RC 21560 (96875) 30APR99 Computer Science/Mathematics

Research Report

High Performance Computing with the Array Package for Java: A Case Study using Data Mining

J. E. Moreira, S. P. Midkiff, M. Gupta, R. D. Lawrence
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

High Performance Computing with the Array Package for Java: A Case Study using Data Mining*

J. E. Moreira S. P. Midkiff M. Gupta R. Lawrence
{*jmoreira, smidkiff, mgupta, ricklawr*}*@us.ibm.com*
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598-0218

Abstract

This paper discusses several techniques used in developing a parallel, production quality data mining application in Java. We started by developing three sequential versions of a product recommendation data mining application: (i) a Fortran 90 version used as a performance reference, (ii) a plain Java implementation that only uses the primitive array structures from the language, and (iii) a baseline Java implementation that uses our Array package for Java. This Array package provides parallelism at the level of individual Array and BLAS operations. Using this Array package, we also developed two parallel Java versions of the data mining application: one that relies entirely on the implicit parallelism provided by the Array package, and another that is explicitly parallel at the application level. We discuss the design of the Array package, as well as the design of the data mining application. We compare the trade-offs between performance and the abstraction level the different Java versions present to the application programmer. Our studies show that, although a plain Java implementation performs poorly, the Java implementation with the Array package is quite competitive in performance with Fortran. We achieve a single processor performance of 109 Mflops, or 91% of Fortran performance, on a 332 MHz PowerPC 604e processor. Both the implicitly and explicitly parallel forms of our Java implementations also parallelize well. On an SMP with four of those PowerPC processors, the implicitly parallel form achieves 290 Mflops with no effort from the application programmer, while the explicitly parallel form achieves 340 Mflops.

1 Introduction

Data mining [25] is the process of discovering useful and previously unknown information in very large data bases. Applications of data mining span scientific (*e.g.*, image processing) as well as commercial computing (*e.g.*, fraud detection). Although typical data mining applications access large volumes of data from either flat files or relational data bases, such applications often involve a significant amount of computation above the data access layer. This computation can be floating-point intensive in the case of a neural network application, or may involve a significant number of integer operations for rule-based algorithms encountered in tree classification [14] and associations algorithms [2]. High-performance data mining applications, therefore, require both fast data access and fast computation.

In many ways Java is an ideal language to implement data mining operations. Java supports portable parallel programming on many platforms, which is useful for developing applications that will be deployed across several organizations with different computing environments. Java is clean and object-oriented, which allows programs to be easily maintained. Standard classes that support domain-specific operations can be effectively implemented in Java, as demonstrated by its large collection of graphics and database APIs. As an additional advantage, Java offers an expanding base of skilled and enthusiastic adepts.

The primary worry about Java has been its efficiency. Execution speed of computation-intensive code in Java can be up to an order of magnitude slower than the equivalent Fortran code [18], even when the best commercially available environments are used. In this paper we use a production data mining application as a case study of the performance and ease-of-use of Java. In particular, we show that the use of the Array package for Java can lead to performance figures that are comparable to what can be achieved with the best Fortran environments. Exploiting parallelism within the Array package can further lead to significant improvements in performance with no effort from the application programmer.

We contrast five implementations of the *scoring* phase of a production quality data mining application. One implementation is written in Fortran 90, and is used as the reference for performance. The first Java implementation is built using plain Java (*i.e.*, only Java and its standard libraries). The second Java implementation uses an Array package [19, 17] that we have developed, and which is being proposed as a standard by the Java Grande Forum [1]. (We emphasize that the Array package is written entirely in Java and requires absolutely no modifications to the language.) The computational core of the program uses a BLAS [8] class that we developed as part of the Array package. This BLAS class can perform its individual operations either sequentially or in parallel, and allows programs written in a completely sequential style to exploit some parallelism. This strategy is similar to that employed by applications built upon the SMP Engineering and Scientific Subroutine Library (ESSL) [12] or parallel DB2 [11] subsystems. This second Java version can be considered both a serial version and an implicitly parallel version. The final Java implementation is explicitly parallel. It also uses the Array package, but exploits parallelism through multiple threads at the application level.

These different implementations allow us to make a fair assessment of the computational speed of Java relative to a language that is tuned for performance. They also allow us to compare the relative benefits of developing explicitly parallel code and building a "sequential" numerical application on top of a parallel subsystem that implements individual operations in parallel. Our Java implementation with the Array package achieves 109 Mflops, or 91% of Fortran performance, on a 332 MHz PowerPC 604e processor. Parallel execution on a 4-way SMP (using the same processors) achieves 290 Mflops for the implicitly parallel version and 340 Mflops for the explicitly parallel version. These correspond to speedups of 2.7 and 3.1, respectively.

The rest of this paper is organized as follows. Section 2 gives a quick overview of the particular data mining application in our case study. Section 3 describes the Array package and the BLAS class that we have implemented in Java. Section 4 describes in some detail the different implementations of the data mining application. Section 5 presents the performance results for these implementations, and discusses the implications of the experimental data. Section 6 discusses related work in the area, and Section 7 presents our conclusions.

2 Data mining and product recommendation

The specific data mining problem considered here involves recommending new products to customers based on previous spending behavior. A number of recommendation systems [20] based on collaborative filtering [24] have been developed over the past several years, and used for recommending items such as books and music. Typically, a new user of such a recommendation system first rates different items on a numerical scale ranging from very positive to very negative. The collaborative filtering system uses this information to establish a user profile, and then finds a group of other users with similar profiles. Once this "peer group" is established, predicted ratings for new items are computed for the target user as a weighted average of the known ratings of the members of the peer group. Higher weights are given to ratings of peer-group members most similar to the target user.

Our methodology differs in several respects from collaborative filtering. As input to our product recommendation system, we have available both detailed purchase data for the customer set, as well as a product taxonomy which generates assignments of each product to a spending category. A 3-level product taxonomy is used: individual products (*e.g.*, "Brooke Bond Tea, 250g") are assigned to one of 2000 product subgroups (*e.g.*, "Premium Tea Bags - Branded"), which in turn are mapped to one of 100 major product groups (*e.g.*, "Tea"). For each customer, a personalized recommendation list is generated by sorting this list of eligible products according to a customer-specific score computed for each product. The score in defined such that higher scores indicate stronger interest in this particular product. In some commercial situations, there may be an incentive to preferentially recommend some products over others due to overstocks, supplier incentives, or increased profit margins. This information can be introduced into the recommendation strategy via product-dependent scaling factors, with magnitudes reflecting the relative priority of recommendation. Details of the scoring algorithm are provided in the following section.

