Towards Automatic Support of Parallel Sparse Computation in Java with Continuous Compilation*

Rong-Guey Chang Cheng-Wei Chen
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Tyng-Ruey Chuang
Institute of Information Science, Academia Sinica, Taipei, Taiwan

Jenq Kuen Lee
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Abstract

In this paper, we present a generic matrix class in Java and a runtime environment with continuous compilations aiming to support automatic parallelization of sparse computations on distributed environments. Our package comes with a collection of matrix classes including operators of dense matrix, sparse matrix, and parallel matrix on distributed memory environments. In our environment, a program such as conjugate gradient solver is written by users with high-level generic matrix notations in Java. At runtime, with the help of profiling information and cost model, our runtime system employs continuous compilation schemes to rewrite the matrix notations into corresponding parallel operations. Our system is particularly useful in optimizing sparse computations on distributed environments. Our runtime compilation environment selects compression and distribution schemes for sparse matrices on distributed memory environments according to the access patterns of the programs and the non-zero structures of the sparse matrices gathered from profiling. Our testbed is currently based on Java and PVM[18] on SUN and DEC workstation clusters. Preliminary experimental result shows our platform can speed up programs significantly with the help of our runtime continuous compilations.

Keywords: Parallel Sparse Matrix Support, Parallel Java, Distributed Memory Machines, Distribution Schemes, Continuous Compilations

1 Introduction

With the recent advent of Internet and World Wide Web, the Web has become an important infrastructure for distributed computing, as it provides rich machine resources connected via internet. At the same time, with growing popularity, Java has gained an important role for developments of future Web and internet softwares as well as for the support of parallel and distributed computings. While Java is gaining momentum, many important questions remain to be addressed. With the emerging of many new technologies and the rapid shift in compilation paradigms in Java and mobile-code environments, one key question we ask is as follows. Are the new technologies and the shift in compilation paradigms providing us with new opportunities to solve some of the hazard problems we faced in the past with parallel computing?

*This paper is submitted to *ACM 1997 Workshop on Java for Science and Engineering Computation*. The correspondence author of this work is Jenq Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw, fax number is +886 3-5723694, phone number is +886 3-5715131 ext. 3519, and postal address is Prof. Jenq Kuen Lee, Department of Computer Science, National Tsing–Hua University, Hsinchu, Taiwan.

In this paper, we use a parallel java environment on distributed memory environments and employ a newly emerging compilation paradigm, called continuous compilation [16], to address a long challenging research topic in parallel computing: the support of parallel executions of sparse matrix computations on distributed environments. We feel that this problem has been difficult at least for the following three reasons. First, the cost of a sparse matrix computation depends greatly on the structures (*i.e.*, the distributions of non–zero elements) of its input matrices [2]. Such information, however, may not be available at static compile–time for optimization purposes. If it is indeed available at compile–time, the non–zero structure of a sparse matrix may change over time during the execution, which makes it more difficult for selecting good compression and distribution schemes. Second, representations of sparse matrices, such as their compression and distribution schemes, are usually hard–wired into the application programs, if conventional languages are used. One must undo the hard–wiring in the program just to get a global view of the computation. Then analyses can be performed to exploit optimization opportunities. This is changing due to the use of high–level array languages, such as APL, Fortran 90, and MATLAB, and portable sparse libraries [11] [17] for large scale computations. Still, programmers face the problem of proper annotations for sparse operations [19]. It is not clear how many of these annotations are needed, and where and when to put them, in order for a compiler to fully utilize them to produce efficient code. Third, even with complete information about its sparsity and computation structures, a sparse application may still need different distribution schemes when executed on different parallel systems. This is due to performance discrepancy (both on computation and communication) between different systems, and is an important issue in the designs and analyses of portable and scalable parallel applications [3] [8] [15].

With the paradigm shifting from static compilation into runtime compilations with Java and Web environments, we see opportunities to explore this new technology to help address the hard problem of parallel sparse computation. In this work, we present a generic matrix class and a runtime environment with continuous compilations to support parallel sparse computations in Java. Our package comes with a collection of matrix classes including operators of dense matrix, sparse matrix, and parallel matrix on distributed memory environments. In our environment, a program such as conjugate gradient solver is written by users with high-level generic matrix notations in Java. At runtime, with the help of profiling information, our runtime system employs continuous compilation schemes to rewrite the matrix notations into corresponding parallel operations. In the continuous compilation model, one execution thread is responsible for executing the program while the compilation module in a separate thread is responsible for translating the interpreted code into a natively executable format or an improved version of the byte code. Previously, in the parallel community, inspector and executor model [20] can be considered as one of the early form of the runtime compilations. Our techniques, partly derived from the emerging continuous compilation model [16], have quite a few new innovations when compared to the inspector and executor model. First the inspector and executor model puts the object code and runtime compiler code into one program. The separation of the target code and the runtime compiler into separate modules can lead to overlap the execution time [16] of these two modules in separate threads and have clear interfaces in the design of the compiler module (which can lead to fast progresses in the runtime compiler design.) Second, the inspector and executor model are not done in the context of object-oriented environments. As we will demonstrate it later in this work, our proposed methodology, called runtime binding, can bind the generic matrix object with the selected optimized distributed and compressed matrix object at runtime. Our runtime binding scheme for objects at runtime by runtime compiler according to the profiling information can significantly improve the performance of the programs, while minimizing the required revisions in the byte codes or target codes. Finally, the continuous compilation model emphasizes the aspects of the the continuous

