The Use of Java Arrays for Matrix Computations

Geir Gundersen[1] and Trond Steihaug[1]

Department of Informatics,
University of Bergen, Norway
{geirg,trond}@ii.uib.no

Abstract. In the paper it is shown how to utilize the flexibility in native Java arrays for matrix computations. Suitable datastructures for symmetric and sparse matrices are introduced. A disadvantage of the native Java arrays is shown when used as two-dimensional array for dense matrix computation. Numerical results show that the efficiency is not lost using the more flexible datastructures compared to classical datastructure for sparse matrices. This flexibility can be utilized for high performance computing (HPC).

1 Introduction

Object-oriented programming have been favored in the last decade(s) and has an easy to understand paradigm. It is straightforward to build large scale application designed in an object-oriented manner. Java's considerable impact implies that it will be used for high performance computing and Java is already introduced as the programming language in introductory courses in scientific computation [3].

Matrix computation is a large and important area in scientific computation. Developing efficient algorithms for working with matrices are of considerable practical interest.

A sparse matrix is usually defined as a matrix where "many" of its elements are equal to zero and we benefit both in time and space by working only on the nonzero elements [6]. The difficulty is that sparse data structures include more overhead (to store indices as well as numerical values of nonzero matrix entries) than the simple arrays used for dense matrices.

There are several different storage schemes for large sparse matrices that are used in languages like FORTRAN, C and C++. These storage schemes have enjoyed several decades of research and the most commonly used storage schemes for large sparse matrices are the compressed row or column storage [6, 7]. The compressed storage schemes have minimal memory requirements and have shown to be convenient for several important operations. With the Java Native Interface (JNI) it is possible to call native methods written in C, FORTRAN and C++. However, after the of release JDK 1.3 it is not advisable to use native methods for improved performance [5].

We discuss the use of Java arrays for storing matrices and particularly four different storage schemes for sparse matrices: Compressed Row Storage, Java Sparse Arrays, Sparse Matrix Concept, and the Skyline Structure. The storage schemes for general sparse matrices are discussed on the basis of performance and flexibility on matrix multiplication and updates.

The compressed storage schemes can be implemented in all languages, while the sparse matrix concept is restricted to object oriented languages. Java Sparse Array [4] is unique for Java.

The timings for the sparse matrix operations where done on Linux with Sun's Java Development Kit (JDK) 1.4.0. The time is measured in milliseconds (mS). The sparse matrices used in the tests are from Matrix Market [14].

2 Java Arrays

Java implements arrays as true objects with defined behaviour. This imposes overhead on a Java application using arrays compared to equivalent C and C++ programs. Creating an array is object creation. When creating an array of primitive elements, the array holds the actual values for those elements. An array of objects stores references to the actual objects. Since arrays are handled through references, an array element may refer to another array thus creating a multidimensional array. A rectangular array of numbers as shown in Figure 4 is implemented as Figure 5. Since each elements in the outermost array of a multidimensional array is an object reference, arrays need not be rectangular and each inner array can have its own size as in Figure 6.

We can expect elements of an array of primitive elements to be stored consecutively , but we cannot expect the objects of an array of objects to be stored consecutively. For a rectangular array of primitive elements, the elements of a row will be stored consecutively, but the rows may be scattered.

A matrix is a rectangular array of entries and the size is described in terms of the numbers of rows and columns. The entry in row i and column j of matrix A is denoted A_{ij}. To be consistent with Java, the first row and column index is 0 and element A_{ij} will in Java be `A[i][j]` and a matrix will be a rectangular array of primitive elements. A vector is either a matrix with only one column (column vector) or one row (row vector).

Consider computing the sum of the elements of the $m \times n$ matrix A

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij}. \qquad (1)$$

The code examples in Figure 1 and 2 show two implementations of (1) in Java. The only difference between the two implementations is that the two `for` loops are interchanged. Loop-order (i,j) implies that the elements of the matrix are accessed row-by-row and loop-order (j,i) implies that the access of the elements is column-by-column. Figure 3 shows that traversing columns is much less efficient than traversing rows when the array gets larger. This demonstrates the

basic observation that accessing the consecutive elements in a row is faster than accessing consecutive elements in a column. Traversing consecutive elements in a matrix (either row or column) is a common operation in matrix computation routines.

```
double s = 0;
double[] array = new double[m][n];
for(int i = 0;i<m;i++){
   for(int j = 0;j<n; j++){
     s+=array[i][j];'
   }
}
```