2.1 The scoring algorithm

Let there be m products eligible for recommendation, p customers, and n spending categories. Each product i has an *affinity vector* A_i, where A_{ik} is the affinity of product i with spending category k. These affinities can be interpreted as the perceived appeal of this product to customers with participation in this spending category. They are determined by pre-computing associations [2] at the level of spending categories. The collection of affinity vectors from all products forms the $m \times n$ affinity matrix A. This matrix is sparse and is organized so that products with the same nonzero patterns in their affinity vectors are adjacent. Therefore, matrix A can be represented by a set of groups of products, as shown in Figure 1. Each group i is defined by f_i and l_i, the indices of the rows representing the first and last products in the group, respectively. All products with the same nonzero pattern belong to the same group. The nonzero pattern for the products in group i is represented by an index vector I_i, where I_{ik} is the position of the k-th nonzero entry for that group. Let $\text{len}(I_i)$ be the length of index vector I_i. Also, let $g_i = l_i - f_i + 1$ be the number of products in group i. Then the affinity matrix G_i for group i is the $g_i \times \text{len}(I_i)$ dense matrix with only the nonzero entries from rows f_i to l_i of A.

$$A = \left[\begin{array}{cccccc} \times & \times & & & \times & \\ \times & \times & & & \times & \\ & \times & \times & \times & & \\ & \times & \times & \times & & \\ \times & & \times & \times & & \times \\ \times & & \times & \times & & \times \\ \times & & \times & \times & & \times \end{array}\right]$$

(a)

$$f_1 = 1, l_1 = 2 \quad G_1 = \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \end{array}\right] \quad I_1 = \left[\begin{array}{ccc} 1 & 2 & 5 \end{array}\right]$$

$$f_2 = 3, l_2 = 4 \quad G_2 = \left[\begin{array}{ccc} \times & \times & \times \\ \times & \times & \times \end{array}\right] \quad I_2 = \left[\begin{array}{ccc} 2 & 3 & 4 \end{array}\right]$$

$$f_3 = 5, l_3 = 7 \quad G_3 = \left[\begin{array}{cccc} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{array}\right] \quad I_3 = \left[\begin{array}{cccc} 1 & 3 & 4 & 6 \end{array}\right]$$

(b)

Figure 1: Structure (a) and representation (b) of the affinity matrix A.

Each customer j has an associated *spending vector* S_j, where S_{jk} is the normalized spending of customer j in category k. The collection of spending vectors of all customers forms the $p \times n$ spending matrix S. This matrix is also sparse, but customers are not organized in any particular order. (The affinity matrix is relatively static and can be built once, whereas the spending matrix may change between two executions of the scoring algorithm.) We also define a $p \times n$ normalizing matrix N, which has the same nonzero pattern as S, but with a 1 wherever S has a nonzero:

$$N_{jk} = \begin{cases} 1 & \text{if } S_{jk} \neq 0 \\ 0 & \text{if } S_{jk} = 0 \end{cases}. \tag{1}$$

The structure of matrices S and N are shown in Figure 2(a). The nonzero pattern of row j of matrix S is represented by an index vector J_j, where J_{jk} is the position of the k-th nonzero entry for that row (customer). The actual spending values are represented by a dense vector σ_j, of the same length as J_j, which has only the nonzero entries for row j of S. This representation of S is shown in Figure 2(b). Matrix N does not have to be represented explicitly.

$$S = \left[\begin{array}{ccccc} \times & & \times & & \times \\ \times & \times & & & \times \\ \times & \times & \times & \times & \\ & \times & \times & \times & \end{array}\right] \quad N = \left[\begin{array}{ccccc} 1 & & 1 & & 1 \\ 1 & 1 & & & 1 \\ 1 & 1 & 1 & 1 & \\ & 1 & 1 & 1 & \end{array}\right]$$

(a)

$$\begin{array}{ll} \sigma_1 = \left[\begin{array}{ccc} \times & \times & \times \end{array}\right] & J_1 = \left[\begin{array}{ccc} 1 & 3 & 5 \end{array}\right] \\ \sigma_2 = \left[\begin{array}{ccc} \times & \times & \times \end{array}\right] & J_2 = \left[\begin{array}{ccc} 1 & 2 & 5 \end{array}\right] \\ \sigma_3 = \left[\begin{array}{cccc} \times & \times & \times & \times \end{array}\right] & J_3 = \left[\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}\right] \\ \sigma_4 = \left[\begin{array}{ccc} \times & \times & \times \end{array}\right] & J_4 = \left[\begin{array}{ccc} 2 & 3 & 4 \end{array}\right] \end{array}$$

(b)

Figure 2: Structure of the spending matrix S and normalizing matrix N (a), representation of matrix S (b).

Let ρ_i be the product dependent scaling factor for product i as discussed in the preceding section. The *score* Φ_{ij} of customer j against product i is computed as:

$$\Phi_{ij} = \rho_i \frac{\sum_{k=1}^{n} A_{ik} S_{jk}}{\sum_{k=1}^{n} A_{ik} N_{jk}}. \tag{2}$$

That is, we first compute the sum of the spending of customer j on all categories, weighted by the affinity of each category with product i ($\sum_{k=1}^{n} A_{ik} S_{jk}$). We then divide it by the sum of all affinities of product i with categories for which customer j has any spending ($\sum_{k=1}^{n} A_{ik} N_{jk}$). The result is an affinity-weighted average of the spending habits of customer j, and therefore a normalized measure of how well product i matches those habits. Finally, we multiply the result by the priority scaling factor to capture product specific contributions in the actual Φ_{ij} score.

We note that $\sum_{k=1}^{n} A_{ik} S_{jk}$ in Equation (2) is the dot-product of row i of A by column j of S^T. Similarly, $\sum_{k=1}^{n} A_{ik} N_{jk}$ is the dot-product of of row i of A by column j of N^T. Let $\Lambda_p(\rho_1, \rho_2, \ldots, \rho_m)$ denote an $m \times p$ matrix Λ with $\Lambda_{ij} = \rho_i, 1 \le i \le m, 1 \le j \le p$. Then, Equation (2) can be rewritten as a matrix operation:

$$\Phi = \Lambda_p(\rho_1, \rho_2, \ldots, \rho_m) * (A \times S^T) \div (A \times N^T) \tag{3}$$

where $\times$ denotes matrix multiplication, $\div$ denotes two-dimensional array (or element-by-element) division, and $*$ denotes two-dimensional array (element-by-element) multiplication.

