Storage Formats for Sparse Matrices in Java

Mikel Luján*, Anila Usman, Patrick Hardie, T.L. Freeman, and John R. Gurd

Centre for Novel Computing, The University of Manchester,
Oxford Road, Manchester M13 9PL, United Kingdom
{mlujan, ausman, hardiep, lfreeman, jgurd}@cs.man.ac.uk

Abstract. Many storage formats (or data structures) have been proposed to represent sparse matrices. This paper presents a performance evaluation in Java comparing eight of the most popular formats plus one recently proposed specifically for Java (by Gundersen and Steihaug [6] – Java Sparse Array) using the matrix-vector multiplication operation.

1 Introduction

Sparse matrices are those matrices which have a substantial minority of nonzero elements – normally less than 10% are nonzero elements. These matrices are pervasive in many computational science and engineering (CS&E) applications. The storage formats for sparse matrices have been proposed to better suit particular CS&E applications or computer architectures. The significant number of different storage formats is the source of a research problem. For example, consider the recently published Basic Linear Algebra Subroutines (BLAS) standard and the part dedicated to sparse matrices (*Sparse BLAS*)[4]. The Sparse BLAS do not state which storage formats must be supported or used. Each specific hardware vendor has the freedom (or problem) to select the storage format (or formats) that perform best for its hardware. In the context of iterative methods [2] and Java, this papers investigates the performance delivered by different storage formats considering a wide variety of sparse matrices.

The structure of the paper is as follows. Section 2 introduces the most commonly used storage formats for sparse matrices. The Java Sparse Array (JSA) storage format was recently proposed by Gundersen and Steihaug [6] to take advantage of Java arrays; Section 3 briefly describes JSA. The performance evaluation (see Section 5) consider a specific kernel from iterative methods, namely matrix-vector multiplication, and compares this operation on two different computational platforms with nine different storage formats. The Java implementation of this matrix operation is described in Section 4. The performance study considers around 200 different sparse matrices representing various CS&E applications as recorded by the Matrix Market repository [1]. To the best of the authors' knowledge, there is no other performance evaluation of storage formats for sparse matrices which consider such a variety of matrices and storage formats. Conclusions and future work are given in Section 6.

* ML acknowledges a postdoctoral fellowship from the Basque Government. AU acknowledges a postdoctoral fellowship from the HEC Pakistan.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 364–371, 2005.

2 Storage Formats for Sparse Matrices

The objective of storage formats for sparse matrices is to best exploit certain matrix properties by (1) reducing memory space, by storing only nonzero elements of a sparse matrix, and (2) by storing these elements in contiguous memory locations, for more efficient execution of subroutines on the matrix data.

From an implementation point of view, there are two categories of storage formats. *Point entry* is used to categorise storage formats where each entry in the storage format is a single element of the matrix. *Block entry* refers to storage formats where each entry defines a dense block of elements of any two dimensions. For both cases, programming languages provide static and dynamic data structures. However since Fortran 77 has been the dominant language in CS&E and does not support dynamic data structures, the most commonly used storage formats are array-based.

There are many documented versions of different storage formats for sparse matrices. One of the most complete sources is the book by Duff *et al.* [3] (for a historical source see [7]).

2.1 Point Entry Storage Formats

Coordinate Format (COO) — Possibly the most intuitive storage format for a sparse matrix is in terms of coordinates. Instead of storing the matrix densely, a list of the coordinates in terms of row and column numbers is stored, with the associated nonzero values. COO requires no specific structure of the matrix and is a very flexible format. It requires three (unordered) arrays and a single scalar recording the total *number of nonzero elements*, *nnz*. The combination of the three arrays provides a row i and column j coordinate pair for an element in the matrix along with its value a_{ij}. In general, for a matrix with *nnz*, COO requires three 1-dimensional arrays of length *nnz* plus a scalar.

Compressed Sparse Row/Column Storage Formats (CSR/CSC) — As with COO, CSR and CSC storage formats can store any matrix. In CSR, the nonzero values of every row in the matrix are stored, together with their column number, consecutively in two parallel arrays, *Value* and *j*. There is no particular order with respect to the column number, *j*. The *Size* and *Pointer* for each row define the number of nonzero elements in the row and point to the relative position of the first nonzero element of the next row, respectively. The column based version, CSC, instead stores *Value* and *i*, in two parallel arrays and *Size* and *Pointer* of each column allows each member of *Value* to be associated with a column as well as the row given in *i*. The storage requirements are two arrays, each of length the number of rows (or columns), and two further arrays of length *nnz*, and a scalar to point to the next free location in the arrays i (or j) and *Value*.

