RC 21326 (96083) 02NOV98 Computer Science/Mathematics

Research Report

Parallel Data Mining in Java

José E. Moreira, Samuel P. Midkiff, Manish Gupta, Richard D. Lawrence
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center [Publications 16-220 ykt] P. O. Box 218, Yorktown Heights, NY 10598. email: `reports@us.ibm.com`
Some reports are available on the internet at `http://domino.watson.ibm.com/library/CyberDig.nsf/home`

This page intentionally left blank.

Parallel Data Mining in Java

J. E. Moreira S. P. Midkiff M. Gupta R. D. 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. Three sequential versions of the data mining application were developed: A sequential Fortran 90 version used as a performance reference, a plain Java implementation that only uses the primitive array structures from the language, and a baseline Java implementation that uses a Java Array package designed by us. When desired, this Array package provides parallelism at the individual Array and BLAS operation level. We have also developed two parallel Java versions: one that relies entirely on the parallelism from 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 presented to the application programmer with the different Java versions. Our studies show that not only is the Java implementation of this program competitive with the Fortran implementation, but that it parallelizes well, both in the implicit as well as the explicit form. The Java implementation achieves a performance of 109 Mflops, or 91% of Fortran performance, on a 332 MHz PowerPC 604e processor. On an SMP with four of those 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

Organizations with large volumes of data have always tried to maximize the useful information that can be extracted from that data. Historically, the organization's knowledge about relationships in the data have been used to arrange and access the data. In the last few years, self-training systems have been used to *mine* data with the goal of uncovering new relationships. Because the volume of data is large, high performance implementations and parallelism are important in these *data mining* systems.

In many ways Java is an ideal language to implement data mining operations. Java supports portable parallel programming on several 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 an additional advantage, there is a growing number of Java programmers.

The primary worry about Java has been its efficiency. In this paper we study the trade-offs between performance and the other benefits of Java by contrasting five implementations of the *scoring* phase of a production quality data mining application. The first implementation is written in Fortran 90, and is used as the reference for performance. The first Java implementation is built using plain Java, that is only Java and its standard libraries. The second Java implementation uses an Array package we have developed, and that is being proposed as a standard by the Java Grande Forum [10]. The computational core of the program uses a BLAS [8] class that we developed for the aforementioned 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) [14] or parallel DB2 [13] subsystems. This second Java version can be considered both a serial version and an implicitly parallel version. The final Java implementation is explicitly parallel, also using the Array package and BLAS, but exploiting parallelism through multiple threads at the application level.

These different implementations allow us to make a fair assessment of Java performance 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 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 data mining, and the particular data mining problem we solved to obtain our experimental results. 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 mentioned above. Section 5 presents the performance results for these implementations, and Section 6 discusses the implications of the experimental data. Section 7 discusses related work in the area, and Section 8 presents our conclusions.

2 Data mining and product recommendation

Data mining [23] is the process of discovering useful and previously unknown information in very large data bases, with applications that span scientific (*e.g.*, image processing) and commercial (*e.g.*, fraud detection) computing. 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 [16] and associations algorithms [1]. High-performance data mining applications, therefore, require both fast data access and fast computation.

The specific data mining problem considered here involves recommending new products to customers based on previous spending behavior. A number of recommendation systems [19] based on collaborative filtering [22] have been developed over the past several years, 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. Several different metrics have been used to compute similarity between user profiles.

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 spending categories. In the specific application discussed here, we utilize approximately two thousand such categories. Recommendations are drawn from a subset of eligible products. For each customer, a personalized recommendation list is generated by re-sorting this list of eligible products according to a customer-specific score computed for each 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 is defined by 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 [1] 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 organized so that products with the same nonzero patterns in their affinity vectors are adjacent. This is shown in Figure 1. Because of this organization, matrix A can be represented by a set of l groups of products. Each group i is defined by f_i and l_i, the indices of the first and last products in the group, respectively. All products with the same nonzero pattern belong to the same group. This 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 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 $g_i \times$ len(I_i) dense matrix with only the nonzero entries or rows f_i to l_i of A. This representation of A is shown in Figure 1.