Fig. 1. Loop-order (i,j) (row wise)

```
double s = 0;
double[] array = new double[m][n];
for(int j = 0;j<n;j++){
  for(int i = 0;i<m; i++){
      s+=array[i][j];
  }
}
```

Fig. 2. Loop-order (j,i) (column wise)

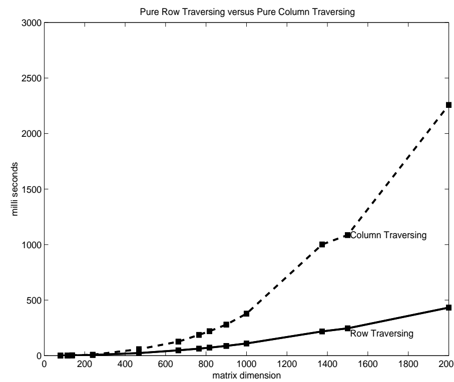

Fig. 3. Time accessing the array matrix row wise and column wise

An element of `double[][]` is the object `double[]`. When an object is created and gets heap allocated, the object can be placed anywhere in the memory. This implies that the elements `double[]` of a `double[][]` may be scattered throughout the memory space, thus explaining a part of the time differences in row and column wise loop order.

Since a 2D Java array can have different row lengths it is straightforward to store a symmetric matrix. A matrix is symmetric if $A_{ij} = A_{ji}$ and only the lower or upper triangular part need to be stored. To store the lower part of a matrix the first row has length one, the second length two, and so on, as illustrated in Figure 7. JAMA[1] has no support for symmetric matrices in their package, so a suggestion might be to use this rather obvious approach as an add on to JAMA.

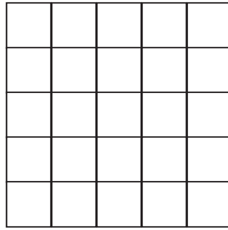

Fig. 4. A *true* 2D array.

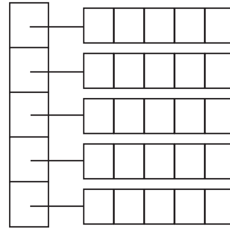

Fig. 5. A 2D Java array.

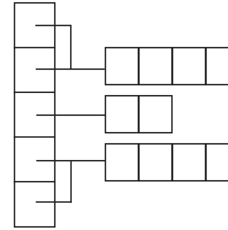

Fig. 6. General Java array.

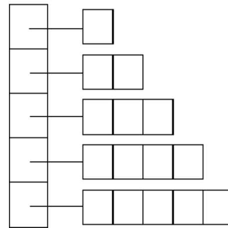

Fig. 7. The lower part of a matrix.

3 Sparse Matrices

Currently there is no *released* packages implemented in Java for numerical computation on sparse matrices, as complete as JAMA and JAMPACK [1, 8] for dense matrices. But there are separate algorithms like [9] using a coordinate storage scheme. The coordinate storage scheme is the most straightforward structure to represent a sparse matrix, it simply records each nonzero entry together with its row and column index. [10] use the coordinate storage format as implemented in C++ in [11]. The coordinate storage format is not an efficient storage format for large sparse compared to compressed row format [3]. There are also some benchmark algorithms like [13] that performs sparse matrix computations using compressed row storage scheme. The sparse structures naturally incorporates symmetric matrices by only storing the lower or upper triangular part of the matrix.

3.1 Compressed Storage Schemes

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in consecutive locations. For a sparse matrix we create three vectors: one for the double type (`value`) and the other two for integers (`columnindex`, `rowpointer`). The `value` vector stores the values of the nonzero elements of the matrix, as they are traversed in a row-wise fashion. The `columnindex` vector stores the column indexes of the elements in the `value` vector. The `rowpointer` vector stores the locations in the `value` vector that start a row. If `value[k]` =

A_{ij} then `columnindex[k]`$= j$ and `rowpointer`$[i] \leq$ `k` $<$ `rowpointer`$[i+1]$. By convention `rowpointer`$[m]$ = nnz, where nnz is the number of nonzero elements in the matrix and m is the number of rows. Consider the sparse 6×6 matrix A.

