Sampling and Analytical Techniques for Data Distribution of Parallel Sparse Computation*

Tyng–Ruey Chuang† Rong–Guey Chang‡ Jenq Kuen Lee‡

Abstract

We present a compile–time method to select compression and distribution schemes for sparse matrices which are computed using Fortran 90 array intrinsic operations. The selection process samples input sparse matrices to determine their sparsity structures. It is also guided by cost functions of various sparse routines as measured from the target machine. The Fortran 90 array expression is then transformed into a sparse array expression that calls the selected compression and distribution routines.

1 Introduction

It has long been a challenging research topic to devise general guidelines for selecting efficient compression and distribution schemes for parallel executions of sparse matrix computations. We feel that this problem is 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 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 library [8] [11] for large scale computations. Still, programmers face the problem of proper annotations for sparse operations [12]. 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] [6] [10].

We show in this paper a general approach to selecting profitable compression and distribution schemes for sparse matrix computations. This approach is based on

*This work is supported, in part, by National Science Council, Taiwan, under contracts NSC 85-2213-E-007-050, NSC 85-2221-E-007-031, and NSC 86-2213-E-001-008.

†Institute of Information Science, Academia Sinica, Taipei 115, Taiwan. E–mail: trc@iis.sinica.edu.tw .

‡Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan. E–mail: rgchang@eagle.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw .

probabilistic analyses of non–zero structures of the input matrices, and and on cost models of parallel sparse matrix operations. The structural analyses are performed at compile–time, assuming the availability of input matrices. The cost models are constructed in a pre-processing stage from performance measurements of sparse matrix routines on the specific machine on which sparse applications will be executed. It is important to note that we measure only intrinsic matrix operations, not the sparse application itself. Therefore, once built, the cost models can be used to optimize all sparse programs.

We use the high–level array language Fortran 90 as our source language. We assume sparse matrix computations are formulated using only Fortran 90 intrinsic array operators and functions. The source program of the sparse computation is the same as of dense matrix computation. The only difference is that a programmer annotates array declarations with a `sparse` attribute if the arrays are sparse. We do not require programmers to specify compression and distribution schemes for sparse arrays. The selections of such schemes are left to compile–time analyses. The source is compiled to Fortran 90 code with calls to Message Passing Interface (MPI) library and to sparse routines for Fortran 90 array intrinsic operations.

This approach, that one uses high–level language primitives to express computations but leaves efficient low–level implementations to the compilers, is quite natural and has been used to compile high–level programs written in APL [4], Fortran 90 [9], and MATLAB [7]. Our approach also takes into considerations the performance characteristics of the target machines. See figure 1 for an illustration of our framework.

The remaining part of this paper is organized as the following. We describe in section 2 the sampling and analytical method for extracting non–zero structures from sparse matrices, and for formulating the cost functions of sparse Fortran 90 array intrinsic operations. Section 3 outlines the cost models. Examples are given in section 4. Related work is discussed, and a short conclusion is made, in section 5.

2 Sparsity Analyses and Cost Estimations

We first assume the probability that a matrix element is zero is the same for all elements in the matrix. That is, the distribution of non–zeros is uniform matrix–wise. This restriction is not so severe since we can partition a sparse matrix into smaller blocks whose non–zeros are more uniformly distributed, see section 2.3. Then, for each Fortran 90 intrinsic array operation, given non–zero ratios of the input arrays, we derive the formulae for computing the non–zero ratio of the resulting array. These formulae are used to infer non–zero ratios of sparse matrices that are generated during the execution.

Similarly, given non–zero ratios of the input arrays, we can formulate the computation cost for each Fortran 90 intrinsic array operation, as if they are sparse. Coefficients in cost functions will be determined by performance measurements of the corresponding sparse routines on the target machine. In addition, we also measure the cost functions for sending/receiving a block of data using MPI for various Fortran 90 intrinsic data types. Now, given a Fortran 90 array intrinsic operation, say transposing an integer matrix, the non–zero ratio of the input array, and the dimension of the array, we can easily estimate the total cost (including computation and distribution) of the corresponding sparse routine on the target machine.

At compile–time, we calculate the non–zero structures of input matrices, as well as infer those of the matrices that are generated by the program. We then select among feasible compression and distribution schemes those that will have minimal execution time, according to the cost functions we have gathered.

