Jan WICIJOWSKI

QUICK OFFLINE SPARSE MATRICES*)

ABSTRACT *When dealing with large datasets, computer memory constraints are a common problem. With the volumes of data exceeding 1 GiB of size, storage of the whole datasets in RAM becomes infeasible. Since in most applications one deals with only a portion of dataset at a time, the rest may be kept offline on non-volatile memory that provides larger capacities. The access to non-volatile memory is typically a few orders of magnitude slower than of RAM, so an efficient method of storage should be proposed to keep the number of disc accesses count as small as possible. In the paper I describe the offline storage of sparse matrices that is built on top of Hierarchical Data Format (precisely, on the latest revision – HDF5) addressing the problem of matrix-vector multiplication.*

Keywords: *Sparse matrix, multiplication, offline storage, huge datasets*

1. PROBLEM STATEMENT

Large sparse matrices appear in numerous fields, i.e. statistical linguistics, mechanical engineering, data mining. There are numerous tools available to process the sparse matrices in memory. Most operations on such data come from linear algebra, with matrix-vector and matrix-matrix multiplication being the most common. As long as there is sufficient memory in the system to hold

*) This work was supported by MNISW grant OR00001905.

Jan WICIJOWSKI, M.Sc., Eng.
e-mail: wici@agh.edu.pl

Department of Electronics
AGH University of Science and Technology

both operands and results, direct RAM calculations are the fastest one can get on selected machine with the algorithm of choice. The problem arises when the amount of memory necessary to store the data exceeds the capacity of RAM available to the computing process, as all popular libraries require complete sets to be available at hand. The sets may be divided manually and loaded incrementally, but the technique requires careful crafting for each individual case. Another option is to resort to the operating system, which may give the process additional virtual memory that is not physically present on the machine. In the process which is called paging, the memory which is not currently in use is then stored on auxiliary storage (most often, a hard drive).

Leading a whole system to the point, where all physical memory is occupied by calculation and the additional data is swapped can be considered catastrophic from the point of view of performance. The whole operating system is busy moving data from and to hard drive, which has been called “thrashing” from the early days of computing. Moreover, the virtual memory has its limits as well – it seldom exceeds the amount of RAM by an order of magnitude. On high performance computing (HPC) machines, a user may be given only limited amount of RAM per job and no swapping possibility at all.

Matrix multiplication by vector algorithm can be formulated is such a way to maximize memory locality. The data kept in memory can be minimal, in the simplest formulation only one row from the matrix, an operand vector and result vector can be stored. If the number of input/output (IO) operations are drawn to minimum, limited to fetching the operands only one can expect best performance available for given hardware. In the next section I will enumerate sparse matrices storage methods, which are applicable to both in-memory and offline storage. Section 3 presents file formats used for serializing sparse matrices to disk. In section 4 I discuss offline storage possibility based on popular HDF5 library. Section 5 describes benchmarks on comparison of matrix-vector multiplication time of the presented formats with direct RAM multiplication and relational database query on PC and HPC cluster.

2. SPARSE MATRICES STORAGE OPTIONS

The most naive approach to optimize the sparse format storage is to organize the nonzero cells from sparse matrix in triplets: (row index, column index, value). This format will be further referred to as COO (from COOrdinate format). Note that this format does not specify the ordering of the elements, and the same cell can

appear in the format more than once[1]. This format is rapidly constructed and can be then converted to other formats.

To enhance COO format is to introduce ordering on one or both row and column subscripts. A format with both subscripts sorted will be further referred to COS (COordinate Sorted). Assuming that no preordering on matrices is set, the average time of conversion between COO and COS format is $O(k^2 m^{-1} \log k \log (k/m))$, constrained by sorting time.

There is a straightforward modification of COS format, which eliminates logarithmic lookup behaviour from the first column. The idea is to store the 2nd and the 3rd column as in COS format, and replace the 1st column with separate table of indices pointing at the beginning of subsequent rows. This is roughly equivalent to C language representation of an array of pointers to contiguous block of memory with subscripts/values structures. The formats in the family of formats are called CSC and CSR (Compressed Sparse Column/Row) and they differ only by transposition.

