On the Efficiency of Arrays in C, C# and Java

Geir Gundersen and Trond Steihaug
Department of Informatics, University of Bergen
Norway
{geirg, Trond Steihaug}@ii.uib.no

12 October, 2004

Abstract

In this paper we will discuss the efficiency and flexibility of linear, jagged and multidimensional arrays for several numerical linear algebra algorithms. We make a comparison of these algorithms on two platforms using the languages C, C# and Java. Numerical results shows that there is no loss in using more dynamic data structures as jagged arrays instead of linear array. Numerical results also shows that there is no loss in going from C to Java and C#.

1 Introduction

Matrix computations and efficiency is of utmost importance for high performance computing, it is therefore crucial that algorithms and data structures on basic linear algebra routines are efficient

A matrix is a rectangular array of numbers as shown in (1). We have implemented several basic linear algebra operations and will in this paper focus on three operations: matrix vector products, LU decomposition and element sum for three-dimensional matrices.

Arrays are the most important data structure for storing matrices and vectors in a numerical context. In this paper we will investigate three different data structures for storing vectors and two-and three-dimensional matrices they are linear, jagged (array of arrays) and multidimensional arrays. We perform our experiments using the languages C, C# and Java.

Multidimensional arrays can be implemented as array of arrays in the languages C, C# and Java. $True$ multidimensional arrays are only possible in C and C#. Jagged arrays in Java are considered inefficient for numerical computing [11]. Many proposals has been made to replace Java's native arrays for numerical computing and two of the most prominent proposals are the MultiArray package from IBM [7] and the multidimensional array implementation in Spar [8]. Both these suggestions would demand extensions to the Java language. A practical and simple proposal for storing matrices, is to use a single linear (or one-dimensional) array instead of Java's native two-dimensional arrays of arrays [1]. This proposal is not yet fully explored but we will give some preliminary results using linear arrays.

[1] On a page previously available at java.sun.com by James Gosling titled "The Evolution of Numerical Computing in Java".

The language C# developed by Microsoft have both jagged and *true* multidimensional arrays [1]. In [12] there are reported several benchmarks where jagged arrays outperforms rectangular or multidimensional arrays in C#. We will also show similar results for numerical algorithms.

It has also been a common belief that C would outperform Java significantly on most numerical intensive algorithm, while this might be true for early releases of Java [2], we will show numerical results that there is not a significant loss in going from C to Java and C# for various numerical linear algebra algorithms.

A basic difficulty with the static structures, as linear and multidimensional arrays, is associated with inserting new elements. This arises for operations like matrix updates and matrix decomposition where back-fill or fill-ins are created. To illustrate the additional flexibility we get from using jagged arrays instead of linear or multidimensional arrays, we address the problem of fill-ins during an LU decomposition for variable banded matrices. Banded matrices appears in many applications [3, 4].

2 Basic Storage Schemes for Matrices

A matrix is a rectangular array of numbers. The numbers in the array are called the entries in the matrix.

A general matrix may be written as

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}. \tag{1}$$

Where $A \in \Re^{m \times n}$ and the entry in row i and column j is A_{ij} of the matrix A. To be consistent with Java first row and column index is 0. A vector is a matrix with only one row or column, and we denote a vector with a lower case letter. Arrays are the most common storage for matrices in C, C# and Java.

In a jagged array the content of the array are arrays which may hold instances of a type or references to other arrays. Jagged arrays in C, C# and Java can be created as a one-dimensional array of arrays, and has the syntax `[][]...[]`, thus multidimensional arrays can therefore be implemented as array of arrays. Jagged arrays in C# and Java have the same syntax, as seen in the code below. Jagged arrays in C are created with pointers `double **A` and with `malloc` memory is allocated for the array, as in the below code example.

However, in C and C# one can also define *true* multidimensional arrays, the rows of *true* multidimensional arrays must be of uniform length. The multidimensional arrays in C are similar to the creation of multidimensional arrays in Java, `double A[m][n]`, without the `new` operator. The syntax for multidimensional arrays in C# are `[,,...,]` [1]. The memory layout for multidimensional arrays in C# are similar to statically created multidimensional arrays in C. That is, the memory of a multidimensional array are laid out as a contiguous block containing members of the same type.