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.) The distribution of non–zero is assumed uniform matrix–wise. We often omit

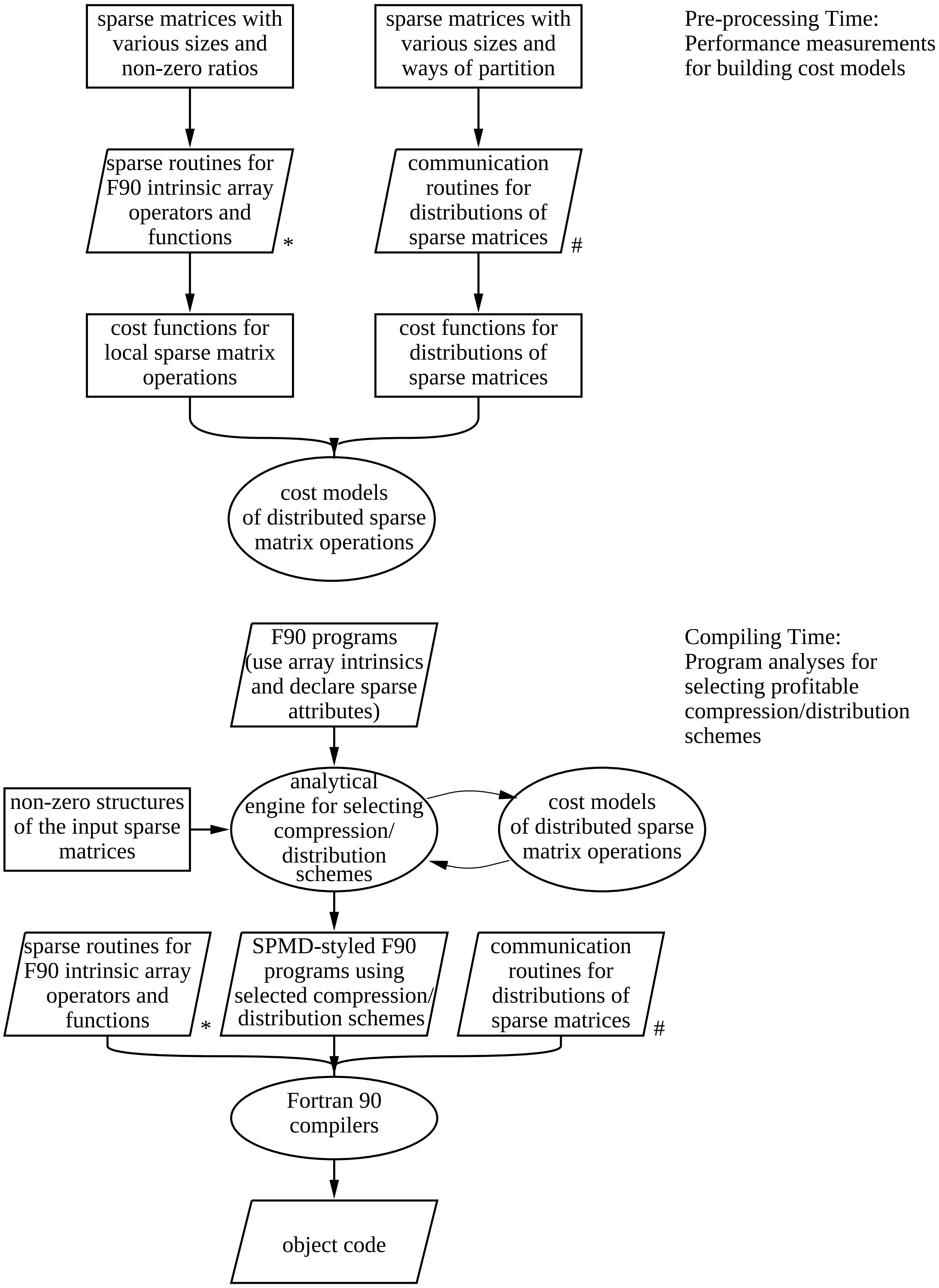

FIG. 1. *Process diagrams for building cost models and for analyzing Fortran 90 programs.*

NOTE: There are separate sets of sparse routines (marked by *) for Fortran 90 array intrinsic operations, one for each compression scheme (CRS, CCS, *etc*). Similarly, there are separate sets of communication routines (marked by #) for distributions of sparse matrices, one for each distribution scheme ((Block, *), (*, Block), *etc*). The cost functions are formulated for each set of routines as well.

the matrix sizes if they are not significant in the discussion. $S[0]$ is an all–zero matrix, while $S[1]$ is a matrix with almost no zero. In the same spirit, we use $s[\alpha]$ to denote a scalar of which non–zero occurs with probability (at most) α.