Using the representations of A and S discussed previously, the scores for a customer j against all products in a product group i can be computed in the following way. First, two vectors ε and η of size n are initialized to 0:

$$\begin{aligned} \varepsilon[1:n] &= 0, \\ \eta[1:n] &= 0. \end{aligned} \tag{4}$$

Then, the compressed vector σ_j is expanded by assigning its values to the elements of ε indexed by J_j. (This is a scatter operation.) Similarly, an expanded representation of the j-th row of N is stored into η:

$$\begin{aligned} \varepsilon[J_j] &= \sigma_j, \\ \eta[J_j] &= 1. \end{aligned} \tag{5}$$

Now, the scores of customer j against products f_i to l_i can be computed by:

$$\Phi[f_i : l_i, j] = \Lambda_1(\rho_{f_i}, \rho_{f_i+1}, \ldots, \rho_{l_i}) * (G_i \times \varepsilon[I_i]) \div (G_i \times \eta[I_i]) \tag{6}$$

where $\times$ now denotes matrix-vector multiplication, and $\div$ and $*$ denote one-dimensional array (element-by-element) division and multiplication respectively. Note that we have two gather operations, one each in $\varepsilon[I_i]$ and $\eta[I_i]$. If we group several customers together, forming a block from customer j_1 to customer j_2, the operations in Equation (6) become dense matrix multiplication, array division, and array multiplication:

$$\Phi[f_i : l_i, j_1 : j_2] = \Lambda_{j_2 - j_1 + 1}(\rho_{f_i}, \rho_{f_i+1}, \ldots, \rho_{l_i}) * (G_i \times \varepsilon[I_i, j_1 : j_2]) \div (G_i \times \eta[I_i, j_1 : j_2]). \tag{7}$$

Matrix multiplication can be implemented more efficiently than matrix-vector multiplication. This is particularly true on cache-based RISC microprocessors, since matrix multiplication can be organized through blocking to better exploit the memory hierarchy.

3 The Array package

Array and matrix operations are building blocks for many numerically intensive applications, including our data mining code. To address various issues in coding these applications in Java, we have developed an Array package. The goal of this package is to introduce in Java the functionality and performance usually associated with Fortran arrays. In this section we first discuss the basic features of multidimensional Arrays in the package. (We use the term *Array*, with a capital first *A*, to denote the multidimensional structures implemented by the Array package.) We then proceed to describe the BLAS class of the Array package, which implements several matrix operations, including the matrix-multiply we use in the data mining code. Finally, we discuss how to exploit the parallelism that is implicit in array and matrix operations.

3.1 The basic package

The Array package for Java is a class library that implements true multidimensional rectangular Arrays. Each class name is of the form <*type*>**Array**<*rank*>, where <*type*> is the type of the Array elements and <*rank*> is the rank (number of axes or dimensions) of the Array. Supported types include all the primitive Java types as well as complex numbers. Supported ranks go from 1D (one-dimensional) to 7D (seven-dimensional). We have, however, currently implemented only one- through three-dimensional Arrays. A k-dimensional Array A is characterized by its *shape vector* $(n_0, n_1, \ldots, n_{k-1})$, where n_j is the extent of Array A along its j-th axis. The shape of an Array is defined at creation time and is immutable during the lifetime of the Array. In this discussion we focus on classes **doubleArray1D** and **doubleArray2D** (one- and two-dimensional arrays of **double**s), since these are the two kinds of arrays used in our data mining code. Regular ranges of indices for arrays are implemented by **Range** objects. Irregular ranges of indices are implemented by **Index** objects. These objects have properties that facilitate bounds checking optimizations.

Java does not directly support multidimensional arrays. Instead, it provides *arrays of arrays*. To understand the difference between a Java **double**[ ][ ] array of arrays (hereafter called Java arrays) and a **doubleArray2D**, consider the example of Figure 3. Both X and Y are of type **double**[ ][ ], while Z and T are of type **doubleArray2D**. The structure of Java arrays is fragmented into rows. Accessing an element (*e.g.*, $X[4][2]$) requires both pointer chasing (to find row $X[4]$) and indexing (to find element $X[4][2]$). Java arrays are not rectangular structures, since different rows of X can have different lengths. Also, a row of an array can be aliased to other rows of the same or another array. (Rows $X[1]$ and $X[2]$ are aliased to each other in Figure 3(a), the same is true for rows $X[4]$ and $Y[3]$.) Finally, the shape of a Java array can be changed at run time. For example, the operation $X[4]$ = **new double**[n] associates a new row with $X[4]$. This operation can even be performed by one thread while another thread is concurrently performing a computation with X. All these properties give great flexibility to Java arrays, but they also hinder performance. Aliasing disambiguation becomes extremely difficult for elements of Java arrays, and determining bounds of (pseudo) multidimensional Java arrays is, in general, an expensive operation.

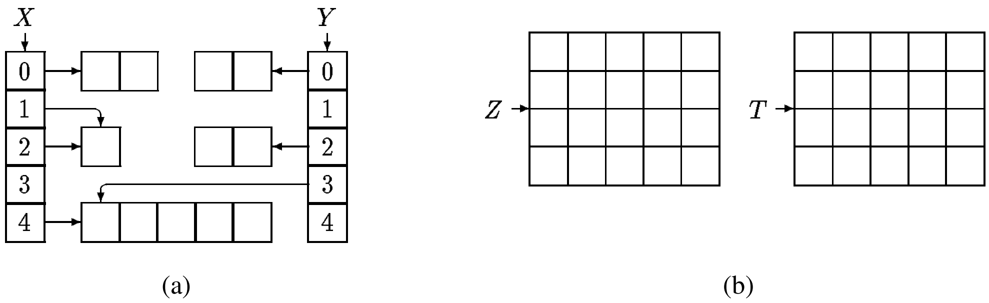


Figure 3: Examples of different array types: (a) **double**[ ][ ], (b) **doubleArray2D**.

