Data Structures in Java for Matrix Computations *

Geir Gundersen
Department of Informatics
University of Bergen
Norway
geirg@ii.uib.no

Trond Steihaug†
Department of Informatics
University of Bergen
Norway
Trond.Steihaug@ii.uib.no

October 15, 2002

Abstract

In this paper it is shown how to utilize Java arrays for matrix computations. We discuss the disadvantages of Java arrays when used as two-dimensional array for dense matrix computation, and how to improve the performance. We show how to create efficient dynamic data structure for sparse matrix computation using Java's native arrays. We construct a data structure for large sparse matrices that is unique for Java. This data structure is shown to be more dynamic and efficient than the traditional storage schemes for large sparse matrices. Numerical results show that this new data structure, called Java Sparse Array (JSA), is competitive with the traditionally Compressed Row Storage scheme (CRS) on matrix computation routines. Java gives flexibility without loosing efficiency. Compared to other object oriented data structures it is shown that JSA has the same flexibility.

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 (limited) numerical computations and Java is already introduced as the programming language in the introductory course in scientific computation Grunnkurs i matematiske beregninger at University of Oslo.

Matrix computation is a large and important area in scientific computation. Developing efficient algorithms for working with matrices are of considerable practical interest. Matrix multiplication is a classic example of an operation, which is very dependent on the details of the data structure. This operation is used as an example and we discuss several different implementations using Java arrays as the underlying data structure. We

†This work has been supported by the Research Council of Norway.
demonstrate the row-wise layout of a two-dimensional array and implement a straightforward matrix multiplication algorithm that takes the row-wise layout into consideration. We present a package implementation (JAMA) [1] of matrix multiplication and compare our straightforward matrix multiplication algorithm with JAMA.

We introduce the use of Java arrays for storing sparse matrices and discuss different storage formats and implementations. Java’s native arrays have a better performance inserting and retrieving elements than the utility classes java.util.Vector, java.util.ArrayList, and java.util.LinkedList [2].

The timings for the dense matrix operations where done on Solaris Ultrasparc with Sun’s Java Development Kit (JDK) 1.3.1. 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).

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 5 is implemented as Figure 4. Since each element 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 continuously, but we cannot expect the objects of an array of objects to be stored continuously. For a rectangular array of primitive elements, the elements of a row will be stored continuously, but the rows may be scattered. A basic observation is that accessing the consecutive elements in a row will be faster than accessing consecutive elements in a column.

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 the sum s of the elements in the $n \times m$ matrix A

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

The code examples in Figure 1 and 2 show two implementation 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
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];
    }
}

Figure 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];
    }
}

Figure 2: Loop-order (j,i) (column wise)

Figure 3: Time accessing the array matrix row wise and column wise

elements in a matrix (either row or column) is a common operation in matrix computation routines.

Java has no support for true two-dimensional arrays, as shown in Figure 4. Therefore
Java implements a two-dimensional array with Java's arrays of arrays, as shown in Figure 5. An element of a double[][] is a double[], that is Java arrays are array of arrays.
The double[][] is an object and its elements, double[], are objects. 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 the time differences in row and column wise loop order.

Figure 4: The representation of a true two-dimensional array.

Figure 5: The layout of a two-dimensional Java arrays.

Figure 6: A general 2D Java array with different row lengths.
Matrix Multiplication: $A \times B$

<table>
<thead>
<tr>
<th>$n = p = m$</th>
<th>Pure Row</th>
<th>Partial</th>
<th>Pure Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k,i,j)</td>
<td>(i,k,j)</td>
<td>(i,j,k)</td>
<td>(j,k,i)</td>
</tr>
<tr>
<td>80</td>
<td>66</td>
<td>63</td>
<td>66</td>
</tr>
<tr>
<td>115</td>
<td>178</td>
<td>174</td>
<td>208</td>
</tr>
<tr>
<td>138</td>
<td>298</td>
<td>257</td>
<td>331</td>
</tr>
<tr>
<td>240</td>
<td>1630</td>
<td>1538</td>
<td>2491</td>
</tr>
<tr>
<td>468</td>
<td>13690</td>
<td>13175</td>
<td>27655</td>
</tr>
</tbody>
</table>

Table 1: The SMM algorithm on input $AB$ with different loop-orders.