A simple illustration of the formats storage for 5×10 matrix with 6 nonzero elements is presented in the following subsections. The complexities of lookup times are gathered in Table 1. All the formats are further explained in the following subsections.

TABLE 1
Asymptotic complexities of chosen factors of different matrix formats

	Dense	**COO**	**COS**	**CSC**	**CSR**
Memory	$\Theta(mn)$	$\Theta(k)$	$\Theta(k)$	$\Theta(2k+m)$	$\Theta(2k+n)$
Element access (average)	$\Theta(1)$	$\Theta(k)$	$\Theta(\log k \log (k/m))$	$\Theta(\log (k/n))$	$\Theta(\log (k/m))$
Row access (average)	$\Theta(1)$	$\Theta(k)$	$\Theta(\log k)$ or $\Theta(k)$	$\Theta(m \log (k/n))$	$\Theta(1)$
Column access (average)	$\Theta(1)$	$\Theta(k)$		$\Theta(1)$	$\Theta(n \log (k/m))$

2.1. Concept

The exemplary matrix that would be analyzed through the article is sketched in Table 2. The row and column indices start at zeros, as in C. The data format are plain integers. In the rest of the article such naming convention will be used:

n – number of columns,
m – number of rows,

[1] It is up to the mathematics library on how to process it. Scipy, for example, sums all duplicate cells.

k – number of nonzero elements,
$O(\cdot)$ – Landau notation of maximal asymptotic complexity,
$\Theta(\cdot)$ – Landau notation of equivalent asymptotic complexity.

TABLE 2
Exemplary matrix without storage scheme

	0	1	2	3	4	5	6	7	8	9
0	1				2					
1			3							
2										
3					4			5		
4										9

2.2. Dense

In small examples and rapid prototyping, the sparse matrix can be represented in memory as a dense format, storing all zero values alongside the nonzero ones. Thus, the matrix would be viewed as contiguous block in memory or disk. This allows for rapid indexing in constant time, as the memory/disk offset is calculated directly from column and row indices. The disadvantages are clear:

- one has to store $\Theta(nm)$ cells for all the zeros, largely limiting the upper bounds on matrix size,
- the nonzero values can be recognized only by scanning and direct comparison – their location is not known before the operations.

The matrix is shown in Table 3.

TABLE 3
Exemplary matrix stored in dense format

1	0	0	0	2	0	0	0	0	0
0	0	3	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	4	0	0	5	0	0
0	0	0	0	0	0	0	0	0	9

2.3. Coordinate format

The coordinate format (COO) is the most straightforward way of representing the sparse matrix as triplets: (row index, column index, cell value). It has the

advantage of storage size requirements as high as $\Theta(k)$. It can also be built incrementally, which is beneficial for interfacing with such tools as mesh generators in mechanical analysis and deserializers of any kind. The simplest approach to COO does not enforce any sorting on the row/column subscripts, so the creation time is $\Theta(k)$. However, the average indexing time is also $\Theta(k)$, because of the need to iterate the whole matrix before successful match on the indices. The exemplary matrix in this scheme is depicted in Table 4.

TABLE 4
Exemplary matrix in coordinate format

row index	0	1	0	4	3	3
column index	4	2	0	9	7	4
value	2	3	1	6	5	4

2.4. Coordinate format with indices

The row and column subscripts, or the combination of both, can be associated with external B-tree indices, leading to logarithmic time on the lookup. Note that such representation is the most straightforward way of encoding sparse matrix in relational database management systems (RDBMS). Entities sufficient for efficient representation are the triplets table and the indices – they are both primitive types in RDBMS, so they are usually aggressively optimized w.r.t. the speed and disk utilization by the libraries authors.

The depiction of such auxiliary indexing does not fit the scope of this article.

The format will be referred to as COI, and depending on the indexing scheme used, it can achieve average indexing times $\Theta(\log n)$ on columns, $\Theta(\log m)$ on rows and $\Theta(\log m + \log n)$ on cells. Nevertheless, each index is an additional constraint on memory and may need longsome rebuild operations on table modifications. The precise complexities are not listed here, as they largely depend on the choice of database engine.