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}. \quad (2)$$

The nonzero structure of the matrix A (2) stored in the CRS scheme:

```
double[] value = {10,-2,3,9,3,7,8,7,3,8,7,5,8,9,9,13,4,2,-1};
int[] columnindex = {0,4,0,1,5,1,2,3,0,2,3,4,1,3,4,5,1,4,5};
int[] rowpointer = {0,2,5,8,12,16,19};
```

3.2 Java Sparse Array

The Java Sparse Array format is a new concept for storing sparse matrices made possible with Java [4]. This concept is illustrated in Figure 8 and uses an array of arrays. There are two arrays, one for storing the references to the value arrays and one for storing the references to the index arrays (one for each row).

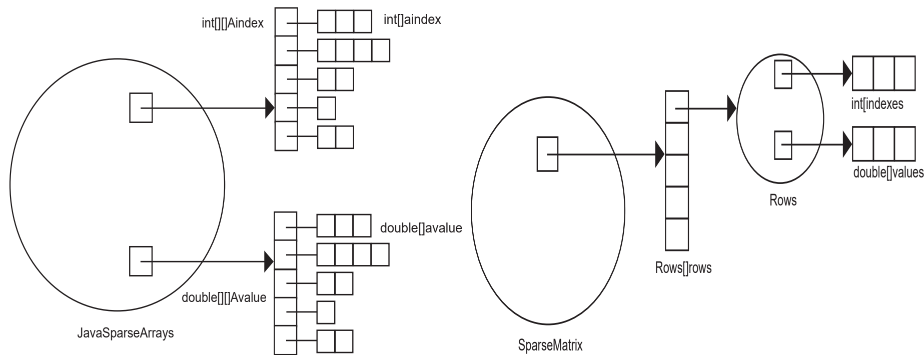

Fig. 8. The Java Sparse Array format. **Fig. 9.** The Sparse Matrix Concept.

With the Java Sparse Array format it is possible to manipulate the rows independently without updating the rest of the structure as would have been necessary with CRS. Each row consist of a value and an index array each with its own unique reference. The storage saving for sparse matrices is significant.

Instead of storing $m \cdot n$ elements, Java Sparse Array use $2nnz + 2m$ storage locations compared to $2nnz + m + 1$ for the CRS format. Here n is the number of columns. The nonzero structure of the matrix A (2) is stored as follows in Java Sparse Array.

```
double[][] value = {{10,-2},{3,9,3},{7,8,7},
                    {3,8,7,5},{8,9,9,13},{4,2,-1}};
int[][] index =    {{0,4},{0,1,5},{1,2,3},{0,2,3,4},
                    {1,3,4,5},{1,4,5}};
```

A `JavaSparseArray` "skeleton" class can look like this.

```
public class JavaSparseArray{
  private double[][] Avalue;
  private int[][] Aindex;
  public JavaSparseArray(double[][] Avalue, int[][] Aindex){
    this.Avalue = Avalue;
    this.Bvalue = Aindex;
  }
  public JavaSparseArray times(JavaSparseArray B){...}
}
```

In this `JavaSparseArray` class, there are two instance variables, `double[][]` and `int[][]`. For each row these arrays are used to store the actual value and the column index.

3.3 The Sparse Matrix Concept

The Sparse Matrix Concept (SMC) is a general object-oriented structure illustarted in Figure 9. It is similar to JSA, but it does not take advantage of the feature that Java's native arrays are true objects. SMC can be implemented in the following way [12].

```
public class SparseMatrix{
      private Rows[] rows;
      public SparseMatrix(Rows[] rows){
          this.rows = rows;
      }
      public SparseMatrix times(SparseMatrix B){...}
}
public class Rows{
      private double[] values;
      private int[] indexes;
      public Rows(double[] values, int[] indexes){
            this.values = values;
            this.indexes = indexes;
      }
      public double elementAt(int k){return values[k];}
}
```

The actual storage of terms of primitive elemets is the same for SMC and JSA, but JSA does not use the extra object layer for each row creating extra references.

Methods that work explicitly on the arrays (`values` and `indexes`) are placed in the `Rows` objects and instances of the `Rows` object are accessed through method calls. Breaking the encapsulation and storing the instances of the `Rows` object as local variables makes the Sparse Matrix Concept very similar to JSA.

Since a Java array can store references to objects the `Rows` objects can be removed and replaced with an `int[][]`, for storing index arrays, and a `double[][]`, for storing value arrays. Storing m (the number of rows) `Rows` objects we have $3m$ references compared to $2m$ references for JSA, each reference is 4 bytes (32 bits).