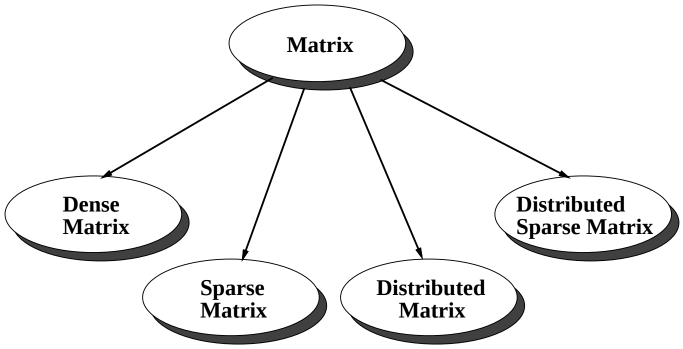

Figure 1: Matrix Class Hierarchies in Java

process of the compilation at runtime to continuously improve the execution codes.

Our runtime compilation environment also selects the optimized compression and distribution schemes for sparse matrices on distributed memory environments according to the access patterns of the programs and the the non-zero of the sparse matrices gathered from profiling. ¡Our testbed is currently based on Java and PVM on SUN and DEC workstation clusters. Preliminary experimental result shows our significantly performance improvement with the help of our runtime compilation techniques.

The remaining part of this paper is organized as the following. Section 2 presents our matrix class support. We then describe in Section 3 the sampling and analytical method for extracting non–zero structures from sparse matrices, and for formulating the cost functions of sparse operations. Next, Section 4 describes our runtime and continuous compilation environment. Preliminary experimental results given in Section 5. Related work is discussed, and a short conclusion is made, in section 6.

2 Generic Matrix Packages

In this section, we present the class hierarchies of our matrix class built in Java. Figure 1 illustrates the class hierarchy. The package comes with a collection of matrix classes including operators of dense matrix, sparse matrix, and parallel dense and sparse matrix on distributed memory environments. In the top of the hierarchy, a generic matrix class is built, and programmers write programs by using the generic matrix notations in the top level. The matrix class supports data parallel programming with the support of matrix addition, substraction, multiplication, etc. In the second level of the hierarchy, several specialized matrix were built with the similar interfaces of the generic matrix in the top level. The specialized matrix class include sequential sparse matrix, parallel dense matrix, distributed sequential matrix, etc. The parallel dense matrix includes the distribution scheme of the matrix, while the parallel sparse matrix can be built according to the specified compression and distribution schemes. To illustrate the idea, we first give part of the code fragments of the generic matrix in the top level below.

```
public class Matrix {
     . . .
     public Matrix();
     public Matrix(int r, int c) ;
     public Matrix set(int r, int c, double val);
     public Matrix addAssign(Matrix m1) ;
```

Table 1: The Domains for Distribution/Compression Scheme Selections

Compression Scheme
Compressed Row Storage
Compressed Column Storage
Dense Representation

Distribution Scheme
(Block, *)
(*,Block)
(Block, Block)

```
        public Matrix add(Matrix m1) ;
        public Matrix mul(Matrix m1) ;
        public Vector mul(Vector v1);
        . . .
    }
```

In our environment, a program such as conjugate gradient solver is written by users with generic matrix notations in Java. The second level of the classes will be loaded at runtime. At runtime, with the help of profiling information, our runtime system rewrites the matrix notations into corresponding parallel operations. Our runtime compilation environment selects compression and distribution schemes for sparse matrices on distributed memory environments according to the access patterns of the programs and the the sampling of the sparse matrices gathered from profiling. Currently, our distributed sparse matrix class supports the compression schemes, both CRS and CCS, and the distribution schemes MRD and Cyclic style distribution among processors [19]. Table 1 lists the compression and distribution schemes for parallel sparse matrix on distributed environments.

The specialized matrix were built with the similar interfaces of the generic matrix in the top level. We show part of the interfaces of the parallel sparse matrix below.

```
public class DistributedSparseMatrix extends Matrix {
        double DA[];
        int CO[];
        int RO[];
        . . .
        public DistributedSparseMatrix();
        public DistributedSparseMatrix mulAssign(DistributedSparseMatrix m1) ;
        public DistributedSparseMatrix mul(DistributedSparseMatrix m1) ;
        public Vector mul(Vector v1) ;
        . . .
    }
```

