CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2004; **16**:799–815 (DOI: 10.1002/cpe.793)

Data structures in Java for matrix computations

Geir Gundersen and Trond Steihaug*,†

Department of Informatics, University of Bergen, Norway

SUMMARY

In this paper we show how to utilize Java's native arrays for matrix computations. The disadvantages of Java arrays used as a 2D array for dense matrix computation are discussed and ways to improve the performance are examined. We show how to create efficient dynamic data structures for sparse matrix computations using Java's native arrays. This data structure is unique for Java and shown to be more dynamic and efficient than the traditional storage schemes for large sparse matrices. Numerical testing indicates that this new data structure, called Java Sparse Array, is competitive with the traditional Compressed Row Storage scheme on matrix computation routines. Java gives increased flexibility without losing efficiency. Compared with other object-oriented data structures Java Sparse Array is shown to have the same flexibility. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: Java; linear algebra; data structures; sparse matrices

INTRODUCTION

Object-oriented programming has been favored in the last decade(s) and has an easy to understand paradigm. It is straightforward to build large scale applications designed in an object-oriented manner. Java's widespread acceptance in commercial applications, and the fact that it is the main language introduced to students, insure that it will increasingly be used for numerical computations [1,2].

Matrix computation is a large and important area in scientific computation. Developing efficient algorithms for working with matrices is of considerable practical interest. Matrix multiplication is a classic example of an operation, which is 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 demonstrate the well-known row-wise layout of a 2D array and

*Correspondence to: Trond Steihaug, Department of Informatics, University of Bergen, Postbox 7800, N-5020 Bergen, Norway.
†E-mail: trond.steihaug@ii.uib.no

Contract/grant sponsor: Research Council of Norway

Received 10 December 2002
Revised 5 May 2003
Accepted 24 July 2003

Table I. Testing environments.

	Sun Ultrasparc	Dell
Processor	Ultra 1	Intel Pentium 4
Processor speed	300 MHz	2.26 GHz
Memory	128 MB	500 MB
Operating System	Solaris	Red Hat Linux
JVM	Sun 1.3.1	Sun 1.4.0/1.4.1.01, IBM 1.3.1
JIT Enabled	Yes	Yes

implement straightforward matrix multiplication algorithms that take this layout into consideration. Implementation of dense matrix multiplication is also illustrated using the package JAMA [3]. Even in such a simple computational task, no dense matrix multiplication algorithm can be expected to be superior on all possible input parameters and on different computers, operating systems and Java Development Kit (JDK)/Java Virtual Machines (JVMs). The testing environments used in this paper are presented in Table I and the timings in the numerical testing are in milliseconds.

This paper focus on utilizing Java's native arrays for numerical operations. Java's native arrays have not been regarded as efficient enough for high-performance computing. Replacing the native arrays with a multidimensional array [4,5], and extensions like Spar [6] have been suggested. However, packages like JAMA [3], JAMPACK [7] and Colt [8] use the native arrays as the underlying storage format. Investigating how to utilize the flexibility of Java arrays for creating elegant and efficient algorithms is of great practical interest.

A sparse matrix is usually described 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 non-zero elements [9]. The difficulty is that sparse data structures include more overhead since indexes as well as numerical values of non-zero matrix entries are stored. 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 [9,10].

We introduce the use of Java arrays for storing sparse matrices and discuss different implementations. We discuss the Compressed Row Storage (CRS) and two flexible implementations made possible in Java: Java Sparse Array (JSA), and Sparse Matrix Concept (SMC). These storage schemes are discussed on the basis of performance and flexibility. The compressed storage schemes can be implemented in most languages, while the SMC is restricted to object-oriented languages. JSA is new and unique for languages that allow jagged arrays like C# and Java (see [11] and the discussion therein).

JAVA ARRAYS

Java's native arrays are objects, inherit the `Object` class and are handled with references. However, Java's native arrays is not an extendable class, and adding self-defined behavior to an array we must

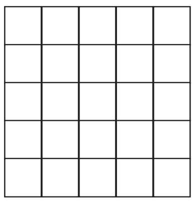
Figure 1. *True* 2D array.