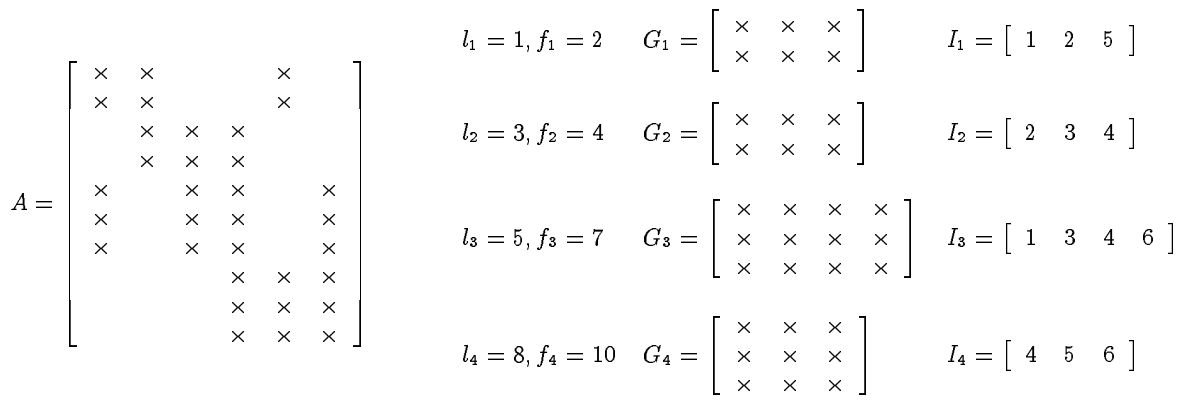


Figure 1: Structure and representation of the affinity matrix A.

Each customer j is defined by its *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. 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. Matrix N does not have to be represented explicitly.

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

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

$$S = \begin{bmatrix} \times & & \times & & \times & \\ \times & \times & & & \times & \\ \times & \times & \times & \times & & \\ & \times & \times & \times & & \\ \times & \times & & \times & & \times \\ \times & & \times & \times & & \times \\ & & \times & \times & & \times \\ \times & & & \times & \times & \end{bmatrix} \quad N = \begin{bmatrix} 1 & & 1 & & 1 & \\ 1 & 1 & & & 1 & \\ 1 & 1 & 1 & 1 & & \\ & 1 & 1 & 1 & & \\ 1 & 1 & & 1 & & 1 \\ 1 & & 1 & 1 & & 1 \\ & & 1 & 1 & & 1 \\ 1 & & & 1 & 1 & \end{bmatrix}$$

$$\begin{array}{lll} \sigma_1 = [\ \times\ \times\ \times\ ] & & J_1 = [\ 1\ 3\ 5\ ] \\ \sigma_2 = [\ \times\ \times\ \times\ ] & & J_2 = [\ 1\ 2\ 5\ ] \\ \sigma_3 = [\ \times\ \times\ \times\ \times\ ] & & J_3 = [\ 1\ 2\ 3\ 4\ ] \\ \sigma_4 = [\ \times\ \times\ \times\ ] & & J_4 = [\ 2\ 3\ 4\ ] \\ \sigma_5 = [\ \times\ \times\ \times\ \times\ ] & & J_5 = [\ 1\ 2\ 4\ 6\ ] \\ \sigma_6 = [\ \times\ \times\ \times\ \times\ ] & & J_6 = [\ 1\ 3\ 4\ 6\ ] \\ \sigma_7 = [\ \times\ \times\ \times\ ] & & J_7 = [\ 3\ 4\ 6\ ] \\ \sigma_8 = [\ \times\ \times\ \times\ ] & & J_8 = [\ 1\ 4\ 5\ ] \end{array}$$

Figure 2: Structures of the spending matrix S and normalizing matrix N, and representation of matrix S.

Note that $\sum_{k=1}^{n} A_{ik} S_{jk}$ 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 $\Lambda(\rho_1, \rho_2, \ldots, \rho_m)$ denote the $m \times m$ diagonal matrix Λ with $\Lambda_{11} = \rho_1, \Lambda_{22} = \rho_2, \ldots, \Lambda_{mm} = \rho_m$. Then, Equation (2) can be rewritten as a matrix operation:

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

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

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:

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

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:

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

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

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

where $\times$ now denotes matrix-vector multiplication and $\div$ denotes one-dimensional array division. Note that we have two gather operations, 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 multiplications and array division:

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

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.

