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

We present a generic matrix class facility in Java and an on-going project for a runtime environment with continuous compilation aiming to support automatic parallelization of sparse computation on distributed environments. Our package comes with a collection of matrix classes with a uniform interface for operations on dense and sparse matrices. These matrix operations are implemented both for sequential and parallel executions on distributed memory environments. In our environment, a program such as conjugate gradient solver is written by users using high-level generic matrix notations in Java. At runtime the generic notations are mapped to specific implementations. Our approach is particularly useful at optimizing sparse computation for distributed environments because, with the help of profiling information and a cost model, it can automatically selects suitable compression and distribution schemes according to access patterns of the programs and non-zero structures of the matrices. Our testbed is currently based on Java and PVM on an IBM SP2 workstation cluster. Preliminary experimental results show that our approach is promising in speeding up sparse matrix computations on distributed memory environments.

1 Introduction

With the advent of Internet and World Wide Web, the Web has become an important infrastructure for distributed computing, as it connects rich computational resources of unprecedented scale for easy access. At the same time, with growing popularity, Java has gained an important role for the development of Web and Internet software, as well as for the support of parallel and distributed computing. With the emerging of many new technologies and the rapid shift in compilation paradigms for Java and its environments, one key question we ask ourselves is as follows. Are the new technologies and paradigms providing us with new opportunities to solve some of the hazard problems we faced in the past with parallel computing?

In this paper, we use Java on a distributed memory environment and employ a emerging compilation paradigm, called continuous compilation [12], to address a long challenging research topic in parallel computing: the support of parallel executions of sparse matrix computations on distributed memory 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 sparsity structures (*i.e.*, the distributions of non-zero elements) of its input matrices [1]. Such information, however, may not be available at compile-time for optimization purposes. If it is indeed available at compile-time, it may change over time during program execution, which makes difficult the task of static selections of profitable compression and distribution schemes for the sparse matrices. Second, representations of sparse matrices, such as their compression and distribution schemes, are usually over-specified in the application programs, if conventional languages are used. One must undo the over-specification in the program just to get a clear view of the computation

*This work has been supported in part by National Science Council of Taiwan under grant no. NSC 86-2213-E-001-008 and NSC 86-2213-E-007-043. 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, and phone number is 886-3-5715131 EXT. 3519.

structure. After 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 90/95, and MATLAB, as well as portable sparse libraries [8, 13]. Still, programmers face the problem of proper annotations for sparse operations [15]. For a programmer, it may not be clear how many of these annotations are needed, where and when to put them, in order for the compiler to fully utilize them to produce efficient code. Third, even with complete information about its sparsity and computation structure, 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 [5, 11].

With the paradigm shifting from static compilation into runtime compilation with Java and Web environments, we see opportunities to explore this new paradigm to help address the problem of parallel sparse computation. In this work, we report a generic matrix class facility in Java and an on-going project for a runtime environment in Java with continuous compilation to support parallel sparse computation. Our package comes with a collection of matrix classes with a uniform interface for operations on dense and sparse matrices. These matrix operations are implemented for sequential and parallel execution on distributed memory environments. In our environment, a program such as conjugate gradient solver is written by users using high-level generic matrix notations in Java. At runtime, with the help of profiling information and a cost model, our runtime system employs continuous compilation schemes to map the generic notations into specific implementations. In the ideal continuous compilation model, one execution thread is responsible for executing the program while the compilation thread is responsible for optimizing the current code into an improved version for later execution. Previously, in the parallel compiler community, the inspector and executor model [16] can be considered as one of the early forms of the runtime compilation. Our approach, partly derived from the continuous compilation model [12], have quite a few differences when compared to the inspector and executor model. First, the inspector and executor model puts the execution code and the runtime compilation code into one program. Separation of the two into two program modules can lead to overlapped execution time if separate executing threads are used. A clear interface between the two modules also help rapid progress in the design and development of the two modules. Second, the inspector and executor model is not done in the context of object-oriented frameworks. As we will show later in this paper, runtime binding of generic matrix objects to optimized implementations of their derived matrix objects can significantly improve performance, while minimizing the required revisions in the source code or target code. Finally, the continuous compilation model emphasizes the continuous aspect of the compilation process at runtime to produce successive and improved execution code. Our testbed is currently based on Java and JavaPVM [14] on an IBM SP2 cluster. Preliminary experimental result shows that our approach, continuous compilation using Java, is promising in speeding up sparse matrix computation on distributed memory environments.