An example of the conjugate gradient solver is given below.

```
    public class cgsolve {

      public static Vector cgsolve(Matrix A, Vector b, double tol) {
      //function x = cgsolve (A, b, tol)
```

```
            Vector x = new Vector(b.getSize()); //x = zeros(size(b));
            Vector r = new Vector(b); //r = b;
            double rtr = r.innerProduct(r); //rtr = r' * r;
            Vector p = new Vector(b.getSize()); //p = zeros(size(b));
            double beta = 0.0; //beta = 0;

            Vector Ap = new Vector(b.getSize());
            double alpha;
            double rtrold;

            while ( r.norm() > tol * b.norm() ) { //while ( norm(r) > tol * norm(b) )
                    p.set(r.add(p.mul(beta))); //p = r  + beta * p;
                    Ap.set(A.mul(p)); // Ap = A * p;
                    alpha = rtr / p.innerProduct(Ap); //alpha = rtr / (p' * Ap);
                    x.addAssign(p.mul(alpha)); //x = x + alpha * p;
                    r.subAssign(Ap.mul(alpha)); //r = r - alpha * Ap;
                    rtrold = rtr;
                    rtr = r.innerProduct(r); //rtr = r' * r;
                    beta = rtr / rtrold;
           } //end

           return x; //x = cgsolve (A, b, tol)
      }
  }
```

In our environment, a program such as conjugate gradient solver is written by users with generic matrix notations in Java. At runtime, our runtime system will rewrite the matrix notations into corresponding parallel operations.

3 Runtime Sampling and Analysis

The Java language and environment provide ample opportunities to experiment with dynamic program analysis and transformation. The Java language and runtime allow one to load and execute code on demand, to perform object introspection at runtime (via the Core Reflection API), and to spawn new control threads as needed (while sharing the runtime data). These language features make it possible to perform program profiling and performance analysis (which traditionally are done after program execution) at the time of program execution, and the result can be used immediately to improve the on-going execution. Our framework of runtime program transformation can be carried out at the language level (by writing reflective programs), with the minimum requirements in going to the implementation level (by modifying the compiler or runtime).

We describe in this section our sampling and analysis techniques in the context of sparse matrix computation. While continuous compilation (or runtime compilation) usually refers to the generation of native code from source code (or intermediate code), in this section we concentrate on the selection of suitable compression and distribution scheme for matrices. Currently we use a set of different implementations of a matrix class, and dynamically select profitable implementations for matrix computations for the on-going program execution. The selection depends on the non-zero ratios of the matrices, on the structures of the matrix computation, and on the performance characteristics of the underlining execution environment. Technieques for dynamic binding of matrix implementations to matrix variables in a program will be described in section 4.

For each supported matrix operation, and given the non-zero ratios of its input arguments, we give formulae to estimate the non-zero ratio of the resultant matrix, as well as the cost of the operation. For matrices that are read from files or are generated incrementally, the distributions of non-zeros are sampled on–line when the matrix is being built. Coefficients in cost formulae are determined by performance measurements of the corresponding sparse routines on the underlining environment. In addition, we also formulate the cost functions of sending/receiving a block of sparse matrix using PVM routines. For parallel computation of a matrix operation, we also describe how the resultant matrix is partitioned among the processors, and how non-zero ratio of each partition aftre the computation can be calculated using ratios from partitions before the computation. The non-zero ratios of each matrix, as well as its dimensions and its compression and distribution details, are stored along with the matrix. During execution, such information are used along with the above mentioned formulae to select profitable compression and distribution scheme. The sampling and analysis prcoess can be executed in another thread.

We assume the distribution of non-zeros in a matrix is uniform matrix-wise. This restriction is not so severe since we can partition a sparse matrix into blocks whose non-zeros are more uniformly distributed. We use the following notations when discussing non-zero ratios of sparse matrices. $S[\alpha, m \times n]$ represents a matrix of $m \times n$ elements, of which αmn are non-zero. (The estimation α in fact is an upper bound of the non-zero ratio. This will become clear later in sectionsec:ratio.) The distribution of non-zero is assumed uniform matrix-wise. We often omit the matrix sizes if they are clear from the context. $S[0]$ is an all-zero matrix, while $S[1]$ is a matrix with almost no zero. Similarly, we use $s[\alpha]$ to denote a scalar of which non-zero occurs with probability (at most) α.

3.1 Non-Zero Ratios

We list in this section the formulae for estimating non-zero ratios of the resultant matrix for operations supported by our matrix class. We assume that zeros resulting from cancellation of additions is unlikely. (Hence α is an over-estimation of the actual non-zero ratio). In the following, `add` and `times` perform point-wise addition and multiplication for each matrix element, while `sum` and `product` accumulate over all matrix elements. `innerProduct` computes the inner product of two vectors, and `mul` performs a matrix multiplication.

$$\begin{aligned}
\texttt{add}(S[\alpha], S[\beta]) &= S[1-(1-\alpha)(1-\beta)] \\
\texttt{times}(S[\alpha], S[\beta]) &= S[\alpha\beta] \\
\texttt{sum}(S[\alpha, n]) &= s[1-(1-\alpha)^n)] \\
\texttt{product}(S[\alpha, n]) &= s[\alpha^n] \\
\texttt{innerProduct}(S[\alpha, n], S[\beta, n]) &= s[1-(1-\alpha\beta)^n] \\
\texttt{mul}(S[\alpha, m \times n], S[\beta, n \times l]) &= S[1-(1-\alpha\beta)^n, m \times l] \\
\texttt{transpose}(S[\alpha]) &= S[\alpha]
\end{aligned}$$

3.2 Computation and Communication Costs

For simplicity, we assume the matrices are organized in compressed row storages (CRS). Similar estimations for matrices using other compression schemes, such as compressed column storages (CCS), or for matrices of higher ranks, can be formulated as well but are omitted here. The positive constants $(c_0, c_1, c_2, \ldots)$ in each of the formulae will be determined from performance measurements on the underling execution environment using the corresponding sparse routine with matrices of various sizes and non-zero ratios.