The *Diagonal Storage Format* (DIA) and *Skyline Storage Formats* (SKS) are also part of the performance evaluation described in Section 5, but their description is omitted for brevity.

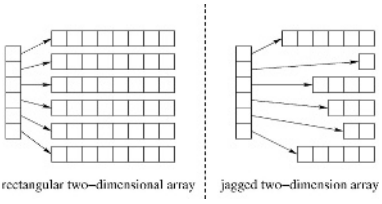

Fig. 1. Examples of two-dimensional arrays in Java

2.2 Block Entry Storage Formats

Block entry storage formats form an extension of certain point entry storage formats based on partitioning matrices into blocks of elements (i.e. sub-matrices). An example of a variable block matrix A is as follows:

$$A = \left(\begin{array}{cc|ccccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{22} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ \hline a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} \end{array}\right) \text{ or } \left(\begin{array}{ccc} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{array}\right) \text{ where, for example, } A_{11} =$$

$$\begin{pmatrix} a_{11} & a_{12} \end{pmatrix}, A_{13} = \begin{pmatrix} a_{17} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{23} & a_{24} & a_{25} & a_{26} \\ a_{33} & a_{34} & a_{35} & a_{36} \end{pmatrix} \text{ and } A_{33} = \begin{pmatrix} a_{47} \\ a_{57} \\ a_{67} \end{pmatrix}.$$

In the point entry storage formats, the storage format describes the position in the (*Value*) array of single matrix elements. Block entry storage formats (with length of block lb), instead have a scheme to describe the position of a single block in a $n/lb \times n/lb$ blocked matrix. Each block contains lb^2 elements. In this way, most point entry storage formats can be blocked to generate Block Coordinate storage format (BCO), Block Sparse Row/Column storage format (BSR/BSC) and others where the block does not have constant dimensions (e.g. Variable Block Compressed Sparse Row).

3 Java Sparse Array (JSA)

The storage formats covered so far have been in use for several years. In contrast, a more recent storage format, JSA, has been created to exploit Java's flexible definition of multi-dimensional arrays. In Java, every array is an object storing either primitive types (i.e. float, double, etc.) or other objects. A two-dimensional array is formed as an array of arrays. This definition enables developers to create both rectangular and jagged arrays (see Fig. 1).

JSA is a row oriented storage format similar to CSR. It uses two arrays, each element of which is itself an array (object). One of these arrays, *Value*, stores arrays of the matrix elements – each row in the matrix has its elements in a separate array. All the separate arrays are elements of the *Value* array; that is an array of array objects. The second main array *Index* stores arrays containing the column numbers of the matrix, again one array per row. The memory re-

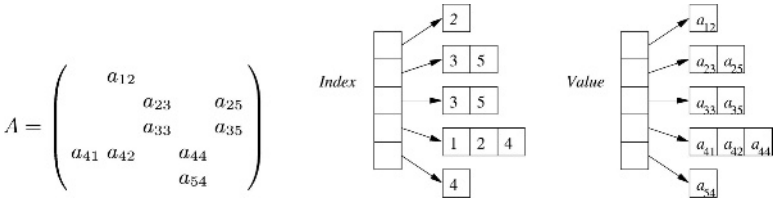

Fig. 2. An example sparse matrix stored using JSA

quirements to store a sparse matrix in JSA are $2nnz + 2n$ array locations. Figure 2 shows an example sparse matrix stored using JSA.

4 Sparse Matrix-Vector Multiplication

The performance evaluation presented in the following section is predicated upon Java implementations of sparse matrix-vector multiplications. These implementations have been developed for this work and follow the structure of the Fortran 95 reference implementation of the Sparse BLAS developed by CERFACS [4]. Object-oriented features are used only for passing parameters to methods (subroutines), but are not used inside the kernels that actually implement the matrix operation. Some simplifications are made compared with the Sparse BLAS reference implementation. The matrix data is assumed to be static once created. This does not modify the implementation of the matrix-vector multiplication, but simplifies the code to create, access and destroy matrices. The implementations concentrate on square and double precision matrices. The Java implementations incorporate external code, such as JSA.