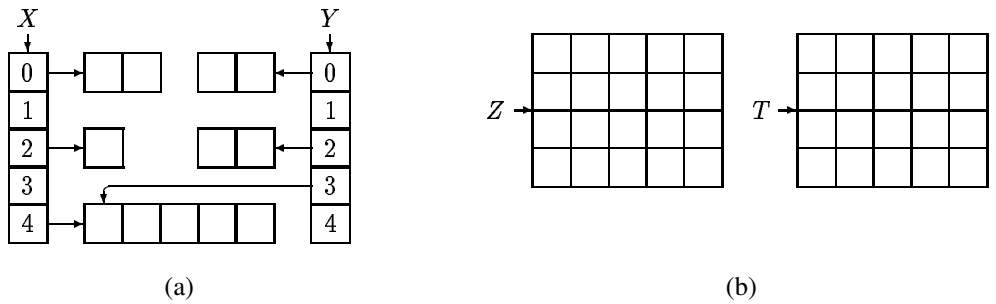
3 The Array package and BLAS class

This section describes a package that implements true multidimensional array classes, a BLAS class for these arrays, and a technique for transparently exploiting parallelism within operations supported by these classes.

3.1 The Array package

The array package provides a collection of classes that implement 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). (So far, we have restricted our implementation to one- through three-dimensional arrays of **double**s.) 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.

Java does not directly support multidimensional arrays. Instead, it simulates them to some degree with *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 **doubleArray2D**s. Storage in Java arrays is fragmented among 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 $X[4]$ and $Y[3]$ are aliased to each other in Figure 3(a).) 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. A final, and very important, benefit is that the array class allows bounds checking optimizations [17, 18] 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.



(a) (b)

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

The multidimensional arrays in the array package fix many of the performance problems 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. Finally, the rectangular nature of the data structures lends itself nicely to parallelization. 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.

To illustrate the difference 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 [18] 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 $c \neq b$. (In fact, because we support array sections, we have to show that $c.x \neq a.x$ and $c.x \neq b.x$, where x is an internal scalar field of our array classes.) 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 $c[i] \neq c[j]$ $\forall i, j$ $i \neq j$.)

```
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**.

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 $m - 1$ 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 away bounds checking during the computation, according to the techniques described in [11, 12].

The `divide` code illustrates an important design principle for the array package: We do not start any visible computation before we are sure we will complete it. That means aggressive optimizations can be performed in the actual computation part of the method, without having to worry about possible exceptions. In particular, the computation can be parallelized as long as data dependences are honored. Parallelization of the division operation is accomplished by splitting the `i`-loop among several threads using a block distribution. This parallelization is totally transparent to the caller of the method. The precise mechanism adopted for parallelization will be discussed in Section 3.3.

```
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 Basic Linear Algebra Subprograms (BLAS) [8] are the building blocks of efficient linear algebra codes. BLAS implements a variety of elementary vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3) operations. We have designed the BLAS class as part of the array package. It provides the functionality of the BLAS operations for the multidimensional arrays in that package. 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 would work 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, to linear algebra in general and to our data mining code in particular, 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. In the actual

code for dgemm the loops are blocked and unrolled for better memory behavior, register allocation, and instruction-level parallelism [18]. This leads to good sequential performance, as described below. The blocked structure is also used for internal parallelization of dgemm, using the approach described in [18].

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

    if (transa) a = a.permuteAxes(1,0);
    if (transb) 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 shows the performance achieved by our dgemm implementation executing on an SMP machine with 4×332 MHz PowerPC 604e processors. 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 execution achieves consistent performance ($\sim$ 180 Mflops.) throughout the entire range of problem sizes. The performance of the multithreaded 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.