3 Matrix Multiplication Algorithms

Let $A$ be a $n \times m$ and $B$ be $m \times p$ matrices. The matrix product $C = AB$ is a $n \times p$ matrix with elements

$$C_{ij} = \sum_{k=0}^{m-1} A_{ik}B_{kj} \quad i = 0,1,\ldots,n-1, \quad j = 0,1,\ldots,p-1 \quad (2)$$

A straightforward implementation of (2) using Java's native arrays is given in [3, 4].

```java
for(int i = 0; i<m;i++){
    for(int j = 0;j<n;j++){
        for(int k = 0;k<p;k++){
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

By interchanging the three `for` loops there are six distinct ways of doing matrix multiplication. We can group them into pure row, pure column, and partial row/column. If for each row of $A$ the elements of $B$ are accessed row-by-row the resulting matrix $C$ is constructed row-by-row. This is a pure row loop-order denoted (i,k,j) and in the implementation the second and third `for` loop are interchanged compared to the straightforward implementation above which will be (i,j,k).

Table 1 shows the results of performing the six straightforward matrix multiplication (SMM) algorithms on $AB$. It is evident from the table that the pure column algorithms are the least efficient algorithms, while the pure row algorithms are the most efficient implementations. This is due to accessing different object arrays when traversing columns as opposed to accessing the same object array several times (when traversing a row). Differences between row and column traversing is also an issue in FORTRAN, C and C++ but the differences are not so significant.

To further improve the performance we traverse one-dimensional arrays, `double[]`, instead of two-dimensional arrays, `double[][]` in the innermost for-loops. Traversing 1D arrays instead of 2D arrays could be a factor of two more efficient [2]. The algorithm in Figure 8 with loop-order (i,k,j) was more efficient than the best effort algorithm with loop-order (k,i,j).
public Matrix times(Matrix B){
    Matrix X = new Matrix(m,B.n);
    double[][] C = X.getArray();
    double[][] Bcolj = new double[n];
    for(int j = 0; j < B.n; j++){
        for(int k = 0; k < n; k++){
            Bcolj[k] = B.A[k][j];
        }
        if(j < m) {
            double[] Arowi = A[i];
            double s = 0;
            for(int k = 0; k < n; k++){
                s += Arowi[k] * Bcolj[k];
            }
            C[i][j] = s;
        }
    }
    return X;
}

public Matrix times(Matrix B){
    Matrix X = new Matrix(m,B.n);
    double[][] C = X.getArray();
    double[][] BA = B.A;
    double[][] Arowi, Crowi, Browi;
    int Bn = B.n, Bm = B.m;
    double a = 0.0;
    int i = 0, j = 0, k = 0;
    for(i = 0; i < m; i++){
        Arowi = A[i];
        Crowi = C[i];
        for(k = 0; k < Bn; k++){
            Browi = BA[k];
            a = Arowi[k];
            for(j = Bm; j > 0; j--){
                Crowi[j] += a*Browi[j];
            }
        }
    }
    return X;
}

Figure 7: JAMA’s algorithm with loop-order (j,i,k)
Figure 8: The pure row-oriented algorithm with loop-order (i,k,j)

3.1 JAMA

JAMA[1] is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real dense matrices. It is meant to provide sufficient functionality for routine problems, packaged in a way that is natural and understandable to non-experts. It is intended to serve as the standard matrix class for Java. JAMA is comprised of six Java classes: Matrix, CholeskyDecomposition, LU Decomposition, QR Decomposition, SingularValueDecomposition and EigenvalueDecomposition.

JAMA’s matrix multiplication algorithm, the A.times(B) algorithm, is part of the Matrix class. In this algorithm the result matrix is constructed column-by-column, loop-order (j,i,k), as shown in Figure 7.

3.2 The pure row-oriented versus JAMA

In this section we compare A.times(B) of the pure row-oriented algorithm Figure 8, to JAMA’s implementation Figure 7. The pure row-oriented algorithm does not traverse the columns of any matrices involved and we have eliminated all unnecessarily declarations and initialisations. When one of the factors in the product is a vector we have a matrix vector product. If the first factor is [1][m] then we have the product \( b^T A \). If the second factor is \([n][1]\) we have \( Ab \). We use lower case to denote a (column) vector. Experiments show that there is no difference in time traversing an \([1][n]\) array compared to an \([n][1]\) array \([5]\) of primitive elements.

Figure 9 shows that JAMA’s algorithm is more efficient than the pure row-oriented algorithm on input \( Ab \) with an average factor of two. There is a significant difference
