Comparison of Vector Operations of Open-Source Linear Optimization Kernels

Péter Böröcz, Péter Tar, István Maros

University of Pannonia
10. Egyetem Str., Veszprém, Hungary H-8200
Tel.: +36-88-624020
{borocz, tar, maros}@dcs.uni-pannon.hu

Optimization is a widely used field of science in many applications. Optimization problems are becoming more and more complex and difficult to solve as the new models tend to be very large. To keep up with the growing requirements the solvers need to operate faster and more accurately. An important field of optimization is linear optimization which is very widely used. It is also often the hidden computational engine behind algorithms of other fields of optimization. Since linear optimization solvers use a high amount of special linear algebraic vector operations their performance is greatly influenced by their linear algebraic kernels. These kernels shall exploit the general characteristics of large-scale linear optimization problem models as efficiently as possible. To construct more efficient linear algebraic kernels the critical implementational factors influencing operation performance were identified via performance analysis and are presented in this paper. With the results of this analysis a new kernel has been developed for the open-source linear optimization solver called Pannon Optimizer developed at the Operations Research Laboratory at the University of Pannonia. A novel application of indexed dense vectors is also introduced which is designed specifically for linear optimization solvers. Finally a computational study is performed comparing the performance of vector operations of different linear optimization kernels to validate the high efficiency of our kernel. It shows that in case of large scale operations the indexed dense vector outperforms the state-of-the-art open-source linear optimization kernels.

Keywords: Linear optimization; Simplex method; Optimization software; Computational linear algebra; Sparse data structures

1 Introduction

Nowadays, there are numerous performance-critical algorithms based on linear algebraic operations. As the efficiency of such algorithms is strongly influenced by the characteristics of the used linear algebraic kernels, the usage of appropriate data structures and their implementations are very important. In many cases it is possible to achieve better overall performance with a kernel exploiting the characteristics of

the nature of the problem. A common characteristic is sparsity, which heavily appears in the field of linear optimization (LO). The implementation of data structures used by LO solvers is not a trivial matter. Several critical factors highly influence the performance of operations.

In this paper the key implementational factors of sparse linear algebraic data structures are gathered and analyzed. Based on these a linear algebraic kernel has been created for the open-source simplex-based LO solver, Pannon Optimizer [1], developed by the Operations Research Laboratory at the University of Pannonia. A specialized indexed dense vector has also been designed and implemented to maximize performance in such applications [2]. Usage and efficiency of such vectors have not been published in the literature of linear optimization. The aim of this paper is to show that using a specialized version of indexed dense storage can lead to major performance improvements in linear optimization.

Besides the used data structures LO has many interesting computational aspects. Numerical stability can be crucial depending on the nature of the problem since the side effects of floating point numerical computations can heavily affect the performance of the solution algorithm [3]. It can also happen that numerical errors lead to cycling or stalling [4] of the algorithm which can prevent it from finding the optimal solution in reasonable time. Although these aspects play important role in the performance of LO solvers they are out of the scope of this paper since these are typically handled with high level logics.

The paper is structured as follows. The introduction briefly describes sparse computing techniques and the commonly used linear algebraic data structures of an LO solver. It also includes the standard form of the LO problem but it does not explain LO in a detailed manner as the focus of the paper is about the implementation aspects of low-level data representation and usage. The second section discusses how linear optimization solvers benefit from the sparse data structures during vector transformations and highlight the importance of these operations. The third section introduces appropriate tools for benchmarking these data structures, while section four gives an overview of the most widely used open-source LO kernels and the Pannon Optimizer. Finally, the performance of the new linear algebraic kernel and two open-source kernels are compared using the CLP [5] and GLPK [6] solvers to support our findings.