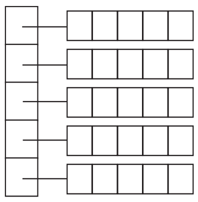
Figure 2. 2D Java array.

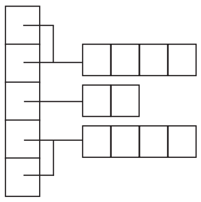
Figure 3. General Java array.

use a wrapper/enclosing class. This is also an issue for primitive types like `Double`, `Integer` and `Boolean` API classes that enclose the primitive types. Java's native arrays can in some cases be seen as a hybrid between an object and a primitive. The object nature of Java arrays imposes overhead on Java applications compared with 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 1 is implemented as Figure 2. 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 3.

We can expect elements of an array of primitive elements to be stored contiguously, but we cannot expect the objects of an array of objects to be stored contiguously. For a rectangular array of primitive elements, the elements of a row will be stored contiguously, 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 be referenced as `A[i][j]`. A vector is either a matrix with only one column (column vector) or one row (row vector).

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

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij} \tag{1}$$

This is the basic operation in computing the square of the Frobenius-norm $\|A\|_F^2 = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij}^2$ of matrix A.

The code examples in Figures 4 and 5 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 6 shows that traversing columns is much less efficient than traversing rows when the square matrix gets larger. This demonstrates the basic observation that accessing the consecutive elements in a row is faster than accessing consecutive elements in a column.

Following the 'do's and don'ts for numerical computing in Java' [12] and [13] the performance improves if 1D arrays, `double[]`, are traversed instead of 2D arrays, `double[][]`.

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

Traversing a 1D array is an average of 30% more efficient than traversing a 2D array as shown in Figure 6. This is referred to as upacked. If the dense $m \times n$ array is copied to an mn array [14] a minor improvement is found using the Sun JDK, while using IBM JDK 1.3.1 the gain is an average factor of three.

Differences between row and column traversing is also an issue in FORTRAN, C and C++. For FORTRAN 77 and ANSI C the average factor is around two while for Java the average factor is five. Since Java implements a 2D array with array of arrays, as shown in Figure 2, the elements of the object `double[][]` are objects `double[]`. When an object is created and gets heap allocated, the object can be placed anywhere in the memory. These issues partially explain the increase in time differences in row- and column-wise loop order in Java compared with FORTRAN, C and C++.

```
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 4. 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 5. Loop-order (j, i), column wise.

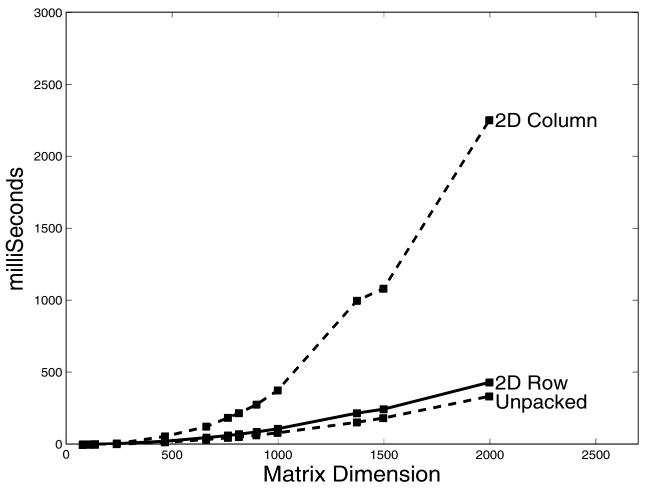

Figure 6. Time accessing a 2D Java array row wise and column wise $(m = n)$.

Table II. The SMM algorithm on input AB with different loop-orders. The time is in milliseconds and the measuring is done on Red Hat Linux with Sun's JDK 1.4.1.01.

	Matrix multiplication: `A.times(B)`					
	Inner row		Mixed		Inner column	
$m = n = p$	(k, i, j)	(i, k, j)	(i, j, k)	(j, i, k)	(j, k, i)	(k, j, i)
80	13	12	13	14	14	14
115	25	24	29	35	30	28
138	37	36	47	46	70	62
240	205	190	323	297	408	413
468	1478	1319	3061	2461	4715	4716

MATRIX MULTIPLICATION ALGORITHMS

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

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

A straightforward implementation of (2) using Java's native arrays is like the following code.