2.5. Coordinate format sorted

An alternative reiteration of COO format is to enforce sorting on row and column subscripts in any order (in the example the primary sort is on rows). Starting from an unsorted COO, full sorting time is $O(k^2 m^{-1} \log k \log (k/m))$,

which is tolerable, as it is mostly done once for a given problem. The coordinate format sorted (COS) shares its advantages with COO. Moreover, it has the following benefits:

- retrieving an entire row takes $O(\log k)$,
- retrieving a single element takes $O(\log k \log (k/m))$,.

The logarithmic speedup can be implemented i.e. by bisection algorithm and does not require any additional storage. Unfortunately, the performance can be compromised by the number of disk accesses needed to reach an appropriate row. For the illustration, see Table 5.

TABLE 5
Exemplary matrix in coordinate format sorted

row index	0	0	1	3	3	4
column index	0	4	2	4	7	9
value	1	2	3	4	5	6

2.6. Compressed row format

The compressed row format (CSR) is a direct optimization of COS format. At the same time, it removes logarithmic lookup time on rows and may reduce the space (depending on k/n factor). It is achieved by replacing the table with row indices with a separate table, which holds the offsets of (column subscripts, values) tuples, which itself is indexed by row indices. It introduces a level of indirection, as the arrays are stored separately, but the row indexing is now $\Theta(1)$. As for the (column subscripts, values) tuples are still sorted by columns, the average cell retrieval becomes logarithmic w.r.t. k/m. The representation example is shown in Table 6.

TABLE 6
Exemplary matrix in compressed row format. The arrays in parentheses are pure memory offsets and are not stored. They are shown here as they are key features to the functionality

Row array for tuple table indexing

(row index)	0	1	2	3	4	5	(end marker)
tuple table offset	0	2	3	3	5	6	(end)

Column/value tuple table

(offset)	0	1	2	3	4	5	6 (end)
column index	0	4	2	4	7	9	
value	1	2	3	4	5	6	

2.7. Compressed column format

The compressed column format (CSC) is nothing more than a transposition of the CSR format. The exemplary matrix is illustrated in Table 7. Note that in the example $k < n$, so the memory requirements are bigger than those for an equivalent for COO representation.

TABLE 7
Exemplary matrix in compressed column format. The arrays in parentheses are pure memory offsets and are not stored. They are shown here as they are key features to the functionality

Column array for tuple table indexing

(column index)	0	1	2	3	4	5	6	7	8	9	10	(end marker)
tuple table offset	0	1	1	2	2	4	4	4	5	5	6	(end)

Row/value tuple table

(offset)	0	1	2	3	4	5	6 (end)
row index	0	1	0	3	3	4	
value	1	3	2	4	5	6	

The nomenclature for the arrays differs with each implementation. The popular ones use names as described in Table 8.

TABLE 8
Compressed matrix nomenclature as used in popular storage matrix (CSC flavour)

Format	tuple table offset	col index	cell value
Matlab V7.3	ir	jr	data
Scipy	indptr	indices	data
Harwell-Boeing	colptr	rowind	values
SparseLib++ [1]	colptr	col_ind	val

3. POPULAR FORMATS

The common storage formats of sparse matrices include Harwell-Boeing and Matrix Market files from open formats and Matlab `.mat` files as the most popular propertiary one.

The Harwell-Boeing and Matrix Market files are plain text files with numbers formatted as a text compatible with FORTRAN I/O. Plain ASCII encoding accorded for its portability across systems for many years. Harwell-Boeing format is practically a CSC matrix, while Matrix Market is COO format. Matlab `.mat` files of versions ≤ V7.0 store their sparse matrix files in binary CSC format. Matlab `.mat` files of versions ≥ V7.3 store all matrices in HDF5 format.