For a point-wise matrix operation `op`, the computation cost is proportional to the total number of non-zeros, and proportional to the number of rows (since they are in CRS). The formula is:

$$Cost(\texttt{op}(S[\alpha, m \times n], S[\beta, m \times n])) = c_2(\alpha + \beta)mn + c_1 m + c_0$$

where constants c_0, c_1, c_2 depend on `op` and the execution environment.

For unary matrix operations `sum`, `product`, and `transpose`, the formulae are:

$$\begin{aligned}
Cost(\texttt{sum}(S[\alpha, n])) &= c_1 \alpha n + c_0 \\
Cost(\texttt{product}(S[\alpha, n])) &= c_1 \frac{1 - \alpha^{(n+1)}}{1 - \alpha} + c_0 \\
Cost(\texttt{transpose}(S[\alpha, m \times n])) &= c_2 \alpha mn + c_1(m + n) + c_0
\end{aligned}$$

Note that the cost of a matrix transpose is exactly the cost of converting a matrix from CRS to CCS. In addition, if the required compression scheme of the result of a `transpose` is the opposite scheme of its argument, then the transpose costs only constant time. In the following, we assume that for `mul`, the two matrices are in CRS and CCS respectively. If not, a simple `transpose` will transform each to the required compression scheme. The resultant matrix from a matrix multiplication will be in CRS.

$$\begin{aligned}
Cost(\texttt{innerProduct}(S[\alpha, n], S[\beta, n])) &= c_2(\alpha + \beta)n + c_1 \alpha \beta n + c_0 \\
Cost(\texttt{mul}(S[\alpha, m \times n], S[\beta, n \times l])) &= c_4(\alpha + \beta)mnl + c_3 \alpha \beta mnl + c_2 ml + \\
&\quad\; c_1(m + l) + c_0
\end{aligned}$$

Note that `innerProduct` performs an element-wise vector multiplication (costs: $c_2(\alpha+\beta)n$), sums up the resulting non-zeros (cost: $c_1 \alpha \beta n$), and needs an initial setup (cost: c_0). For `mul`, there are ml `DOT_PRODUCT` operations (cost: $(c_4(\alpha + \beta)n + c_3 \alpha \beta n + c_2)ml$). But carrying the loops along CRS and CCS needs $c_1(m + l)$. c_0 is the setup cost.

The cost function for sending/receiving a sparse matrix to/from other processor is simply formulated as:

$$Cost(\texttt{send/receive}(S[\alpha, n])) = c_1 \alpha n + c_0$$

where the coefficients will depend on the data type of the matrix elements, and be determined by performance characteristics of the PVM routines and the execution environments.

3.3 Global Structures

A sparse matrix may be partitioned into disjoint blocks for parallel computation. Assume the non-zero ratio of each partition of the input matrix is known in advance, we show in the following how to estimate the non-zero ratio for each partition of the resultant matrix.

Let *Ratio* be the sampling function such that $Ratio(A) = S[\alpha]$ for any sparse matrix A whose non-zero ratio is α. Likewise, $Ratio(a) = s[\alpha]$ whenever a is zero with probability α. For a point-wise matrix operations `op`, non-zero ratio of each of the resultant matrix partitions can be computed from the respective input matrix partitions. Let A_i be a partition of A. Then,

$$Ratio(\texttt{op}(A,B)_i) \quad = \quad \texttt{op}(Ratio(A_i), Ratio(B_i))$$

Note that `op` operates on different domains at the two sides of the above equation. It is a matrix operator at the left-hand side, but a ratio operator (as in section 3.1) at the right hand side.

For reduction operations such as `SUM` and `PRODUCT`:

$$\begin{aligned}
Ratio(\texttt{sum}(A)) &= s[1 - \prod_i (1-\alpha_i)], \quad \text{where } \texttt{sum}(Ratio(A_i)) = s[\alpha_i] \\
Ratio(\texttt{product}(A)) &= s[\prod_i \alpha_i], \quad \text{where } \texttt{product}(Ratio(A_i)) = s[\alpha_i]
\end{aligned}$$

and for `innerProduct` and `mul`:

$$\begin{aligned}
Ratio(\texttt{innerProduct}(A,B)) &= s[1 - \prod_i (1-\alpha_i)], \\
&\quad \text{where } \texttt{innerProduct}(Ratio(A_i), Ratio(B_i)) = s[\alpha_i] \\
Ratio(\texttt{mul}(A,B)_{i,k}) &= S[1 - \prod_j (1-\alpha_j)], \\
&\quad \text{where } \texttt{mul}(Ratio(A_{i,j}), Ratio(B_{j,k})) = S[\alpha_j]
\end{aligned}$$

Finally, for `transpose`:

$$Ratio(\texttt{transpose}(A)_{i,j}) \quad = \quad Ratio(A_{j,i})$$

3.4 Cost Models

Given k sparse arrays, $x_1, x_2, \ldots, x_k$, and m consecutive array operations, $op_1, op_2, \ldots, op_m$, in a basic block, we want to select the proper distribution and compression schemes for the k arrays. By analyses in section 3, we can estimate the sparsity structures of the k arrays and the computation cost for each of the m array operations.