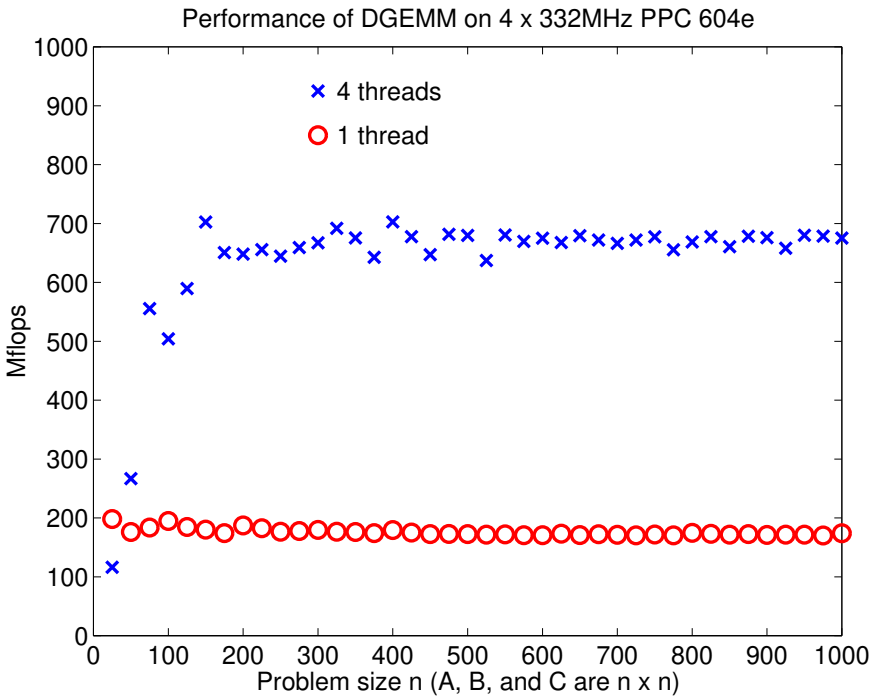


Figure 7: Performance of `dgemm` in the Blas class.

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 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.

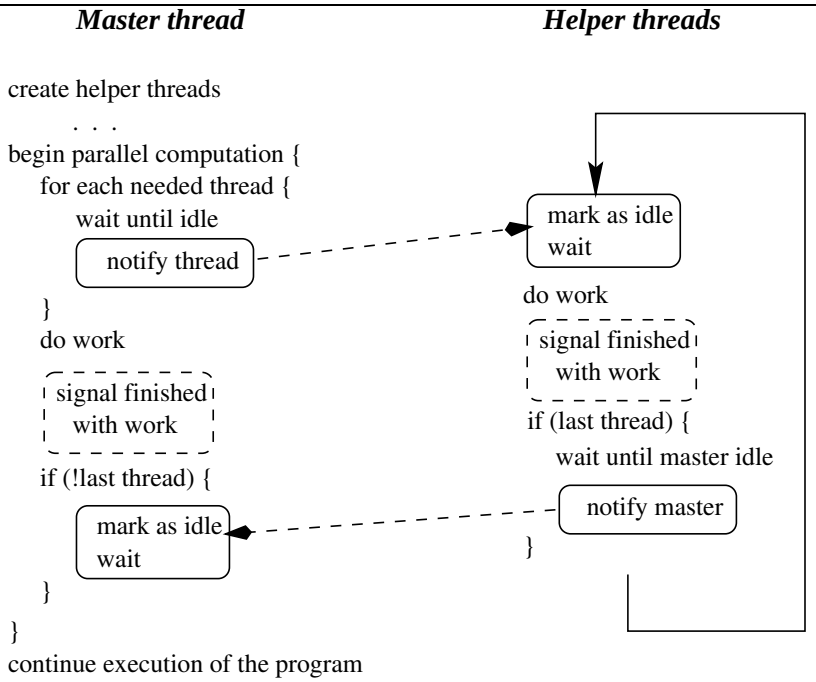


Figure 8: Overview of the thread scheduling operations.

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 using those classes. From the baseline we derive *implicitly* and *explicitly* parallel 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.

