SpMachO - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation

David Kernert
Technische Universität Dresden
Database Technology Group
Dresden, Germany
david.kernert@sap.com

Frank Köhler
SAP SE
Dietmer-Hopp-Allee 16
Walldorf, Germany
frank.koehler@sap.com

Wolfgang Lehner
Technische Universität Dresden
Database Technology Group
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

In the age of statistical and scientific databases, there is an emerging trend of integrating analytical algorithms into database systems. Many of these algorithms are based on linear algebra with large, sparse matrices. However, linear algebra expressions often contain multiplications of more then two matrices. The execution of sparse matrix chains is nontrivial, since the runtime depends on the parenthesization and on physical properties of intermediate results. Our approach targets to overcome the burden for data scientists of selecting appropriate algorithms, matrix storage representations, and execution paths. In this paper, we present a sparse matrix chain optimizer (SpMachO) that creates an execution plan, which is composed of multiplication operators and transformations between sparse and dense matrix storage representations. We introduce a comprehensive cost model for sparse-, dense- and hybrid multiplication kernels. Moreover, we propose a sparse matrix product density estimator (SpProdest) for intermediate result matrices. We evaluated SpMachO and SpProdest using real-world matrices and random matrix chains.

Categories and Subject Descriptors

G.1.3 [**Numerical Linear Algebra**]: Sparse, structured, and very large systems

General Terms

sparse linear algebra, optimization

1. INTRODUCTION

In the era of big data and data deluge, scientists and data analysists are confronted with a time-consuming implementation overhead, when they want to scale and speed up their existing, handcrafted code that has been working for years on small data sets. This set the way for new database applications in the fields of scientific computations and advanced analytics on large data. However, since most of the algorithms in science and data mining are composed of linear algebra expressions, conventional SQL-based relational database management systems (RDBMS's) did not match the requirements. On the other hand, numerical algebra systems like R are known for efficient linear algebra algorithms, but they lack scalability and data manipulation capabilities. As a consequence, the demand of data scientists for a scalable system that provides a basic set of efficient linear algebra primitives attracted the attention of the database community [11]. Recently emerged systems like SciDB [8] or SystemML [6] reacted by providing a R or R-like interface and deep integrations of linear algebra primitives, such as sparse matrix-matrix and matrix-vector multiplications.

In business environments, data analysts often load data from a relational database into a numerical algebra system to perform their analysis by means of linear algebra. For example, financial analysts that use a RDBMS for storing stock price data need the functionality of SQL, e.g., in order to get the average stock prizes, or to find all stocks that belong to a certain group. On the other side, they might want to use matrix multiplications to find correlations [24] of stocks and derivatives.

Matrices as first-class citizens have been integrated in many systems, for example in array DBMS's [8], data warehouses [19], or in-memory column stores [20], but only little has been done in the direction of optimizing the execution of linear algebra expressions. In this paper, we focus on optimizing the execution of sparse linear algebra expressions based on physical properties. The idea is the following: next to the RDBMS optimizer that creates optimized execution plans from SQL expressions, we propose a component (Fig. 1) that generates an optimal execution plan for linear algebra, which is using the native storage and execution engine of the system, and additional operators for linear algebra, such as a matrix multiplication operator.

As most of the big matrices occurring in the real world are *sparse*, linear algebra expressions often contain multiplications of three or more sparse, or mixed dense and sparse matrices, e.g. in transitive closure computations, Markov chains [27], linear transformation [13], or linear discrete dynamical systems [4]. An efficient execution of sparse matrix chain multiplications is nontrivial, in particular, if intermediate result matrices become dense. In many situations

10.5441/002/edbt.2015.26

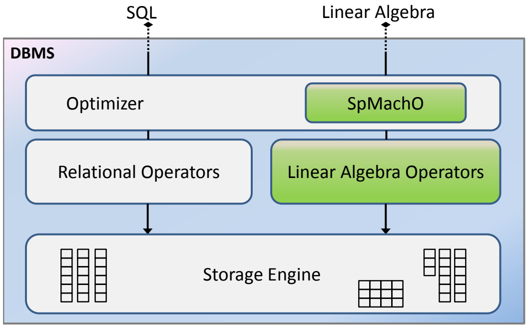


Figure 1: DBMS example architecture.

the runtime performance can be significantly improved by changing the execution order, or by switching from a sparse to a dense multiplication algorithm in later stages. Since data scientists are usually not familiar with algorithmic details of multiplication kernels and system parameters, and do not have profound knowledge of the characteristics of their matrices, it makes sense to leave these decisions to the system.

In particular, the contributions of this work are:

- **SpMachO**, a general matrix chain multiplication optimizer based on a dynamic programming approach, which leverages density estimations of intermediate results and different multiplication kernels to minimize the total execution runtime. The optimization problem and the SpMachO algorithm are presented in sections 2 and 3.
- **SpProdest**, a sparse matrix density estimator, which predicts the density structure of intermediate and final result matrices, by using a novel skew-aware stochastic density propagation method. It is described in detail in sections 4 and 5.
- An extensive evaluation and comparison of the execution runtime of the SpMachO-generated plan against alternative execution strategies and other numerical algebra systems, which is presented in section 6.

Finally, we will discuss the related work in section 7, followed by the conclusion in section 8.

2. EXPRESSION OPTIMIZATION

A lot of data scientists work with numeric algebra systems to run their linear algebra algorithms. However, the user is often let alone with the execution order, although the way of executing large sparse matrix expressions contains a significant optimization potential.

Consider a set of linear algebra expressions that consist of matrix multiplication and addition on general $\mathcal{R}^{m\times n}$ matrices. Further operations, like subtraction or division by a matrix $\boldsymbol{A}$ can be represented by using the corresponding inverse of addition $(-\boldsymbol{A})$, or inverse of multiplication $\boldsymbol{A}^{-1}$, respectively. Thus, the expression can be reduced to a form:

$$\boldsymbol{C} = \boldsymbol{A}_0 + \boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot ... \cdot \boldsymbol{A}_p + \boldsymbol{A}_{p+1} \cdot ... \boldsymbol{A}_l + ...,$$

From a mathematical perspective, the number of operations needed for the element-wise addition of intermediate results, or any element-wise operation in general, is independent from the execution order, thus, it is the same for $(\boldsymbol{A}+\boldsymbol{B})+\boldsymbol{C}$ as for $\boldsymbol{A}+(\boldsymbol{B}+\boldsymbol{C})$. Although this indifference might not hold in practise, when different physical representations are used, the addition part in the computation of $\boldsymbol{C}$ is rather cheap, since the complexity is at most $\mathcal{O}(mn)$ for dense matrices. Most of the execution time is spent in the computation of multiplications

$$\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot ... \cdot \boldsymbol{A}_{p-1} \cdot \boldsymbol{A}_p, \quad \boldsymbol{A}_i \in \mathcal{R}^{m_i \times m_{i+1}}, \tag{1}$$

so the work of this paper focus on the optimization of matrix chain multiplications. The algebraic degree of freedom to execute expression (1) consists in the setting of parenthesis, since matrix multiplications are associative. Altering the parenthesization does not change the result, but the number of operations required in the computation of the complete chain can vary significantly. Finding the optimal parenthesization for a dense, non-square matrix chain multiplication is well understood and serves as a text book example for the use of dynamic programming [16, 12]. The idea is to iteratively construct the optimal parenthesization as a combination of optimal sub-parenthesizations, by minimizing a cost recurrence expression with respect to the split point k

$$C_{\pi^{\mathrm{B}}(ij)} = \min_{i\leq k<j} \{ C_{\pi^{\mathrm{B}}(ik)} + C_{\pi^{\mathrm{B}}((k+1)j)} + \mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right) \}. \tag{2}$$

For the mathematical formulation, we introduce

- $\pi(ij)$: a parenthesization for the matrix (sub)chain $\boldsymbol{A}_{[i...j]}$
  $\pi^{\mathrm{B}}(ij)$ denotes the optimal ("best") one.
- C_π: the cost for executing the matrix chain multiplication, given a certain parenthesization π.
- $\mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right)$: the cost function for multiplying the two matrices that result from the subchains $\boldsymbol{A}_{[i...k]}$ and $\boldsymbol{A}_{[k+1...j]}$

In the textbook case, the combination cost $\mathtt{TM}\left(\boldsymbol{A}_{[i...k]}, \boldsymbol{A}_{[k+1...j]}\right)$ is set equal with the number of flops for multiplying the two dense intermediate result matrices. By using the classical inner product algorithm the costs can be exactly determined a priori as $m_i m_{k+1} m_{j+1}$.