The remaining part of this paper is organized as the following. Section 2 presents our matrix class facility in Java. We then outline 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. Section 4 describes our runtime and continuous compilation environments. Preliminary experimental results are given in Section 5. Related work is discussed, and a short conclusion is made, in section 6.

2 A Generic Matrix Package

In this section, we present the class hierarchy of our matrix classes in Java. The package comes with a collection of matrix classes including classes for both dense and sparse matrices, and their implementations on distributed memory environments. All the matrix classes are extended from a generic matrix class to form a two-level hierarchy as illustrated in Figure 1. Programmers use only the generic matrix notations of the top level in their programs. The operations supported by the generic matrix class include data parallel programming methods for element-wise matrix addition, subtraction, multiplication, *etc.* as well as matrix-vector and matrix-matrix multiplication. In the second level of the hierarchy, specialized implementations are built, but with identical interfaces of the generic matrix in the top level. Furthermore, for distributed implementations, the classes will include distribution schemes of the matrix, and for distributed sparse implementations, the classes in addition will include compression schemes. Currently, the distribution schemes being considered are general block partitions based on number of non-zeros. The compression schemes being considered are CRS, CCS, and dense representations. Table 1 lists the compression and distribution schemes for distributed sparse matrices. Note that by (Block, *) scheme, we mean distributing the sparse matrix by row. The distribution is not guided by an even partition of the row index range, but rather by a partition based on non-zero structure in the matrix. The partition may even produce uneven distribution of

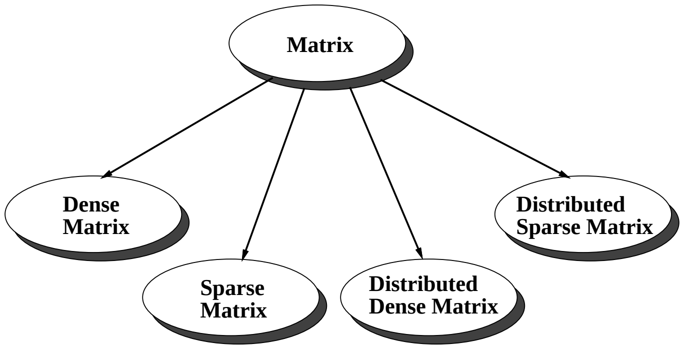


Figure 1: Matrix class hierarchies in Java.

Compression Scheme
Compressed Row Storage
Compressed Column Storage
Dense Representation

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

Table 1: The Domains for Distribution/Compression Scheme Selections

non-zeros among processors, as long as it achieves over-all balanced computation and communication. Likewise, (*, Block) partitions the matrix by column, while (Block, Block) partitions the matrix both by row and column.

To give an idea of how the generic matrix class looks like, we show in the following part of its code fragments.

```
public class Matrix {

int rows;
int cols;
int countOfNonZero;
...
public Matrix(int r, int c);                // initialization, r rows c columns
public Matrix(int r, int c, double initVal); // initialized with value initValue
public Matrix(int r, int c, String fname);   // initialized from file fname
...
public double elementAt(int r, int c);       // m[r][c], m the current matrix
public Matrix addAssign(Matrix m1);          // m  = m + m1, elem.-wise add.
public Matrix add(Matrix m1);                // m2 = m + m1, elem.-wise add.
public Matrix mul(Matrix m1);                // m2 = m x m1, mat.-mat. mult.
public Vector mul(Vector v1);                // v2 = m x v1, mat.-vec. mult.
...
public double getSparsity();                 // non-zero ratio
}
```