If we have a lower triangular $m \times m$ matrix, that is $A_{ij} = 0$, $i > j$, then we do not need to store these zero elements. Row one will have one element, row two have two element and so on, the last row will have have m elements. The rows in jagged

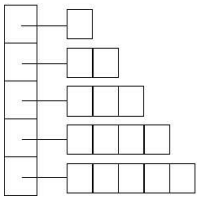

Figure 1: Data structure for a lower triangular matrix.

arrays do not need to have uniform length thus we can use jagged arrays to store a lower triangular matrix. This is illustrated in Figure 1, and the following code examples shows the implementation

```
/**Dynamically creation of a two-dimensional jagged array in C*/
double **jaggedarray;
jaggedarray = malloc((m)*sizeof(double));
for(i=0;i<m;i++)
    jaggedarray[i] = malloc((i+1)*sizeof(double));

/**Dynamically creation of a two-dimensional jagged array in Java*/
double[][] jaggedarray = new double[m][0];
for(int i = 0;i<m;i++)
    jaggedarray[i] = new double[i+1];

/**Dynamically creation of a two-dimensional jagged array in C#*/
double[][] jaggedarray = new double[m][];
for(int i = 0;i<m;i++)
    jaggedarray[i] = new double[i+1];
```

Jagged arrays have its typical use when we do not know the length of each row at compile time, and when the rows have different lengths as for the case of sparse matrices (where we usually only store the nonzero elements in each row) [3].

Linear or one-dimensional arrays are of interest since they have all their elements stored contiguously in memory for C, Java and C#. James Gosling also propose that a two-dimensional array should be stored as a linear array, this should supposedly lead to optimal and convenient behavior.

Several numerical linear algebra packages store matrices as linear array, see for example [10].

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

We can store a two-dimensional matrix as shown in (2) by using a linear array.

```
double[] A = {10,0,0,0,0,3,9,0,0,1,0,7,8,7,0,3,0,8,7,0};
```

The matrix `A` is here laid out row-wise in the linear array, as long as the row and column dimension m and n respectively is provided it is straightforward to traverse and access elements in this array.

Using linear, jagged and multidimensional arrays as data structure for the chosen algorithms, we use the same type of implementation, that is the same number of nested for-loops and so on.

3 Algorithms

The numerical linear algebra algorithms we have chosen as illustrations are matrix vector product, LU decomposition and voxel sum (or three-dimensional element sum).

We make a distinction between row-oriented and column-oriented algorithms when we implement the various operations. They are mathematically equivalent, but their efficiency can vary significantly. We say that we have a row-oriented algorithm if the inner loop access of the of the data (the matrix) is row-wise. The row-orientation of jagged arrays means that, as in C, row-oriented algorithms may be preferred over column-oriented algorithms. The possible non-contiguity of rows implies that the effectiveness of block-oriented algorithms may be highly dependent on the particular implementation. All the algorithms presented in this section are row-oriented.

Matrix vector products arises frequently in many algorithms. If A is an $m \times n$ matrix and b is a (column) vector of length n, then the matrix vector product is (3). Where c is the resulting (column) vector of length m ($c = Ab$).

$$c_i = \sum_{k=0}^{n-1} A_{ik} b_k \qquad i = 0, ..., m-1 \tag{3}$$

Which leads to the following code implementations, first with a linear array then with a two-dimensional array

```
/**A matrix vector product using linear array storage in Java*/
double[] c = new double[m];
for(int i=0;i<m;i++){
   int im = i*m;
   double s = 0.0;
   for(int j = 0;j<n;j++){
      s += A[im+j]*b[j];
   }
   c[i]=s;
}
```

```
/**A matrix vector product using jagged array storage in Java*/
double[] c = new double[m];
for(int i = 0;i<m;i++){
    double s = 0.0;
    for(int k = 0;k<n;k++){
            s += A[i][k]*b[k];
    }
    c[i] = s;
}
```

LU decomposition is a procedure of decomposing an $n \times n$ matrix A into a product of lower triangular matrix L and an upper triangular matrix U. The matrix P is the

permutation matrix where each element is 1 or 0 and each row and column only has one nonzero element. We then have a rearrangement of the $n \times n$ identity matrix I, and as a data structure we use a vector of the row indices. The purpose of the permutation matrix P is to permute the rows of A to gain numerical stability, this is known as partial pivoting, we then have $PA = LU$. After row interchange if $|A_{kk}| \leq \epsilon$, for $\epsilon \geq 0$, the matrix A is singular and the decomposition fails. The LU decomposition is described in Algorithm 1.

