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 [1]. 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 [13]. (We emphasize that the Array package is written entirely in Java and requires absolutely no modifications to the

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

language.) The computational core of the program uses a BLAS [7] 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) [11] or parallel DB2 [10] 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 [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 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

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

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 from rows f_i to l_i of A.

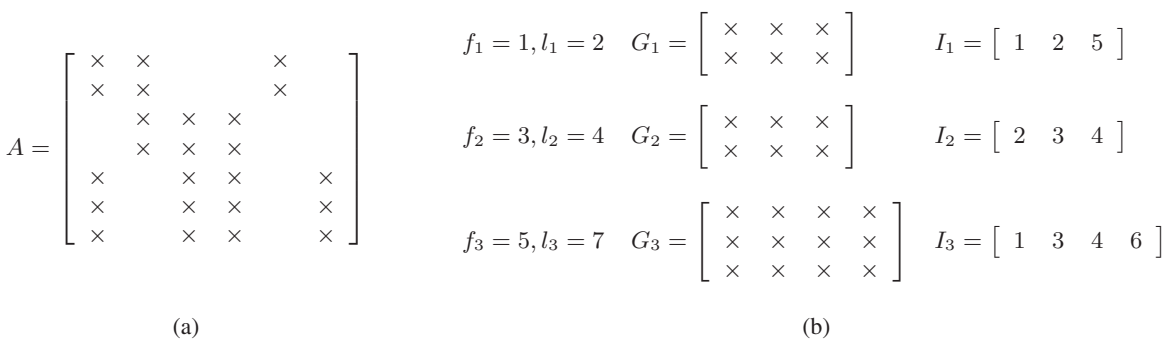


Figure 1: Structure (a) and representation (b) of the affinity matrix A.

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.

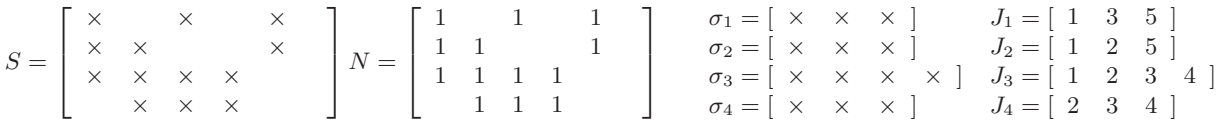


Figure 2: Structure of the spending matrix S and normalizing matrix N (a), representation of matrix S (b).

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

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

Λ with $\Lambda_{ij} = \rho_i, 1 \leq i \leq m, 1 \leq j \leq p$. 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:

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

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:

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

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

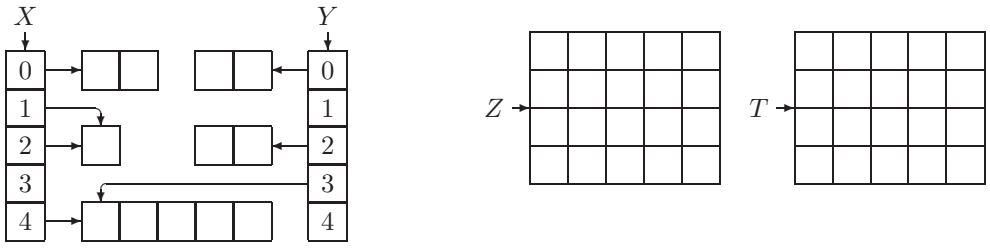
Array and matrix operations are building blocks for many numerically intensive applications, including our data mining code. To address various issues in coding these applications in Java, we have developed an Array package. The goal of this package is to introduce in Java the functionality and performance 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.

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

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.



(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 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 $m - 1$ 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 [8, 9].

Proceedings of the ACM/IEEE SC99 Conference (SC'99)
1-58113-091-0/99 $ 17.00 © 1999 IEEE

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.