The multidimensional Arrays in the Array package fix many of the performance problems associated with Java arrays. Although the storage layout is not visible from outside the classes, it can be implemented in a contiguous format to avoid fragmentation. Pointer chasing is not necessary to access elements, since the location of an element in storage can be found through index arithmetic. Rows cannot be aliased to each other and are all of the same length. Another important benefit of the Array package is that it allows bounds checking optimizations [15, 16] to be performed without privatizing the array descriptors. Because Java requires a precise exception to be thrown for any **null**-pointer dereference or any out-of-bounds array access, good bounds checking optimization is necessary for compilers to perform any transformation that alters data access orders. Finally, the rectangular nature of the data structures lends itself nicely to parallelization.

To illustrate differences in coding styles, we show in Figure 4 an example of how a user might code a matrix multiply routine using (a) Java arrays and (b) multidimensional Arrays. The two versions are very similar in structure, but they require different optimization strategies. The fixed shapes of a, b, and c in Figure 4(b) allow the bounds checking optimizations to be performed without any need for privatization. The high order transformations described in [16] can also be applied if we can disambiguate between a, b, and c. (Note that, in Java, a, b, and c are really pointers to **doubleArray2D** objects.) Disambiguation between a, b, and c in Figure 4(b) can be done with a constant time test. All we need to show is that $c \neq a$ and $c \neq b$. (To be precise, we have to show that $c.x \neq a.x$ and $c.x \neq b.x$, where x is an internal field of our array classes which points to the data storage area.) Disambiguation in Figure 4(a) requires disambiguation for each and every row of a, b, and c. (We need to show that $c[i] \neq a[j]$ and $c[i] \neq b[j]$ $\forall i, j$ and that $c[i] \neq c[j]$ $\forall i, j$ $i \neq j$.)

Some of the design principles of the Array package can be understood by examining the method `divide` of class **doubleArray2D**. This method (shown in Figure 5) computes the array division $c = a \div b$ (which would be coded in Java as `c = a.divide(b)`). Before performing any computation, we check that `b` has the same shape as `a`. If the shapes are not the same we exit the method with an exception. The access to the Array elements in the actual division operation are safe with respect to Array bounds. Note that the range of i is from 0 to $m - 1$

```
void matmul(   double[][] a,
               double[][] b,
               double[][] c,
               int m, int n, int p ) {
  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
      for (k = 0; k < n; k++)
        c[i][j] += a[i][k] * b[k][j];
}
```

(a)

```
void matmul(   doubleArray2D a,
               doubleArray2D b,
               doubleArray2D c,
               int m, int n, int p ) {
  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
      for (k = 0; k < n; k++)
        c.set(i, j, c.get(i, j) + a.get(i, k) * b.get(k, j));
}
```

(b)

Figure 4: Matrix-multiply code: (a) with **double**[ ][ ], (b) with **doubleArray2D**.

and the range of j is from 0 to $n-1$. Furthermore, `a.size(0)` $=$ `b.size(0)` $=$ `c.size(0)` $= m$ and `a.size(1)` $=$ `b.size(1)` $=$ `c.size(1)` $= n$. This property is used to optimize bounds checking during the computation, according to the techniques described in [9, 10].

```
public doubleArray2D divide(doubleArray2D b)
   throws NonconformingArrayException {

    doubleArray2D a = this;
    doubleArray2D c = null;

    int m = a.size(0);
    int n = a.size(1);

    if (n != b.size(0)) throw new NonconformingArrayException();
    if (m != b.size(1)) throw new NonconformingArrayException();

    c = new doubleArray2D(m,n);

    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            c.set(i,j,a.get(i,j)/b.get(i,j));
        }
    }

    return c;
}
```

Figure 5: Implementation of Array division.

3.2 The BLAS class

The Array package provides a Basic Linear Algebra Subprograms (BLAS) [8] class as a facility for writing efficient linear algebra codes. BLAS implements a variety of elementary vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3) operations. Note that this is a 100% Java implementation of BLAS, and not an interface to already existing native libraries. We chose this approach for two main reasons: (i) to demonstrate that Java can achieve high performance in numerical computing and (ii) to have a completely portable parallel implementation of BLAS that works well with the Array package.

The most important of the many BLAS operations, from a performance perspective, is `dgemm`, which implements the generic matrix multiplication.

$$C = \alpha A^{\star} B^{\star} + \beta C \tag{8}$$

where A, B, and C are matrices, α and β are scalars, $A^{\star}$ can be either A or A^T and $B^{\star}$ can be either B or B^T. Linear algebra codes are often written to maximize the use of `dgemm`, since this is typically the highest performing operation in BLAS. Matrix multiplication is also the main operation in our scoring code, as described in Section 2.1. Because of its importance, we focus this discussion on the implementation of `dgemm`.

A simplified version of `dgemm` from the BLAS class is shown in Figure 6. First, the Arrays `a` and `b` are transposed, if necessary, as indicated by the two flags `transa` and `transb`. This transposition is performed in place at negligible cost (no data copying) and allows us to use just one version of code for the actual multiplication. Following the design principles for the Array package, all Arrays are tested for conformity before we start any computation. The computational loop nest is guaranteed not to throw any exceptions and therefore can be optimized extensively. (For clarity, we omit the code that performs runtime alias disambiguation. If `c` is aliased to `a` or `b`, the result is first computed into a temporary Array, to guarantee proper semantics.) In the actual code for `dgemm` (not shown here), the loops are blocked and unrolled for better memory behavior, register allocation, and instruction-level parallelism [16]. This leads to good sequential performance, as discussed below. The blocked structure is also used for internal parallelization of `dgemm`, using the approach described in [16].

```
public static void dgemm(int transa, int transb, double alpha,
                         doubleArray2D a, doubleArray2D b,
                         double beta, doubleArray2D c)
    throws NonconformingArrayException {

    if (transa == Transpose) a = a.permuteAxes(1,0);
    if (transb == Transpose) b = b.permuteAxes(1,0);

    int m = a.size(0);   int n = a.size(1);    int p = b.size(1);

    if (n != b.size(0)) throw new NonconformingArrayException();
    if (p != c.size(1)) throw new NonconformingArrayException();
    if (m != c.size(0)) throw new NonconformingArrayException();

    for (int i=0; i<m; i++) {
        for (int j=0; j<p; j++) {
            double s = 0;
            for (int k=0; k<n; k++) {
                s += a.get(i,k)*b.get(k,j);
            }
            c.set(i,j,alpha*s+beta*c.get(i,j));
        }
    }
}
```

Figure 6: An implementation of `dgemm` using the Array classes.