Algorithm 1 LU Decomposition $PA = LU$

for $k = 0$ to $n - 1$ **do**
  {Find the index $p \geq k$ such that $|A_{pk}| = max_{i=k,k+1,\ldots,n-1}|A_{ik}|$.}
  if $p \neq k$ **then**
    {Swap rows p and k of matrix A.}
  end if
  if $|A_{kk}| \leq \epsilon$ **then**
    {Singular matrix, exit algorithm.}
  end if
  for $i = k + 1$ to $n - 1$ **do**
    $A_{ik} \leftarrow A_{ik}/A_{kk}$
    for $j = k + 1$ to $n - 1$ **do**
      $A_{ij} \leftarrow A_{ij} - A_{kj}A_{ik}$
    end for
  end for
end for
{The matrix A contains now both L and U}
{$L_{ij} \leftarrow A_{ij}$ $i > j$, $L_{ii} \leftarrow 1$}
{$U_{ij} \leftarrow A_{ij}$ $i \leq j$}

Partial pivoting is when we swap rows, as described in Algorithm 1. With jagged arrays we can swap the rows by interchanging the references for each row. Reference swapping is not possible with multidimensional arrays, where we have to traverse and overwrite the rows. Each row interchange with jagged arrays takes only $O(1)$ operations, but with multidimensional arrays it takes $O(n)$ operations.

Let A be a $m \times n \times p$ volume metric dataset and consider computing the sum of all the voxel values in A as

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{p-1} A_{ijk}. \tag{4}$$

and (4) is implemented with the following code

```
/**Summation of a three-dimensional array in C#*/
double s = 0;
for(int i=0;i<m;i++)
    for(int j=0;j<n;j++)
      for(int k=0;k<p;k++)
          s+=A[i,j,k];
```

4 LU Decomposition on Variable Band Storage

Before we present the numerical results, we make a detour to show the flexibility jagged arrays have. To do that we modify the LU decomposition algorithm described in the previous section, to be able to handle variable banded matrices. We have as input in Algorithm 2 a variable banded $n \times n$ matrix A. Output is the LU factorization of A, and the permutation matrix P such that $PA = LU$. A is overwritten and contains L and U. f_i and e_i are the first and last column index in row i of A. The nonzero entries are defined as $0 \leq f_i \leq e_i \leq n-1$. We say that the matrix A has a variable band if $A_{ij} = 0$ if $j < f_i$ or $j > e_i$. By setting $f_i = 0$ and $e_i = n-1$ in Algorithm 2 for all rows we have Algorithm 1.

Algorithm 2 LU Decomposition $PA = LU$ on Variable Band Storage

```
{f_i and e_i are first and last column index in row i of A.}
{A_{i,j} = 0, j < f_i or j > e_i.}
for k = 0 to n − 1 do
  p = k
  {Find index p ≥ k such that |A_pk| = max{|A_ik| |i ∈ {k ≤ i ≤ n − 1|f_i ≤ k ≤
  e_i}}}
  if p ≠ k then
    {Swap rows p and k of matrix A.}
  end if
  if |A_kk| ≤ ε then
    {Singular matrix, exit algorithm.}
  end if
  for i = k + 1 to n − 1 do
    if f_i ≤ k ≤ e_i then
      {Add elements to row i if e_k > e_i, e_i ← e_k.}
      A_ik ← A_ik/A_kk
      for j = k + 1 to e_k do
        A_ij ← A_ij − A_kj ∗ A_ik
      end for
    end if
  end for
end for
{The matrix A contains now both L and U.}
{L_ij ← A_ij, L_ii ← 1, f_i ≤ j < i}
{U_ij ← A_ij, i ≤ j ≤ e_i}
```

Data Structure: Jagged Variable Band Storage

In this section we show how we store and access the nonzero elements in each row of the variable banded matrix in Java. This implementation is also valid for C#, only minor modifications are needed. This data structure for storing banded matrices were first introduced in [6]. A variable band matrix is also sparse, that is many of its entries are zero. To exploit this sparsity structure we only store and work on the nonzero structure of the banded matrix. We refer to this data structure as Jagged Variable Band Storage (JVBS), since we use Java's jagged arrays to store

the matrix. Consider the sparse 5×5 matrix A.