However, the dense matrix case has certain limitations that restricts its relevance in practice. Most important,

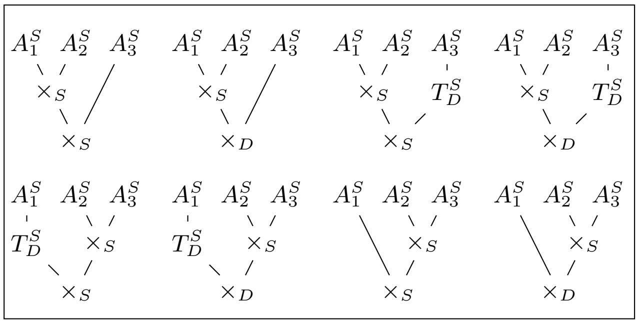


Figure 2: Eight of the possible 128 execution plans for the multiplication of three sparse matrices $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. $\times$ are binary multiplication operators, S/D denotes the internal *sparse*/*dense* representation type of matrices, and T are unary storage type transformations of matrices.

many of the real-world matrices in big data environments are sparse. A sparse matrix is not only defined by its row and column dimensions m and n, but also by the number and the pattern of non-zero elements N_{nz}, or the *density*[1] $\rho = \frac{N_{nz}}{mn}$. As a matter of fact, algorithms on sparse matrices have a different complexity: unlike the naive inner product algorithm for dense matrices, the cost of a multiplication of two sparse matrices rather depends on the number of non-zero elements than on their dimensions. Moreover, sparse matrices are stored in a different data structure than dense matrices, which leads to different actual costs depending on the characteristics of the physical representation. Most of the related work on matrix chain multiplications consider either dense-only [16, 12] or sparse-only [9] multiplications, being agnostic to the fact that the densities of the intermediate result matrices can vary significantly from the initial matrices. For example, the density of the result matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be much higher, or even less than that of both $\boldsymbol{A}$ and $\boldsymbol{B}$ (see Fig. 5). Despite the mathematical complexity, it is in many cases more efficient to continue using algorithms on dense matrix representations, if the density exceeds a certain threshold. This can be reasoned with the efficient and well-tuned implementations of dense matrix multiplication kernels in BLAS[2].

Our idea is to take the individual differences of the different matrix representations and multiplication kernels into account, and to exploit the potential performance benefits from changing the physical implementation of the initial matrices or intermediate results. We construct an execution plan for the chain expression that can contain dense, sparse and mixed dense/sparse matrix multiplications. Furthermore, the execution plan can contain conversions from a sparse into a dense representation. Therefore, we adopted the idea of dynamic programming and modified it in such a way that it incorporates the physical properties of the matrices. We extended the recurrence (2) by adding the input and output storage types as independent dimensions, and added cost functions for the storage type conversions:

$$
\begin{aligned}
&C_{\Pi^{\mathrm{B}}(ij)S^o} = \\
&\min_{\substack{i \le k < j \\ S^l, S^r, S^1, S^2 \in \mathcal{S}}} \Big\{ C_{\Pi^{\mathrm{B}}(ik)S^l} + C_{\Pi^{\mathrm{B}}((k+1)j)S^r} + \mathtt{TT}_{S^1}\left(\boldsymbol{A}_{[i..k],S^l,\rho}\right) \\
&+ \mathtt{TT}_{S^2}\left(\boldsymbol{A}_{[k+1..j],S^r,\rho}\right) + \mathtt{TM}_{S^o}\left(\boldsymbol{A}_{[i..k],S^1,\rho}, \boldsymbol{A}_{[k+1..j],S^2,\rho}\right) \Big\}
\end{aligned} \tag{3}
$$

- $\Pi(ij)$: execution plan for a matrix (sub)chain multiplication. It contains the execution order as well as all storage transformations. Π^{B} denotes the optimal plan.
- S^X: storage type, which is either dense or sparse. The superscript X labels each of the five matrices that are considered per execution node. $X = l$: left subplan output-, r: right subplan output-, 1: left input-, 2: right input-, o: current product output matrix .
- $\mathtt{TT}_{S^Y}\left(\boldsymbol{A}_{[i..k],S^X}\right)$: cost function for the conversion of a matrix from type S^X into type S^Y.

[1]In the remainder of this paper, we refer to the non-zero structure using ρ rather than N_{nz}

[2]Basic Linear Algebra Subprograms, http://www.netlib.org/blas/

Since the cost functions TM in equation (3) depend on the storage types of the input and output matrices, as well as their densities, it could be beneficial to convert a matrix from one into the other representation prior to the multiplication because conversions are usually less costly than multiplications. For example, if the initial matrices are in a sparse representation, and the dense multiplication kernel plus the conversion has a far lower cost than the sparse multiplication, then they are first converted into the dense representation. As a matter of fact, the value of the conversion cost $\mathtt{TT}_S(A, S)$ equals zero for identity transformations, i.e., when a matrix is already in the optimal representation. Hence, besides the parenthesization split point k, we vary the input and output storage types for each step in recurrence (3).

Some of the parameters that contribute to the cost functions $\mathtt{TM}/\mathtt{TT}(\cdot)$, for example the density ρ of intermediate results, are not known prior to the execution and have to be estimated. Therefore, we developed the sparse matrix product density estimator SpProdest, which is described in section 4 and 5 of this paper. The resulting costs derived from recurrence (3) are minimal, given that the estimated costs encoded in TM and TT are determined precisely. In particular, the optimality, or *goodness*, of SpMachO depends on two parts, which potentially contain uncertainties: first, the accuracy of the quantitative cost model of the multiplication kernels, and second, the precision of the density estimates provided by SpProdest.

The total number of the possible execution plans using our model (3) for a matrix chain multiplication of length p is

$$\mathcal{C}_{p-1} \cdot 2^{3(p-1)}, \tag{4}$$

where $\mathcal{C}_{p-1}$ denotes the Catalan number $\mathcal{C}_n = \frac{(2n)!}{(n+1)!n!}$.

$\mathcal{C}_{p-1}$ reflects the number of possible parenthesizations, which is the same as for the textbook case [12]. The second factor is related to the 2^3 {left input-, right input-, output-} storage type combinations that are connected with each of the $p-1$ type multiplication nodes. The number in (4) resembles the size of the search space, which grows exponentially. To give an example, it yields 2560 for a matrix chain of length $p = 4$ and already 1376256 for $p = 6$. As in [12], SpMachO solves the recurrence (3) in $\mathcal{O}(p^3)$ time using a bottom-up dynamic programming approach.

3. SpMachO

The pseudocode of SpMachO is sketched in Algorithm 1. The cost of the optimal sub-chain multiplications and the relevant plan information (split points and storage types per sub-chain) are cached in three-dimensional array structures. For each combination in the inner loop, the method chkMemLimit checks if the total memory consumption of the matrices and intermediate results in the current plan configuration would exceed the system limit. The memory required for dense $m \times n$ matrices is $\mathcal{O}(mn)$, and $\mathcal{O}(N_{nz} = mn\rho)$ for sparse matrices. Since SpMachO optimizes the runtime performance, the plan may contain conversions from sparse into dense matrix representations whenever the dense kernel leads to a lower overall runtime, due to the efficient dense kernel implementation. However, the conversions into dense representations potentially increase the memory consumption compared to a sparse-only plan. Our strategy is that every conversion is allowed, as long as the total memory consumption at every point in time does not exceed a hard memory

Algorithm 1 SpMachO

1: **function** spMacho(MatrixChain $\boldsymbol{A}_{[1..p]}$, TM, TT)
2: $\hat{\boldsymbol{\rho}}[][] \leftarrow$ spProdest($\boldsymbol{A}_{[1..p]}$)
3: **for** $1 \leq j < p, j > i \geq 0$ **do**
4: **for** $i \leq k < j$ **do**
5: **for** types $\in \{\text{sparse}, \text{dense}\}$ **do**
6: **if** !chkMemLimit(A, k, ρ, types) **then**
7: **continue**
8: $q \leftarrow \text{TT}(\boldsymbol{A}_{[i..k..j]}, \hat{\boldsymbol{\rho}}[][], \text{types})$
9: $q \leftarrow q + \text{TM}(\boldsymbol{A}_{[i..k..j]}, \hat{\boldsymbol{\rho}}[][], \text{types})$
10: **if** $q < \text{cost}[i][j][S^o]$ **then**
11: $\text{cost}[i][j][S^o] \leftarrow q$
12: $\text{plan}[i][j][S^o] \leftarrow \text{k, types}$
13: **if** $cost[1][p][\cdot] \geq$ MAXVAL] **then**
14: /* memory exceed exception */
15: **else**
16: **return** min(plan[1][p][sparse], plan[1][p][dense])

limit. Execution paths that would exceed the memory limit are automatically skipped (line 6). We assume that there is at least one plan that does not exceed the memory limit. If not, SpMachO returns with an exception (line 13). Finally, the resulting plan, which can be converted into a directed acyclic graph representation as of Fig. 2, is returned to the system for execution.