1.1 Sparse computing techniques

If z denotes the number of nonzero valued elements in an m dimensional vector v the density of v is defined as: $\rho(v) = \frac{z}{m}$. Sparsity is nothing but low density. Vectors and matrices of real-life problems submitted to LO solvers are usually very sparse. To provide high-performance data structures for LO solvers it is necessary to exploit sparsity, which is a key issue for a meaningful implementation of the revised simplex method. Stored sparse vectors generally do not explicitly keep information about every element. They only store the index-value pairs of nonzero entries. The storage of sparse vectors is much more efficient in this way but this representation

lacks the direct accessibility of elements. There are many factors influencing the performance of such data structures. As an example, element access can be speeded up by storing the pairs in a sorted order by indices. However, the initialization and insertion of a new element are slower due to the necessity of sorting (Table 1) [7].

Table 1
Operation complexity for different vector types

Operation	Dense	Sparse sorted	Sparse unsorted
Access element	$O(1)$	$O(log(z))$	$O(z)$
Initialization	$O(m)$	$O(z \cdot log(z))$	$O(z)$
Insert new element	$O(m)$	$O(log(z))$	$O(z)$

1.2 Operations of sparse linear algebraic kernels

Sparse linear algebraic kernels usually implement both dense and sparse vectors since using sparse storage techniques on vectors with high density is very inefficient. Because of this and the basic differences between dense and sparse vectors dense—dense, sparse—dense and sparse—sparse operations need to be implemented separately. The efficiency of these operations is heavily influenced by the characteristics of the vector implementations and the algorithm that use these structures. Since sparse—dense operations are faster most sparse—sparse operations are more efficient if one of the vectors is converted to dense representation and after the operation is executed the result is converted back to sparse.

Converting a sparse vector into dense representation is called scattering and converting a dense vector into sparse representation is called gathering. Scatter and gather are elementary operations of sparse linear algebraic kernels.

When a high amount of sparse—sparse operations is to be executed and the maximal dimension of the vectors is known it is more efficient to maintain a static array as working vector and use it for scattering sparse vectors. A static array means that it is a fixed dimension array allocated at the beginning of the algorithm and is kept throughout the whole solution process. This working vector needs to be cleared after every operation which can be done very efficiently if the nonzero pattern of the vector is known.

1.3 Linear optimization problems

The history of linear optimization was originated in the 1950's when Dantzig formulated the LO problem [8]. One of the formulations of the LO problem is the standard form:

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{x}, \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} = \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$; $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$.

The two main solution algorithms are the simplex method [9, 10, 11] and the interior point methods [12, 13] which give the global optimum of a linear optimization problem. In this paper, we show our results using the revised simplex method from which the main operations are highlighted in section 2. The simplex method is an iterative algorithm. The whole process is started from an initial basis and results in an optimal one if it exists, while the method iterates through a series of neighboring bases of the linear equation system. During these a large amount of linear algebraic operations must be performed. This paper is not intended to compare or evaluate different theoretical approaches of the simplex method but focuses on implementational details, which can be commonly applied for each method.

1.4 Linear algebraic data structures of LO solvers

In the past five decades the performance of LO solvers increased by 5 orders of magnitude. From this 3 orders of magnitude are due to the hardware development of computers and 2 are due to algorithmic improvements. The need for further performance improvements is still an issue, significant breakthroughs were not published in the past decade. Since the solution time of LO problems is crucial, efficient data structures can be exploited to achieve good optimization performance. Today's LO solvers generally use three different vector representations for storing operational data as they are shown in figure 1. They have different characteristics in term of operational complexity and memory requirements [7].

Dense storage

Value	2	0	0	5	0	-3

Sparse storage

Value	2	5	-3
Nonzero index	3	0	5
Dimension	6		

Indexed storage

Value	2	0	0	5	0	-3
Nonzero index	3	0	5			
Nonzero count	3					

Figure 1
The storage types used in large-scale LO solvers. Note, components of a vector are indexed $0, \ldots, m-1$.