```
for(int i = 0; i<m;i++){
    for(int j = 0;j<p;j++){
        for(int k = 0;k<n;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. These can be divided into three groups based on the innermost loop: inner row, mixed column and row, and inner 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). In the implementation the second and third for-loops are interchanged compared with the straightforward implementation above which will be (i, j, k), where the inner loop is an innerproduct of a row and a column.

Table II shows the results of performing the six straightforward matrix multiplication algorithms on AB. It is evident from the table that the *inner column* algorithms are the least efficient algorithms, while the *inner 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).

The pure inner row-oriented algorithm with loop-order (i, k, j) will be more efficient than the inner row-oriented algorithm with loop-order (k, i, j), since the latter traverses the columns of matrix A in the loop-body of the second for-loop.

JAMA

JAMA [3] 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 composed of six Java classes where JAMA's matrix multiplication algorithm, the `A.times(B)` algorithm, is part of the `Matrix` class. In this algorithm the resulting matrix is constructed column-by-column, loop-order (j, i, k), as shown in Figure 7.

The pure row-oriented versus mixed

In this section we compare `A.times(B)` of the pure row-oriented algorithm in Figure 8 to JAMA's implementation in Figure 7. The pure row-oriented algorithm does not traverse the columns of any matrices involved and we have eliminated all unnecessary declarations and initialization. When one of the factors in the product is a vector we have a matrix vector product. We use lower case to denote a (column) vector. If the first factor is $1 \times n$ we have the product $b^T A$. If the second factor is $n \times 1$ we have Ab. Experiments show that there is only a minor difference in time traversing a `double [1][n]` array compared with a `double [m][1]` array [15].

Figure 9 shows that JAMA's algorithm is slightly more efficient than the pure row-oriented algorithm on input Ab. There is a significant difference between JAMA's algorithm compared with the pure row-oriented algorithm on $b^T A$. Figure 10 shows that JAMA is a an average factor of 7 slower than pure row. This is consistent for the whole testing environment (Table I).

In Figures 11 and 12 a comparison on input AB is shown for square matrices. In Figure 11 the pure row-oriented algorithm is better than JAMA's algorithm, with an average factor of 2 on Sun JDK 1.4.0. However, on IBM JDK 1.3.1, JAMA's algorithm is on average 20% more efficient than the pure row-oriented algorithm as shown in Figure 12. This difference is due to the cost of the inner-most loop constructions. Since these two constructions are executed n^3 times, the difference in execution time will have a significant impact on the difference in performance between JAMA and pure row-oriented algorithms. On Sun JDK 1.4.0 and 1.4.1 the ratio of execution times for `s+=Arowi[k]*Bcolj[k]` and `Crowi[j] += a*Browk[j]` was on average 2, explaining why the pure row algorithm was the most efficient of the two. However, for IBM JDK 1.3.1 the factor was 0.8.

