gemulla, R.; Balmin, A.; Eltabakh, M.; Kanne, C.-C.; Ozcan, F.; Shekita, E. J.: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In: Proceedings of VLDB Conference. Vol. 4, pp. 1272–1283, 2011.

[BK07] Bliss, N. T.; Kepner, J.: pMATLAB Parallel MATLAB Library. International Journal of High Performance Computing Applications 21/3, pp. 336–359, 2007.

[Bo14a] Boehm, M.; Burdick, D. R.; Evfimievski, A. V.; Reinwald, B.; Reiss, F. R.; Sen, P.; Tatikonda, S.; Tian, Y.: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. IEEE Data Eng. Bull. 37/3, pp. 52–62, 2014.

[Bo14b] Boehm, M.; Tatikonda, S.; Reinwald, B.; Sen, P.; Tian, Y.; Burdick, D. R.; Vaithyanathan, S.: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. Proceedings of the VLDB Endowment 7/7, pp. 553–564, 2014.

[Da10] Das, S.; Sismanis, Y.; Beyer, K. S.; Gemulla, R.; Haas, P. J.; McPherson, J.: Ricardo: Integrating R and Hadoop. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 987–998, 2010.

[DG08] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51/1, pp. 107–113, 2008.

[DM82] Damas, L.; Milner, R.: Principal Type-Schemes for Functional Programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, pp. 207–212, 1982.

[El16] Elgohary, A.; Boehm, M.; Haas, P. J.; Reiss, F. R.; Reinwald, B.: Compressed Linear Algebra for Large-Scale Machine Learning. Proceedings of the VLDB Endowment 9/12, pp. 960–971, 2016.

[Fu09] Furr, M.; An, J.-h. D.; Foster, J. S.; Hicks, M.: Static Type Inference for Ruby. In: Proceedings of the 2009 ACM symposium on Applied Computing. ACM, pp. 1859–1866, 2009.

[Gh11] Ghoting, A.; Krishnamurthy, R.; Pednault, E.; Reinwald, B.; Sindhwani, V.; Tatikonda, S.; Tian, Y.; Vaithyanathan, S.: SystemML: Declarative Machine Learning on MapReduce. In: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. IEEE, pp. 231–242, 2011.

[Gu12] Guha, S.; Hafen, R.; Rounds, J.; Xia, J.; Li, J.; Xi, B.; Cleveland, W. S.: Large Complex Data: Divide and Recombine (D&R) with RHIPE. Stat 1/1, pp. 53–67, 2012.

[Hi69] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic. Transactions of the American Mathematical Society 146/, pp. 29–60, 1969.

[KA04] Kepner, J.; Ahalt, S.: MatlabMPI. Journal of Parallel and Distributed Computing 64/8, pp. 997–1005, 2004.

[Ku16] Kunft, A.; Alexandrov, A.; Katsifodimos, A.; Markl, V.: Bridging the Gap: Towards Optimization Across Linear and Relational Algebra. In: Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. BeyondMR '16, ACM, 1:1–1:4, 2016.

[Li10] Liu, C.; Yang, H.-c.; Fan, J.; He, L.-W.; Wang, Y.-M.: Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In: Proceedings of the 19th international conference on World wide web. WWW '10, ACM, pp. 681–690, 2010.

[LP16] Lyubimov, D.; Palumbo, A.: Apache Mahout: Beyond MapReduce. CreateSpace Independent Publishing Platform, 2016.

[Maa] MathWorks: Matlab Distributed Computing Server, URL: http://www.mathworks.com/products/distriben/, visited on: 10/06/2016.

[Mab] MathWorks: Matlab Parallel Computing Toolbox, URL: http://www.mathworks.de/products/parallel-computing/, visited on: 10/06/2016.

[Ma84] MathWorks: Matlab - The Language for Technical Computing, 1984, URL: http://www.mathworks.com/products/matlab/, visited on: 10/06/2016.

[Mi78] Milner, R.: A Theory of Type Polymorphism in Programming. Journal of computer and system sciences 17/3, pp. 348–375, 1978.

[Pa98] Page, L.; Brin, S.; Motwani, R.; Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. In: Proceedings of the 7th International World Wide Web Conference. Pp. 161–172, 1998.

[Ro14] Rohrmann, T.: Gilbert: A Distributed Sparse Linear Algebra Environment Executed in Massively Parallel Dataflow Systems, MA thesis, Technische Universität Berlin, 2014.

[Sa09] Samatova, N. F.: pR: Introduction to Parallel R for Statistical Computing. In: CScADS Scientific Data and Analytics for Petascale Computing Workshop. Pp. 505–509, 2009.

[Sc15] Schelter, S.; Soto, J.; Markl, V.; Burdick, D.; Reinwald, B.; Evfimievski, A.: Efficient Sample Generation for Scalable Meta Learning. In: 2015 IEEE 31st International Conference on Data Engineering. IEEE, pp. 1191–1202, 2015.

[Sc16] Schelter, S.; Palumbo, A.; Quinn, S.; Marthi, S.; Musselman, A.: Samsara: Declarative Machine Learning on Distributed Dataflow Systems. Machine Learning Systems workshop at NIPS/, 2016.

[SL01] Seung, D.; Lee, L.: Algorithms for Non-Negative Matrix Factorization. Advances in neural information processing systems 13/, pp. 556–562, 2001.

[Ve16] Venkataraman, S.; Yang, Z.; Liu, D.; Liang, E.; Falaki, H.; Meng, X.; Xin, R.; Ghodsi, A.; Franklin, M.; Stoica, I.; Zaharia, M.: SparkR: Scaling R Programs with Spark. In: Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data. SIGMOD '16, ACM, pp. 1099–1104, 2016.