Let δ_i be a selection of the compression and distribution scheme for array x_i. Our goal is to find a sequence of selections $\delta_1, \delta_2, \ldots, \delta_k$, such that it minimizes

$$\sum_{i=1}^{m} Total\_Cost(op_i, \delta_1, \ldots, \delta_k), \quad \forall \delta_j \in D, 1 \le j \le k$$

where D is the domain of all possible combinations of compression/distribution schemes. Currently we use a rather limited domain for D. It is summarized in Table 1.

Note that by (Block, *) we mean distributing the sparse matrix by row. The partition is not by an even division of the row index-range, but by the sparsity structure of the given matrix. The partition may even produce uneven distribution of non-zeros among the processors, as long as it achieves over-all balanced computation and communication. See section 5 for an example. Likewise, (*, Block) partitions the matrix by column, while (Block, Block) partitions the matrix both by row and column. Function *Total_Cost* calculates the total cost of computation and communication for an array operation for the specific compression/distribution schemes.

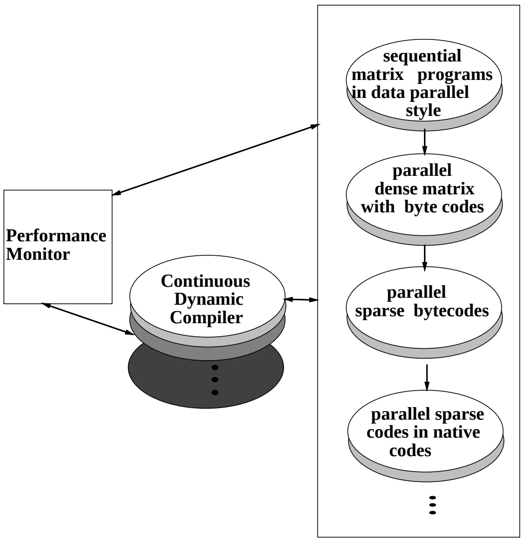

Figure 2: Continuous Compilation Support for Parallel Java Programs

4 Runtime Compilations and Implementations

Figure 2 illustrates the runtime compilation process of our system. It illustrates one scenario in our system. A Java program written with our generic matrix notation is first executed sequentially with the default selection of dense matrix computation. After being executed for a short while, with the profiling information the runtime compiler finds that the program is using matrix and data parallel programming notations, and is doing serious computation. The runtime compiler then rewrites the sequential byte code program into parallel matrix programs on distributed environments. (Our current environment is based on DEC and SUN workstation clusters.) The program then continues to execute in parallel on distributed environments. After a while, the runtime compilation process continues. Later, the profiling information indicates that some matrix of the major computation are with sparse structures. Therefore, the runtime compiler at this point re-writes the parallel dense program into parallel sparse programs on distributed environments. The issues involved at this point include compression scheme and distribution scheme selections by the runtime compiler. With the compilation process continues to evolve, the byte code program is eventually re-written into a parallel sparse program in native codes.

In this paper, we mainly focus on the issues involved in rewriting dense programs into sparse programs. This includes both rewriting sequential dense programs into sequential sparse programs, and rewriting parallel dense programs into parallel sparse programs on distributed environments. The issues to be considered in the runtime compilations include the followings.

- the loading status and availability of the computing resources in the Web.
- the profiling and the timing information on major computing blocks of the program.
- the major loops in the program.
- the appearances of the usage of operations of matrix packages.
- the runtime profiling information about the sparsity of major arrays.
- the access patterns to matrix class.