2.1 Non–Zero Ratios

We list in this section the formulae for estimating non–zero ratios of the result array for several Fortran 90 intrinsic array operations. They are: $+$, $*$, $**$, `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`, `PRODUCT`, `SUM`, `DOT_PRODUCT`, `MATMUL`, and `TRANSPOSE`. Formulae for other array operations (such as `PACK` and `UNPACK`) are more complicated and omitted here. We make three assumptions in the formulations: that zeros resulting from cancellation of additions is unlikely, that two non–zeros are unlikely to be equal, and that one non–zero is equally likely to be greater or smaller than the other. We view the logical value `.FALSE.` as zero.

$$
\begin{aligned}
S[\alpha] + S[\beta] &= S[1-(1-\alpha)(1-\beta)] \\
S[\alpha] * S[\beta] &= S[\alpha\beta] \\
S[\alpha] ** S[\beta] &= S[1-\beta+\alpha\beta] \\
S[\alpha]\ \texttt{.EQ.}\ S[\beta] &= S[(1-\alpha)(1-\beta)] \\
S[\alpha]\ \texttt{.NE.}\ S[\beta] &= S[1-(1-\alpha)(1-\beta)] \\
S[\alpha]\ \texttt{.GT.}\ S[\beta] &= S[(\alpha+\beta-\alpha\beta)/2] \\
S[\alpha]\ \texttt{.GE.}\ S[\beta] &= S[1-(\alpha+\beta-\alpha\beta)/2] \\
S[\alpha]\ \texttt{.LT.}\ S[\beta] &= S[(\alpha+\beta-\alpha\beta)/2] \\
S[\alpha]\ \texttt{.LE.}\ S[\beta] &= S[1-(\alpha+\beta-\alpha\beta)/2] \\
\texttt{SUM}(S[\alpha,n]) &= s[1-(1-\alpha)^n)] \\
\texttt{PRODUCT}(S[\alpha,n]) &= s[\alpha^n] \\
\texttt{DOT\_PRODUCT}(S[\alpha,n], S[\beta,n]) &= s[1-(1-\alpha\beta)^n] \\
\texttt{MATMUL}(S[\alpha, m\times n], S[\beta, n\times l]) &= S[1-(1-\alpha\beta)^n, m\times l] \\
\texttt{TRANSPOSE}(S[\alpha]) &= S[\alpha]
\end{aligned}
$$

2.2 Computation and Communication Costs

We assume that 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 the formulae will be determined from performance measurements of the corresponding sparse routine using matrices of various sizes and non–zero ratios.

For array operations $+$, $*$, $**$, `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, and `.LE.`, each has its computation cost proportional to the total number of non–zeros, and proportional to the number of rows (since they are in CRS), in the matrices. The formula is:

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

where constants c_0, c_1, c_2 depend on `op` and the target machine.

For unary array operations `SUM`, `PRODUCT`, and `TRANSPOSE`, the formulae are:

$$
\begin{aligned}
Cost(\texttt{SUM}(S[\alpha,n])) &= c_1\alpha n + c_0 \\
Cost(\texttt{PRODUCT}(S[\alpha,n])) &= c_1\frac{1-\alpha^{(n+1)}}{1-\alpha} + c_0
\end{aligned}
$$

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

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 `MATMUL`, the two matrices are in CRS and CCS respectively. If not, a simple `TRANSPOSE` will transform each to the required compression scheme. The result of the multiplication is in CRS.

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

Note that `DOT_PRODUCT` 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 `MATMUL`, 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 target machine. Note that currently we have omitted the costs of generating the necessary communication sets for `Send` and `Receive` operations. These costs will be incorporated later on into our implementation.

2.3 Global Structures

A sparse matrix may be partitioned into disjoint sub–matrices for distributed 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 resulting 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 element–wise array operations, such as $+$, $*$, $**$, `.EQ.`, `.NE.`, `.GT.`, `GE.`, `.LT.`, `.LE.`, non–zero ratio of an output matrix partition can be computed from the respective input matrix partitions. Let A_i be a partition of A. Then,

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

where `op` is an element–wise array operation. 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 2.1) at the right hand side.

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

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

Finally, for `DOT_PRODUCT` and `MATMUL`:

$$Ratio(\texttt{DOT\_PRODUCT}(A, B)) = s[1 - \prod_i (1-\alpha_i)],$$

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)

$$\text{where } \mathtt{DOT\_PRODUCT}(Ratio(A_i), Ratio(B_i)) = s[\alpha_i]$$

$$Ratio(\mathtt{MATMUL}(A,B)_{i,k}) = S[1 - \prod_j (1-\alpha_j)],$$

$$\text{where } \mathtt{MATMUL}(Ratio(A_{i,j}), Ratio(B_{j,k})) = S[\alpha_j]$$

3 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 2, 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 guided by 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 4 for an example. Likewise, (*, Block) partitions the matrix by column, while (Block, Block) partitions the matrix both by row and column.