For a matrix-vector multiplication like $u = A \times v$, where A a matrix, u and v vectors, one simply writes `u.set(A.mul(v))` in the program. Figure 2 shows a conjugate gradient solver written entirely using the generic matrix notations. Specific implementations from the second level matrix classes will be chosen at runtime to improve performance. The runtime system will use profiling information which includes access patterns of the program and non-zero structures of the matrices. The cost model will be explained in Section 3, and the techniques for dynamic binding of specific implementations to generic matrix objects will be discussed in Section 4.

```
public class cgsolve {
  int iteration;                        // benchmark purpose only, see Sec. 5

  public static Vector cgsolve(Matrix A, Vector b, double tol) {
                                        //   x = cgsolve(A, b, tol)
  Vector x  = new Vector(b.getSize()); //   x = zeros(size(b));
  Vector p  = new Vector(b.getSize()); //   p = zeros(size(b));
  Vector Ap = new Vector(b.getSize()); //  Ap = zeros(size(b));
  Vector r  = new Vector(b);           //   r = b;
  double rtr = r.innerProduct(r);      // rtr = r' * r;
  double rtrold;
  double beta = 0.0;
  double alpha;

  iteration = 0;                        // benchmark purpose only, see Sec. 5
  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;
    // A = monitor.select(A);          // representation selection, see Sec. 4
    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;
    // iteration++;                    // benchmark purpose only, see Sec. 5
    // if(iteration >= 10) break;      // benchmark purpose only, see Sec. 5
  }
  return x;
  }                                     // end of method cgsolve

}                                       // end of class  cgsolve
```

Figure 2: Conjugate gradient in Java, with generic matrix notations.

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 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 minimal requirements of going into the implementation level (by modifying the compiler or runtime system).

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 the source code or the 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 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. Similar analysis has been carried out in the context of Fortran 90 array intrinsic operators [4]. However, we emphasize that Java and its environment allow us to perform such analysis continuously at runtime. This is contrary to our previous work on analyzing Fortran 90 array intrinsics for sparse computation, which is a compile-time effort.

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 generated incrementally, the distributions of non-zeros are sampled on-line when the matrix is being built. Coefficients in cost formulae are determined in a preprocessing stage by performance measurements on the underlining execution environment. In addition, we formulate the cost functions of sending/receiving a block of sparse matrix using PVM routines. 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 schemes. The sampling and analysis process 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.We omit the matrix sizes if they are clear from the context. Similarly, we use $s[\alpha]$ to denote a scalar of which non-zero occurs with probability (at most) α.

Take operations `add`, `innerProduct`, and `mul` as examples. The non-zero ratios of their resultant matrices can be estimated to be the following:

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

We assume that zeros resulting from cancelation of additions are unlikely. (Hence α is an over-estimation of the actual non-zero ratio).

For an element-wise matrix operation like `add`, its computation cost is proportional to the total number of non-zeros, and proportional to the number of rows if it is in CRS. The cost formula is the following.

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

For `mul`, we assume the two matrices are in CRS and CCS respectively. The resultant matrix will be in CRS. The cost formulae for `innerProduct` and `mul` are the following.

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

Operation `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 `innerProduct` 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 positive constants $(c_0, c_1, c_2, \ldots)$ in each of the formulae will be calibrated from performance measurements on the underling execution environment using the corresponding sparse routine with matrices of various sizes and non-zero ratios.

The cost formulae 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.

4 Runtime Compilations

4.1 Proposed Framework for Runtime Compilations

Figure 3 illustrates the runtime compilation process of our proposed 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

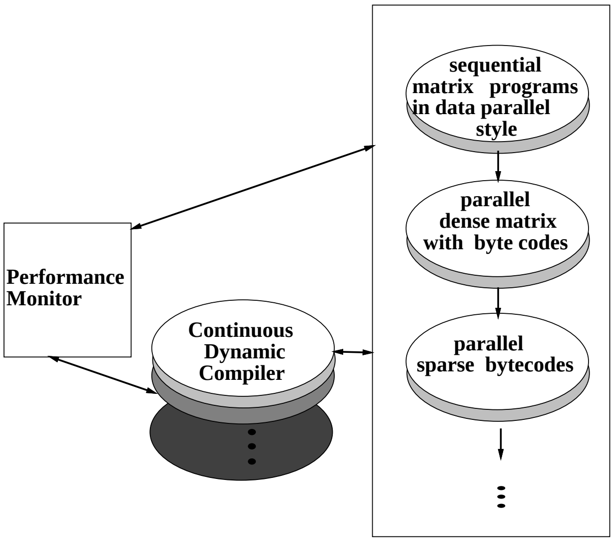


Figure 3: Continuous Compilation Support for Parallel Java Programs

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. 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 matrices of the major computation are sparse. At this point, the runtime compiler re-writes the parallel dense program into parallel sparse programs on distributed environments. The issues involved at this point include selections of compression schemes and distribution schemes. With the compilation process continues to evolve, the byte code program is eventually re-written into a parallel sparse program in native code.

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 one needs to consider for the runtime compilations include the following.