Dense vectors are stored as simple arrays. This allows direct access to the elements and an efficient way of storage if the vector is static (e.g. an auxiliary working vector that speeds up some computations) and not very sparse. This storage is wasteful in case of sparse vectors since zeros are stored explicitly. Furthermore dense storage can cause serious memory issues if large-scale LO problems are considered.

Sparse vectors are stored as index-value pairs. In this way, elements are only accessible via searching, but the storage need is low if the vector is highly sparse. The mathematical models of LO problems are generally stored in sparse format. Since the model is nothing but a sparse matrix it can be stored as a set of sparse row or column vectors. The state-of-the-art solvers usually sacrifice some memory and store the coefficient matrix in both ways in order to enhance computational efficiency of the simplex method [3].

Indexed storage of vectors maintains dense and sparse representations of a vector in parallel [14]. This enables direct element access and exploits sparsity at the same time. However, it uses considerably more memory. Operations executed with indexed vectors are even more efficient than with sparse vectors, but changing element values is costly due to the necessity of maintaining both representations. Indexed vectors are generally used in sparse operations that involve scattering a sparse vector into a dense array.

A fourth storage type is based the indexed dense vector introduced in [2], the modified version of this vector used for linear optimization is introduced by the kernel of Pannon Optimizer which is shown in figure 2 [15]. Indexed dense vectors are similar to indexed vectors with the addition of an index pointer vector. This index pointer vector is a full-length array connecting the nonzero values of the dense representation to their copies in the sparse representation. If the element at index i is nonzero the index pointer vector has a pointer at index i pointing to the sparse representation value i. With this the complexity of changing values is reduced to constant. Indexed dense vectors generally offer better performance than traditional indexed vectors as it will be shown in section 5. There are several differences with our vector implementation and the previously published method. The differences of our indexed dense implementation is as follows:

- In our case the vector uses permanent storage capacity for each indexed dense vector while traditional methods collect the indices of nonzero pointers for temporary usage only. In our case the pointers are stored and maintained while temporary usage of these pointers have to be initialized at the beginning of each vector operation.
- To utilize index pointers the temporary storage only exploits the pointers for one operand of the vector operation.
- Traditional methods do not use index pointers to handle canceled nonzero elements. Numerically sensitive situations often generate many zero elements which shall be noticed to maintain the sparsity of the representation and the efficiency of further calculations.

An LO implementation usually uses multiple vector types to store data. Matrix $\mathbf{A}$ is represented using sparse storage for real-life large-scale problems because with dense methods it can take a huge amount of memory. For example storing a matrix with $n = m = 50000$ takes 20 GB of RAM using double precision floating point numbers which makes it impossible to handle with commonly available computing hardware. Conversely, sparse-sparse operations are not efficient on their own, because the nonzeros are usually not ordered (ordering takes computational time) and

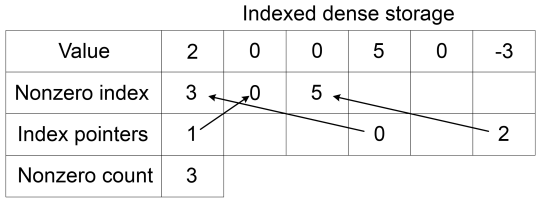

Figure 2
The indexed dense storage type. Note, components of a vector are indexed $0,\ldots,m-1$.

searching must be used to access elements (Table 1). To avoid searching working vectors containing dense arrays (dense, indexed or indexed dense) should be used to scatter the elements thus making them easily available for computation. When a series of computations should be done using the same vectors this working vector is extremely valuable and the result can be quickly gathered to a sparse format. In section 5 the impact of using different vector types as a working vector is investigated in detail.

2 Computational elements of the simplex method

In this section, the most commonly used elementary linear algebraic operations of the revised simplex method are presented. The following data structures are used in the state-of-the-art simplex based LO solvers to achieve high performance [3]. The tools supporting the identification of critical implementational factors are performance analysis tools presented in section 3.