The complexity of Algorithm 1 is $\mathcal{O}(p^3)$, which can be derived from the dynamic programming loop and is the same as in the dense-only problem. The additional complexity by the introduction of storage type transformations yields a constant factor, since the inner loops over the storage types do not depend on the chain length p. A few pruning methods can be applied to reduce the execution plan space, for example, by excluding that the product of two dense can be sparse. However, they do not lower the asymptotic complexity of the algorithm.

The system executes the plan using the corresponding transformation and matrix multiplication operators. While the unary transformation operator either performs a dense-to-sparse or a sparse-to-dense storage transformation, the eight-fold multiplication operator delegates the execution to one of the multiplication kernels. We implemented all algorithms in our prototype using row-major 2D-arrays for dense- and the columnar compressed sparse row layout (CSR) for sparse matrices, since they are to our notion the most common physical representations, and are used in many numerical libraries, e.g. the Intel Math Kernel Library [1]. As mentioned in the beginning, we already showed in [20] that these representations can be mapped onto a columnar storage layout of an in-memory columnar DBMS.

3.1 Multiplication Kernels

There are eight different **ge**neral **m**atrix **m**ultiply (gemm) kernels that are used in our system. We will use the notation *xyz_gemm* to denote a multiplication kernel, where x is the left-hand input-, y is the right-hand input type and z the output matrix storage type, which can be either sparse (sp) or dense (d). Some of the kernels, for example the standard BLAS *ddd_gemm*, or *spdd_gemm*, are implemented by vendor-tuned C++ libraries, so we call the library instead of providing an own implementation. As of the current status, this is done only for *ddd_gemm*, for which we use the Intel MKL implementation.

Table 1: The matrix multiplication kernels for the product $C^{m\times n} = A^{m\times k} \cdot B^{k\times n}$ and their cost functions used in SpMachO. $N_\times$ denotes the number of actual multiplications, the hat indicates the corresponding estimated value. α, β, γ are constant parameters.

Kernel	Cost Function
ddd_gemm	$\alpha(mkn)$
spdd_gemm	$\alpha\hat{N}_\times$
dspd_gemm	$\alpha(mk) + \beta\hat{N}_\times$
spspd_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times$
ddsp_gemm	$\alpha\hat{N}_\times + \beta\hat{N}^C_{\text{nz}} + \gamma(mn)$
spdsp_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times + \gamma\hat{N}^C_{\text{nz}}$
dspsp_gemm	$\alpha\hat{N}_\times + \beta\hat{N}^C_{\text{nz}} + \gamma(mk)$
spspsp_gemm	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times + \gamma\hat{N}^C_{\text{nz}}$

Table 1 lists the kernels that are used in our system. In order to obtain the optimal execution plan via solving recurrence (3), the cost model and its corresponding parameters have to be determined accurately for each multiplication kernel. Since the actual runtime depends on many parameters and external influences, an exact determination cannot be guaranteed. However, even for small variations, the plan generated by SpMachO is still near-optimal, which we verified in the evaluation in section 6.

3.2 Execution Time Cost Model

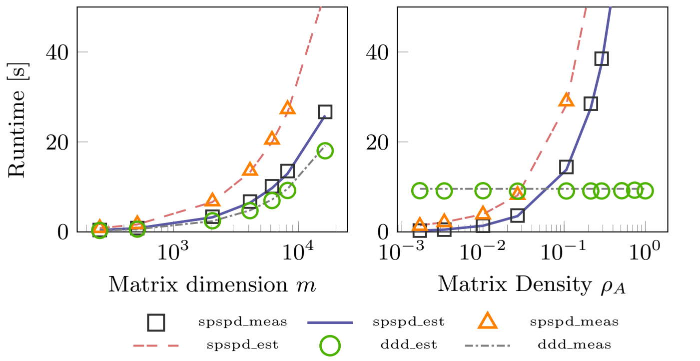
The cost for multiplying two dense or sparse matrices generally depends on the matrix dimensions m, k, n, the number and pattern of non-zero elements of both the input and the result matrices, and the implementation details of the corresponding kernel algorithm. The idea is to reduce the dimensions to a set of only a few, significant dimensions, and create the cost model based on the reduced dimension set. On average, it is a fair approximation to assume that the runtime of a single multiplication only depends on the number of non-zero elements, and not on the individual non-zero pattern variations. Hence, we are able to reduce the parameter space to the dimensions m, k, n, ρ_A, ρ_B, and $\hat{\rho}_C$, which corresponds to the product density estimation, which is determined by SpProdest.

As an example, we will examine the *spspsp_gemm* kernel that uses the popular Gustavson algorithm [18]. The algorithm is based on the *sparse accumulator* method, which is still commonly used for sparse matrix implementations [15]. In order to not repeat the algorithm description in detail, we only sketch the derivation of our cost model for *spspsp_gemm*, which we consider as the most interesting kernel.

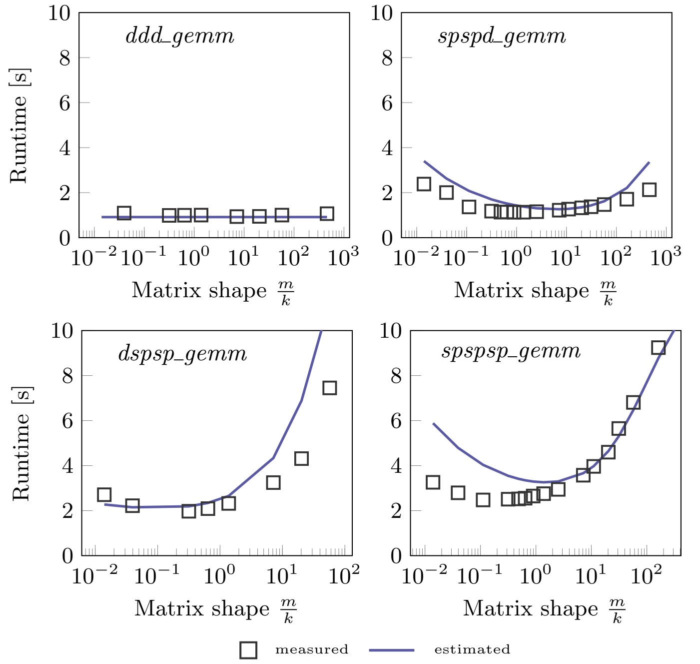
For reasonably large sparse matrices, the runtime of sparse kernels is often dominated by main memory bandwidth. Hence, mainly the memory accesses contribute to the runtime of an algorithm, which is in conformance to the external memory (I/O) model. For *spspsp_gemm*, the access pattern can be formulated as

$$\texttt{TM}(m, k, n, \rho_A, \rho_B) = m \times (\text{read}_{rowA} + k\rho_A \times (\text{read}_{colA,rowB} + n\rho_B \times (\text{read}_{colB,valA,valB} + \text{write}_{colC,valC})) + n\hat{\rho}_C \text{write}_{colC,valC}),$$

where the read/write denote the accesses to the respective data structures (*rowA, rowB,* etc.) in memory. For example,



(a) Scaling behavior: varying the dimension (left) and density (right). Other dimensions are fixed: $k = 8192, n = 8192, \rho_B = 0.1$



(b) Varying the matrix shape m/k. $N_\times$ is fixed in each plot.

Figure 3: Measured runtimes (markers) and time estimates (lines) for different matrix multiplication kernels.

a read on *colA* has a higher average cost than a read on *colB*, since a complete row of matrix $\boldsymbol{B}$ is touched in-between two consecutive *colA*-reads, thus, the system has most probably evicted the cache line of *colA* and has to fetch it again. The exact number of cycles, and hence, the required time per read or write access depends on whether the addressed cache line resides in the system cache. However, since we consider large matrices with sizes that are by factors larger than the last level system cache, we approximate the read and write accesses as fixed time constants in our model. Moreover, instead of determining the individual time constants for the $\text{read}_X/\text{write}_X$ access, we abstracted them into few parameters, which can then be determined empirically. Therefore, we expand the expression of `TM` and accumulate the time constants of the read/write accesses into the constant parameters α, β, γ, and obtain a simple time approximation for the

Wall-clock time

$$T \approx \alpha(\underbrace{m \cdot k \cdot \rho_A}_{N_{\text{nz}}^A}) + \beta(\underbrace{m \cdot k \cdot \rho_A \cdot n \cdot \rho_B}_{\hat{N}_\times}) + \gamma(\underbrace{m \cdot n \cdot \hat{\rho}_C}_{\hat{N}_{\text{nz}}^C}),$$

which only depends on the constant parameters

$$\alpha = T(read_{colA,rowB})$$
$$\beta = T(read_{colB,valA,valB} + write_{colC,valC})$$
$$\gamma = T(write_{colC,valC})$$

and the *derived* dimensions:

- N_{nz}^A: the number of non-zeros in matrix $\boldsymbol{A}$
- $\hat{N}_\times$: the estimated number of multiplications
- $\hat{N}_{\text{nz}}^C$: the estimated number of non-zero elements in the result matrix

For the other kernels, the cost function can be deduced in a similar manner. Because of space limitations, we will not discuss them in this paper. Table 1 lists the cost function for each kernel. Each cost function is a linear combination of different derived dimensions, weighted by constant parameters α, β, γ. The constant parameters are estimated for each kernel by a multilinear least-squares fit. Since they are dependent on the system hardware, the fit has to be done once for each system configuration.