- the loading status and availability of the computing resources in the network,
- the profiling and the timing information on major computing blocks of the program,
- the major loops in the program,
- the usage of operations in our matrix package,
- the sparsity of major arrays,
- the access patterns to matrix classes,
- the compression schemes which should be used for sparse or parallel sparse computations, and
- the distribution schemes which should be used for matrices or sparse matrices on distributed environments.

In the example of the conjugate gradient solver listed in Section 2, with all the necessary information available, the runtime compiler will select CRS compression scheme and (Block, *) distribution scheme for the major matrix `A` in this case, if `A` turns out to be sparse, and parallel environments are available. The selection is based on our model in Section 3.

4.2 Implementation Status and Current Emulations

Now we discuss the ways target code are produced by a rewriting system. 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. The second way for dynamic code substitution is to utilize a set of properly layered interfaces to the matrix classes. We use a technique based on dynamic binding of runtime objects. At runtime, matrix objects may come from different layers and support methods of specific execution requirement (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 to change matrix computation from one form into another at any program point. First, the runtime compiler needs to create a new matrix object with, say, the selected compression and distribution schemes. Second, the system needs to convert the original matrix object into a new matrix object of the specialized class. Finally, we cast the specialized matrix object into the generic matrix object.

Currently, we adopt the second method described above. In addition, we have an emulating environment to emulate our proposed runtime environment. In our emulating experiment, we have a performance monitor class to perform the analysis and object transformation described above. Programmers will have to insert a line as follows in the loop of the conjugate gradient solver as described in Section 2.

```
A = monitor.select(A);
```

We will report experimental results in the next section based on our matrix class library and emulation environment.

5 Preliminary Results

In this section, we report preliminary performance results from our experiments with generic matrix class facility in Java to support sparse computation. The high-level generic matrix notations described in Section 2 are used to write programs, but difference implementations (dense *vs.* sparse, uniprocessor *vs.* distributed environment) are used for actual executions.

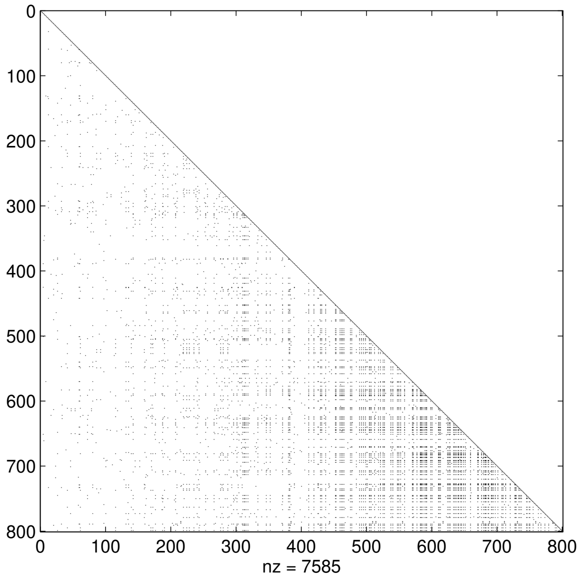
We use the two sparse data sets in Figure 4 for our experiments. The first is an artificial 800 by 800 positive definite matrix generated by us. The second is a 479 by 479 matrix from the toolbox in MATLAB and is a well-known test case [6]. We use the matrix-vector multiplication and the conjugate gradient solver in Figure 2 as our testing programs. The test programs use only generic matrix notations. Our experimental platform is an IBM SP2 workstation cluster connected by FDDI network. Each node runs a copy of the Java virtual machine with the JavaPVM interface to the underlining PVM library. Table 2 shows the performance results.

Note that Table 2 shows only the results of fixed selections of class implementations. At the beginning of an execution, we may not know that the matrix is sparse or dense, and an automatic selection of specific class implementations is needed at run-time. Currently, automatic selections of class implementations are invoked by hints from the programmers (*i.e.*, `monitor.select()` method in Figure 2), but there is no reason why this cannot be done automatically by matrices themselves once they are made into reflective objects. The method `monitor.select()` will change the current implementation of the matrix to a more suitable implementation (or make no change at all) based on the sparsity of the matrix, and the current execution environment. From our measurements, the dense-to-sparse matrix conversion cost can be amortized over a long execution. Also, in Table 2, the sparse matrix class being selected uses the (Block, *) scheme for distribution and the CRS scheme for compression. These two schemes are most suitable for the two test programs. For other test programs, certainly different schemes can be selected. Part of the code from the class hierarchy can be found in `http://puma.cs.nthu.edu.tw/~project/ParallelJavaTool`.