The most natural notation of retrieving an element A_{ij} is `A[i][k]` where `Aindex[i][k]`$=j$ in JSA. Unlike SMC which first has to get the `Rows` object `i`, then the element `k` by `rows[i].elementAt(k)`. This is an object-oriented approach to retrieve instances from an object oriented datastructure.

3.4 Skyline Structure Matrices

In this datastructure, all matrix elements from the first nonzero in each row to the last nonzero in the row are explicitly stored. For each row i define upper u_i and lower l_i bandwidths

$$l_i = \max_j \; i - j \text{ for nonzero } A_{ij},$$
$$u_i = \max_j \; j - i \text{ for nonzero } A_{ij}.$$

The skyline storage requirement for an $m \times n$ matrix is $\sum_{i=0}^{m-1} l_i + u_i + m$.

$$A = \begin{pmatrix} 0 & 0 & 0 & x & x \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & x & x & x & 0 \\ x & x & 0 & 0 & 0 \end{pmatrix}.$$

Fig. 10. A skyline structure matrix.

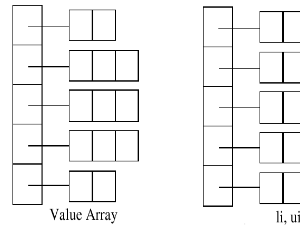

Fig. 11. Storing a skyline structure matrix.

In Figure 11 the index array only stores u_i and l_i since the column indexes are $i - l_i \leq j \leq i + u_i$.

For tri-diagonal ($u_i = l_i = 1$), matrices, as shown, in Figure 12, the rows $i = 1, 2..., m-2$ have the same upper and lower bandwidth and if we know there are nonzero diagonal elements, we can store the tri-diagonal matrix as shown in Figure 13.

$$A = \begin{pmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix} .$$

Fig. 12. A tri-diagonal skyline structure matrix.

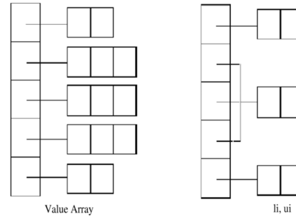

Fig. 13. Storing a tri-diagonal skyline structure matrix.

3.5 Sparse Matrix Multiplication

A problem in a matrix multiplication $C = AB$ algorithm on CRS is that we do not know the actual size (nnz) or structure of the resulting matrix C. This structure can for sparse general matrices be found by using the datastructures of A and B. The implementation used is based on FORTRAN routines [6,15] using Java's native arrays.

Table 1. Matrix multiplication $C = AA$ and update $A = A + ab^T$.Times in mS for the algorithms.

A		Sparse Matrix Multiplication				Sparse Matrix Update				
n	$nnz(A)$	$nnz(AA)$	CRS	JSA	SMC	$nnz(ab^T)$	$nnz(\text{new}A)$	CRS	JSA	SMC
115	421	1027	1	0	1	7	426	11	0	0
468	2820	8920	19	11	13	148	2963	13	1	1
2205	14133	46199	21	19	19	449	14557	44	8	3
4884	147631	473734	185	130	130	2365	149942	183	8	9
10974	219512	620957	207	160	158	1350	220104	753	8	8
17282	553956	2525937	829	600	545	324	554138	1806	11	11

For each row of A those rows of B indexed by the index array of the sparse row of A are traversed. Each element in the value array of A, is multiplied with the value array of B, which is indexed by the index array of that row. The result is stored in the result row of C, which is indexed by the index value of the element in the B row.
We can construct `double[n][1]` and `int[n][1]` to contain the rows of the resulted elements of matrix C, before we actually creates the actual rows of the resulting matrix [15].

Comparing in Table 1 CRS to JSA we see that JSA is more efficient than CRS with an average factor of 1.37. It is important to state that we cannot draw too general conclusions on the performance of these algorithms on the basis of the test matrices we used in this paper. But these results give a strong indication on their overall performance and that matrix multiplication on Java Sparse Array is

both fast and reliable. The SMC approach have to create a `Rows` object for each value and index array in the resulting matrix in addition to always access the value and index array from method calls. The object creation profiler JProfiler [18] is used to analyze the memory usage of JSA and SMC for a sparse matrix multiplication. For $n = 17282$ the profiler indicates that in SMC the `Rows` object causes 2.3% of the overall memory use. SMC uses only 1% more memory than JSA. For all testing performed JSA gives a small memory gain, but no significant gain in time compared to SMC.