Fig. 3a shows the scaling behavior of the matrix multiplication kernels *spspsp_gemm*, *spspd_gemm* and *ddd_gemm* with respect to matrix dimension and density. We observe that our cost models (lines) conform well with the actual algorithm runtimes (markers). From the right plot in Fig. 3a we can infer the following example scenarios: if the density ρ_A has a higher value than about 0.1, then it can be worthwhile to convert A into a dense representation and continue with the dense kernel – of course only if the dense representation does not exceed the available memory. If the density is below, it is probably best to select the *spspd_gemm* kernel. For $\rho_A \ll 0.01$ and depending on the following multiplications and the estimated intermediate result densities, it might be best to take with the *spspsp_gemm* kernel.

In Fig. 3b, we fixed the densities ρ_A, ρ_B and the product $(m \cdot k \cdot n)$, and only varied the relative matrix shapes $\frac{m}{k} \equiv \frac{k}{n}$. At both edges of each plot the matrices are extremely rectangular. The deviation of the actual times from our estimates shows that our cost model has limited accuracy in these extreme cases. However, the deviation is still acceptable, since the times are always overestimated. Overestimation is more robust than underestimation, because SpMachO would then just select another multiplication kernel, whereas in the latter case the deviation would propagate into the overall estimation. It is worthwhile mentioning that one conclusion we deduce from Fig. 3b is that simplistic cost models, which solely depend on the number of non-zero multiplications $N_\times$, are not able to describe the shape dependency, since $N_\times \equiv const.$ is fixed in each plot.

4. DENSITY ESTIMATION

The estimation of the intermediate result matrix densities is a crucial part of the SpMachO optimizer, since the cost models of the sparse multiplication kernels primarily depend on the number of non-zero elements, and hence, the matrix densities ρ. Our approach is to encode the non-zero structure into the smallest possible set of values without losing too much information.

a) $\begin{pmatrix} 0 & 2 & 0 & 0 \\ 5 & 0 & 0 & 8 \\ 0 & 0 & 8 & 0 \\ 4 & 0 & 0 & 0 \end{pmatrix} \quad \rho = \begin{pmatrix} \frac{5}{16} \end{pmatrix}$

b) $\begin{pmatrix} 0 & 2 & 0 & 0 \\ 3 & 9 & 0 & 1 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 2 & 0 \end{pmatrix} \quad \begin{pmatrix} \rho_{11} & \rho_{12} \\ \rho_{21} & \rho_{22} \end{pmatrix} = \begin{pmatrix} 0.75 & 0.25 \\ 0 & 0.75 \end{pmatrix}$

Figure 4: Assignment of non-zero population densities for two different matrices using a) scalar density, b) density map.

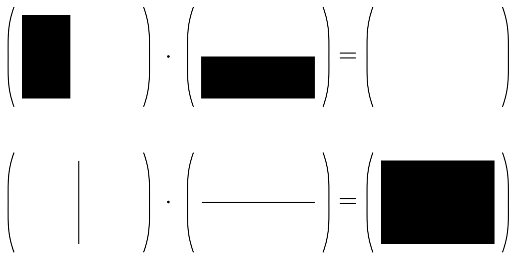


Figure 5: Product density in extreme non-uniform cases. The upper row shows how two half-populated matrices cancel to zero. The lower row shows how two, almost empty sparse matrices produce a full matrix (outer vector product).

4.1 Scalar Density

A matrix can be considered as a two-dimensional object which has a certain population density ρ of non-zero matrix elements. As an example, Fig. 4 a) shows a 4×4 matrix, which has five non-zero elements, and thus, a total population density of $\rho = 5/16 \approx 0.31$. The scalar density value does not reflect any patterns in the matrix, but it contains all relevant information if, and only if, the matrix is *uniformly* populated with non-zero elements. Then, the probability of a randomly picked matrix element for being non-zero is $p((A)_{ij} \neq 0) = \rho$, which is 0.31 in the example of Fig. 4 a).

Lemma 4.1. *Under the condition that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of a product of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation*

$$\hat{\rho}_C = \hat{\rho}_{A \cdot B} = 1 - (1 - \rho_A \rho_B)^k. \tag{5}$$

The estimate of Eq. (5) is unbiased, i.e. $E[\hat{\rho}_C] \equiv \rho_C$.

For the sake of simplicity, we denote the operation in Eq. (5) with the symbol $\odot$, thus, $\hat{\rho}_C = \rho_A \odot \rho_B$. Lemma 4.1 can be derived as follows: using the inner product formulation, the elements c_{ij} of the result matrix are calculated as $\sum_k a_{ik} b_{kj}$. The probability for a_{ik} being non-zero is $p(a_{ik} \neq 0) = \rho_A$, for b_{kj} accordingly: $p(b_{kj} \neq 0) = \rho_B$. Thus, every summand $a_{ik} b_{kj}$ is nonzero with probability $p_{nz}(a_{ik}) \wedge p_{nz}(b_{kj}) = \rho_A \rho_B$. c_{ij} is non-zero if any of the summands $a_{ik} b_{kj}$ is non-zero. We leverage the inverse probability and obtain $p(c_{ij} = 0) = \Pi_k(1 - \rho_A \rho_B)$. Finally, with $p(c_{ij} \neq 0) = 1 - p(c_{ij} = 0)$ and $p(c_{ij} \neq 0) = \rho_C$, equation (5) results. We remark that we are assuming *no cancellation*, i.e., a sum of products of overlapping non-zero elements never cancel to zero, which is a very common assumption in the mathematical programming literature [10].

Eq. (5) can be used as an $\mathcal{O}(1)$ estimator for the result density prediction of a multiplication of two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ that have uniform non-zero patterns. Hence, the prediction of a chain multiplication of p matrices has a linear time complexity $\mathcal{O}(p)$. Moreover, the density prediction is independent of the parenthesization.

4.2 Estimation Errors

However, the obvious disadvantage of maintaining a scalar density is that $\hat{\rho}$ is only valid for matrices with uniformly distributed non-zero elements. Although the uniform assumption holds for many matrices to a certain degree, there are many matrices that have distinguishable non-zero patterns, i.e. a topology with some regions that are significantly more dense than others. For these matrices, a density prediction according to equation 5 does not provide an accurate, unbiased result.

In extreme non-uniform cases, equation (5) could produce an asymptotic maximum error of 100%. Fig. 5 shows two example cases where the $\hat{\rho}$ estimate according to (5) fails significantly: In the first case, the product of two $n \times n$ matrices with each $\rho = 0.5$ cancel out into an empty matrix with $\rho = 0$, whereas the naive estimate according to Eq. 5 is $\hat{\rho} = 1 - (0.75)^n \xrightarrow{n \to \infty} 1$. Hence, the density value is maximal overestimated. The second example is a multiplication of two sparse $\rho = \frac{1}{n}$ matrices, which are zero except one column in $\boldsymbol{A}$ and the matching row in $\boldsymbol{B}$. The resulting full matrix has $\rho = 1$, whereas the naive prediction gives $\hat{\rho} = 1 - (1 - \frac{1}{n^2})^n \xrightarrow{n \to \infty} 0$.

In order to lower the average estimation error, we estimate matrix densities on a finer granularity. SpProdest uses a *density map* for non-uniform sparse matrices, which is able to reflect a $2D$ matrix pattern on a configurable, granular level.

4.3 Density Map

The density map $\boldsymbol{\rho}_A$ of a $m \times n$ sparse matrix $\boldsymbol{A}$ is effectively a $\frac{m}{b} \times \frac{n}{b}$ density histogram. It consists of $\left(\frac{mn}{b^2}\right)$ density values $(\rho)_{ij}$, each referring to the density of the corresponding block $\boldsymbol{A}_{ij}$ of size $b \times b$ in matrix $\boldsymbol{A}$. As an example, Fig. 4b) shows the density map of a 4×4 matrix with blocks of size 2×2.