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

The jagged variable banded structure of the matrix (5) is stored in JVBS as

```
double[][] A = {{-1,2,2},{13,9,9},{-2,10},{3,1,3,9},{5,3,5,7}};
int[] Aindex =  {0,0,1,0,1};
```

The rows are stored independently, from the first nonzero entry in the row to the last nonzero entry.

In JVBS we get the f_i in row i of A by `Aindex[`i`]` and e_i in row i of A by `A[`i`].length + Aindex[`i`] - 1`.

The elements below the diagonal in column k in A are `A[`i`][`p`]`, *i=k+1,...,m-1* where *p=k-*`Aindex[`i`]` and $f_i \leq p$+`Aindex[`i`]` $\leq e_i$.

Fill-ins Issues

When we perform LU decomposition on banded matrices with pivoting there is no guarantee that the nonzero structure is preserved, i.e. we can get new entries (fill-ins) in the data structure [3]. We cannot predict fill-ins in an LU decomposition with a symbolic phase since the pivot choices are based on numerical values, during the decomposition. In this section we discuss how to add fill-ins that occur when performing partial pivoting.

There are two approaches to this problem: 1) create a data structure with linear or multidimensional arrays with only the nonzero elements that initially occurs in A and resize the whole data structure for each row that has new fill-ins. The cost of this operation are creation of a new and larger data structure and the insertion of all the initial elements. 2) create a data structure with jagged arrays with only the nonzero elements that initially occurs in A and only resize each row that has new fill-ins. The cost of this operation are creation of a new and larger row array and the insertion of the initial row elements.

The second approach is clearly the most efficient, and since we do not have any loss using jagged arrays as shown in section 5, we will get the best overall efficiency using jagged arrays in any language, for this operation.

5 Numerical Results

In this section we present some of the numerical results of the benchmarking we did with C, Java and C# for the basic numerical linear algebra algorithms presented in the section 3. The testing environments are described in Table 1.

As a unit of measurement we use Million Floating Points Operations per Second (MFLOPS). MFLOPS is used to measure the number of arithmetic operations that a computer can perform in one second. We use MFLOPS as a unit of measurement since our algorithms are exclusively numerical. It's important to understand the numerical results we present that on Red Hat Linux for ANSI C and Java we have 100 MFLOPS for 3D element sum, 200 MFLOPS for matrix vector and 300 MFLOPS

Testing Environments		
	PC	Dell
Processor	AMD	Intel Pentium 4
Processor speed	2.93GHz	2.26GHz
Memory	512MB	500MB
Operating System	Windows XP	Red Hat Linux
CLR(C#)/JVM(Java)	.NET Framework SDK 1.1/Sun 1.4.2	Sun 1.4.2.03/ANSI C
JIT Enabled (C#/Java)	Yes	Yes

Table 1: Testing Environments

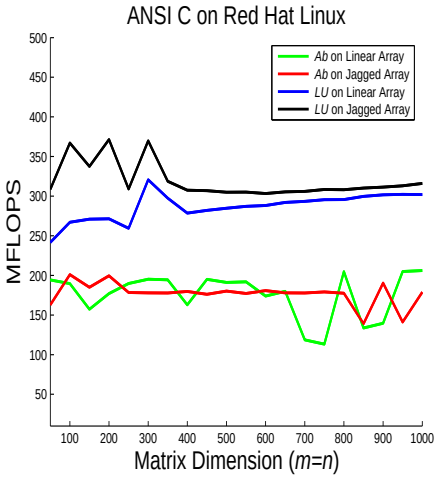

Figure 2: *Ab* and *LU* decomposition comparisons on Red Hat Linux using ANSI C.