4. OFFLINE SPARSE MATRICES USING HDF5

HDF5 format is a fast, portable and robust file format for storing arrays of numeric data. For two decades it has become a de-facto standard in numerous fields of engineering, including astronomy, geospatial data, meteorology and biochemistry [2, 3]. In recent versions of Matlab (from 7.3 or 2006b upwards), it has become the optional underlying storage format for `.mat` files [4], as the dense arrays are easily represented in HDF5, and sparse matrices can be stored using any of described schemes. Still, for Matlab, the format is meant for serialization/deserialization only; when one wants to perform calculations on the data, it needs to be loaded into the memory as a whole.

Due to the demanding needs of storing and retrieving huge datasets, a format based on HDF5 is proposed. For the underlying storage it uses plain tables of 64-bit unsigned integers for the matrix subscripts and 64-bit IEEE floating point values for matrix cell values. The matrix is stored in CSC format as described in subsections 2.5 and 2.6.

5. BENCHMARK

The proposed approach of offline representation of sparse matrices on HDF5 format was compared with an alternative method of storage – a relational database with B-tree indices on row and column subscripts. Direct calculations in RAM are conducted along those two schemes, serving as a reference.

Sparse matrices were generated pseudo-randomly, to have uniform distribution of nonzero elements along the matrix. Tests were conducted on two sets of matrices – very dense ones, with density about 1/2, and relatively sparse ones, with about 1 nonzero element in 1000.

TABLE 9
Sizes of test matrices used in benchmark

Dense sets			Sparse sets		
k	m,n	$k/(mn)$	k	m,n	$k/(mn)$
10000	140	0.51	10000	3260	0.00094
30000	245	0.50	30000	5470	0.001
100000	447	0.50	100000	10000	0.001
300000	775	0.50	300000	17320	0.001
1000000	1400	0.51	1000000	31600	0.001
3000000	2450	0.50	3000000	54700	0.001
10000000	4470	0.50	10000000	100000	0.001
77000000	12400	0.50	77000000	277500	0.001

Each matrix from the set was converted to the following formats:

- Sqlite3 table with three columns as COO format. Two database indices were created on matrix row and matrix column columns.
- HDF5 file as described in Section 4.
- Matlab V7.0 .mat file to store the sparse matrix for easy deserialization into RAM.

A batch of multiplications was performed as follows. A sparse vector was generated by randomly selecting indices from the range of columns of the matrix with uniform distribution. Its entries were filled with random floating-point values from the [0,1) range[2]. The number of the nonzero elements of the vector was selected to be 10 and 100. For the smallest matrices, the number was trimmed to the matrix columns count. The sparse vector operand is kept in memory as a hashtable, to provide $O(1)$ time lookup on its elements. The matrices were multiplied by the vector in the following way:

- For Sqlite3 table, a “SELECT” query is performed to pick only the triplets subject to multiplication by nonzero values. The result vector is first created, and then populated with incremental results. The triplets are fetched in arbitrary order, then multiplied by the vector elements and added to the result. The result is a dense numerical array.
- For sparse matrix built upon HDF5 tables only the columns subject to multiplication were chosen on the base of `indptr` table. The result vector is first created, and then populated with incremental results. The columns from the matrix are fetched in the ascending order of nonzero entries subscripts of the operand vector. For each column, the `indptr` table is queried once, then column data is fetched from

[2] Which is not very relevant for the algorithm benchmark.

tuple table. In the best case, the disk access count can be as low as twice the number of nonzero vector elements.

- For the direct RAM calculations I resort to Scipy sparse matrix routines, which are low-level C++ functions.

The software was modelled in Python, which serves as a glue language between high performance libraries written in C, due to the rapid speed of prototyping and easiness of testing. Unfortunately, the overhead introduced by high level wrappers over the libraries can be significant. The resulting code runs proportionally slower than equivalent low-level software. Even though, the researchers show, that the slowdown factor is from the range of (1,20), it is largely dependent on a given problem [5, 6]. The profiling tools show marginal overhead of Python runtime with respect to IO and compiled-in C routines.

The tests were run on modern PC with 4 GiB of memory on Intel(R) Core(TM)2 Quad CPU Q9400 with 2.67 GHz as a 32-bit processes and on Intel Server with four Intel(R) Xeon(R) CPU X5550 running 2.67 GHz as 64-bit processes. The partitions were of ext4 type in both cases. There were no attempts to optimize the code beyond default setup; single core was used, no cache optimization techniques were utilized. The gathered times are taken for the sole multiplication time, including neither the loading of the datasets and libraries nor the vector generation. There were 50 multiplications for each size of vector and matrix, then the average of each batch was taken.