When we deal with sparse problems the application of the revised simplex method is inevitable which uses some special representation of the basis. The two main representations are the Lower-Upper (LU) factorization [16] and the Product Form of the Inverse (PFI) [17]. Most of the solvers use the LU factorization, but it has been shown that none of them is superior because the PFI with a proper implementation can be as good as the LU [18] in several cases. The two most time consuming linear algebraic operations of the simplex algorithm are the FTRAN and BTRAN operations [3]. They involve the computation of $\mathbf{B}^{-1}\mathbf{a}$ and $\mathbf{a}^T\mathbf{B}^{-1}$, where $\mathbf{a}\in\mathbb{R}^m$, and $\mathbf{B}$ is the actual basis. In the formulas below we use the PFI form to represent the computational aspects of vector transformations but it can be adopted to the LU form as well.

FTRAN is a sequence of simple elementary transformations of the form [3]:

$$\begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} a_1+a_p\eta^1 \\ \vdots \\ a_p\eta^p \\ \vdots \\ a_m+a_p\eta^m \end{bmatrix}, \tag{1}$$

which can be written in vector form as:

$$\mathbf{c} = \mathbf{a} + \lambda \mathbf{b} \tag{2}$$

This is called a daxpy product of two vectors if double precision values are used. Since we are dealing with sparse basis representations the daxpy product is widely used on sparse vectors throughout the solution process.

BTRAN can be decomposed similarly. It is also a sequence of elementary steps, which are of the form:

$$[a_1,\ldots,a_p,\ldots,a_m]\begin{bmatrix} 1 & & \eta 1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix} = \left[a_1,\ldots,\sum_{i=1}^{m} a_i\eta^i,\ldots,a_m\right]. \tag{3}$$

It can be written in vector form as a dot product:

$$a'_p = \mathbf{a}^{\mathbf{T}}\eta \tag{4}$$

Since these computations usually take $> 50\%$ of the total solution time, the performance of them is critical and the data structures must be highly efficient.

3 Tools for performance analysis

Performance of different implementations of a given sparse linear algebraic operation can be compared by measuring the execution times on a fixed architecture. Some open-source tools exist to make such measurements. However, creating a comprehensive performance analysis is cumbersome due to the lack of tools to process and display the obtained data. As a measuring engine the Blazemark benchmark suite of the open-source Blaze library was used [19].

The Blazemark suite has several features making it a good choice to measure performance of sparse linear algebraic operations. It gives measurement results in million floating point operations per second (MFLOPS) rather than execution time making comparison between different vector sizes and the theoretical peak of the processor possible. In order to provide credible results for sparse computations as well, only the necessary floating point operations are considered. It means that the minimal number of additions that must be computed in order to get the proper result. In case of sparse-sparse addition the number of effective operations is determined by the number of nonzero elements rather than vector dimensions. Blazemark also makes measurements iteratively to filter out false results. It can be parameterized to measure operations with data structures of the desired size and sparsity.

In order to extend the functionality of the Blazemark suite and support advanced measurements and performance analysis we have created a tool called Blazemark-Analyser [15]. It is able to configure and run the Blazemark suite, parse the test results and store them in a database. It supports library-specific parameterization such

as tolerances or other simplex-related numerical settings. It is also able to execute parameter sweeps and draw a plot of performance as a function of a given parameter (e.g. dimension or sparsity). The software offers a graphical user interface and can be used conveniently to interpret and compare the results. This software makes it possible to identify how different implementational factors influence operational performance. With the help of performance analysis a new linear algebraic kernel was created for Pannon Optimizer.

4 Kernels of open source LO solvers

There are several open-source LO solvers from which only a few is capable of solving large-scale LO problems. The linear algebraic kernels used in them need to fulfil the requirements of the LO solver and offer the best possible performance. This section describes the main characteristics of the kernels of two of the most widely used open-source LO solvers GLPK [20], CLP [21] and the Pannon Optimizer.