In the following we sketch how the density map $\hat{\boldsymbol{\rho}}_C$ of the result matrix $\boldsymbol{C}$ are estimated from the density maps $\boldsymbol{\rho}_A$ and $\boldsymbol{\rho}_B$ of its factor matrices. Therefore, it is necessary to take a glance at blocked matrix multiplication. Assuming square blocks, the product matrix $\boldsymbol{C}$ can be represented as

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \\ \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{22} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \end{pmatrix}. \tag{6}$$

First, we define an estimator for the *addition* of two matrices:

Lemma 4.2. *Under the condition that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of the addition of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation*

$$\hat{\rho}_{A+B} = \rho_A + \rho_B - (\rho_A \rho_B) \equiv \rho_A \oplus \rho_B. \tag{7}$$

The derivation of Lemma 4.2 is similar to that of Lemma 4.2, which is why we leave it out for space reasons.

Then, combining Eq. (6) with (5) and (7), one obtains

$$\hat{\boldsymbol{\rho}}_C = \begin{pmatrix} \rho_{A_{11}} \odot \rho_{B_{11}} \oplus \rho_{A_{12}} \odot \rho_{B_{21}} & \rho_{A_{11}} \odot \rho_{B_{12}} \oplus \rho_{A_{12}} \odot \rho_{B_{22}} \\ \rho_{A_{21}} \odot \rho_{B_{11}} \oplus \rho_{A_{22}} \odot \rho_{B_{21}} & \rho_{A_{21}} \odot \rho_{B_{12}} \oplus \rho_{A_{22}} \odot \rho_{B_{22}} \end{pmatrix}$$

for the density propagation of a 2×2 map. Density maps of a finer granularity, i.e. with more than four blocks, are calculated accordingly.

As a result, the average density estimation error is significantly lowered when using density maps compared to the scalar density estimation, which we verified empirically in the evaluation section (Fig. 6.) As a matter of fact, the smaller the block size and the higher the granularity, the more information is stored in the density map and finer structures can be resolved. However, the runtime of the density map estimation also grows with the granularity, since its complexity is in $\mathcal{O}(\left(\frac{n}{b}\right)^3)$, and hence, $\mathcal{O}(p\left(\frac{n}{b}\right)^3)$ for a chain estimation of length p. For infinitesimal block sizes $b \to 1 \times 1$, the estimation error vanishes completely, but the determination of $\hat{\boldsymbol{\rho}}_C$ is then equivalent with the corresponding boolean matrix multiplication of $\boldsymbol{A} \cdot \boldsymbol{B}$, and has the same problem complexity as the actual multiplication. Thus, the block size configuration is generally a trade-off between accuracy and runtime of the prediction.

However, we employ a greedy strategy, which reduces the runtime by using density maps only for matrices with a skewed non-zero distribution, and the scalar density for matrices with an approximately uniform distribution. To decide whether a given matrix has an uniformly distributed or a skewed non-zero pattern we define a quantitative disorder measure for sparse matrices.

4.4 Matrix Disorder Measures

We introduce two measures to quantify how the non-zero pattern of a sparse matrix deviates from an (approximate) uniform distribution. Since we already introduced the density map that involves blocks of different densities, it is natural to approach the problem from same the block-granular level.

4.4.1 *Variance Analysis*

One way of deciding whether or not a matrix is approximately uniformly distributed is to make use of a statistical hypothesis testing method. The scalar density propagation formulas as shown in Lemmas 4.1 and 4.2 are based on the assumption that every element of the matrix has the same probability to be populated, and the probability is equal to the overall density ρ. Using this assumption as the null hypothesis H_0, the number of non-zero elements in each $b \times b$-block would follow a binomial distribution $\mathcal{B}(N, p)$ with $N = b^2$ and $p = \rho$. This can be deduced in the same way as a coin toss experiment, where the number of experiments N is equal to the number of potential elements in a block, and the *success* probability $p = \rho$ equals the global population density. From elementary statistics [7] we get

$$E[N_{\text{nz}}^{b\times b}] = b^2\rho, \qquad E[V(N_{\text{nz}}^{b\times b})] = b^2\rho(1-\rho) \tag{8}$$

for the expectation values of for the number of non-zero elements N_{nz} in one $b \times b$ and the variance of $N_{nz}^{b\times b}$ when assuming a binomial distribution $\mathcal{B}(b^2, \rho)$.

The (dis-)conformance of the null hypothesis H_0 with the reality can be determined using a simple one-factorial variance analysis. Therefore, we use the F-test [7], a likelihood quotient test, which checks the conformance of the observed variance of two random, normally distributed[3] variables X and Y. If the test statistic $f = V_X/V_Y$ (ratio of variances in X and Y) exceeds a critical value f_{crit} according to an α-quantile of the F-distribution, then H_0 is rejected, meaning that X and Y are not of the same distribution with probability 1-α.

For a matrix with N_B $b \times b$ blocks, we define

$$V_{\text{observed}} = \frac{1}{N_B - 1} \sum_{ij} (N_{\text{nz}}(ij) - E[N_{nz}^{b\times b}])^2 \tag{9}$$

$$f = \left(\frac{V_{\text{observed}}}{V_{\text{expected}}}\right) = \left(\frac{V_{\text{observed}}}{E[V(N_{nz}^{b\times b})]}\right) \tag{10}$$

For uniformly distributed matrices, f has the expectation value $E[f] = 1$. The exact choice of the threshold, however, depends on the sample size, i.e. the number of blocks N_B and the desired accuracy.

4.4.2 *Entropy*

A known measure for the disorder of elements is the entropy

$$\sum_i^N -p_i \ln p_i, \tag{11}$$

which is defined over a space with N entities (or states) i that have a relative probability p_i. The entropy is used in a variety of contexts. To name an example, the Shannon entropy [26] is used in information theory to quantify the information content of a message with N characters as entities. In a similar manner, we can define and quantify the information content of the non-zero pattern of a sparse matrix, by identifying the p_i with the local block density ρ_{ij}.

The entropy (11) is maximal if each entity has the same probability. In the terms of sparse matrices, the theoretical disorder is maximized if every block has the same local density $\rho_{ij} \equiv \rho$. The scaled entropy

$$\tilde{H} = \frac{H}{H_{\max}} = \frac{\sum_{ij}^{N_B} \rho_{ij} \ln \rho_{ij}}{N_B \rho \ln \rho} \in [0, 1] \tag{12}$$

is sensitive to the matrix density skew, which we evaluated in section 6. However, in contrast to f, the entropy is rather suited for measuring *relative* changes in the non-zero disorder, and it is hard to interpret the absolute value of $\tilde{H}$.

5. SpProdest

SpProdest is sketched in algorithm 2. First, the disorder measure $\delta = \sqrt{1/f}$ is retrieved (getDisorder) for each matrix, which is a modified version of f according to Eq. (10). Then, δ is used to decide whether to store a only scalar density value, or a density map (line 4). If δ is lower than a certain threshold δ_T, then the density map is created, if the disorder is higher, then a scalar density is chosen. Finally, the density estimates are calculated by using the probabilistic density propagation method (EstProdDensity), according to equations (5) and (7). Note that instead of calculating δ and $\hat{\boldsymbol{\rho}}$ for each expression (line 4-7), they can be cached as matrix statistics in the system, and reused for further multiplications.

The complexity of SpProdest depends on the actual granularity of the density map. Assuming a chain multiplication

[3]for sufficiently large N and $Np \to$ const., the binomial distribution can be approximated by a normal distribution

Algorithm 2 SPPRODEST

```
function SPPRODEST(MatrixChain A[1..p])
    ρ̂[][] ← 0
    for Matrix A_i ∈ A[1..p] do
        δ ← GETDISORDER(A_i)
        if δ < δ_T then
            ρ̂[i][i] ← SCALARDENSITY(A_i)
        else
            ρ̂[i][i] ← DENSITYMAP(A_i)
    for 1 < j < p do
        for j > i > 0 do
            ρ̂[i][j] ← ESTPRODDENSITY(ρ̂[i][j − 1], ρ̂[j][j])
    return ρ̂[][]
```

of p square matrices, the time complexity would be in the best case $\mathcal{O}(p)$ (no map) and in worst case $\mathcal{O}(p(\frac{n}{b})^3)$, which is equal to the chain multiplication of p $\frac{n}{b} \times \frac{n}{b}$ matrices, where $\frac{n}{b}$ is the dimension of the density grids. Analogously, the space complexity of SPPRODEST is best case $\mathcal{O}(p)$, worst case $\mathcal{O}(p(\frac{n}{b})^2)$. In practice, the overhead of the SPPRODEST component is negligible against the potential speedup gained by the SPMACHO, which is manifested in our evaluation in section 6.3.