The results of the benchmark are presented in Figures 1, 2, 3 and 4.

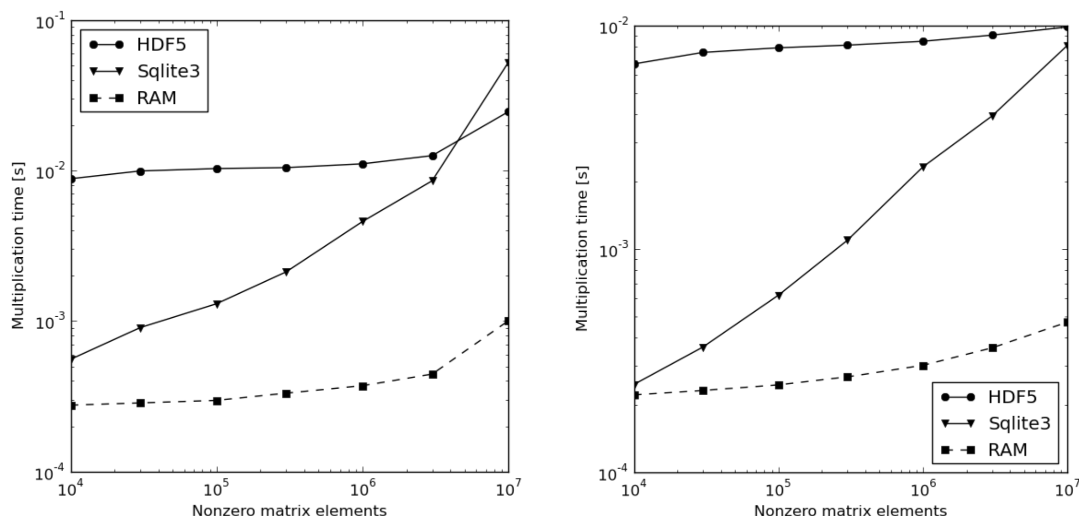

Fig. 1. Sparse matrix-vector multiplication time against sizes of matrices. Nonzero elements of the vector are as high as 10, average density is about 0.001. Left: PC, Right: Server

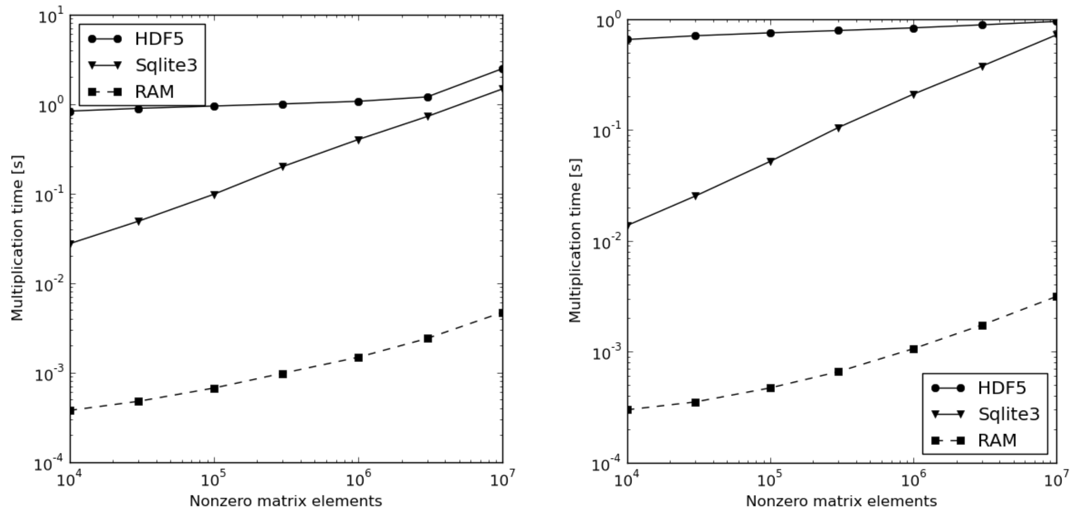

Fig. 2. Sparse matrix-vector multiplication time against sizes of matrices. Nonzero elements of the vector are as high as 1000, average density is about 0.001. Left: PC, Right: Server