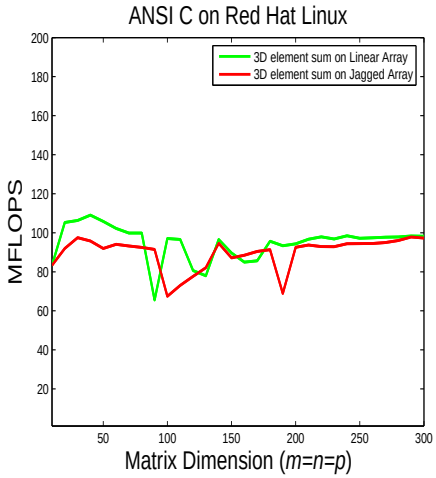

Figure 3: 3D element sum comparisons on Red Hat Linux using ANSI C.

for *LU* decomposition. The 3D element sum operation is dominated by memory access while *LU* is dominated by floating point operations. This partially explains the inconsistent values for lower dimensions.

The C Programming Language: In Figure 2 and 3, we see that there are no indication that using a linear array is significantly more efficient than jagged array for matrix vector product and three-dimensional element sum. For the *LU* decomposition, the jagged array was more efficient than linear array for all of the input sizes.

The Java Programming Language: In Figure 4 and 5 we see that on the Red Hat Linux Platform matrix vector product and three-dimensional element sum, that there is no indication that using a linear array is significantly more efficient than jagged array. For the *LU* decomposition, jagged array was more efficient than linear array for most input sizes. For Java on the Windows XP platform we have similar results, as shown in Figure 6 and 7.

The C# Programming Language: In Figure 8 and 9 we see that for three-dimensional element sum jagged array was clearly more efficient than both linear and multidimensional arrays on dimensions ≥ 130, For matrix vector product they are approximately the same. For *LU* decomposition jagged and linear arrays was clearly more efficient than multidimensional arrays. Jagged array are implemented more efficient than the multidimensional array construction in C# so this result

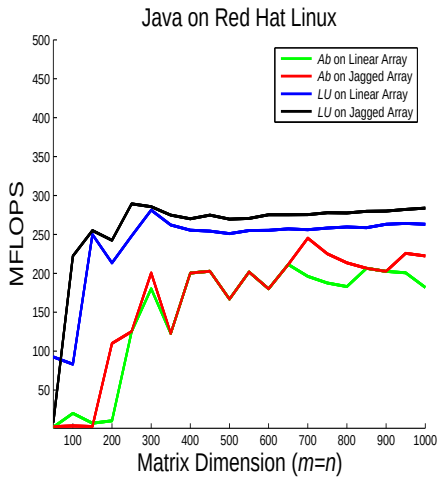

Figure 4: Ab and LU decomposition comparisons on Red Hat Linux using Sun 1.4.2.03.

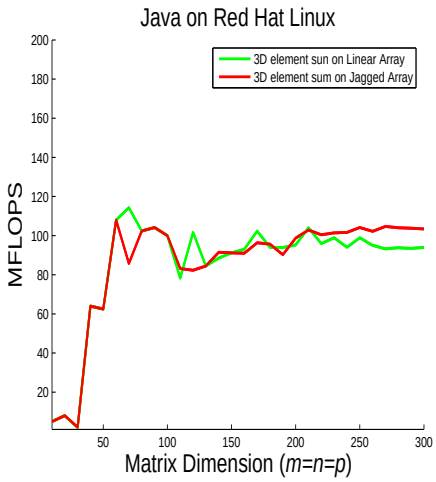

Figure 5: 3D element sum comparisons on Red Hat Linux using Sun 1.4.2.03.

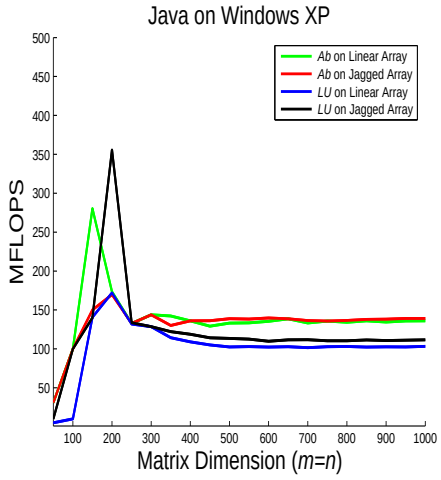

Figure 6: Ab and LU decomposition comparisons Windows XP using Sun 1.4.2.