6. EVALUATION

In this section, we first evaluate the accuracy of SPPRODEST according to the deviation in the result sparse matrix densities. Second, we apply SPMACHO on different matrix chains and compare the execution runtime against R and a popular commercial numerical algebra system for matrix computations (called system A here), and two further execution approaches. The platform for our prototype implementation is a two-socket Intel Xeon X5650 CPU with 2×6 cores with 2.66 GHz and a total of 48 GB RAM.

6.1 Density Estimate Accuracy

As mentioned in section 4, the density ρ_C of a matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ depends not only on the densities of the factor matrices ρ_A and ρ_B, but also on their non-zero patterns, especially on the pattern skew. To show the effect of the non-zero pattern skew on the density estimation, we generated a set of matrices with increasing skew. The skew parameter ξ in our example defines the slope of a linear ascend in the density distribution with increasing row number r: $\rho(r) = \xi \cdot r$. We fixed the total density of $\boldsymbol{A}$, thus, N_{nz} =const.

The left-hand plot in Fig. 6 shows the actual density ρ_C and the estimated densities using the scalar density and the density map approach with different block sizes. In our example, the density of the product matrix ρ_C decreases with increasing pattern skew. This conforms with our notion, that in most cases a higher skew at constant density leads to a lower density of the result matrix, although there are cases which show the opposite behavior, for example, as in Fig. 5.

Nevertheless, it is clearly observable that a finer granularity of the density map, and thus, a smaller block size, results in a better density estimation. The idea is to choose the block size as large as possible, since a finer granular density grid negatively influences the runtime performance of SPPRODEST. We chose a block size of 256×256 as a good compromise between accuracy and estimation runtime. The right-hand plot of Fig. 6 confirms that the disorder measures are both

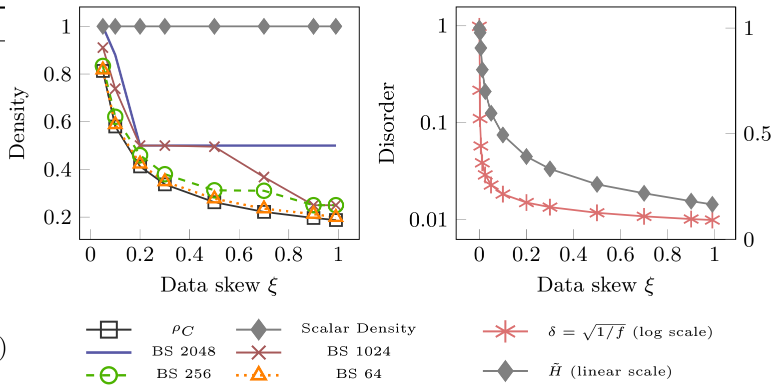


Figure 6: Left: Estimated density $\hat{\rho}_C$ vs. actual density (ρ_C) of the product matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ using the scalar and the density map estimation with different grid block sizes (BS). Matrices: $\boldsymbol{A} \in \mathcal{R}^{4096 \times 2048}, \boldsymbol{B} \in \mathcal{R}^{2048 \times 4096}$ and average density $\langle\rho\rangle = 0.1$. Right: Influence of the matrix $\boldsymbol{A}, \boldsymbol{B}$ nonzero skew on the disorder measures $\delta, \tilde{H}$.

sensitive to the nonzero skew, but the teststatistic-based disorder measure δ is much more sensitive to the skew than the entropy-based $\tilde{H}$. In particular we can observe that the trivial scalar density provides a sufficient accuracy for approximately uniform nonzero patterns ($\xi \to 0$).

6.2 Plan Ranking

In this experiment we evaluate the total cost model accuracy of SPMACHO by comparing the estimated runtime against the actual runtime of each possible plan. Hence, SPMACHO is optimal if the plan with the lowest actual runtime has also the lowest estimated runtime. However, this experimental verification of optimality requires to run all possible execution plans, which is not feasible for longer matrix chains due to the exponentially growing number of plans according to Eq. (4). Thus, we did a "brute-force evaluation" only for matrix chains of length $p = 3$:

$$\boldsymbol{A}_1^S \cdot \boldsymbol{A}_2^S \cdot \boldsymbol{A}_3^S. \tag{13}$$

Although there are only two ways of setting the parenthesis in this expression, with all the possible storage type transformations per multiplication node, one obtains 128 different execution plans. Fig. 2 shows some of the possible plans, to illustrate the problem complexity. The execution plans are composed of

- **multiplication operators** ${}^{D/S}\times^{D/S}_{D/S}$, which can either produce a *sparse* result matrix or a *dense* one. For a chain of length p there are exactly $p-1$ multiplication operators.
- **transformation operator** $T^{S/D}_{S/D}$ that transforms the intermediate result from one storage representation into another (which is either dense or sparse.) There can be none or up to 2^{3p-1} transformation operators.

SPMACHO estimates the cost for each operator via the cost functions `TM`,`TT` described in section 2. As a consequence, each operator estimation potentially contributes to the absolute execution runtime error.

In this experiment, we first executed all 128 plans and measured the *actual* execution runtime. Thereafter, we computed the runtime estimations using SPMACHO's cost model

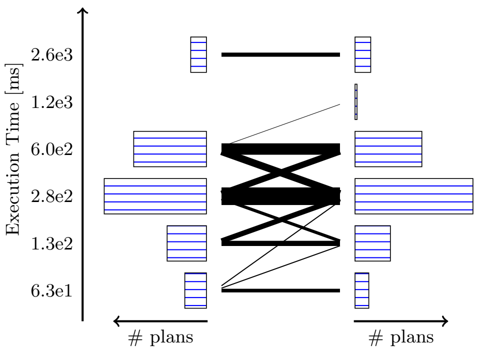


Figure 7: Vertical histogram of the actual (bars on left-hand side) and estimated (bars on right-hand side) runtimes of all 128 possible execution plans. The edges denote the plan (dis-)placement, the linewidth correlates with the corresponding number of plans. The vertical time axis has a logarithmic scale.

for each plan. Fig. 7 shows the histograms for the actually measured (left-hand side) and the estimated (right-hand side) execution runtimes. Note that the vertical axis has a logarithmic scale, hence, the width of the upper bins refers to a larger time interval than the width of the lower bins. The connecting edges in-between the bins of the two histograms show where the plans of the actual runtime histogram are placed in the estimated runtime histogram. If there is an ascending edge, for example from the lowest bin in the left histogram to the second lowest bin of the right histogram, then there is at least one plan, whose runtime was overestimated. If the edge is horizontal, then the estimated runtimes of all plans corresponding to this edge are within the same time interval as their actual runtimes. The width of the edges indicates how many plans are affected. It is worthwhile mentioning that for the selection of the best execution plan, the quantitative estimation of execution runtimes could potentially differ arbitrarily from the actual runtimes, as long as the estimated *order* of the plans preserves the actual runtime order correctly. This condition is only violated for the edges that are crossing another edge. If the total runtime for a plan is significantly under- or overestimated, its corresponding edge crosses multiple other edges. Indeed, the goodness of the cost model can be defined by the number of edges crossings, weighted by the number of plans per crossing edge.

The majority of estimations, which are shown in Fig. 7, are in the correct corresponding time bin. Although there are quite a few crossings, especially in the middle part, most of the edges only span over to the neighboring bin. Moreover, for the selection of the most efficient plan, only the lower part of Fig. 7 is relevant. In particular, the plan with the lowest estimated runtime, which is generated by SpMachO, should be contained in the lowest bin of the actual runtime histogram. Since all edges of the lowest estimated time bin originate from the lowest actual time bin, we observe that the SpMachO selected plan is at least among the top k plans, if not the best.

6.3 Performance Comparison

We compared the absolute execution runtime of a matrix chain multiplication expression using the optimized plan by SpMachO against R and the commercial system A. Both systems contain classes and algorithms for dense and sparse matrices. In R (V3.0.0) we used the Cran R matrix package[2] (V1.0.12), in system A we used the native sparse matrix representation. In addition,we included the following alternative execution approaches to the measurement:

- **Left-deep, sparse only:** All matrices are multiplied using a sparse-sparse into sparse multiplication using the *spspsp_gemm* kernel (${}^{S}\times^{S}_{S}$), starting with the first left pair and proceeding into the right direction.
- **Right-deep, sparse-dense-dense:** The outermost right pair is multiplied using sparse-sparse into dense multiplication (${}^{S}\times^{S}_{D}$). Then, the matrices on the left are consecutively multiplied with the right-hand dense intermediate result matrix using the sparse-dense into dense multiplication kernel (*spdd_gemm*, ${}^{S}\times^{D}_{D}$).