between JAMA's algorithm versus the pure row-oriented algorithm on $b^T A$ as shown in Figure 10, with an average factor of 7. In this case JAMA is less efficient. In Figure 11 a comparison is on input $AB$ is shown for square matrices. Here the pure row-oriented algorithm is better than JAMA's algorithm with an average of 30% better performance.

To find when the pure row-oriented $\text{A.times(B)}$ algorithm is achieving a better performance than JAMA's algorithm we compare the algorithms with input $AB_p$. Matrix $B_p$ has the dimension $m \times p$ where $p = 1, 2, 3, \ldots, m$. We are expecting a break even $p^*$ since increasing the number of columns in the $B_p$ matrix from $p = 1$ when JAMA was most efficient, we are getting closer to a $AB$ operation on square matrices where the pure row-oriented algorithm is more efficient. Table 2 shows break even for small values of $p^*$. The pure row-oriented $\text{A.times(B)}$ algorithm was more efficient for all $p$ larger than the break even.

<table>
<thead>
<tr>
<th>$m$</th>
<th>80</th>
<th>115</th>
<th>138</th>
<th>240</th>
<th>468</th>
<th>663</th>
<th>765</th>
<th>817</th>
<th>900</th>
<th>1000</th>
<th>1374</th>
<th>1500</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^*$</td>
<td>13</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2: $A$ is $m \times m$ and JAMA is most efficient when $p \leq p^*$. 

102
4 Sparse Matrices

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 and unstructured 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 [4]. The compressed storage schemes have minimal memory requirements and have shown to be convenient for several important operations. For matrices with special structures like symmetry the storage schemes must be modified.

Currently there is no released packages implemented in Java for numerical computation on sparse matrices, as complete as JAMA and JAMPACK [1, 7] for dense matrices. But there are separate algorithms like [8] 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. [9] use the coordinate storage format as implemented in C++ in [10]. The coordinate storage format is not an efficient storage format for large sparse compared to compressed row format [5]. There are also some benchmark algorithms like [12] that performs sparse matrix computations using compressed row storage scheme.

In the next sections we will introduce three sparse storage schemes, Compressed Row Storage, Java Sparse Arrays and Sparse Matrix Concept. We will discuss them on the basis of performance and flexibility.

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

All the sparse matrices used as test matrices in this paper where taken from Matrix Market [13]. All the matrices are square and classified as general with no properties or structures where there can be used special storage schemes.

4.1 Compressed Storage Schemes

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in continuous locations. For a sparse matrix we create three vectors: one for the double type (value) and the other two for integers (columnindex, rowpointer). The double type in Java uses 64 bits for storing each element and the int type in Java uses 32 bits for its elements. 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. Let \( n \) be the number of rows and \( m \) be the number of columns. If \( \text{value}[k] = A_{ij} \) then \( \text{columnindex}[k] = j \) and \( \text{rowpointer}[k] < k < \text{rowpointer}[k + 1] \). By convention \( \text{rowpointer}[n] = \text{nnz} \), where \( \text{nnz} \) is the number of nonzero elements in the matrix. The storage savings for this approach is significant for sparse matrices. Instead of storing \( n \cdot m \) elements, we only need \( 2\text{nnz} + n + 1 \) storage
locations. The compressed column storage format is basically CRS on $A^T$. Consider the sparse $6 \times 6$ matrix $A$.

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

The nonzero structure of the matrix $A$ (3) 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};

4.2 Java Sparse Array

The Java Sparse Array format is a new concept for storing sparse matrices made possible with Java. This concept is illustrated in Figure 12. This unique concept uses an array of arrays where each array is an object.

There are two arrays, one for storing the references to the value arrays (one for each row) and one for storing the references to the index arrays (one for each row).

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 consists of a value and an index array each with its own unique reference. Java Sparse Array use $2mnz + 2n$ storage locations compared to $2mnz + n + 1$ for the CRS format.

The nonzero structure of the matrix $A$ (3) is stored as follows in Java Sparse Array.

double[] value = {{10,-2}, {3,9}, {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.