6 Related Work and Conclusion

Representations of sparse matrices play crucial roles in the efficient execution of sparse applications. The selection of representations can be performed automatically by conventional program and data analyses [1, 10], or by programmers using portable sparse libraries [8, 13]. Approaches based on high-level language primitives and structural analyses have been successfully used to parallelize dense matrix computations [3].

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 by a continuous compilation process at program execution time. The way that we analyze a program by carrying out its execution not on its usual domains



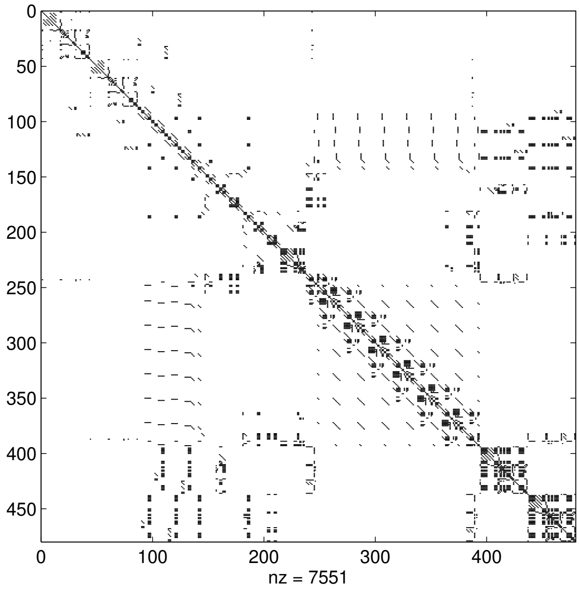


Figure 4: The data sets used in the experiment. The first is a 800×800 matrix with non-zero ratio 1.19%, and the second is a 479×479 matrix with no-zero ratio 3.29%.

No. of Proc.	Dense Matrix Class		Sparse Matrix Class	
	$u = A \times v$	cgsolve	$u = A \times v$	cgsolve
1	6,162	6,357	147	335
2	3,104	3,393	91	266
4	1,584	1,754	71	228
8	1,226	1,449	56	225

No. of Proc.	Dense Matrix Class		Sparse Matrix Class	
	$u = A \times v$	cgsolve	$u = A \times v$	cgsolve
1	2,212	22,590	169	2,664
2	1,114	11,879	117	2,114
4	590	6,551	90	1,750
8	321	3,992	75	1,663

Table 2: Execution time (in ms) on IBM SP2 for the two datasets. $u = A \times v$ is for a matrix-vector multiplication, and cgsolve is for running the conjugate gradient solver (shown in Figure 2) for 10 iterations. The table on the left is for the 800×800 dataset, and the table on the right is for the 479×479 dataset.

but on other domains of interests (*i.e.*, non-zero ratios, compression/distribution schemes, cost functions, *etc.*), is known as abstract interpretation in the functional programming literature.

Several recent works have used statistical information about the applications or the target machines to help select implementation strategies [4, 5, 11]. These methods seem to concentrate on specific applications, while ours center around high-level matrix classes and dynamic selections of specialized classes for faster execution. Runtime compilation techniques and their applications to sparse computation can be found at [12, 16]. For the emerging roles of Java and Web technology in parallel computing, see [2, 7, 9, 14].

References

[1] 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.

[2] Marina Chen and James Cowie. Java's role in distributed collaboration. *Concurrency: Practice and Experience*, 9(6):509–519, June 1997.

[3] 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.

[4] 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, Minnesota, USA, March 1997. 8 pages. SIAM Press.

[5] 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.

[6] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

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

[8] 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.

[9] Susan Flynn Hummel, Ton Ngo, and Harini Srinivasan. SPMD programming in Java. *Concurrency: Practice and Experience*, 9(6):621–631, June 1997.

[10] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel sparse code for user-defined data structures. In *Proceedings of Eights SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, USA, March 1997. SIAM Press.

[11] 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 Computer Society Press.

[12] Michael P. Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late than Never". In *24th Annual SIGPLAN–SIGACT Symposium on Principle of Programming Languages*. Paris, France, 1997. 12 pages. ACM Press.

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

[14] 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.

[15] 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.

[16] 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.