3.6 Sparse Matrix Update

Consider the outer product ab^T of the two vectors $a \in \Re^m, b \in \Re^n$ where many of the elements are 0. The outer product will be a sparse matrix with some rows where all elements are 0, and the corresponding sparse datastructure will have rows without any elements. A typical operation is a rank one update of an $m \times n$ matrix A.

$$A_{ij} = A_{ij} + a_i b_j \; i = 0, 1, \ldots, m-1, \; j = 0, 1, \ldots, n-1 \tag{3}$$

where a_i is element i in a and b_j is element j in b. Thus only those rows of A where $a_i \neq 0$ need to be updated. This can easily be done in JSA while for CRS we need to create new `value` and `columnindex` array and perform either a copy or a multiplication and an addition. This is clearly shown in Table 1 where 10% of the elements in a are nonzero. The JSA algorithm is an average factor of 78 times faster than the CRS algorithm. The overhead in creating a native Java array is propotional to the number of elements thus accounting for the major difference between CRS and JSA on matrix updates.

4 Concluding Remarks

When using Java arrays as two-dimensional arrays we need to consider its row-wise layout. One proposal to make the difference between row and column traversing less significant, is to cluster the row objects together in memory [16]. However this accounts for only a minor part of the difference. Other suggestion is to make a multidimensional Java array class avoiding array of arrays [17]. For sparse matrices we have illustrated the effect of manipulating rows of the structure without updating the rest of the structure. This excludes Compressed Row Storage in favor of Java Sparse Array and Sparse Matrix Concept as the preferable sparse matrix datastructures in Java. The Sparse Matrix Concept has an unique implementation in Java as Java Sparse Array.

References

1. Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington, *JAMA: A Java Matrix Package*, June 1999. http://math.nist.gov/javanumerics/jama.

2. Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A. Remington, and G. W. Stewart *Developing numerical libraries in Java*, Concurrency: Practice and Experience, 10(1998)1117-1129..
3. Geir Gundersen, *The use of Java arrays in matrix computation*, Cand. Scient Thesis, University of Bergen, Bergen, Norway, April 2002.
4. Geir Gundersen and Trond Steihaug, *Datastructures in Java for Matrix Computation*, Submitted for publication.
5. Joshua Bloch, *Effective Java Programming Langauage Guide*, Addison Wesley, New York, 2001.
6. Sergio Pissanetsky, *Sparse Matrix Technology*, Academic Press, Massachusetts, 1984.
7. Yousef Saad, *Iterative Methods for Sparse linear Systems*, PWS Publishing Company, Boston, 1996.
8. G.W. Stewart. *JAMPACK: A Package for Matrix Computations*, February 1999. ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html.
9. Java Numerical Toolkit. http://math.nist.gov/jnt.
10. Rong-Guey Chang, Cheng-Wei Chen, Tyng-Ruey Chuang, and Jenq Kuen Lee, *Towards Automatic Support of Parallel Sparse Computation in Java with Continuous Compilation.* Concurrency: Practice and Experience, 9(1997)1101-1111.
11. Roldan Pozo, Karin Remington, and Andrew Lumsdaine, *SparseLib++ v. 1.5 Sparse Matrix Reference Guide.* http://math.nist.gov/sparselib++.
12. Arnold Nielsen, Thilo Kielman, Henri Bal, and Jason Maaasen, *Object Based Communication in Java*, pages 11-20, Proceedings of the ACM 2001 Java Grande/ISCOPE Conference, June 2-4 Palo Alto.
13. SciMark 2.0. http://math.nist.gov/scimark2/about.html.
14. Matrix Market. http://math.nist.gov/MatrixMarket.
15. Yocef Saad, *SPARSEKIT: A basic tool kit for sparse matrix computations, Version 2*, Technical Report, Computer Science Department, University of Minnesota, June 1994.
16. James Gosling, *The Evolution of Numerical Computing in Java*, http://java.sun.com/people/jag/FP.html.
17. José E. Moreira, Samuel P. Midkiff, and Manish Gupta, *A comparison of three approaches to language, compiler, and library support for multidimensional arrays in Java*, ISCOPE Conference on ACM 2001 Java Grande, pages 116–125, 2001.
18. ej-technologies JProfiler, http://www.ej-technologies.com/products/jprofiler/overview.html