4.1 GNU Linear Programming Kit

The GNU Linear Programming Kit (GLPK) is a collection of ANSI C implementations of mathematical optimization solvers including linear optimization [6]. The GLPK kernel uses computer memory very efficiently. It allocates blocks and stores vectors in a way to minimize the caching operations of the processor. Just as other solver algorithms it implements the revised simplex method. GLPK has a lightweight implementation of dense, sparse and indexed vector representations with minimal overhead. All vector types consist of only the necessary arrays and the vector operations are also implemented without any overhead of sophisticated implementations. This implies that it does not pay particular attention to numerical stability at operational level.

4.2 COIN-OR Linear Program solver

The COIN-OR Linear Program solver (CLP) is an open-source large-scale optimization problem solver written in C++ [5]. It includes an object-oriented implementation of the revised simplex method. The linear algebraic kernel of CLP offers three vector types for the solver: dense, sparse and indexed representations. Dense vectors are implemented traditionally using arrays. The sparse vector representation is sorted by index and only used to store the mathematical model. Indexed vector representation is used during all sparse operations. The CLP kernel is capable of mitigating the negative effects of numerical problems by setting elementary operation results below a given threshold to a pre-determined tiny value. It should be noted that throughout the solution process of CLP it overrides the default behavior of its own kernel with more efficient array operations (similar to GLPK).

4.3 Pannon Optimizer

Pannon Optimizer [1] is a large-scale LO solver using a high-level C++11 implementation of the revised simplex method [22]. It is being developed specifically for research purposes, making the performance impact of subalgorithms measurable. The linear algebraic kernel of Pannon Optimizer was developed considering the results of performance analysis with BlazemarkAnalyzer. It implements dense and sparse vector representations as well as the indexed dense vector which is a uniquely extended implementation of the indexed vector.

5 Computational study

This section presents the summarized results of a computational study on the linear algebraic kernels of the solvers mentioned above. The testing environment was a laptop computer with an Intel Core i5-3230M CPU with fixed clock speed at 2.60GHz, 3MB L3 cache and AVX (Advanced Vector Extensions) support, with 4 Gb DDR3 RAM. The operating system was an Ubuntu 14.04 64-bit system.

The theoretical peak performance of this system is 10400 MFLOPS with 4 double precision floating point operations per clock cycle (2 additions and 2 multiplications). BlazemarkAnalyzer was used for the measurements and it also provided the diagrams that we present in this section. All the figures show the results normalized according to the maximal measured MFLOPS value on the Y axis of the diagrams. The X axis showing the dimensions of the vectors uses a logarithmic scale.

The test vectors used in our performance analysis were composed in a way to mimic the structure of real LP problems. In order to achieve this we have used different vector patterns throughout the measurements. In case of vector additions 70% of the operations were standard additions where the operands and result are nonzero. 10% of the operations were cancellations, where the operands are nonzero but the result is numerically zero. The last 20% were non-overlapping values, thus one of the operands of these operations was zero. In case of dot product operations the vectors did not have the same number of nonzero elements. During the measurements of dot product operations $\mathbf{a}^\mathbf{T}\mathbf{b}$ was always performed together with $\mathbf{b}^\mathbf{T}\mathbf{a}$. This measures the effect of traversing different nonzero patterns.

In the case of dense operations such as the dense—dense vector addition (Figure 3) or the dense—dense vector daxpy product (Figure 4) low-level memory management can result in significantly better performance. This means that simple array implementations are the best for dense—dense operations.

In the case of sparse—sparse vector dot products kernels with sophisticated memory management perform significantly better (Figure 5). Since this operation does not change values in either vectors and the dot product implementation is based on the nonzero indices, the difference between using a sparse, an indexed vector or an indexed dense vector is negligible. This is one of the most performance-critical operations of LO solvers since it forms the foundation of an elementary step of the

BTRAN operation. The results prove that with the application of performance analysis very high performance of linear algebraic operations can be achieved. When it comes to large-scale simplex specific operations the performance of Pannon Optimizer kernel exceeds the other two kernels which it has been compared to.