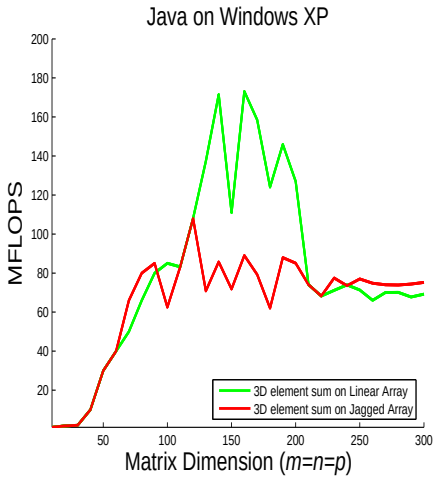

Figure 7: 3D element sum comparisons on Windows XP using Sun 1.4.2.

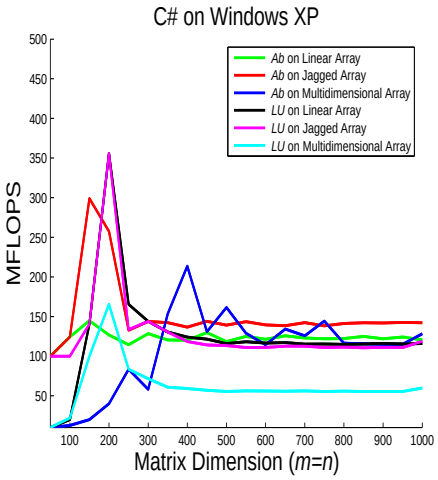

Figure 8: Ab and LU decomposition comparisons on Windows XP using .NET Framework SDK 1.1.

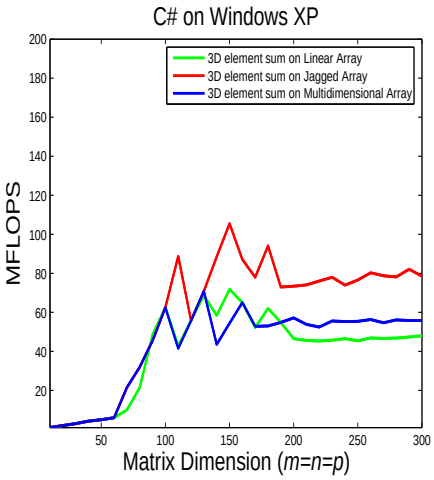

Figure 9: 3D element sum comparisons on Windows XP using .NET Framework SDK 1.1.

was not particularly surprising. This is due to minor limitation in the current JIT compiler regarding the elimination of bound checks on multidimensional arrays, hence accessing a two-dimensional jagged array row-wise is quicker than accessing multidimensional arrays row-wise [2].

6 Java versus C

In Figure 10 and 11 we show the performance of the matrix vector product, 3D element sum and LU decomposition comparing Java against C on Red Hat Linux, see Table 1. Java was more efficient than Java on matrix vector product and 3D element sum for large m. Further C was approximately 10% more efficient than Java on LU decomposition. That some kernels may be faster on Java than C is a fully realistic scenario, since C does not outperform Java on any kernels we have tested. Therefore, it is also possible that some compiler choices for Java favors certain kernels thus making Java faster than C.

7 Conclusions

In this paper we have shown that jagged arrays on row-oriented algorithms in C, Java and C# are competitive with linear arrays, despite non-contiguously memory layout. Jagged arrays have poor performance for column-oriented algorithms. In [5] it is shown for several matrix multiplication algorithms the impact of column-oriented algorithms for jagged arrays in Java. It is clear from those numerical results that column-oriented algorithms can be multiple times slower than their row-oriented versions. Comparing linear array against jagged arrays in Java and C# shows that for our test cases that jagged arrays are in fact competitive and that there is only a minor degradation for linear arrays. The benefit from using linear array may be more optimizing possibilities as removing for-loops for certain operations. Linear arrays will probably also give a more consistent performance during the execution