With unpacking an array of arrays in an algorithm we mean that we only traverse 1D arrays in the for-loop where the actual computation is done. We do this since an array of arrays has array bounds checking for both references and values compared with 1D arrays which only has array bounds checking for values. We unpacked *all* the straightforward matrix multiplication routines as illustrated for pure row loop-order (i, k, j) and JAMA's mixed order (j, i, k) and none of the unpacked versions was slower than the packed versions, with consistent results as in Table II for the whole testing environment. Of the two mixed algorithms the algorithm with loop-order (i, j, k) is more efficient than the algorithm with loop-order (j, i, k), since the elements in the columns of matrix `B` are accumulated into a 1D array in the first for-loop. Of the two column algorithms, the pure column algorithm was

```
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];
      }
      for(int i = 0;i<m;i++){
         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;
}
```

Figure 7. JAMA loop-order (j, i, k).

```
public Matrix times(Matrix B){
   Matrix X = new Matrix(m,B.n);
   double[][] C = X.getArray();
   double[] Arowi, Crowi, Browk;
   for(int i = 0;i<m;i++){
      Arowi = A[i];
      Crowi = C[i];
      for(int k = 0;k<B.m;k++){
         Browk = B.A[k];
         double a = Arowi[k];
         for(int j = 0;j<B.n;j++){
            Crowi[j] += a*Browk[j];
         }
       }
    }
    return X;
}
```

Figure 8. Pure row-oriented loop-order (i, k, j).

most efficient. The pure column algorithm accumulates the columns of matrix `C` into a 1D array and updates the `C` matrix in the loop body of the first for-loop.

Extracting a row from a 2D array to a 1D array is a $\Theta(1)$ operation (`double[] Arowi = A[i]`, where `A` is of type `double[][]`). But extracting a column is $\Theta(m)$, since we must copy all the elements from matrix `A` to a 1D array in a loop.

Matrix vector products are $\Theta(n^2)$ operations while the matrix–matrix products are $\Theta(n^3)$ operations. On average we have a 50% gain in performance if we use the algorithm for matrix vector products instead of the more general algorithms for matrix–matrix products that treats vectors as matrices.

SPARSE MATRICES

There is a released package implemented in Java for numerical computation on sparse matrices, called Colt [8], and there are separate algorithms like [16] using a coordinate storage scheme. The coordinate storage scheme is the most straightforward structure to represent a sparse matrix; it simply records each non-zero entry together with its row and column index. In [17], the coordinate storage format as implemented in C++ in [18] is used. The coordinate storage format is not an efficient storage format for large sparse compared with compressed row format [15]. There are also some benchmark algorithms like [19] that perform sparse matrix computations using a compressed row storage scheme. For matrices with special structures like symmetry the storage schemes must be modified.

All the sparse matrices used as test matrices in this paper were taken from Matrix Market [20]. All the matrices are square and symmetry has not been utilized.

COMPRESSED STORAGE SCHEMES

The CRS format puts the non-zero of a given row in contiguous 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 non-zero elements of the matrix, as they are traversed in a row-wise fashion. The `columnindex` vector stores the indexes of the corresponding elements in the `value` vector. The `rowpointer` vector stores the locations in the `value` vector that start a row. Let m be the number of rows and n be the number of columns. If `value[k]` $= A_{ij}$ then `columnindex[k]`$= j$ and `rowpointer[`i`]` $\leq$ `k` $<$ `rowpointer[`$i+1$`]`. By convention `rowpointer[`n`]` $= nnz$, where nnz is the number of non-zero elements in the matrix. The compressed column storage format is basically CRS on A^T. Consider the sparse 6×6 matrix A [21].