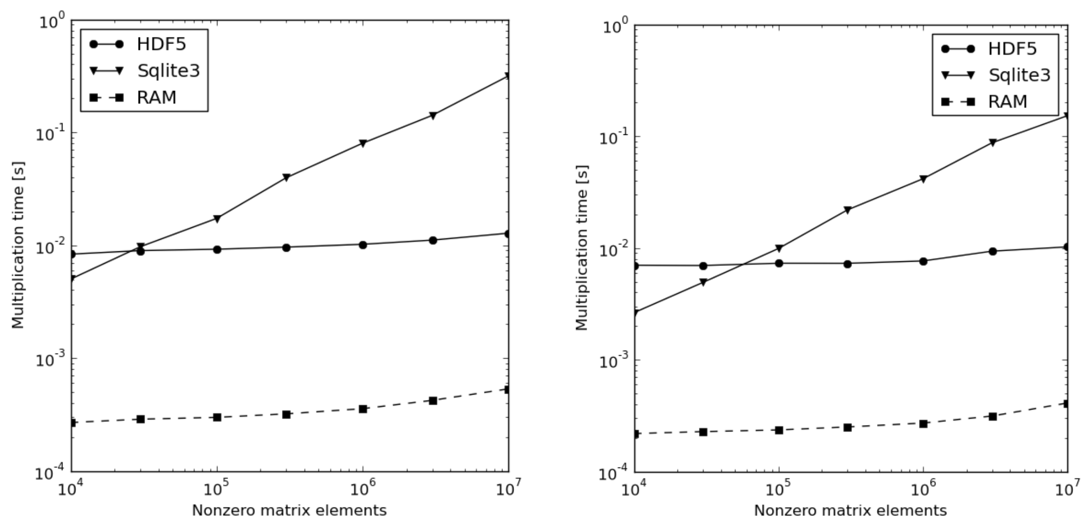

Fig. 3. Sparse matrix-vector multiplication time against sizes of matrices. Nonzero elements of the vector are as high as 10, average density is about 0.05. Left: PC, Right: Server.

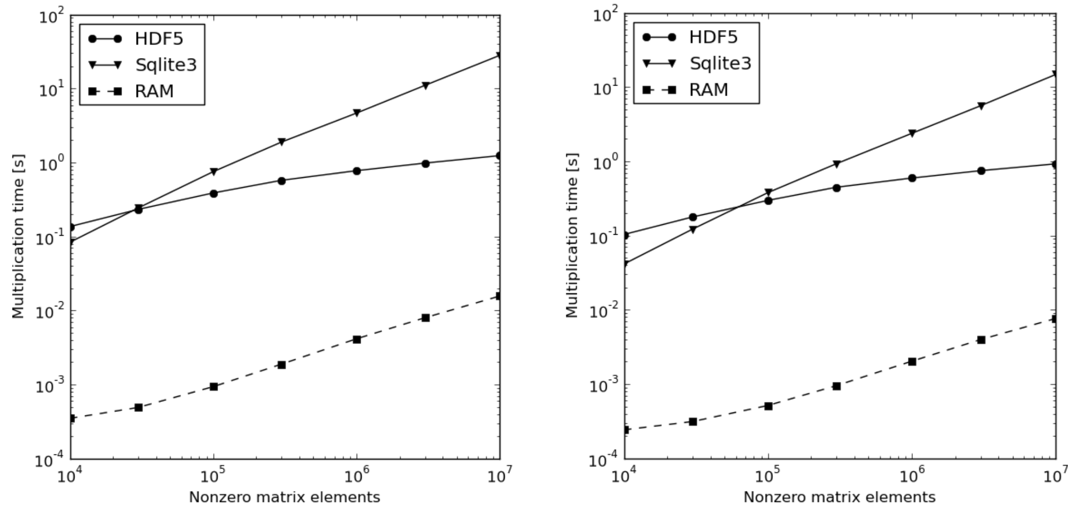

Fig. 4. Sparse matrix-vector multiplication time against sizes of matrices. Nonzero elements of the vector are as high as 1000, average density is about 0.05. Left: PC, Right: Server