The baseline Java implementation: In this implementation, each G_i matrix is represented by a **doubleArray2D** object `G` 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 the block. The diagonal

```
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.

matrix Λ is stored as a **doubleArray2D** object `Lambda`, with all the elements of row k set to ρ_k. This allows the matrix multiplication by Λ to be replaced by an array, or element-by-element, multiplication. 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(false,false,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(false,false,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 call sets up the collection of threads used by the Array package. 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 describe the performance results obtained for different implementations of the data-mining code on an IBM 4-way SMP using 332 MHz PowerPC 604e processors. We show the performance of the scoring algorithm, which is the bulk of the computation in the code. We used the IBM HPCJ compiler [7] (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.

Figure 11 shows the serial performance of the Java and Fortran 90 codes, as a fraction of the performance obtained by the Fortran 90 version. 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.

Figure 12 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.

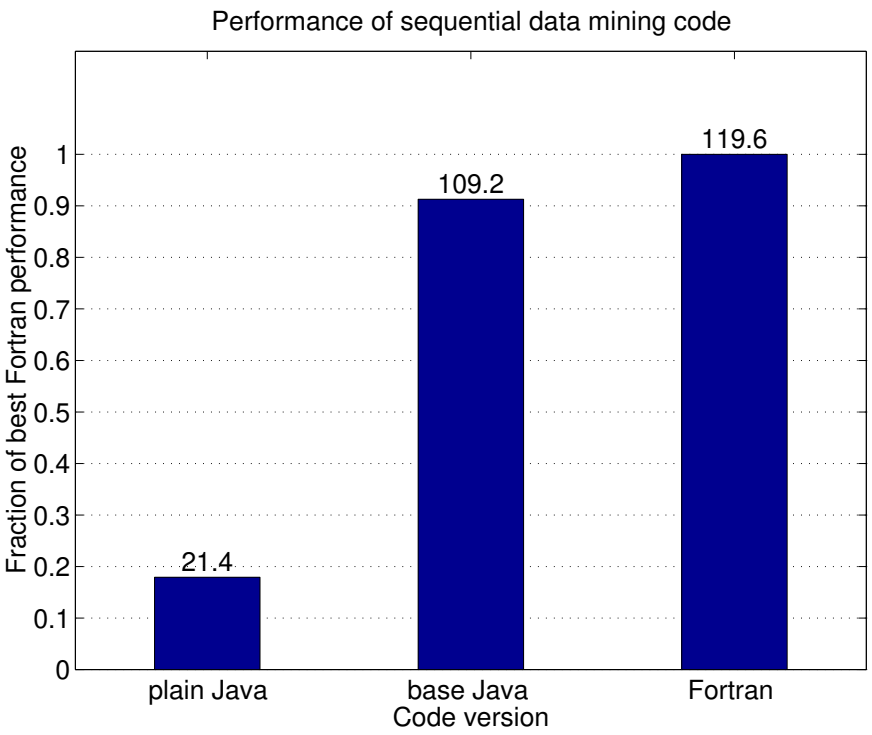


Figure 11: Performance of plain Java, baseline Java, and Fortran 90 versions of the data mining code.

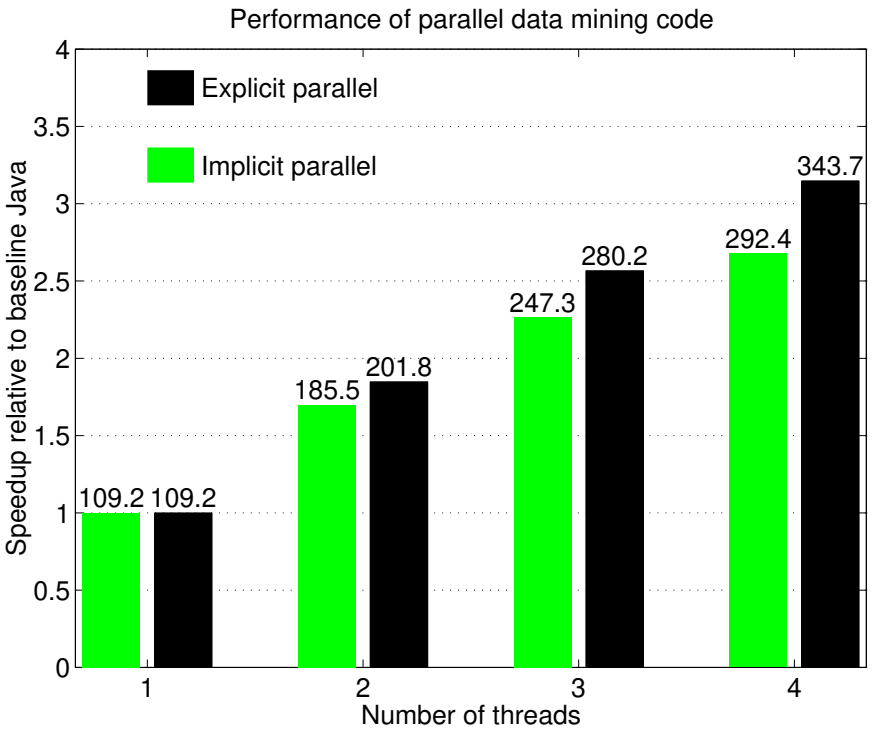


Figure 12: Performance of explicitly parallel and implicitly parallel versions of the data mining code.

6 Discussion

In this section, we discuss the reasons for the performance differences we have seen in the various implementations of the data mining code. We also draw some general conclusions about writing numerically intensive applications using Java. Finally, we compare the two approaches to exploiting parallelism: (i) writing explicitly parallel code, and (ii) using a parallel subsystem to extract parallelism out of an application written in a completely sequential style.

6.1 Performance of Serial Versions

It is clear that the Array package and the BLAS class provide a higher level of abstraction to the programmer than the primitive operations on arrays currently defined in Java. Our experimental results clearly demonstrate an important additional benefit to using these classes, namely, higher performance. There are two main reasons for the difference in performance between the plain Java version that uses primitive array operations and the baseline Java version: (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, used by the baseline Java version.

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 [18] to deal with this problem if the program uses the Array package.

The BLAS class provides the benefit of making a tuned implementation of commonly needed operations, such as matrix-multiplication, available to the programmer. This is similar to the approach adopted by providers of high-performance libraries like LAPACK [8] and ESSL [14]. As with the implementation of the Array package, all checks for conditions leading to exceptions are performed at the beginning, so that compiler optimizations can be applied effectively to performance-critical sections of the code.

We observe that by using the Array package and the BLAS class, which do 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 even on numerically-intensive production quality codes with pure Java.

