Available online at www.sciencedirect.com
ScienceDirect
Transactions of A. Razmadze Mathematical Institute ▮ (▮▮▮▮) ▮▮▮–▮▮▮
Transactions of A. Razmadze Mathematical Institute
www.elsevier.com/locate/trmi

Original Article

Jagged non-zero submatrix data structure

Giga Chalauri[a], Vakhtang Laluashvili[b], Koba Gelashvili[c,*]

[a] *LUXOFT Poland, Regular Java developer, Krakowska 280, 32-080 Zabierzów, Krakow, Poland*
[b] *Public Service Development Agency, Junior Programmer, 67a Tsereteli Avenue, 0154, Tbilisi, Georgia*
[c] *Department of Computer Science, Tbilisi State University, TSU Building/Block 11, 13 University Str., 0186 Tbilisi, Georgia*

Received 21 June 2017; received in revised form 30 September 2017; accepted 2 October 2017
Available online xxxx

Abstract

On the basis of C language matrix having rows of different length, we have developed a new storage format for rectangular matrix. It stores non-zero entries, their column indices and is called jagged non-zero sub-matrix data structure or simply *jnz-format*.

In case of simple applications, when the only requirement from the format is to ensure the serial algorithm of multiplying matrix by vector (e.g. conjugate gradient (CG) method), two following issues are experimentally studied:

- For what amount of zero-entries do we accept the rectangular matrix as sparse, with respect to used memory and speed;
- What should the *jnz-format*'s interface look like.

Determining the interface is comparatively laborious; *jnz-format* is compared to two approved formats—CRS and *Mapped Matrix*. In comparisons, CRS format is considered by using two different implementations, whilst *jnz* and *Mapped Matrix* —by using one. In comparisons, we use *jnz* and CRS formats with our own simple interface implementations and CRS and *Mapped Matrix* with *boost*'s library interfaces and implementations. Experiments' results show *jnz* format's prospect and visible advantage of the relatively easy interface.

All the material regarding experiments can be seen at https://github.com/vakho10/Sparse-Storage-Formats.

Keywords: Sparse matrix; Mapped matrix; Compressed matrix; Compressed sparse row; Java sparse array; jnz-format; GitHub; Boost library; Conjugate gradient method

1. Introduction

Sparse matrices often arise in real-world applications. Matrices, connected with graphs and partial differential equations, always contain a certain number of zero entries. Intensive research of sparse matrices have been performed since 1970s. So far several data structures—storage formats have been introduced. Storage formats are developed

* Corresponding author.
E-mail addresses: giga.chalauri@gmail.com (G. Chalauri), vakho10@gmail.com (V. Laluashvili), koba.gelashvili@tsu.ge (K. Gelashvili).
Peer review under responsibility of Journal Transactions of A. Razmadze Mathematical Institute.

https://doi.org/10.1016/j.trmi.2017.10.002

either for the situations, when sparseness is detected as a pattern of some systematic model (for example, three diagonal or five diagonal matrices), or arrangement of non-zero entries is not subjected to any regularity. In the present work only the second case, which is more complicated, is considered.

From processed formats of sparse matrices some (the most reliable and fastest ones) are implemented in the libraries of modern programming languages (see [1]). One of the most effective and widespread libraries is *boost* (see [2]), well-known scientific external library of C++ language. To benchmark jnz-format, two of the fastest formats of *boost, Mapped Matrix* and *Compressed Matrix* were used. They will be described briefly in the next section.

jnz-format principally differs from these two formats, it uses the abilities of modern programming languages, related to dynamic creation of one-dimensional arrays and jagged two-dimensional arrays. Ideologically, it is generalization of *Ellpack-Itpack* format (see [3]) which was efficient in cases, when maximum number of non-zero entries per rows was known beforehand. In such case, instead of source rectangular matrix, it is possible to consider two relatively smaller rectangular matrices, one composed by non-zero entries of source matrix, and another—by column indices of non-zero entries (of source matrix).

jnz-format is very close to the other generalization (see [4]) of *Ellpack-Itpack* format, which is known as Java Sparse Matrix, representing itself by two two-dimensional jagged arrays, which are received by deleting zeros and their indices from *Ellpack-Itpack* format. In [4] and in other works of the same authors, some tests proving effectiveness of Java Sparse Matrix are conducted. But, it should be noticed, that in calculating needed amount of memory they do not take into account some factors.

Section 2 is devoted to storage formats of sparse matrix of our interest. Strong and weak sides of some formats, including jnz-format are briefly described. It is shown that *jnz-format* is best-suited to matrix operations (matrix–vector multiplication and swapping rows, which are widely spread in parallel implementations of algebraic algorithms).

Section 3 is devoted to determination of the percentage of zeros in dense matrix used in CG-algorithm, which identifies matrix as sparse in case of using *jnz-format*. The problem is interesting, because the "universal" description of the sparse matrix does not exist and the sparseness looks like as dependent from the application, sparse format and its implementation. It turned out, that in case of one third (33%) of zero-entries using of the *jnz-format* instead of usual rectangular form saves memory and accelerates the process of solution with CG-method (providing that real and integer numbers are stored in primitive types *double* and *int*). If the interface and implementation of sparse format are not suited to CG, then the requirements to the memory/speed become more strict. From the materials uploaded on GitHub, this section is included in JNZvsDense project.

To benchmark efficiency of *jnz-format* in real applications, which only use multiplication of a matrix by vector, and to determine its interface, we have chosen the CG-method. It is effective in solving $Ax = b$ systems, where A is symmetric and positively defined sparse matrix. The algorithm is very simple, so substitution of one format by another in implementations is simple. Another reason why we chose CG-method is that for symmetric sparse matrices we can use *jnz-format* in more economical way, saving only diagonal and non-zero entries of upper triangular matrix. In our case, CG-method represents only the tool for comparison of different formats of sparse matrices, so we are not trying to program more sophisticated and faster variants (taking into account preconditioning and parallelism). To the contrary, we accept the simplest code, in order to focus on data structures.

In Section 4, the results of the usage of three different sparse matrix formats and two interfaces are investigated. 85 matrices, taken from the sparse matrix collection of University of Florida (see [5]) with randomly generated right-hand sides (for each matrix) serves us as tests. $Ax = b$ systems with these data are solved. To present results, the well known methodology of benchmarking optimization software [6] is used. The results of the numerical experiments show that formats *jnz* and CRS, which have simplest interface implemented in the C-style, have practically the same speed (in most cases CRS is faster). Whilst, in comparison to CRS and *Mapped Matrix* formats, which are implemented in *boost* library, it is much more efficient. This states the prospect of new format and the obvious advantage of interface adjusted for CG compared to the overloaded interfaces of *boost* library.

In the materials uploaded on GitHub, this section is included in SparseProject project. It evaluates effectiveness of *jnz-format*. The project itself consists of three sub-projects: SparseLib, SparseMatrixProject and UnitTests.

The SparseLib sub-project contains all the necessary classes and functions that we use in two other sub-projects. The SparseMatrixProject is the main sub-project, which, by using classes and functions from SparseLib project, evaluates specific tests to compare sparse formats. The last sub-project, UntiTests consists of the unit tests that are used in development to make sure that everything works correctly after the necessary changes we have made in code.

Please cite this article in press as: G. Chalauri, et al., Jagged non-zero submatrix data structure, Transactions of A. Razmadze Mathematical Institute (2017), https://doi.org/10.1016/j.trmi.2017.10.002.