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

The non-zero structure of the matrix (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};
```

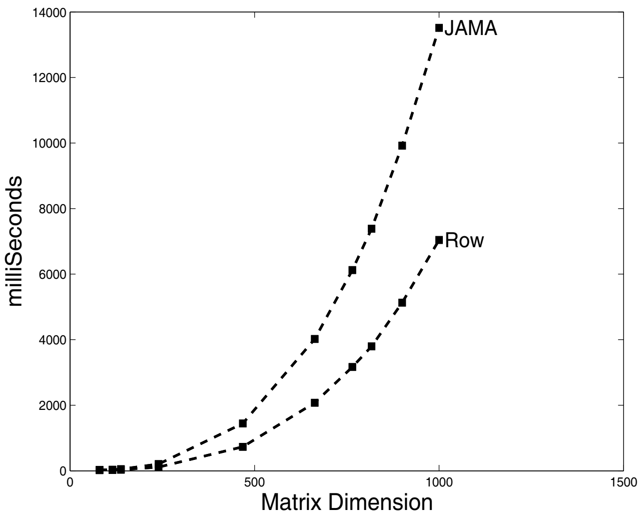

Figure 9. Sun 1.4.1.01 on input Ab.

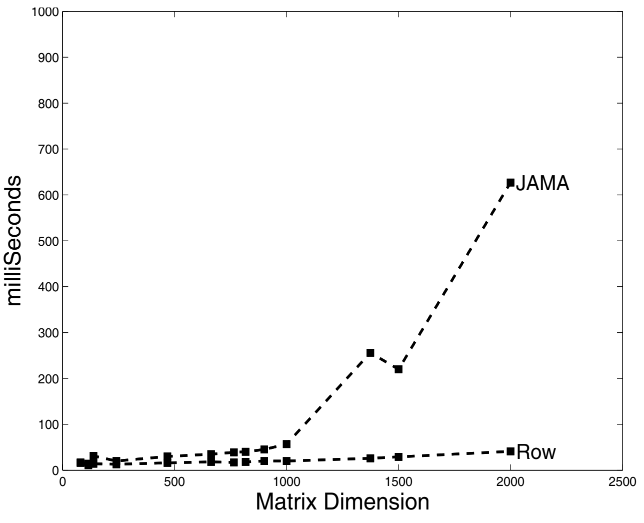

Figure 10. Sun 1.4.1.01 on input $b^T A$.

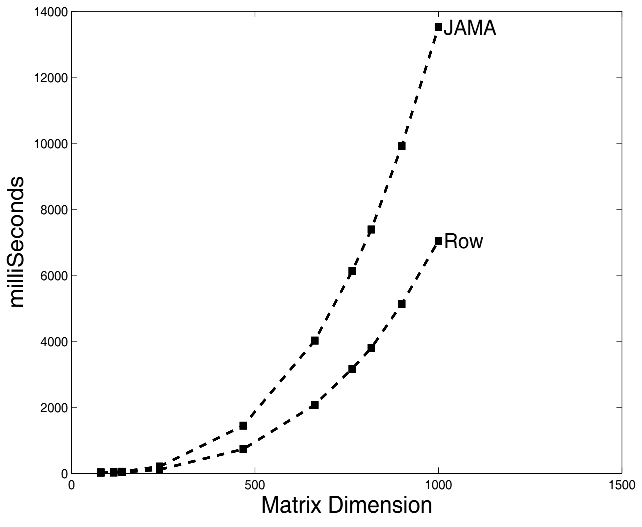

Figure 11. Sun 1.4.1.01 on input AB.

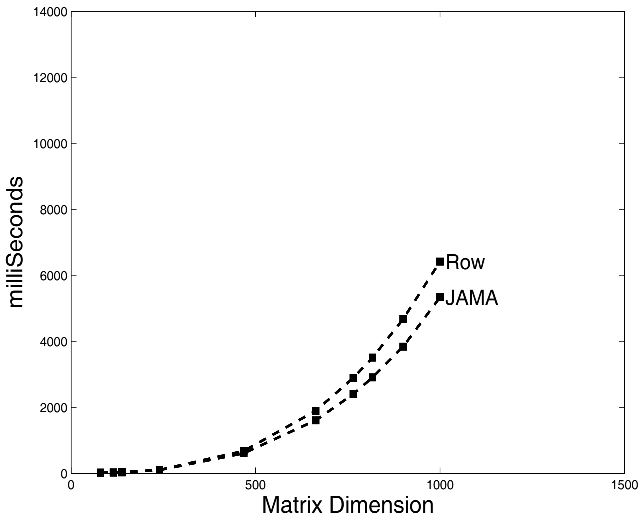

Figure 12. IBM 1.3.1 on input AB.

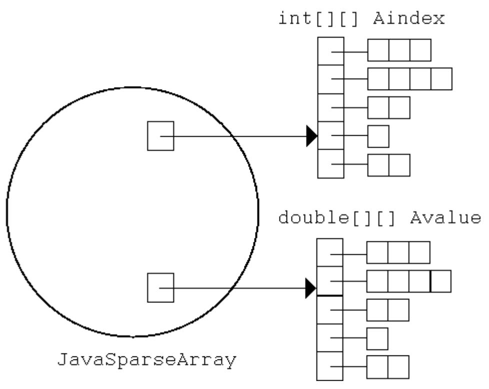

Figure 13. The JSA format.

JAVA SPARSE ARRAY

The JSA format is a new concept for storing sparse matrices made possible with Java and C#, which both support jagged arrays implementation. This concept is illustrated in Figure 13. This unique concept 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 corresponding index arrays (one for each row).

With the JSA format it is possible to manipulate the rows independently without updating the rest of the structure, as would have been necessary with CRS. This means a row can be removed or inserted into the JSA structure without creating a new large 1D array for values and indexes. Each row consists of a value and a corresponding index array each with its own unique reference. JSA uses $2nnz + 2n$ storage locations compared with $2nnz + n + 1$ for the CRS format.

The non-zero structure of the matrix (3) is stored as follows in JSA.