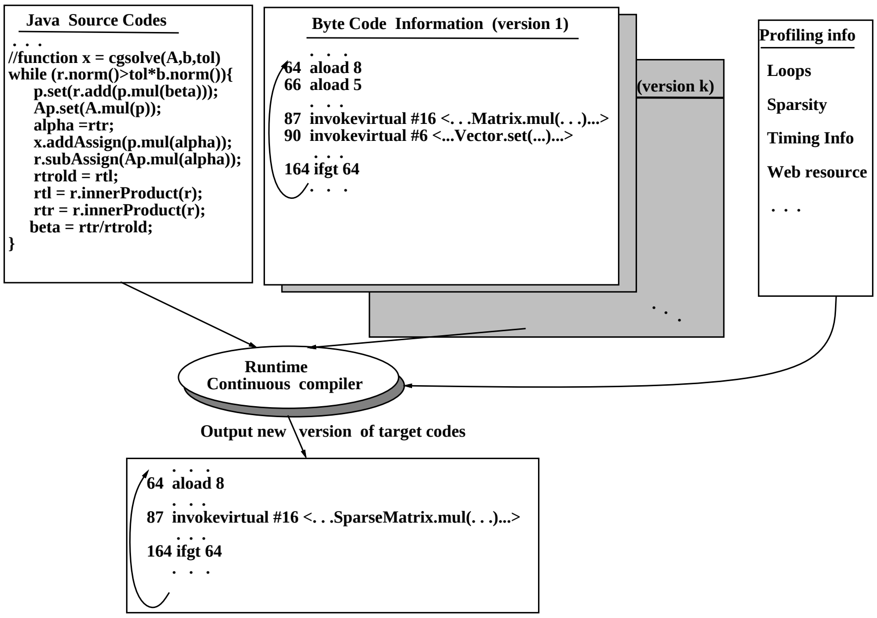

Figure 3: Information Involved in the Runtime Compilation Process

- the compression scheme which should be used for sparse or parallel sparse computations.
- the distribution scheme which should be used for matrices or sparse matrices on distributed environments.

Figure 3 illustrates the information needed by the runtime compilation. In the example of the conjugate gradient solver, with all the needed information available, the runtime compiler will select CRS compression scheme and (BLOCK,*) distribution scheme for the major matrix `A` in this case. Note that by (Block, *) we mean distributing the sparse matrix by row. The partition is not by an even division of the row index range, but by the sparsity structure of the given matrix. The selection is based on our model in Section 3.

Now we describe the way target code is produced by our rewriting system. Figure 4 illustrates the byte code produced for the conjugate gradient solver with our generic matrix notations in Java. Suppose at runtime the sparse matrix class has been selected by the continuous compilation process to substitute the general matrix class. We find two ways to perform the code substitution. The first way is to revise the byte code directly to produce a newer version of the byte code. For example, in this example, we can rewrite line 87 marked in Figure 4 into the following form.

87 invokevirtual #16 <Method SparseMatrix.mul(LVector;)LVector;>

which now uses the matrix-vector multiplication method from the sparse matrix class. In addition, we need to do proper housekeeping work in converting matrix object from one shape into another.

The second way for dynamic code substitution is to utilize a set of properly layered interfaces to the matrix classes. We develop an innovative technique based on dynamic binding of runtime objects. At runtime, matrix objects may come from different layers and support methods of specific requirement of execution (such as dense/sparse, compression/distribution, and sequential/parallel requirement). With our runtime binding scheme, we need not revise the byte codes at all. Three steps are needed when we want to change matrix computation from one format into another at any program point. First, the runtime compiler needs to create a new matrix object with, say, the

```
Method Vector cgsolve(Matrix,Vector,double)
  0 new #4 <Class Vector>
  3 dup
  4 aload_1
  5 invokevirtual #8 <Method Vector.getSize()I>
  8 invokenonvirtual #7 <Method Vector.<init>(I)V>
  11 astore 4
  . . .
  64 aload 8
  66 aload 5
  68 aload 8
  70 dload 9
  72 invokevirtual #5 <Method Vector.mul(D)LVector;>
  75 invokevirtual #9 <Method Vector.add(LVector;)LVector;>
  78 invokevirtual #6 <Method Vector.set(LVector;)LVector;>
  81 pop
  82 aload 11
  84 aload_0
  85 aload 8
  87 invokevirtual #16 <Method Matrix.mul(LVector;)LVector;>
  90 invokevirtual #6 <Method Vector.set(LVector;)LVector;>
  93 pop
  94 dload 6
  96 aload 8
  98 aload 11
  100 invokevirtual #11 <Method Vector.innerProduct(LVector;)D>
  103 ddiv
  104 dstore 12
  106 aload 4
  108 aload 8
  110 dload 12
  112 invokevirtual #5 <Method Vector.mul(D)LVector;>
  115 invokevirtual #10 <Method Vector.addAssign(LVector;)LVector;>
  118 pop
  119 aload 5
  121 aload 11
  123 dload 12
  125 invokevirtual #5 <Method Vector.mul(D)LVector;>
  128 invokevirtual #12 <Method Vector.subAssign(LVector;)LVector;>
  131 pop
  132 dload 6
  134 dstore 14
  136 aload 5
  138 aload 5
  140 invokevirtual #11 <Method Vector.innerProduct(LVector;)D>
  . . .
  162 dmul
  163 dcmpl
  164 ifgt 64
```

Figure 4: Byte Code Fragments of the Conjugate Gradient Solver in Java with Our Matrix Class

Table 2: The time spent for conjugate gradient solver with 10 iterations

Dense Computation	67.2 seconds
Sparse Notation	10.9 seconds
Conversion Time	4.7 seconds

selected compression and distribution schemes. Second, the system needs to convert matrix objects using the original matrix class into the new matrix objects of the specialize class. Finally, we cast the specialized matrix objects into the generic matrix objects. In addition, the binding of the generic matrix object is done by the runtime compiler according to sparsity of the major arrays. The above procedure is helped by the Java Core Reflection API, which allows us to inspect objects of their classes, and move the objects between layers at run-time.