Both approaches use the same infrastructure as SpMachO. The reason why we chose exactly these two specific execution strategies is that either of them turned out to be good (or even optimal) for a reasonable large fraction of matrix chains. In particular, they yield good performance if the inter-matrix skew is low, i.e., the dimensions and the densities do not differ, since in these cases, the impact of parenthesization on the optimization is less significant. We also tried other alternatives to the dynamic programming approach of SpMachO, for example, a method that picks an execution plan based on the metaheuristic simulated annealing. However, due to the high dimensionality of the search space and the large discrepancy in the runtimes, it turned to be out to be far worse in most cases, hence, we did not include it in the measurements.

6.3.1 Data Set

Since there are currently no standardized benchmarks for large scale linear algebra expressions, it is generally difficult to provide a comprehensive performance comparison. Therefore, we created two performance experiments: first, we took real world-matrices of different domains and compared the execution runtime of self-multiplication chains (matrix powers). Thereafter, we generated random matrices of different dimension and density skews in order to study the systematic behavior of SpMachO.

Table 2: Sparse matrices of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density (rounded) of each matrix. All matrices are square ($n \times n$.)

Name	Matrix Domain	Dim.	N_{nz}	$\rho \cdot 10^2$ [%]
NCSM1	Nuclear Physics	3440	2.930 M	24.7
PWNET	Power Eng.	8140	2.017 M	3.0
JACO1	Econometric	9129	56 K	0.07

Tab. 2 lists the matrix data sets which we used in the evaluation. The first matrix NCSM1 is taken from a nuclear physics group, the other two (PWNET, JACO1) are from the Florida Sparse Matrix Collection[4].

In our prototype system, some of the multiplication kernels are implemented single-threaded, whereas other kernels have parallel implementations. Although we emphasize that the

[4]`http://www.cise.ufl.edu/research/sparse/matrices/`

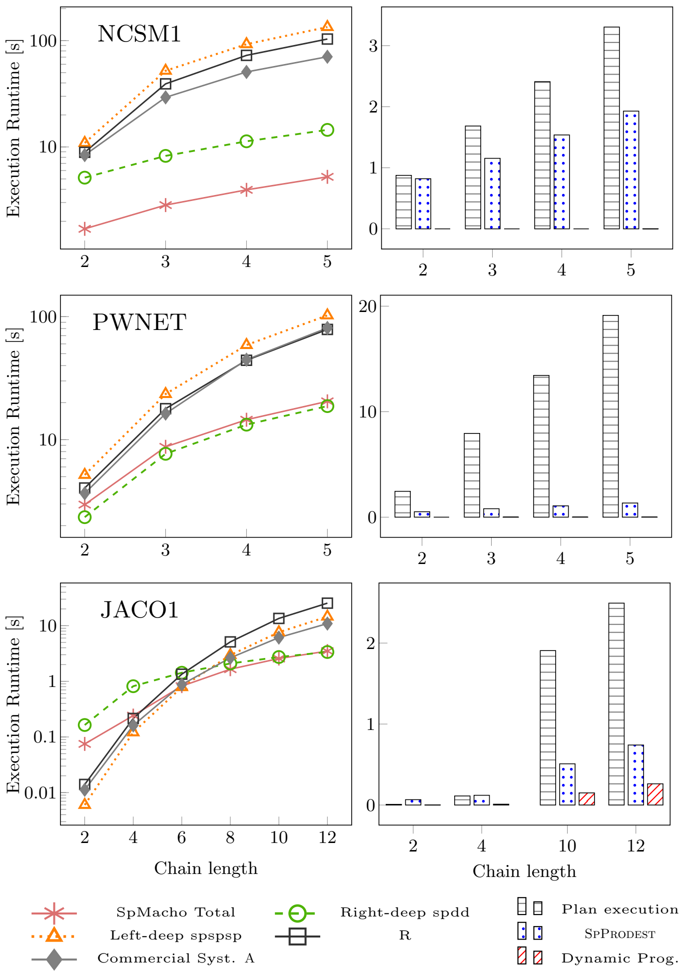


Figure 8: *Left Column*: Measurement of the execution runtime of sparse matrix chains (matrix powers). *Right*: The total runtime of SpMacho is visually separated into its components: the plan execution, the SpProdest runtime and the dynamic programming part.

conceptual execution plan optimization of SpMacho works orthogonal to the individual performance of the multiplication kernels, the absolute execution runtime does obviously also depend on the low level implementation of each algorithm. Hence, there is still potential to reduce the overall runtime further by switching completely to massively parallelized multiplication kernels. However, although we use mostly sequential kernels in the prototype, our algorithm was still able to outperform R and the commercial system A.

6.3.2 Self Multiplications

In this part we discuss the performance of matrix self multiplications (matrix powers), which is for example used for the calculation of Markov chains models.

The left column of Fig. 8 shows the absolute runtimes using SpMacho versus R, commercial system A, and the right-deep *spspsp* and left-deep *spdd* approaches. The first notion is that the relative performance speedup of SpMacho becomes more significant with increasing chain length. For matrices with a relatively high density, e.g. NCSM1, SpMacho outperforms the other systems even for a single multiplication already by several factors. In this case, SpMacho recognizes that it is worth to convert the matrix into dense representations prior to the multiplication. For the second matrix chain (PWNET), the performance gap is increasing with the chain length up to a speedup factor of five. Only in the third plot (JACO1), the overhead of SpMacho amortizes not before a chain length of four. In this relatively simple case of matrix self multiplications, the speedup is related to the density evolution of the intermediate results. When the matrix reaches a relatively high density in an early stage of the execution plan, SpMacho is likely to choose dense formats and proceed with dense multiplication kernels, whereas the use of sparse-only kernels will have a poor performance for every additional multiplication. That also explains why the right-deep *spdd_gemm* strategy is often optimal, e.g. for the PWNET matrix chain.

The right column in Fig. 8 shows the separate runtimes of each component of SpMacho, which are: the plan execution, the SpProdest runtime, and the dynamic programming loop. The runtime of the dynamic programming part is negligible and only visible for longer chains (10-12). Note that we the SpProdest cost can be further reduced, if we cache the density maps. As of now, they are created once prior to each expression execution, consuming most of SpProdest's runtime.

As a side note, the R and commercial system A runtimes show astonishing similarity with our left-deep *spspsp_gemm* approach. We assume that they use a similar way of execution.

6.3.3 Random Matrix Chains

In the next experiment, we used three different, randomly created matrices. Products of *three* matrices are very common in many applications, for example in algorithms that contain matrix factorizations.

In order to observe the systematic influence of the data skew on the execution runtime, we varied three skew dimensions: the matrix shape skew, the inter-matrix density skew and the matrix intra-density skew (as of section 6.1). For each skew dimension, we varied a parameter $\xi \in [0, 1]$ that quantifies the skew in a range from zero (no skew) to one (maximum skew). More precisely, the parameter dimensions $(m/n)_i$, ρ_i and the intra-density skews ξ_i, are randomly picked from a $<$*min*, *max*, *average*$>$ distribution, where ξ corresponds to the deviation from the *average* value.

Since we created the matrices randomly for each skew parameter configuration, one single configuration can have various random instances, which results in a potentially large variety of different runtimes. This is reasoned by the fact that a skew in the matrices can affect the execution runtime in both directions – increasingly or decreasingly. Generally speaking, a large skew in the data can dramatically slow down naive execution approaches, but also reveals a large optimization potential for SpMacho by exploiting the skew. In contrast to the previous self-multiplication experiment, a skew in the matrices leads to a higher influence of the parenthesization, and the selection of storage representations and algorithms.