JSA Implementation — Two Java implementations of sparse matrix-vector multiplication are considered for JSA: JSA-GS and JSA2. The JSA-GS implementation is the code made available by Gundersen and Steihaug [6]. This code does not include a specialised case for symmetric matrices[1]. A new subroutine was incorporated into this code (a new method in class JavaSparseArray) to support symmetric matrices. The JSA2 implementation is a reimplementation of JSA following the structure of the Sparse BLAS reference implementation. The main differences compared with JSA-GS is the code for creating, handling and destroying matrices, and one subroutine (or method) which implements the multiplication rather than two as in JSA-GS. When a matrix is created the Sparse BLAS allow users to specify whether the matrix is symmetric. With JSA-GS a program has to check the information provided about the matrix to call either the general subroutine or the symmetric subroutine. With JSA2, a program simply calls the subroutine and the check is performed internally. Otherwise the sets of instructions that implement the multiplication are identical.

[1] A matrix A is symmetric when $a_{ij} = a_{ji}$.

5 Performance Evaluation

The aim of this performance evaluation is to analyse the circumstances under which a given storage format performs better than the other storage formats.

Table 1. Legend for the 'storage formats'-axis in Fig. 3

1	COO		6	BCO block size 8		11	BSR block size 16		16	BSC block size 32	
2	CSR		7	BCO block size 16		12	BSR block size 32		17	BSC block size 64	
3	CSC		8	BCO block size 32		13	BSR block size 64		18	DIA	
4	JSA-GS		9	BCO block size 64		14	BSC block size 8		19	SKS	
5	JSA2		10	BSR block size 8		15	BSC block size 16				

Experimental Testbed — Matrix test data are (around 200 different matrices) real, symmetric and non-symmetric matrices from Eigenvalue problems and Linear Systems available from the Matrix Market Collection [1]. The test program reads a matrix from file, calculates the multiplication of that matrix with a random vector and records the result as well as the time taken to calculate the product. The test program repeats this computation for the 9 different storage formats (note different block sizes).

The two test machines run the Java Virtual Machines (JVMs) with the minimum and maximum heap sizes of 128 MB and 1536 MB, respectively, and -server flag. System A is an Ultra Sparc 10 at 333 MHz with 256 MB running Solaris 5.8 and Sun Java 2 SDK 1.4.2 Standard Edition (SE). System B is an Intel Pentium 4 at 2.6 GHz with 512 MB running Red Hat 9 kernel 2.4.20-31.9 and Sun Java 2 SDK 1.4.2 SE. The timer accuracy is one millisecond and the time reported is the time spent performing 50 matrix-vector multiplications on System A and 200 on System B. The numbers 50 and 200 are selected so that the times are large enough in relation to the accuracy of the timers.

Performance Results on all Systems and Matrices — Figure 3 presents the average times (out of four runs) for each matrix on both machines. The general pattern is the same on both platforms. The block entry storage formats do not perform significantly better with different block sizes. This suggests that any gain that results from more efficient use of the memory hierarchy is offset by increases in the number of zero elements that need to be processed. Throughout, the point entry storage formats, with the exception of DIA, appear to give the best performance. At best the block entry storage formats get close to the point entry storage formats.

The Fastest Storage Formats — In Fig. 3 the fastest storage formats are COO, CSR, CSC, JSA-GS and JSA2. Figure 4 shows in more detail the execution times for these storage formats. The graphs present the results for JSA-GS, JSA2 and the minimum time taken by the other three storage formats; i.e. min(COO, CSR, CSC) $\equiv$ MCCC. For System A, JSA-GS consistently performs better than JSA2 and JSA2 itself in most cases performs better than or equivalent to MCCC. System B does not offer the same clear conclusion. For the matrices in the region