For sparse operations that result in the change of values a working vector having a dense representation is advisable since during the FTRAN operation multiple daxpy products are to be done using the same vector. The performance of such operations can be significantly increased with the use of indexed dense vector instead of regular indexed vector (Figure 6). The extent of this improvement depends on the number of newly created zero elements. Changing a nonzero element to zero in an indexed vector has logarithmic or linear complexity depending on whether the vector is sorted or not, while in an indexed dense vector it has constant complexity. In the case of operations with very small vectors in dimension lightweight kernels with minimized overheads (additional computing only needed for memory management) perform better.

It is not trivial whether the traditional indexed vector or the new indexed dense vector would perform better as a static operation vector for LO solvers. To examine this issue, numerous measurements were made to compare the indexed dense vector of Pannon Optimizer with regular indexed vector implementations used in CLP and GLPK. When adding a sparse vector to an indexed or indexed dense vector with values of opposite sign where the result is expected to be a null-vector, the operation performance of regular indexed vectors reduces logarithmically with the growth of the dimension of the vectors. In the case of indexed dense vectors operation performance does not reduce (Figure 7). This further emphasizes the efficiency of the indexed dense vector.

When comparing the performance of traditional indexed vectors with indexed dense vectors on the dot product operation with dense vectors the results show that when working with small vectors regular indexed vectors are advisable to be used but in the case of large vectors the indexed dense vector performs significantly better (Figure 8). These characteristics prove that the indexed dense vector is a very efficient working vector of large-scale LO solvers.

Conclusions

The linear algebraic kernel of the Pannon Optimizer was developed, based on results of the performance analysis of sparse data structures and with consideration of computationally heavy simplex-specific operations such as FTRAN and BTRAN. This investigation led us to develop and release a high performance sparse linear algebraic kernel that performs better than its predecessors for solving linear optimization problems.

We have introduced a new kind of indexed vector based on the experiences that we gathered from its alternatives. It appears that our vector type performs very well as a static working vector of large-scale LO solvers, surpassing the performance of traditional indexed vector implementations that most LO solvers use. When working with large vectors, both sparse and dense operations are faster with the new vector

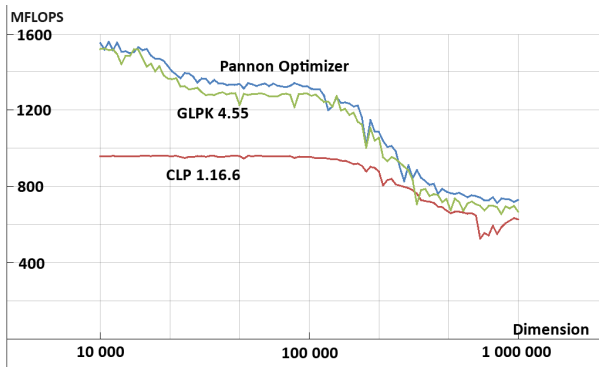

Figure 3
Performance of different implementations of the addition of two dense vectors.

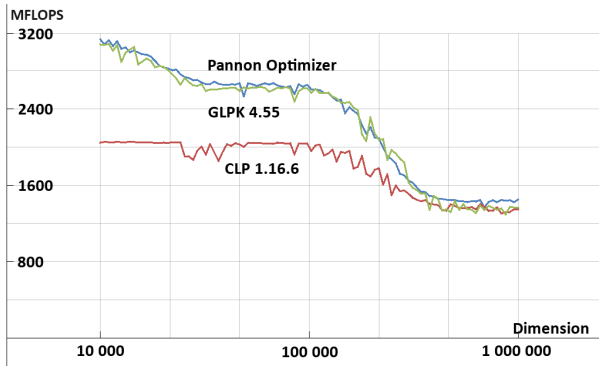

Figure 4
Performance of different implementations of the daxpy product of two dense vectors.