To be independent of particular random matrix instances, we repeated the measurement multiple times, hence, we took 25 different randomly created sparse matrix chains of length three per configuration. Fig. 9 shows for each skew configuration a box plot with the corresponding median, lower quartile,

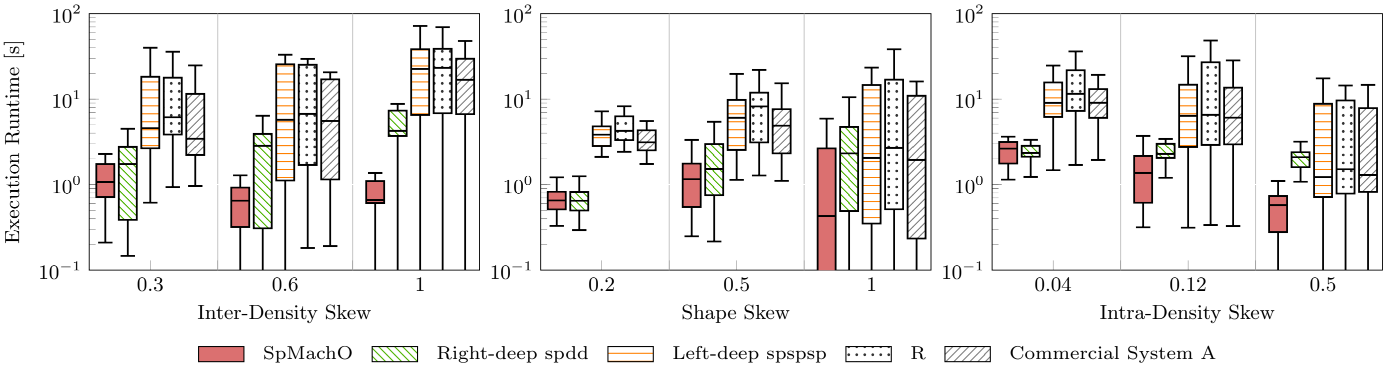


Figure 9: Average (log scale) runtime and variance comparison of SpMachO vs. the right-deep spdd and left-deep spspsp approaches, R, and the commercial system for a multiplication of the expression $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. We varied in x-direction: the inter-density skew (left), the shape skew (middle) and the intra-density skew (right). The bounding *<min,max,avg>* distributions for the data skew are *<0.001, 0.5, 0.025>* for matrix densities, and *<32, 16384, 3072>* for the matrix dimensions. The intra-density skew ξ was chosen according to Fig. 6 with values ranging from 0 to 0.5.

upper quartile and whiskers of the execution runtimes. Note that our measured runtime of SpMachO includes the time for the density estimation (SpProdest). We observe the following characteristics: First, for low skew parameters, both SpMachO and the right-deep spdd outperform the other approaches for most of the instances. For unskewed matrices, this underlines the result that we already obtained in the previous experiment for matrix chains with similar densities, i.e. that the right-deep spdd multiplication execution is optimal if intermediate results are rather dense. Second, and more interesting, is the development of the runtime distributions with higher data skews. In the inter-density skew experiment (left plot), the median execution time in R, the commercial system A and the left-deep spspsp approach increases, whereas the SpMachO median time stays low and gets even lower for $\xi = 1$. Moreover, the variance in time of SpMachO grows notably slower than these of the other systems. In the other two plots, we see a similar picture, although most of the median execution times decrease slightly in the shape skew experiment (middle plot), and more significantly in the intra-density skew experiment (right plot). Here, the increased intra-density skew leads in the majority of cases to a reduced execution runtime, which conforms to our notion. Still, in quite a few cases the runtime of the other systems explodes, leading to the observed high variance and scattering of the execution times. The right-deep spdd approach is more robust, but not optimal for high skews. In contrast, SpMachO is able to reveal the skew and exploit it for optimization. As a result, we observe that SpMachO has a by far better worst-case behavior than the established systems.

7. RELATED WORK

As this work has overlaps with multiple research areas, we subdivide the discussion into the major subtopics:

7.1 Optimization of Linear Algebra Expressions

Despite the optimization potential, we did not find that common numerical algebra systems optimize the execution of linear algebra expressions based on matrix sparsity and dimension characteristics. In contrast, the idea of optimizing linear algebra operations on system level has been mentioned in SystemML [14, 6], which describes a Hadoop-based machine-learning framework with an *R*-like declarative language. However, the cost model they describe in [6] is based on independent, one-dimensional scaling functions, and they assume full density ($\rho = 1$) for intermediate results. In [5] they mention that they optimize by assuming "independence with regard to the sparsity of intermediates". In contrast, we observed that particularly in situations with large density differences (inter-density skew) the density of intermediate results influences the optimization significantly. Since simple models are unable to reconstruct the complicated runtime behaviour of matrix multiplication kernels, we also put our focus on accurate cost models from an algorithmic perspective. In addition, our cost analysis revealed that matrix dimensions and the matrix sparsity can not be regarded as independent parameters.

7.2 Matrix Chain Multiplication and Density Estimation

In contrast to dense matrix chain multiplication, which has been discussed thoroughly in the past decades, e.g. in [12, 16, 21], there is little work about sparse matrix chain multiplications. Interesting work that should be mentioned in this context is from Cohen [9, 10], who extended the dynamic programming approach idea to sparse matrices. In her work, she minimizes the overall number of floating point operations that are needed to compute the matrix chain product by predicting the non-zero structure for intermediate result matrices on row/column-level. The density prediction algorithm of [10] is based on random number propagation in a layered graph, and has a fixed complexity $\Theta(\sum N_{nz,i})$. However, as observed in section 2, the actual runtime cost of sparse matrix multiplication kernels are not just proportional to number of floating point operations. Moreover, coming from a real system perspective, we consider not only pure sparse-sparse matrix multiplications, but leverage sparse-dense transformations and the coexistence of sparse and dense matrices to optimize on a more complete level.

7.3 Join Optimization

The problem of sparse matrix chain multiplication is related to join enumeration and cardinality estimation in a relational database management system RDBMS. This connection is more obvious when sparse matrices are represented as $\langle row, col, val\rangle$ triple tables [20]. In fact, a multiplication can then be expressed as a join aggregation [3]. The use of dynamic programming in join optimization [25, 22] and join plan generation with respect to physical table properties [17] were inspiring for this work, as well as the use of multidimensional histograms [23] for the optimization of queries on multidimensional data. Although the mathematical characteristics of matrices and multiplications require a slightly different perspective, it is an interesting aspect that some of the ideas of relational join optimization can be used for linear algebra.

8. CONCLUSION

In times of emerging analytical and scientific databases, many systems [8, 6] started to deeply integrate linear algebra. This work shows that integrating linear algebra operations, such as matrix multiplications, is not just adding algorithms to the database engine. In fact, due to different matrix representations, algorithms, and the presence of data skew, we observed that a naive execution of sparse matrix products can be up to orders of magnitude slower than an optimized one.

In this paper we presented SpMachO, which optimizes sparse, dense and mixed matrix multiplications of arbitrary length, by creating an execution plan that consists of transformation and multiplication operators. By using detailed cost functions of different sparse, dense and mixed matrix multiplication kernels, SpMachO leads to a faster and more robust execution compared to widely used algebra systems. Moreover, our density prediction approach SpProdest with an entropy-based skew awareness enables accurate memory consumption and runtime estimates at each stage in the execution plan.

To put it in a nutshell, we showed how methods inspired from database technology can improve linear algebra computations, and took a step into the direction of taking complexity from data scientists – who should not be required to have profound knowledge about the connections between mathematical optimizations, matrix characteristics, algorithmic complexities and the hardware parameters of their system.

9. REFERENCES

[1] Intel® Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[2] CRAN R Matrix Package, http://cran.r-project.org/web/packages/Matrix/index.html.

[3] R. R. Amossen and R. Pagh. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT*, 2009.

[4] K. Behrend. Dynamical Systems and Matrix Algebra. 2008.

[5] M. Boehm, R. B. Douglas, A. V. Evfimievski, et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.

[6] M. Boehm, S. Tatikonda, B. Reinwald, et al. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *VLDB*, 7(7), 2014.

[7] G. E. P. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery , 2nd Edition*. Wiley-Interscience, 2 edition, May 2005.

[8] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.

[9] E. Cohen. On Optimizing Multiplications of Sparse Matrices. In *IPCO*, 1996.

[10] E. Cohen. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization*, 2(4), 1998.

[11] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, et al. MAD Skills: New Analysis Practices for Big Data. *VLDB*, 2(2), Aug. 2009.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] A. Edelman, S. Heller, and S. Lennart Johnsson. Index Transformation Algorithms in a Linear Algebra Framework. *IEEE Trans. Parallel Distrib. Syst.*, 5(12), Dec 1994.

[14] A. Ghoting, R. Krishnamurthy, E. Pednault, et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[15] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.*, 13(1), Jan. 1992.

[16] S. S. Godbole. On Efficient Computation of Matrix Chain Products. *IEEE Trans. Comput.*, 22(9), Sept. 1973.

[17] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. Int. Conf. Data Eng.*, 1993.

[18] F. G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3), Sept. 1978.

[19] IBM® Netezza® Analytics. *Matrix Engine Developer's Guide*. IBM, 1994.

[20] D. Kernert, F. Köhler, and W. Lehner. SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System. In *SSDBM*, 2014.

[21] H. Lee, J. Kim, S. J. Hong, and S. Lee. Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(4), Apr. 2003.

[22] G. Moerkotte. Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P. 2003.

[23] M. Muralikrishna and D. J. DeWitt. Equi-depth Multidimensional Histograms. *SIGMOD Rec.*, 17(3), June 1988.

[24] R. Rebonato and P. Jäckel. The Most General Methodology to Create a Valid Correlation Matrix for Risk Management and Option Pricing Purposes, 1999.

[25] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[26] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1), Jan. 2001.

[27] V. Yegnanarayanan. An application of matrix multiplication. *Resonance*, 18(4), 2013.