[2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftips.asp

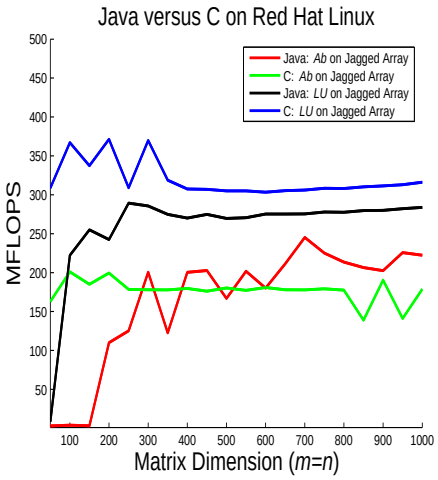

Figure 10: ANSI C versus Sun 1.4.2.03 on input Ab and LU decomposition on Red Hat Linux.

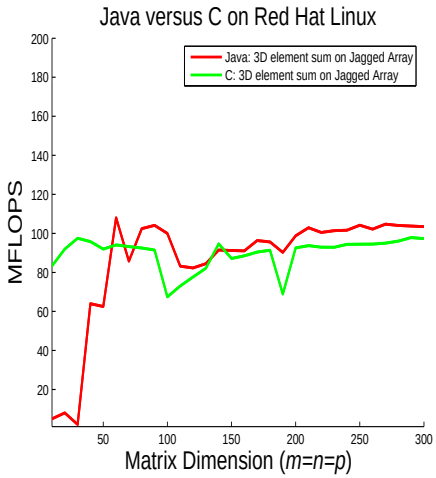

Figure 11: ANSI C versus Sun 1.4.2.03 on input 3D element sum on Red Hat Linux.

of various algorithms, and its input sizes. C# and Java had approximately the same performance for all the kernels we presented. C# is a strong object-oriented language that is still *new* and will grow and develop more over the years.

Our observations on the performance of row-oriented algorithms using jagged arrays are important and indicate that flexibility and efficiency are not mutually exclusive. These observations are based on extensive numerical testing where only few representative results are presented in this paper. The example with LU decomposition on banded matrices shows clearly the need for a more dynamic data structure than a linear or multidimensional array. Jagged arrays as shown in this paper satisfies this need. Area of further research is how utilize jagged arrays for sparse three-dimensional matrices (also known to as tensors).

References

[1] T. Archer. *Inside C#*. Microsoft Press, 2001.

[2] J. M. Bull, L. A. Smith, L. Pottage and R. Freeman. *Benchmarking Java against C and Fortran for Scientific Applications*. In Proceedings of the ACM 2001 Java Grande/ISCOPE Conference, pages 97– 105, 2001.

[3] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[5] G. Gundersen and T. Steihaug. *Data Structures in Java for Matrix Computation*. Concurrency and Computation: Practice and Experience Volume 16, Issue 8, Pages 735-815 (July 2004).

[6] G. Gundersen and T. Steihaug. *On the Use of Java Arrays for Sparse Matrix Computations*. Parallel Computing: Software Technology, Algorithms,

Architectures and Applications, Proceeding of the 10th ParCo Conference, Dresden, 2004. North Holland/Elsevier.

[7] J.E. Moreira, S.P. Midkiff, and M. Gupta. *Supporting Multidimensional Arrays in Java*. Concurrency and Computation: Practice and Experience, 15(2003)317–340.

[8] K.v. Reeuwijk, F. Kuijlman, and H.J. Sips: *Spar: a set of extensions Java for scientific computation*. Concurrency and Computation: Practice and Experience, 15(2003)277–299.

[9] J. Singer. *JVM versus CLR: A Comparative Study*. Proceedings of the Second International Conference on Principles and Practice of Programming in Java", pp.167-169 June, 2003.

[10] The Colt Distribution. *Open Source Libraries for High Performance Scientific and Technical Computing in Java*. http://hoschek.home.cern.ch/hoschek/colt/.

[11] G.K. Thiruvathukal. *Java at Middle Age: Enabling Java for Computational Science*. Computing in Science & Engineering, Vol.4, No.1, pages 74-84, 2002.

[12] W. Vogels. *HPC.NET - are CLI-based Virtual Machine Suitable for High-Performance Computing?* In Proceedings of SuperComputing 2003 (SC2003), Phoenix, AZ, November 2003.