Figure 7(a) shows the performance achieved by our Java `dgemm` implementation executing on an SMP machine with 4×332 MHz PowerPC 604e processors. For comparison, Figure 7(b) shows the performance for the highly optimized `dgemm` in ESSL (note the different scales). We report Mflops achieved when all three matrices are of size $n \times n$, for different values of n. The circles represent results for a single threaded execution, while the crosses represent results for an execution with four threads. The single threaded Java `dgemm` execution achieves consistent performance (~ 180 Mflops) throughout the entire range of problem sizes. The performance of the multithreaded Java execution scales rapidly with the problem size in the lower part of that range, and later stabilizes at approximately 3.5 times the single threaded performance. We note that for small problem sizes ($n = 25$ in particular), the multithreaded execution actually does worse in terms of performance. With such a small problem size, there really is only one block of matrix C to be computed and all the work is done by one thread. Since we still pay the overhead to start and terminate execution on all four threads, we see a performance degradation. From the results with $n = 25$ we can estimate the overhead for starting and terminating parallel execution at approximately 100 microseconds. Performance of Java dgemm is approximately half of ESSL dgemm. (Which, in turn, achieves approximately 50% of the peak machine performance of 664 Mflops per processor.) This is not surprising, since Java cannot use the fma instruction in the PowerPC. The net effect of this limitation is a reduction by a factor of two of the peak machine performance, from 664 Mflops with fmas to 332 Mflops without.

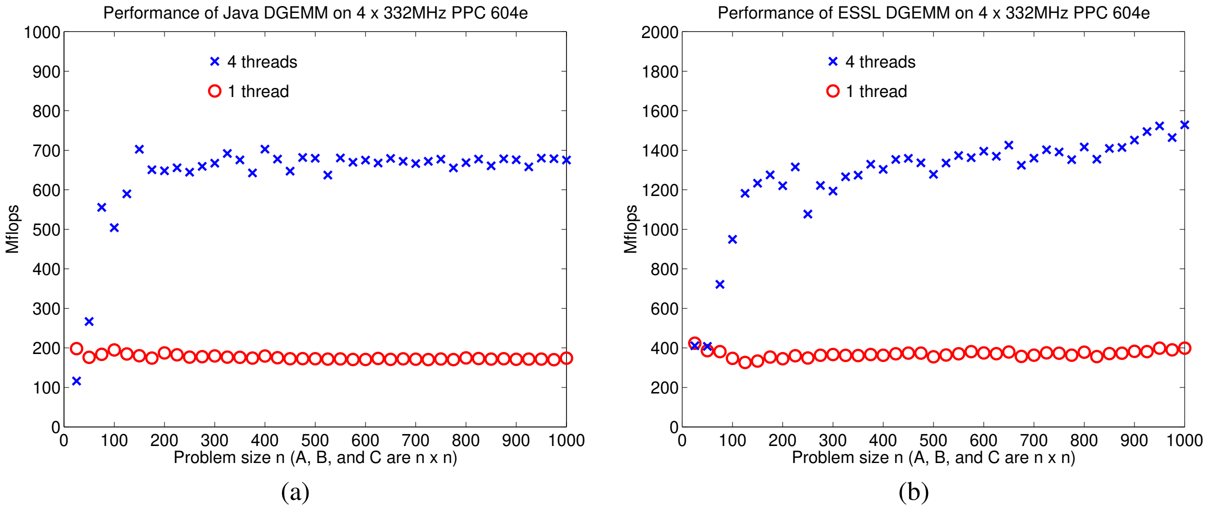


Figure 7: Performance of dgemm in (a) the Java BLAS and (b) ESSL.

3.3 Enabling implicit parallelism

We now describe the master/slave mechanism used by the implicitly parallel routines. Our design principles were: (i) create threads only once and reuse them in different operations; and (ii) allow the master to control the assignment of work to helper threads. The first principle reduces the overhead of excessive thread creation and destructions. The second principle supports better exploitation of data locality between operations.

Figure 8 shows the basic actions performed when an operation is executed in parallel. Solid boxes indicate mutex operations performed on per-thread data structures, while dashed boxes indicate mutex operation performed on a global shared data structure. Solid arcs indicate flow of control, and dashed arcs indicate notify events. The master thread initially creates the number of helper threads specified by an initialization call from the application program. It then begins executing the program. Each newly created helper thread marks itself as idle, and waits for work on a synchronization object associated with it. This allows the Java notify method to be used to restart a specific thread.

When the master thread encounters a parallel operation, it computes T, the desired number of threads to perform the operation. It then notifies helper threads $1, \ldots, T-1$ to start work. All of the threads, including the master, perform their part of the computation. When each thread finishes work on the parallel operation, it signals that by decrementing a shared counter. If the master thread is not the last thread to finish work (as can be determined by examining if the shared counter is zero) it marks itself as idle and waits on its synchronization object. If a helper thread is the last thread to finish, it notifies the master to continue. All helper threads, when finished with the operation, mark themselves as idle, and wait for the next parallel operation. When all threads have finished their part of the parallel computation, the master thread proceeds with the serial computation.

4 The data mining implementations

This section discusses five different implementations of the scoring algorithm described in Section 2.1. The first describes a *plain* Java implementation of the scoring algorithm, that is an implementation that does not make use of the BLAS class or Array package described in Section 3. The second implementation is our *baseline* Java code, which implements the scoring algorithm using those classes. From the baseline we derive *implicitly* and *explicitly* parallel

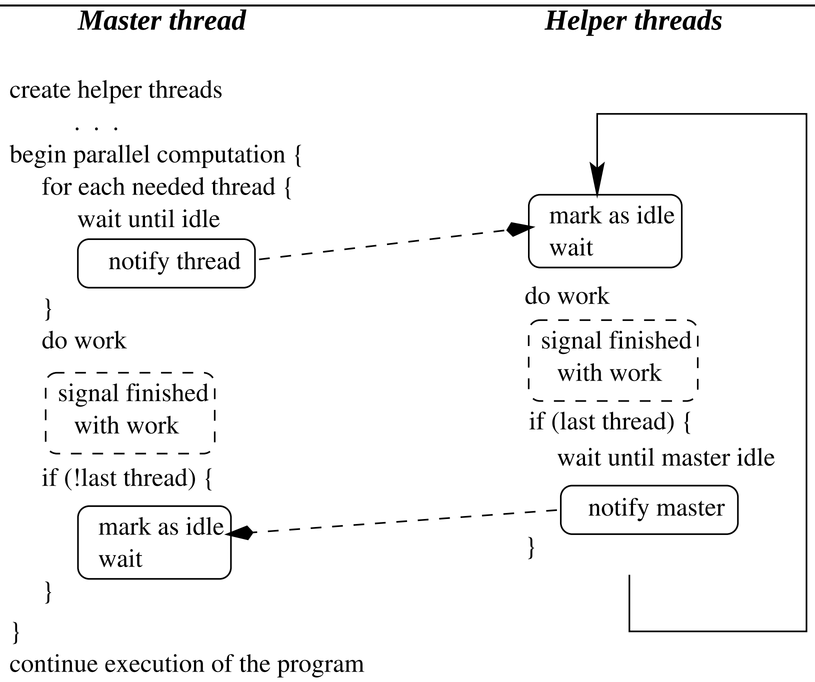


Figure 8: Overview of the thread scheduling operations.