5 Preliminary Results

In this section, we report preliminary performance results for our experiments with automatic support of sparse computation in Java programs. The high-level generic matrix class in Java described in Section 2 is used in our experiment, and part of the class hierarchy can be found in `http://puma.cs.nthu.edu.tw/~project/ParallelJavaTool`. We use two sparse data sets for our preliminary experiments. Figure 5 and 6 show the sparsity of the arrays used in our conjugate gradient solver. The first problem is a 800 by 800 matrix and the second set of the data is a 479 by 479 matrix. The first one is a sparse positive definite matrix generated by us. The second one is a sample from the toolbox of Matlab and a well-known test case [9]. We use the conjugate gradient solver written in Section 2 as our testing program. The program for the conjugate gradient solver is written with generic matrix notations in Java. At runtime, our runtime system running in a separate thread will rewrite the matrix notations into corresponding matrix operations of the matrix object in our second level of the matrix class hierarchy according to the sparsity of the arrays and the characteristics of the programs. In our first experiment, we compare the performance of the CG program using sequential dense matrix operations with using sparse matrix operations. In one of the test run with data set in Figure 6, the program is executed with generic matrix notations and is with default selections of dense operations. In the second case, after initial iterations, the runtime compiler finds that the program is using matrix notation, and the profiling information indicates that some matrix of the major computation are with sparse structures. Therefore, the runtime compiler at this point re-writes the sequential dense program into sequential sparse programs. The performance is shown in Table 1. Without runtime compilation support, it takes 67.2 seconds to finish 10 iterations of the CG solver with default dense selections. With runtime compilations, it takes 4.7 seconds to convert the matrix object from dense notations into sparse objects, and then it only takes 10.9 seconds to finish 10 iterations. The experiment is done on a DEC Alpha/3300 station.

To observe the effect on parallel environments, we then focus on the experiments with parallel environments. Our parallel environment is with DEC and SUN workstation clusters. In the experiment, the program for the conjugate gradient solver is written with generic matrix notations in Java. At runtime, our runtime system first tries to rewrite the matrix notations into parallel dense matrix operations of the matrix object. The generic matrix notation is then re-written into parallel sparse computation according to the sparsity of the programs. The runtime compiler selects CRS compression scheme and (BLOCK,*) distribution scheme for the major array in this case. Note that by (Block, *) we mean distributing the sparse matrix by row. The partition is not guided by the row index range, but by the

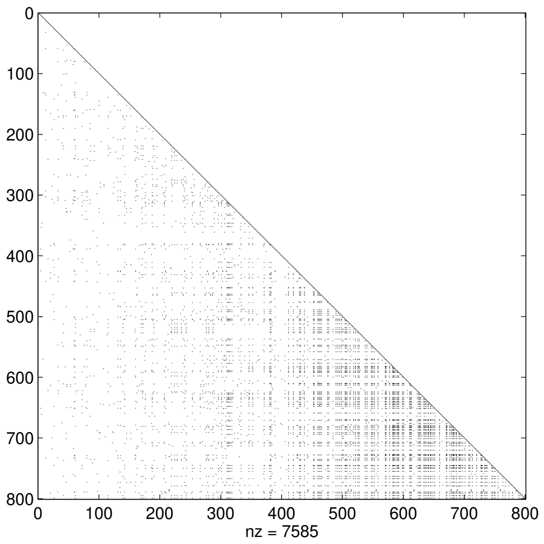

Figure 5: The first data set of our experiment

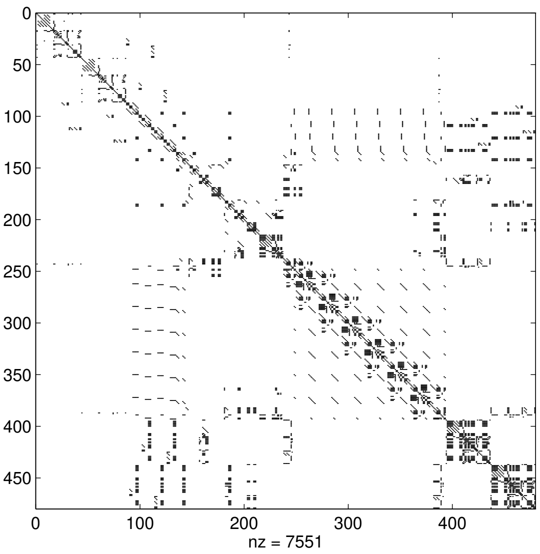

Figure 6: The second data set of our experimental data set

sparsity structure of the given matrix. To evaluate the effectiveness of our cost model and selection scheme, we compare the program with or without our selection model for the distribution scheme. In one of the test run with data set in Figure 6, the CG program is executed with parallel sparse computation on a 8-node DEC Farm. The program with our runtime analysis scheme which distributes elements among processors according to the sparsity of the arrays reports 50% faster than the one simply divides elements among processors according to the index-range of the arrays. Due to that we are still in the process of fine-tuning the performance in our message-passing libraries for parallel sparse computation, the performance data of many other aspects of our environment are still far too preliminary to represent the effects of our optimization. We will try to report many more experimental results for our runtime optimizations at the time of the workshop.