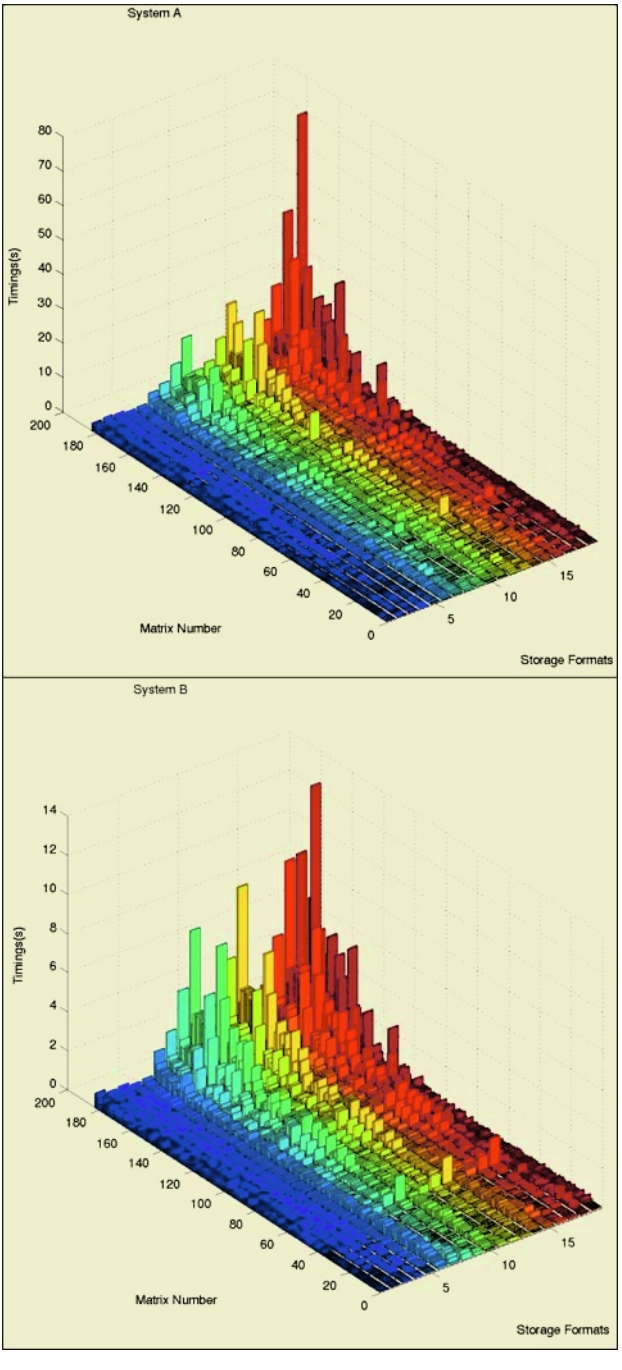

Fig. 3. Time results (seconds) on Systems A and B. The legend for the 'storage formats'-axis is given in Table 1. The 'matrix number'-axis is ordered by *nnz*