Java implementations. Finally, we describe a Fortran 90 version of the scoring algorithm. The Fortran 90 version serves as a metric to determine how well we are doing with Java in absolute terms.

The plain Java implementation: In the plain Java implementation of the scoring algorithm, each G_i matrix is represented as a `double[][] G` and each I_i index vector is represented as an `int[] I`. The compressed spending vector σ_j for customer j is expanded into vectors ε and η according to Equation (5), with these vectors represented by a `double[] E` and `double[] N`, respectively. Scores for customer j against the products in group i are computed according to Equation (6) by the code shown in Figure 9. In that code, `m` and `n` are the number of rows and columns of `G`, respectively. The index f_i of the first product in group i is represented by `f[i]` and the scaling factor ρ_q for product q is represented by `rho[q]`. Results are stored in a `double[][] Phi` structure, which represents Φ.

```
for (int l=0; l<m; l++) {

    double score = 0.0;
    for (int k=0; k<n; k++) {
        score += G[l][k]*E[I[k]];
    }

    double norm = 0.0;
    for (int k=0; k<n; k++) {
        norm += G[l][k]*N[I[k]];
    }

    Phi[f[i]+l][j] = rho[f[i]+l]*score/norm;
}
```

Figure 9: Main computations in the plain Java implementation.

The baseline Java implementation: In this implementation, each G_i matrix is represented by an object `G` of class **doubleArray2D** and each I_i by an **Index** object `I`. Scoring is computed according to Equation (7), which operates on a block of customers, from customer j_1 to customer j_2. In that case, ε and η are matrices, also represented by **doubleArray2D** objects `E` and `N`, respectively. Each column of matrix ε is obtained by expanding the compressed vector σ_j for customer j ($j_1 \leq j \leq j_2$). The matrix η is obtained similarly. We operate on one block of customers at a time to conserve memory: we only have to keep expanded spending vectors for the customers in that block. The matrix Λ is stored as a **doubleArray2D** object `Lambda`, with all the elements of row k set to ρ_k. The computation of $\Phi[f_i : l_i, j_1 : j_2]$ is performed by the code of Figure 10. First, an Array section `results` is created to represent the region of Φ being computed. That region is initialized with the matrix product $G_i \times \varepsilon$, after extracting the relevant rows of the ε matrix according to the I index. The result of $G_i \times \eta$ is similarly stored in an auxiliary Array `normal`. We complete the computation of $\Phi[f_i : l_i, j_1 : j_2]$ with an Array multiplication by Λ and an Array division by the result of $G_i \times \eta$. The `timesAssign` and `divideAssign` methods implement the operators `*=` and `/=` for Arrays.

```
/*
 * Extract the section of the Phi matrix corresponding
 * to group i and customers j1 to j2.
 */
results = Phi.section(new Range(f[i],l[i]),new Range(j1,j2));

/*
 * Compute the compressed matrix of customer spending
 * in "compressed". Multiply the group affinity matrix
 * by it, producing the unnormalized result.
 */
compressed = E.get(I,new Range(j1,j2));
Blas.dgemm(NoTranspose,NoTranspose,1.0,G,compressed,0.0,results);

/*
 * Compute the compressed matrix of "1"s.
 * Multiply the group weights matrix by it,
 * producing the normalization matrix.
 */
compressed = N.get(I,new Range(j1,j2));
Blas.dgemm(NoTranspose,NoTranspose,1.0,G,compressed,0.0,normal);

/*
 * Multiplying the unnormalized results by the scaling
 * matrix Lambda and then dividing by the normalization
 * matrix produces the final, normalized results.
 */
results.timesAssign(Lambda).divideAssign(normal);
```

Figure 10: Main computations in the baseline Java implementation.

The parallel Java implementations: The implicitly parallel Java version is identical to the baseline Java, except for one call at the beginning of the application to an initializer method of the Array package. This method call is of the form

`ParallelEnvironment.createThreads(`n`)`

and it creates the collection of n threads used by the Array package in exploiting parallelism. All parallelism is exploited internally in the Array package and the BLAS class in a totally transparent manner to the application. In the explicitly parallel version, it is the data mining application itself that creates additional helper threads. The customers are partitioned into blocks, just as in the baseline Java, but the blocks themselves are statically distributed among the threads, following a cyclic distribution. In essence, this corresponds to a block-cyclic distribution of customers to threads. Since each thread operates on one block of customers at a time, it can take advantage of the matrix-matrix operations in BLAS. At the same time, the cyclic distribution of blocks helps achieve better load balance.

The Fortran 90 implementation: The Fortran 90 implementation is a straightforward translation of the baseline Java. The **doubleArray2D** objects of Java are represented by two-dimensional Fortran arrays of `real*8`. **Index** objects are represented by one-dimensional arrays of `integer`. Matrix multiplication is implemented through calls to the highly optimized `dgemm` routine in ESSL. Array multiplication and division are performed by using the standard `*` and `/` operators of Fortran 90, applied to two-dimensional arrays.