```
double[][] Avalue = {{10,-2},{3,9,3},{7,8,7},{3,8,7,5},{8,9,9,13},{4,2,-1}};
int[][] Aindex = {{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.Aindex = Aindex;
  }
  public JavaSparseArray times(JavaSparseArray B){...}
}
```

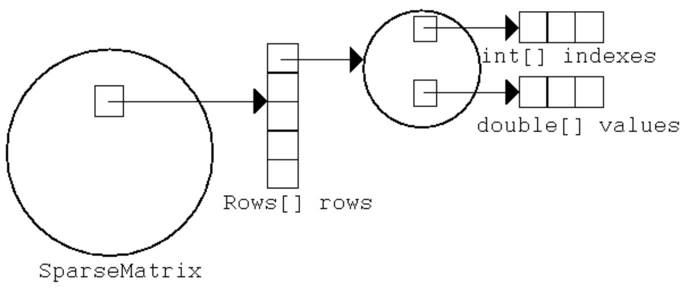

Figure 14. The SMC.

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.

AN OBJECT-ORIENTED APPROACH FOR STORING SPARSE MATRICES

A more object-oriented approach for creating a row-oriented data structure for storing sparse matrices can be made [22]. An example of such a data structure, SMC [23], is shown in Figure 14. Each row in the SMC is an object which can be manipulated (removed/inserted) without effecting the rest of the structure. As a data structure for storing the rows we can, for example, use `java.util.Vector`, `java.util.ArrayList` or `java.util.LinkedList` to store the `Rows` objects.

SMC can be implemented in the following way.