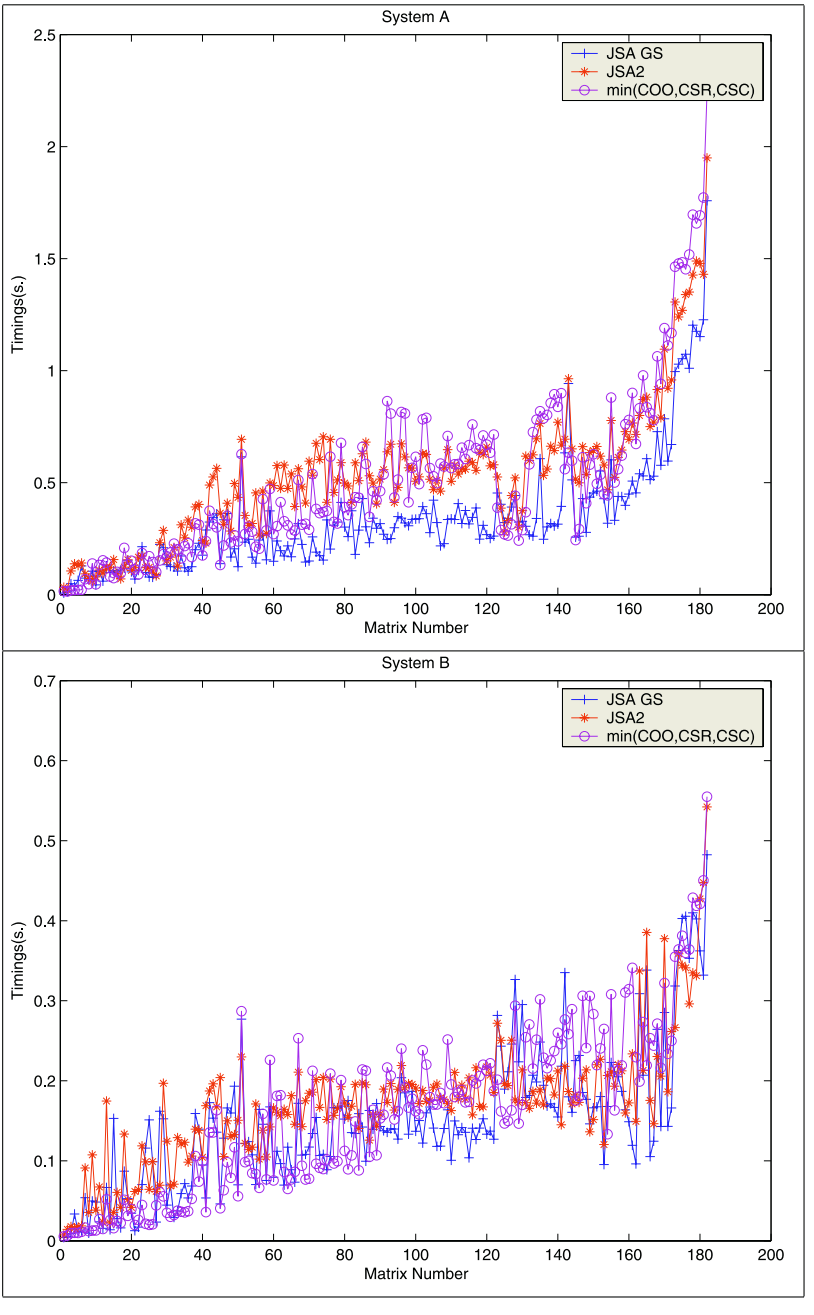

Fig. 4. Time results (seconds) for the fastest storage formats. The matrix number axis is ordered by *nnz*

1 to 90 (matrices with the fewest nnz), MCCC performs in most cases similarly to JSA-GS. For these matrices JSA2 performs similarly, but slightly worse than JSA-GS. For the rest of the matrices, JSA-GS performs better than JSA2 in most cases and JSA2 performs better than or similarly to MCCC. The exception is in the matrix region around 130 where JSA-GS and JSA2 deliver similar execution times, but both are outperformed by MCCC.

As mentioned in Section 4, the computationally intensive code in JSA-GS and JSA2 is identical. Thus, a reasonably expectation would be that the performance delivered by JSA-GS and JSA2 should be similar. One possible explanation for the observed different performances is that the JVMs find it more amenable to optimise the smaller subroutines (methods) in JSA-GS (symmetric vs. non-symmetric support).

6 Conclusions and Future Work

It would be presumptuous to say that all the storage formats for sparse matrices are covered by this work, especially since there are many minor variations which can create entirely new storage formats. Nonetheless, this paper has presented a comprehensive performance comparison of storage formats for sparse matrices. The results have shown that JSA performes better than the other storage formats for most matrices. The block entry storage formats have not performed as well as the point entry storage formats. Future work underway is to include a similar set of experiments with Fortran implementations and then other operations supported by the Sparse BLAS. The most relevant related work is the Sparsity project [5]. For a given sparse matrix the Sparsity project has developed compile time techniques to optimise automatically several sparse matrix kernels using a specific block entry storage format.

References

1. The matrix market. http://math.nist.gov/MatrixMarket/.
2. R. Barrett *et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.
3. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.
4. I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, 2002.
5. Eun-Jin, K. A. Yelick, and R. Vuduc. SPARSITY: An optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.
6. G. Gundersen and T. Steihaug. Data structures in Java for matrix computations. *Concurrency and Computation: Practice and Experience*, 16(8):799–815, 2004.
7. U. W. Pooch and A. Nieder. A survey of indexing techniques for sparse matrices. *ACM Computing Surveys*, 5(2):109–133, 1973.