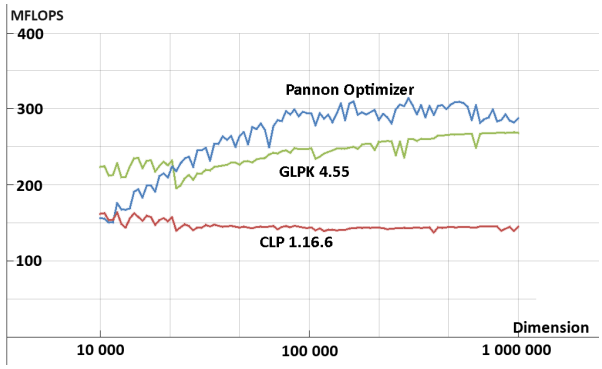

Figure 5
Performance of different implementations of the dot product of two sparse vectors (0.1% density).

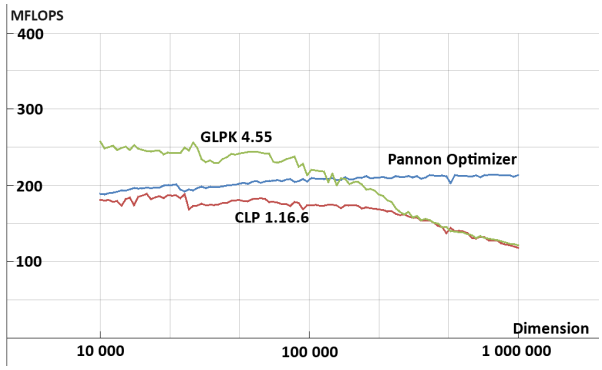

Figure 6
Performance of different implementations of the daxpy product of two sparse vectors (0.1% density) using an indexed (CLP, GLPK) or an indexed dense (Pannon Optimzier) static working vector.

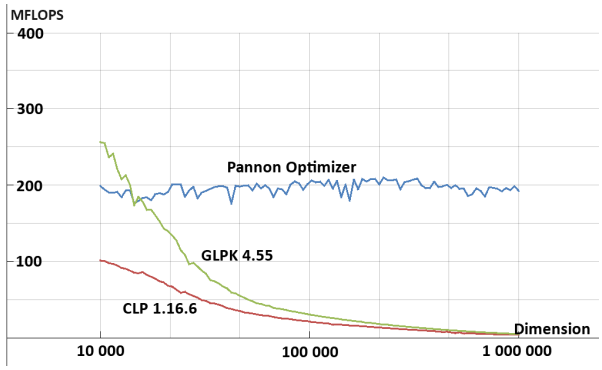

Figure 7
Performance of the addition of a traditional indexed vector (CLP, GLPK) or an indexed dense vector (Pannon Optimizer) (0.1% density) and a sparse vector (0.1% density) using a nonzero pattern where the result of the addition is algebraically a null-vector.

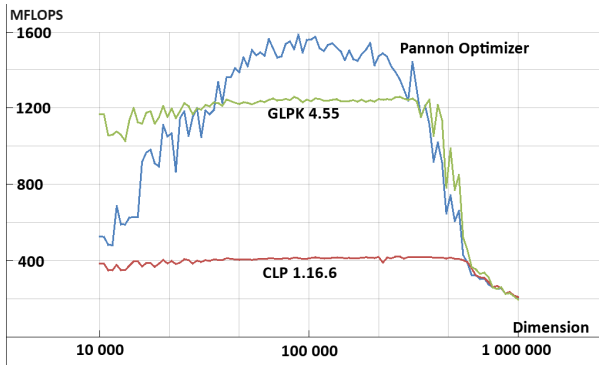

Figure 8
The performance of the dot product operation of a traditional indexed vector (CLP, GLPK) or an indexed dense vector (Pannon Optimizer) (0.1% density) and a dense vector.

type that has been validated by performance analysis. As a conclusion of the performance analysis, we highly recommend the usage of the indexed dense vector if the dimension of the vectors is greater than 10^4. It can also be noted that the usage of the indexed dense vector instead of other vector types does not affect the performance negatively if used as a static working vector. Altogether, the overall performance of the linear algebraic kernel of the Pannon Optimizer seems to be better than kernels of other open-source, large-scale LO solvers.