```
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;
      }
}
```

Table III. The matrix multiplication algorithms for $C = AA$. The time is in milliseconds and the measuring is done on Red Hat Linux with Sun's JDK 1.4.0.

Name	$m = n$	$nnz(A)$	$nnz(C)$	CRS	CRS (*a priori*)	JSA	SMC
GRE 115	115	421	1027	1	0	0	1
NOS5	468	2820	8920	19	11	11	13
ORSREG 1	2205	14 133	46 199	21	14	19	19
BCSSTK16	4884	147 631	473 734	185	125	130	130
BCSSTK17	10 974	219 512	620 957	207	161	160	158
E40R0100	17 281	553 956	2 525 937	829	395	600	545

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 SMC very similar to JSA. If all the methods that work explicitly on the instances of `Rows` are in the `Rows` class the efficiency is improved.

The actual storage in terms of primitive elements is the same for SMC and JSA, but JSA does not use the extra object layer for each row creating extra references ($2m + 2$ compared with $3m + 1$). The object creation profiler JProfiler [24] is used to analyze the memory usage of JSA and SMC for a sparse matrix multiplication. For $n = 17\,281$ the profiler indicates that in SMC the `Rows` object causes 2.3% of the overall memory use. SMC uses 1% more memory than JSA.

SPARSE MATRIX MULTIPLICATION

On performing a matrix multiplication algorithm on CRS, we do not know the actual size (nnz) or structure of the resulting matrix. This structure can for general matrices be found by using the data structures of A and B. The implementation used is based on FORTRAN routines [9,25] using Java's native arrays.

The CRS approach when we know the exact size (nnz) of C performs well. It does have, not surprisingly, the best performance of all the matrix multiplication algorithms we present in this paper, see Table III. However, it is not a practical solution, since *a priori* information is *usually* not available. If no information is available on the number of non-zero elements in C, a 'good guess' or a symbolic phase is needed to compute it.

JSA is in a row-oriented manner and the matrix multiplication algorithm is performed with the loop-order (i, k, j). 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. An element in the value array of A is multiplied by the value array of B. The result is accumulated in a row of C, indexed by the index value of the element in the B row. We can construct `double[m][1]` and `int[m][1]` to contain the rows of the resulting elements of matrix C, before we actually create the actual rows of the resulting matrix.

Table IV. The matrix update for $A = A + B$, $B = ab^T$. The time is in milliseconds and the measuring is done on Red Hat Linux with Sun's JDK 1.4.0.

$m = n$	$nnz(A)$	$nnz(B)$	$nnz(\text{new}A)$	CRS	JSA	SMC
115	421	7	426	11	0	0
468	2820	148	2963	13	1	1
2205	14 133	449	14 557	44	8	3
4884	147 631	2365	149 942	183	8	9
10 974	219 512	1350	219 812	753	8	8
17 281	553 956	324	554 138	1806	11	11

When comparing CRS with a symbolic phase on AB with JSA, we see that JSA is slightly more efficient than CRS with an average factor of 1.4. This CRS algorithm is the most realistic considering a package implementation, and therefore the most realistic comparison to the JSA timings. The SMC approach has to create a `Rows` object for each value and index array in the resulting matrix, in addition to always accessing the value and index array from method calls. 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 of their overall performance and that matrix multiplication on JSA is both fast and reliable. The most natural notation of retrieving an element from a matrix A is `A[i][k]`. This notation is possible with JSA, unlike SMC, which first has to get the `Rows` object `i`, then the element `k` by `A.rows[i].elementAt(k)`.

Java's native arrays, whose elements are of primitive types, have better performance inserting and retrieving elements than the utility classes `java.util.Vector`, `java.util.ArrayList`, and `java.util.LinkedList` [13].

SPARSE MATRIX UPDATE

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

$$A_{ij} = A_{ij} + a_i b_j, \quad i = 0, 1, \ldots, m-1, \; j = 0, 1, \ldots, n-1 \tag{4}$$

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. The update method (sometimes referred to as additionEquals) updates row by row. For JSA we only need to copy data for the row we are updating to a larger resulting row array. Using CRS we must copy the whole non-zero structure over to a larger CRS structure (creating a new `value` and `columnindex` array); this must be done for each row. This extensive copying creates the overhead for update on CRS compared with update on JSA. This is clearly shown in Table IV where 10% of the elements in a are non-zero. 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.