6 Related Work and Conclusion

Representations of sparse matrices play crucial roles in the efficient implementations of sparse applications. The selection of representations can be performed automatically by conventional program and data analyses [2], or by programmers using portable sparse libraries [11] [17]. Aapproaches based on high–level language primitives and structural analyses have been successfully used to parallelize dense matrix computations [5] [14].

We have presented an approach based on high–level matrix classes in Java for the parallelization of sparse matrix computation. The program transformation and parallelization is carried out a continuous compilation process that performs profiling and analyzing tasks at program execution time. Our approach uses cost models that are specific to the execution environment to help select profitable compression and distribution schemes. The way we analyze a program by carrying out its execution not on its usual domains but on other domains of interests (*i.e.*, non–zero ratios, compression/distribution schemes, cost functions, *etc.*), is known as abstract interpretation [1].

Several recent works have used statistical information about the applications or the target machines to help select implementation strategies [3] [6] [7] [8] [15]. These methods seems to concentrate on specific applications, while ours centers around high–level matrix classes for and dynamic selections of specialized classes for faster execution. We are currently applying our method to larger applications to show that the method is effective in practice.

Runtime compilation techniques and their applications to sparse computation can be found at [16] [20]. For the emerging roles of Java and Web technology in parallel computing, see [4] [10] [12] [13] [18].

References

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Computers and Their Applications. Ellis Horwood, 1987.

[2] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, February 1996.

[3] Eric A. Brewer. High–level optimization via automated statistical modeling. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 80–91. Santa Barbara, California, USA, July 1995. ACM Press.

[4] Byran Carpenter, Yuh-Jye Chang, Geoffrey Fox, and Donald Leskiw an Xiaoming Li. Experience with "HPJava". In *Java for Science and Engineering Computation — Simulation and Modelling.* Syracuse University, New York, USA, December 1996. 23 pages. http://www.npac.syr.edu/projects/javaforcse/cpande/suhpjava.ps.

[5] Wai-Mee Ching and Alex Katz. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of Supercomputing '94*, pages 59–68. Washington, D. C., USA, November 1994. IEEE Computer Society Press.

[6] Tyng–Ruey Chuang, Rong–Guey Chang, and Jenq Kuen Lee. Sampling and analytical techniques for data distribution of parallel sparse computation. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing.* Minneapolis, Minnestota, March 1997. 8 pages. SIAM Press.

[7] Tyng–Ruey Chuang and Wen L. Hwang. A probabilistic approach to the problem of automatic selection of data representations. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 190–200. Philadelphia, Pennsylvania, USA, May 1996. The proceedings also appears as *ACM SIGPLAN Notices*, Volume 31, Number 6, June 1996. ACM Press.

[8] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[9] I. S. Duff, R. G. Grimes, and J. G. Lewis. *Sparse matrix test problems*, ACM, Trans. Math. Software, 15 (1989), pp. 1-14.

[10] Geoffrey Fox and Wojtek Furmanski. Java and web technologies for simulation and modelling in computational science and engineering. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing.* Minneapolis, Minnestota, March 1997. 8 pages. SIAM Press.

[11] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333—356, January 1992.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison–Wesley, August 1996.

[13] Susan Flynn Hummel, Ton Ngo, and Harini Srinivasan. SPMD programming in Java. In *Java for Science and Engineering Computation — Simulation and Modelling.* Syracuse University, New York, USA, December 1996. 14 pages. http://www.npac.syr.edu/projects/javaforcse/cpande/IBMspmdjava_new.ps.

[14] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. An array operation synthesis scheme to optimize Fortran 90 programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 112–122. Santa Barbara, California, USA, July 1995. ACM Press.

[15] Lionel M. Ni, Hong Xu, and Edgar T. Kalns. Issues in scalable library design for massively parallel computers. In *Proceedings of Supercomputing '93*, pages 181–190. Portland, Oregon, USA, November 1993. IEEE Compuer Society Press.

[16] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late then never". In *24th Annual SIGPLAN–SIGACT Symposium on Principle of Programming Languages.* Paris, France, 1997. 12 pages. ACM Press.

[17] Youcef Saad. SPARSKIT: A basic tool kit for sparse computations, VERSION 2. Technical report, Compuer Science Department, University of Minnesota, June 1994.

[18] David A. Turmann. JavaPVM. http://www.isye.gatech.edu/chmsr/JavaPVM, December 1996. Center for Human–Machine Systems Research, School of Industrial and Systems Engineering, Georgia Institute of Technology, USA.

[19] Manuel Ujaldon, Emilio Zapata, Barbara M. Chapman, and Hans P. Zima. New data–parallel language features for sparse matrix computations. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 742–749. Santa Barbara, California, USA, April 1995. IEEE Computer Society Press.

[20] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transaction on Computers*, 44(6):737–753, June 1995.