6. CONCLUSIONS

The benchmark results are mixed and no solution fitting all needs can be designated. Certainly, for a class of problems, the proposed CSC format based of HDF5 behaves fast enough for it may be worth implementing a low level, high performance sparse library on top of HDF5.

As the benchmarks indicate, the proposed format outperforms the relational database solution, when the number of entries is high, and whole columns of numbers can be fetched from the disk in continuous operation. Even if the RDBMS is faster in the beginning, its performance degrades quicker than of the presented schema. However, the performance of RDBMS solution is incomparable, when the number of data to be read is relatively low, so it is closer to the typical use cases of the engine, where the indices implementation excel.

Unfortunately, offline matrices are always slower than their counterparts in RAM by constant order of magnitude, which is also seen on the charts. Such constraint limits the application possibility of offline matrices to the problems with high locality, like the presented one. For problems which require random access to whole matrix, such as factorization or clustering, the speed regression would be proportional to the proportion of access speed to RAM and non-volatile memory, which in the presented case was about 15. This motivates the progress in agglomerative algorithms, where the dataset is divided into separate chunks by design, instead of the necessity, such the popular ones presented in [7, 8].

LITERATURE

1. Dongarra, J., Xz, J. D., Lumsdaine, A., Niu, X., Pozo, R. and Remington, K.: *A sparse matrix library in C++ for high performance architectures*, 1994.
2. Georgieva, J., Gancheva, V. and Goranova, M.: *Scientific data formats*. In AIC'09: Proceedings of the 9th WSEAS international conference on Applied informatics and communications, pages 19–24, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).
3. HDF Group. *Hdf5 users*. Online, Last modified January 28th 2010.
4. Sonnenburg, S.: *Matlab(TM) 7.3 file format is actually HDF5 and can be read from other languages like Python*. Online, November 2009.
5. Cai, X., Langtangen, P.T., and Moe, H.: *On the performance of the Python programming language for serial and parallel scientific computations,* Scientific Programming, Volume 13, pages 31-56, number 1/2005.
6. Cannon, B.: *Localized type inference of atomic types in Python*, Online, 2005.
7. Gorrell, G.: *Generalized hebbian algorithm for incremental latent semantic analysis*. In Proceedings of Interspeech, 2006.

Manuscript submitted 17.08.2010
Reviewed by Jacek Starzyński, D.Sc., Eng.

SZYBKIE RZADKIE MACIERZE PRZECHOWYWANE NA DYSKU

Jan WICIJOWSKI

STRESZCZENIE *Ograniczenia pamięci komputera są powszechnym problemem przy obliczeniach przeprowadzanych na wielkich zbiorach danych. Przy danych roboczych przekraczających 1 GiB, składowanie całości w pamięci operacyjnej staje się utrudnione, a często nawet nieosiągalne. Ponieważ w większości aplikacji wykonuje się działania jedynie na fragmencie zbioru danych, reszta może być przechowywana w pamięci stałej, która zapewnia dużo większe pojemności. Dostęp do pamięci stałej jest zazwyczaj kilka rzędów wielkości wolniejszy niż do RAMu, zatem należy przedstawić metodę składowania ograniczającą do minimum ilość dostępów do dysku. W artykule opisuję format przechowywania macierzy rzadkich na dysku, zbudowanym na bazie formatu HDF5 (Hierarchical Data Format) pod kątem minimalizacji czasu mnożenia tej macierzy przez wektor.*

Jan WICIJOWSKI, M.Sc. Electronics and Telecommunications graduated from AGH University of Science and Technology in Krakow (2009). Currently he pursues Ph.D. in Electronics on AGH-UST. In his work at Signal Processing Group he does research on automated speech recognition and semantic modelling of Polish.