6.2 Performance of Parallel Versions

Our experiments with explicitly parallel and implicitly parallel implementations of the data mining code 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 and the BLAS class entirely hides the complexity of parallelization from the end-user. The user merely has to specify the degree of parallelism desired.

Clearly, 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, (ii) greater load imbalance, (iii) more serial components, and (iv) poorer cache utilization 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.

7 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 [13] and SMP ESSL [14]. 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 [5] 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 [2]. In the Harpoon project [3], Blount and Chaterjee 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 [6, 24, 4] and subsystem [15, 21, 20] levels, with different degrees of transparency to the application programmer.

Our approach to producing a standard array package [9] has been to develop implementations, in Java itself, that cleanly express the semantics of the class, while allowing compiler optimizations to deliver good performance [25]. This allows us to also achieve good performance on array based computations that do not use higher-level array

operations. For programs that make use of the higher-level array operations, we build high performance numerical functions on top of this standard package. These functions 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.

8 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 [9]. 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.

Acknowledgments: The authors wish to thank Marc Snir for for strongly supporting our research. We also wish to thank Ven Seshadri of IBM Canada for helping and supporting our experiments with HPCJ.

References

[1] 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.

[2] 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. available at `http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html`.

[3] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proceedings of ISCOPE'98*, 1998. *To appear*.

[4] 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.

[5] 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.

[6] 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.

[7] IBM Corporation. IBM High Performance Compiler for Java, 1997. Information available in Web page at `http://simont01.torolab.ibm.com/hpj/hpj.html`, available for download at `http://www.alphaWorks.ibm.com/formula`.

[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] Java Grande Forum. Java Grande Charter. `http://www.javagrande.org/public.htm`.

[10] Java Grande Forum. Issues in numerical computing with Java. Document available at URL `http://math.nist.gov/javanumerics/issues.html`, March 1998.

[11] 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.

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

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

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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`.

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

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

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

[25] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Improving java performance through semantic inlining. Technical Report 21313, IBM Research Division, 1998. Submitted for publication.