Acknowledgement

This publication/research has been supported by the European Union and Hungary and co-financed by the European Social Fund through the project TÁMOP-4.2.2.C-11/1/KONV-2012-0004 - National Research Center for Development and Market Introduction of Advanced Information and Communication Technologies.

References

[1] University of Pannonia: Pannon Optimizer, http://sourceforge.net/projects/pannonoptimizer/, Online, 2017.09.

[2] S. Pissanetzky: Sparse Matrix Technology, Academic Press, 1986.

[3] I. Maros: Computational Techniques of the Simplex Method, Kluwer Academic Publishers, Norwell, Massachusetts, 2003.

[4] M. Padberg: Linear Optimization and Extensions, Springer, Berlin, 1999.

[5] COIN-OR Project: COIN-OR Linear Optimization Solver, https://projects.coin-or.org/Clp, Online, 2017.09.

[6] GNU Project: GNU Linear Programming Kit, https://www.gnu.org/software/glpk/, Online, 2017.09.

[7] I. Maros: Essentials of Computational Linear Algebra, Department of Computer Science, Veszprém, 2009.

[8] G. B. Dantzig: Maximization of a Linear Function of Variables Subject to Linear Inequalities In T.C. Koopmans, editor, *Activity analysis of production and allocation*, pages 339–347. Wiley, 1951.

[9] G. B. Dantzig: Linear Programming and Extensions, Princeton University Press, 1963.

[10] K. G. Murty: Linear and Combinatorial Programming, John Wiley & Sons, 1976.

[11] A. Prékopa: A Very Short Introduction to Linear Programming, RUTCOR Lecture notes, pages 2–92, 1992.

[12] C. Roos, T. Terlaky, and J-Ph Vial: Theory and Algorithms for Linear Optimization: An Interior Point Approach, John Wiley & Sons, 1997.

[13] T. Illés and T. Terlaky: Pivot Versus Interior Point Methods: Pros and Cons, European Journal of Operations Research, 140:6–26, 2002.

[14] A. Koberstein: Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation, Computational Optimization and Applications, 41(2):185–204, 2008.

[15] P. Böröcz, P. Tar, and I. Maros: Performance Analysis of Sparse Data Structure Implementations, MACRo 2015, 1(1):283–292, 2015.

[16] H. M. Markowitz: The Elimination Form of the Inverse and Its Application to Linear Programming, Management Science, 3(3):255–269, 1957.

[17] G.B. Dantzig and Wm. Orchard-Hays: The Product Form for the Inverse in the Simplex Method, Mathematical Tables and Other Aids to Computation, 8(46):64–67, 1954.

[18] P. Tar and I. Maros: Product Form of the Inverse Revisited, OpenAccess Series in Informatics (OASIcs), 22:64–74, 2012.

[19] K. Iglberger, G. Hager, J. Treibig, and U. Rüde: Expression Templates Revisited: A Performance Analysis of the Current ET Methodology, SIAM Journal on Scientific Computing, 34(2):42–69, 2012.

[20] J. L. Gearhart, K. L. Adair, R. J. Detry, J. D. Durfee, K. A. Jones, and N. Martin: Comparison of Open-Source Linear Programming Solvers, Technical report, 2013.

[21] B. Meindl and M. Templ: Analysis of Commercial and Free and Open Source Solvers for the Cell Suppression Problem, Transactions on Data Privacy, 6:147–159, 2013.

[22] B. Stágel, P. Tar, and I. Maros: The Pannon Optimizer - A Linear Programming Solver for Research Purposes, Proceedings of the 5th International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics, 1(1):293–301, 2015.