CONCLUSIONS AND FUTURE WORK

We need to consider the row-wise layout when we use Java arrays as 2D arrays. This well-known fact is easily illustrated with the matrix sum example. To eliminate this effect arrays must either be unpacked or the algorithms must be row-oriented. For sparse matrices we have illustrated this manipulating only the rows of the structure without updating the rest of the structure. This excludes CRS in favor of JSA and SMC as the preferable sparse matrix data structures in Java. The SMC has a unique implementation in Java, as JSA. We know that JSA could replace CRS for most algorithms. However, the benefits of JSA compared with SMC need further studies. Block matrices are fundamental in numerical linear algebra and will be an extension of this work. However, this will have to be a more object-oriented approach.

REFERENCES

1. Bischof CH, Bücker HM, Henrichs J, Lang B. Hands-on training for undergraduates in high-performance computing using Java. *Applied Parallel Computing: New Paradigms for HPC in Industry and Academia*, Sørevik T, Manne F, Moe R, Gebremedhin AH (eds.), *Proceedings of the 5th International Workshop, PARA 2000*, Bergen, Norway, June 2000 (*Lecture Notes in Computer Science*, vol. 1947). Springer: Berlin, 2001; 306–315.
2. Langtangen HP, Tveito A. How should we prepare the students of science and technology for a life in the computer age? *Mathematics Unlimited—2001 and Beyond*, Enquist B, Schmid W (eds.). Springer: Berlin, 2000; 809–826.
3. Hicklin J, Moler C, Webb P, Boisvert RF, Miller B, Pozo R, Remington K. JAMA: A Java matrix package. http://math.nist.gov/javanumerics/jama [5 May 2003].
4. Moreira JE, Midkiff SP, Gupta M. Supporting multidimensional arrays in Java. *Concurrency and Computation: Practice and Experience* 2003; **15**(3–5):317–340.
5. Gosling J. The evolution of numerical computing in Java. *Technical Memorandum*, Sun Microsystems Laboratories, 1997.
6. van Reeuwijk K, Kuijlman F, Sips HJ. Spar: A set of extensions to Java for scientific computation. *Concurrency and Computation: Practice and Experience* 2003; **15**(3–5):277–299.
7. Stewart GW. JAMPACK: A package for matrix computations. ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html [5 May 2003].
8. The Colt Distribution. Open source libraries for high performance scientific and technical computing in Java. http://hoschek.home.cern.ch/hoschek/colt/ [5 May 2003].
9. Pissanetsky S. *Sparse Matrix Technology*. Academic Press: Boston, MA, 1984.
10. Saad Y. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company: Boston, MA, 1996.
11. Thiruvathukal GK. Java at middle age: Enabling Java for computational science. *Computing in Science and Engineering* 2002; **4**(1):74–84.
12. Boisvert RF, Moreira J, Philippsen M, Pozo R. Java and numerical computing. *Computing in Science and Engineering* 2001; **3**(2):18–24.
13. Shirazi J. *Java Performance Tuning*. O'Reilly: Cambridge, MA, 2000.
14. Blount B, Chatterjee S. An evaluation of Java for numerical computing. *Scientific Programming* 1999; **7**(2):97–110.
15. Gundersen G. The use of Java arrays in matrix computation. *Candidatus Scientarium (Master in Science) Thesis*, Department of Informatics, University of Bergen, Bergen, Norway, 2002.
16. Java Numerical Toolkit. http://math.nist.gov/jnt [5 May 2003].
17. Chang R-G, Chen C-W, Chuang T-R, Lee JK. Towards automatic support of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience* 1997; **9**:1101–1111.
18. Pozo R, Remington K, Lumsdaine A. SparseLib++ version 1.5 Sparse Matrix reference guide. *Technical Report*, Mathematical and Computational Sciences Division, National Institute of Standards and Technology, MD, April 1996.
19. SciMark 2.0. http://math.nist.gov/scimark2/about.html [5 May 2003].

20. Matrix Market. http://math.nist.gov/MatrixMarket [5 May 2003].
21. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, van der Vorst H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Publications: Philadelphia, PA, 1994.
22. LASPack Reference Manual. http://www.tu-dresden.de/mwism/skalicky/laspack/laspack.html [5 May 2003].
23. Nelisse A, Maassen J, Kielmann T, Bal HE. CCJ: Object-based message passing and collective communication in Java. *Concurrency and Computation: Practice and Experience* 2003; **15**(3–5):341–369.
24. ej-technologies. JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html [5 May 2003].
25. Saad Y. SPARSEKIT: A basic tool kit for sparse matrix computations, version 2. *Technical Report*, Computer Science Department, University of Minnesota, June 1994.