```java
public class JavaSparseArray{
    private double[][] Avalue;
```
private int[][] Aindex;
public JavaSparseArray(double[][] Avalue, int[][] Aindex) {
    this.Avalue = Avalue;
    this.Aindex = 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. We will see that this structure can compete with CRS when it comes to performance and memory use.

4.3 The Sparse Matrix Concept

The Sparse Matrix Concept is a general object-oriented structure illustrated in Figure 13. It is similar to JSA, but it does not take advantage of the feature that Java’s native arrays are true objects. The Sparse Matrix Concept can be implemented in the following way [11].

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;
    }
}

The actual storing is the same for SMC and JSA, but JSA does not use the extra object layer for each row.

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. However, JSA is preferable compared to any of the two implementations of the Sparse Matrix Concept, since they have the same flexibility, without the extra object layer.

4.4 Sparse Matrix Multiplication

The problems with performing a matrix multiplication algorithm on CRS, is that we do not know the actual size (m*n) or structure of the resulting matrix. This structure can for
general matrices be found by using the datastructures of $A$ and $B$. The implementation
used is based on FORTRAN routines [6, 14] using Java’s native arrays.

The nonzero structure, the index and value array, are created with an a priori size. The row
pointer can be created with a fixed size since it is the size of the row dimension
of $A$ ($C=AB$), plus one extra element. The CRS approach when we know the exact
size ($nnz$) performs well. It has not surprisingly, the best performance of all the matrix
multiplication algorithms we present in this paper, see Table 3. However, it is not a
practical solution, since a priori information is usually not available. If no information
is available on the number of nonzeros, a ”good guess” is needed or a symbolic phase to
compute the number of nonzero elements in the product. The next paragraphs show
how to perform the matrix multiplication operation using Java Sparse Array format. This
format addresses the problem of inserting rows without needing to update the rest of the
nonzero structure.

Java Sparse Array is in a row-oriented manner and the matrix multiplication algorithm
is performed with the loop-order (i,j,k). The matrices involved are traversed row-wise and
the resulting matrix is created row-by-row.

For each row of matrix $A$ 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 $\texttt{double [n]}$ [1] and $\texttt{int [n]}$ [1] to contain the rows of the resulted
elements of matrix $C$, before we actually creates the actual rows of the resulting matrix.

The key to this approach is the the SPARSEKIT approach [14]. Considering the memory
use this approach uses three additional accumulators (two $\texttt{int []}$ and one $\texttt{double []}$)
and creates two new array objects for each new row.

When comparing CRS with a symbolic phase on $AB$ to JSA we see that JSA is
significantly more efficient than CRS with an average factor of 3.54. This CRS algorithm
is the most realistic considering a package implementation, and therefore the most realistic
comparison to the JSA timings. When comparing CRS with a priori information on $AB$
to Java Sparse Array we see that the CRS approach is slightly more efficient with an
average factor of 1.55 for moderately sized structures. It is important to state that we
cannot draw too general conclusions on the performance of these two 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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$nnz(A)$</th>
<th>$nnz(C)$</th>
<th>CRS</th>
<th>CRS (a priori)</th>
<th>JSA</th>
<th>SMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>115</td>
<td>421</td>
<td>1027</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>468</td>
<td>2820</td>
<td>8920</td>
<td>17</td>
<td>11</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>2205</td>
<td>14133</td>
<td>46199</td>
<td>58</td>
<td>14</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>4884</td>
<td>147631</td>
<td>473734</td>
<td>322</td>
<td>125</td>
<td>169</td>
<td>165</td>
</tr>
<tr>
<td>10974</td>
<td>219512</td>
<td>620957</td>
<td>1067</td>
<td>161</td>
<td>228</td>
<td>278</td>
</tr>
<tr>
<td>17282</td>
<td>553956</td>
<td>2525937</td>
<td>2401</td>
<td>395</td>
<td>642</td>
<td>628</td>
</tr>
</tbody>
</table>

Table 3: The CRS and JSA algorithms for $C = AA$. 

106
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.

4.5 Sparse Matrix Update

Consider the outer product $ab^T$ of the two vectors $a, b \in \mathbb{R}^n$ 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 $n \times n$ matrix $A$.

$$A_{ij} = A_{ij} + a_i b_j^T \quad i = 0, 1, \ldots, n - 1, \quad j = 0, 1, \ldots, n - 1$$

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 4 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 proportional to the number of elements thus accounting for the major difference between CRS and JSA on matrix updates.

5 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 [15]. 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 [16]. For sparse matrices we have illustrated the effect of having the possibility to manipulate only the rows of the structure without updating the rest of the structure. We still do not know if Java Sparse Array could replace CRS for all algorithms, this is worth further research.
References