5 Experimental Results

In this section, we present a detailed discussion of the performance results obtained for different implementations of the scoring algorithm on an IBM 4-way SMP using 332 MHz PowerPC 604e processors (IBM RS/6000 model F50). We report the performance, in Mflops, of the entire scoring algorithm, as discussed in Section 2.1. The scoring algorithm is the bulk of the computation in the data mining code. We used the IBM HPCJ compiler [23] (with -O optimization level) to compile Java programs, and the XLF 5.1 compiler (with -O3 optimization level) for the Fortran 90 program. The problem parameters are shown in Table 1, which describes the sizes and number of nonzeros in the three main matrices of our application. We remind the reader that the affinity matrix A is stored as a set of dense blocks, one for each group (93 in total). The S matrix is stored in compressed rows. Finally, the Φ matrix, because of its relatively large number of nonzeros, is stored in expanded form.

matrix	size (rows $\times$ columns)	number of nonzeros	fraction of nonzeros
A	10350×2103	397,559	1.83%
S	4800×2103	231,194	2.29%
Φ	10350×4800	20,759,569	41.79%

Table 1: Problem parameters for our data mining application.

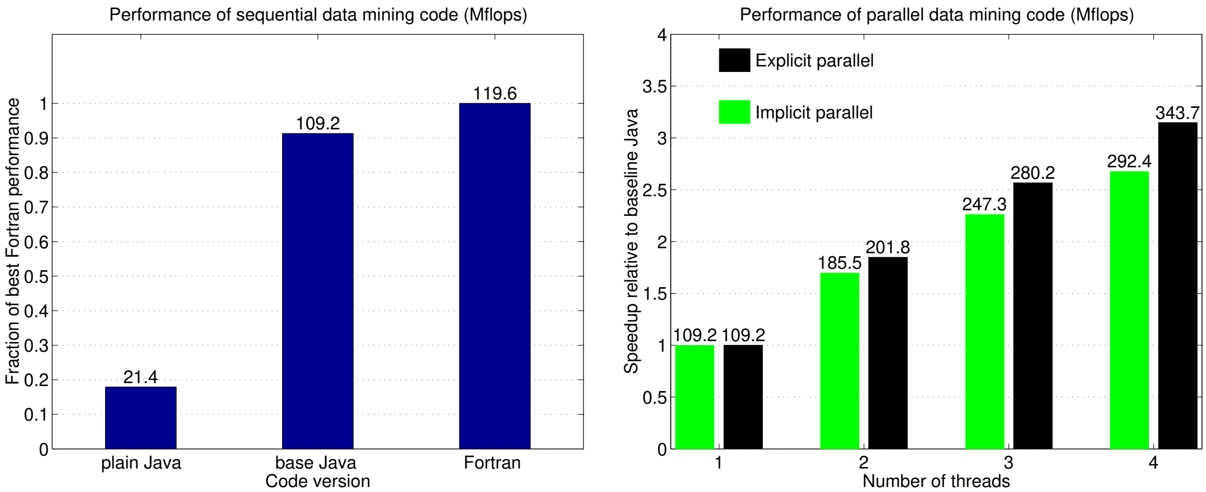
5.1 Performance of serial versions

Figure 11(a) shows the serial performance of the Java and Fortran 90 codes, as a fraction of the performance obtained by the Fortran 90 version (119.6 Mflops). The plain Java program achieves 21.4 Mflops, which is 18% of the Fortran 90 performance. The baseline Java version performs significantly better, achieving 109.2 Mflops, or 91% of the Fortran 90 performance. The Array package provides a higher level of abstraction to the programmer than the primitive operations on arrays currently defined in Java. Our experimental results demonstrate that it also leads to higher performance, which is primarily due to: (i) greater impact of array bounds checking on the plain Java version, and (ii) better utilization of the memory hierarchy by the BLAS class library code.

In the plain Java program, bounds checking is performed for all array accesses. Since Java supports precise exceptions, all code-reordering transformations must be disabled in loops that have bounds checking. An additional measurement showed that the performance of this program jumps to 36.5 Mflops if all array bounds checks are optimized away. As discussed earlier in Section 3, it is easier for the compiler to apply transformations [16] to deal with this problem if the program uses the Array package.

The BLAS class provides to the application programmer the benefits of a tuned implementation of commonly needed operations, such as matrix-multiplication. This is similar to the approach adopted by providers of high-performance libraries like LAPACK [8] and ESSL [12]. We emphasize that all checks for conditions leading to exceptions are performed at the beginning of the code for the BLAS methods, so that compiler optimizations can be applied effectively to performance-critical sections of the code.

We observe that by using the Array package, which does not use any native libraries, the Java version of the data mining code achieves performance close to that delivered by the Fortran 90 version. Thus, we show that it is possible to obtain competitive performance with pure Java, even on numerically-intensive production quality codes.



(a) sequential code performance

(b) parallel code performance

Figure 11: Performance of different versions of the data mining code.

5.2 Performance of parallel versions

Figure 11(b) shows the performance of the parallel codes for different numbers of threads. The implicitly parallel code achieves a speedup of 2.7 with four threads, reaching a performance of 292.4 Mflops. The explicitly parallel code consistently delivers better performance, achieving 343.7 Mflops on four threads, which represents a speedup of 3.1 over the baseline Java version.

Our experiments illustrate the classical trade-off between ease of programming and efficiency of generated code. The support for implicit parallelism in our implementation of the Array package entirely hides the complexity of parallelization from the end-user. The user merely has to specify the degree of parallelism desired. This transparency of parallelization comes at the expense of some loss of performance. The implicitly parallel code suffers from four main sources of performance degradation: (i) higher overhead of threading operations (because each operation in the implicit version requires enqueuing and dequeueing threads from the work queue), (ii) greater load imbalance, (iii) more serial components, and (iv) poorer cache utilization (due to dynamic scheduling of operations to processors in the implicitly parallel version) and a greater amount of cache traffic due to interference in shared data. These factors are largely a consequence of the fact that the implicitly parallel code exploits parallelism at a finer granularity than the explicitly parallel code.

The explicitly parallel version has a single dominant outer loop which can be parallelized, so that the overhead of waking up helper threads and waiting for them to finish is incurred only once. In contrast, the implicitly parallel code has many calls to the parallelized versions of the Array and BLAS methods, which are repeated inside the loop over blocks of customers. Any load imbalance among different threads during each of these operations is additive: there is no possibility of these imbalances canceling each other out over multiple operations, as with the explicitly parallel case. The implicitly parallel version also has more serial components in the code, because for some scatter and gather operations the gains from parallelization are not enough to offset the overhead of threading and synchronization. Hence, those operations are not parallelized. Finally, most of the temporary Arrays used in the explicitly parallel code are read and written by a single thread, so that this version attains very good cache utilization and avoids the problems of cache line invalidations due to false sharing. The implicitly parallel code, by its very nature, relies on library calls over shared data.

We observe that even though the implicitly parallel approach does not lead to the best possible performance, it consistently delivers good performance as the number of threads is increased. On four processors, it leads to a respectable speedup of 2.7 over the sequential code, as compared with a speedup of 3.1 obtained using the explicitly parallel approach.

Similar issues are likely to arise in the context of parallelizing any non-trivial application, using a hand-optimized approach versus using a library which transparently parallelizes various operations. A library-based approach must inherently rely on local optimizations. So even if the individual library routines are highly optimized, the overall performance may not match the performance of a globally optimized version of the code. Furthermore, exclusive reliance on a library-based approach precludes parallelization at a higher level, where different library calls are executed in parallel or grouped together to form partitions of work that are executed in parallel. On the other hand, the library-based approach provides a very convenient framework for end-user applications, written in a serial style, to obtain the benefits of parallelism in a transparent manner. For many users, the performance differential would be acceptable, in return for the benefits of easier development and maintenance of their codes.

6 Related work

Related work on data mining was already discussed in Section 2. Here we focus more on work related to our Java approach. Our implicitly parallel approach was motivated by the examples of DB2 [11] and SMP ESSL [12]. DB2 is a parallel database subsystem that presents a sequential interface to the user. Parallelization within queries is accomplished entirely within the database engine. Programs using SMP ESSL are coded in the same way as programs using the sequential ESSL library. SMP ESSL transparently provides parallel implementations of the required function.

Scientific libraries for performing array and BLAS operations are available for practically all architectures that are used extensively for technical computing. In the more specific area of libraries for Java, there have been several efforts. Casanova and Dongarra [6] describe wrappers for native implementations of ScaLAPACK and other numerical routines that allow them to be called from Java. Bik and Gannon take a similar strategy in [3]. In the Harpoon project [4], Blount and Chatterjee have developed a Java version of LAPACK. All of these efforts have attempted to develop high performance numerical libraries that rely very little on compilers for performance.

Array operations are natural candidates for exploitation of parallelism. This has been done extensively before both at the language [7, 26, 5] and subsystem [13, 22, 21] levels, with different degrees of transparency to the application programmer.

Our approach to producing a standard Array package [1] has been to develop a pure Java implementation that clearly expresses the semantics of the package and allows compiler optimizations to deliver good performance. The functions in the package are parallelized using standard Java features. Thus, in addition to good sequential performance, we offer a completely portable and transparent parallel subsystem for high-performance array processing in Java.

7 Conclusions

By comparing the performance of several production quality implementations of a data mining application, we have demonstrated that very high performance – on par with Fortran 90 – can be achieved using Java. Moreover, we have shown that not only can parallelism be successfully exploited in Java but that, with properly designed standard classes, applications can be successfully parallelized without compiler or programmer intervention. Finally, we show that a well-designed standard Array package not only simplifies program development, but is essential to enable important compiler optimizations and achieve high performance. These results support the inclusion of a standard Array package in Java, as proposed by the Java Grande Forum [1]. We conclude that Java will be an important force in technical computing, and that parallel subsystems will be an important tool for fully utilizing small-scale parallel systems.

References

[1] Java Grande Forum report: Making Java work for high-end computing. Java Grande Forum Panel, SC98, Orlando, FL, November 1998. URL: `http://www.javagrande.org/reports.htm`.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. In *Proceedings of the ACM-SIGMOD 1993 International Conference on Management of Data*, pages 207–216, Washington, DC, May 1993.

[3] A. Bik and D. B. Gannon. A note on level 1 BLAS in Java. In *Proceedings of the Workshop on Java for Computational Science and Engineering - Simulation and Modeling II*, June 1997. URL `http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html`.

[4] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proceedings of ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1998.

[5] F. Bodin, P. Beckmann, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[6] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency, Pract. Exp. (UK)*, 9(11):1279–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[7] W.-M. Ching and D. Ju. Execution of automatically parallelized APL programs on RP3. *IBM Journal of Research and Development*, 35(5/6):767–777, 1991.

[8] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[9] R. Gupta. A fresh look at optimizing array bounds checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[10] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[11] International Business Machines Corporation. *DATABASE 2 Parallel Edition for AIX Administration Guide*, June 1996.

[12] International Business Machines Corporation. *IBM Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

[13] International Business Machines Corporation. *IBM Parallel Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

[14] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ, a fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[15] S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[16] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to Megaflops: Java for technical computing. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*, 1998. IBM Research Report 21166.

[17] J. E. Moreira, S. P. Midkiff, and M. Gupta. Ninja: Numerically Intensive Java. Project page for the Numerically Intensive Java project at URL `http://www.research.ibm.com/ninja`, December 1998.

[18] J. E. Moreira, S. P. Midkiff, M. Gupta, and P. V. Artigas. Java programming for high performance numerical computing. Technical Report 21481, IBM Research Division, 1999.

[19] J. E. Moreira, S. P. Midkiff, M. Gupta, and P. V. Artigas. Numerically Intensive Java. URL: `http://www.alphaWorks.ibm.com/tech/ninja`, April 1999. A sequential version of the Array package is available for download.

[20] Communications of the ACM. *Special Issue on Information Filtering*, volume 35(12). Association for Computing Machinery, December 1992.

[21] R. Parsons and D. Quinlan. Run time recognition of task parallelism within the P++ parallel array class library. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 77–86. IEEE Comput. Soc. Press, October 1993.

[22] J. V. W. Reynders, J. C. Cummings, M. Tholburn, P. J. Hinker, S. R. Atlas, S. Banerjee, M. Srikant, W. F. Humphrey, S. R. Karmesin, and K. Keahey. POOMA: A framework for scientific simulation on parallel architectures. In *Proceedings of First International Workshop on High Level Programming Models and Supportive Environments, Honolulu, HI*, pages 41–49, April 16 1996. Technical report available at `http://www.acl.lanl.gov/PoomaFramework/papers/papers.html`.

[23] V. Seshadri. IBM high performance compiler for Java. AIXpert Magazine, September 1997. URL: `http://www.developer.ibm.com/library/aixpert`.

[24] U. Shardanand and P. Maes. Social information filtering: algorithms for automated "word of mouth". In *CHI-95 Conference Proceedings*, Denver, CO, May 1995.

[25] E. Simoudis. Reality check for data mining. *IEEE Expert: Intelligent systems and their applications*, 26(3), October 1996.

[26] R. G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4):498–512, 1991.