We are investigating more complex compression/distribution schemes for inclusion in domain D. Function *Total_Cost* calculates the total cost of computation and communication for an array operation under a particular set of compression/distribution schemes.

4 Preliminary Results

We have implemented on IBM SP2 a set of sparse routines for basic Fortran 90 intrinsic array operations, with one set of routines for each compression scheme. Communication routines for sending/receiving sparse matrices using various distribution methods have been implemented as well. The following are the cost functions of several routines on IBM SP2 (time in microsecond, and compressed by CRS):

$$Cost(S[\alpha, m\times n] + S[\beta, m\times n]) = \begin{cases} 0.703(\alpha+\beta)mn + 0.611m + 1487.18 & \text{integer} \\ 2.05(\alpha+\beta)mn + 0.854m + 3538.485 & \text{real} \\ 2.05(\alpha+\beta)mn + 0.854m + 3538.485 & \text{double} \end{cases}$$

$$Cost(\mathtt{TRANSPOSE}(S[\alpha, m\times n])) = \begin{cases} 0.904\alpha mn + 0.405(m+n) + 1857 & \text{integer} \\ 1.771\alpha mn + 0.634(m+n) + 2825 & \text{real} \\ 1.765\alpha mn + 0.602(m+n) + 3740 & \text{double} \end{cases}$$

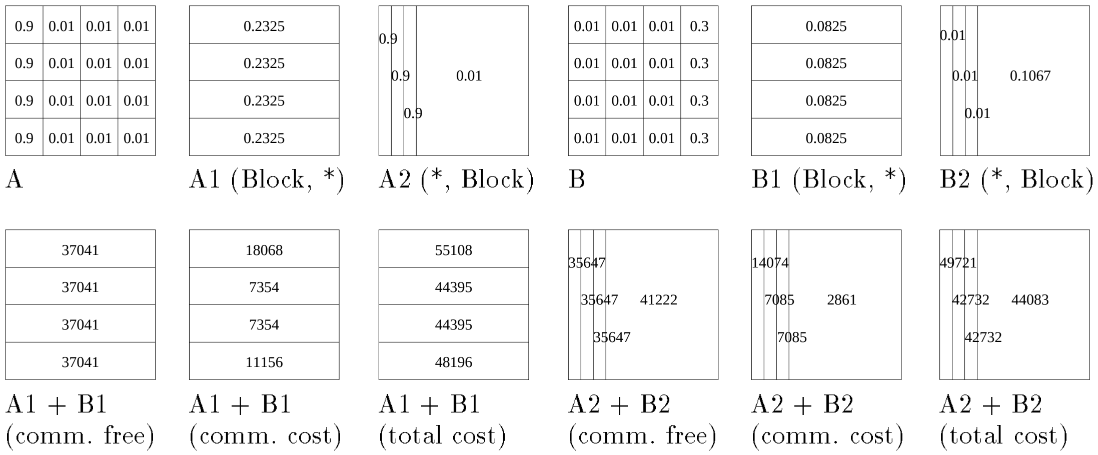

FIG. 2. *Compression/Distribution scheme selections: an example.*
NOTE: Both A and B are 800×800 sparse integer matrices. The first row of the figure shows the sparsity structures of A and B, as well as when they are partitioned using (Block, *) and (*, Block) distribution schemes. The number in each block is its non–zero ratio. The second row shows costs of adding the two matrices under the two distribution schemes. The number in each block is the cost in microseconds for processing that block. Note that the sizes of the blocks are proportional to their index ranges. For example, each block in A is a 200×200 matrix. Likewise, each of the first three blocks in $A2$ is $800 \times 200/3$, and the last block is 800×600. First it is assumed that both A and B is pre–distributed and computation at each PE is communication free. Then it is assumed that A and B completely reside in PE 1 and PE4, respectively. We then calculate the distribution cost and total cost. The numbers are derived using cost functions described in this section. For example, under (Block, *) distribution, PE 1 (for the first 200 rows) will need $3 \times Cost(\texttt{Send}(S[0.2325, 200 \times 800])) + Cost(\texttt{Receive}(S[0.08225, 200 \times 800])) = 18068$ microseconds to send/receive all the necessary matrix elements before the computation can start. Furthermore, addition at that PE costs $Cost(S[0.2325, 200 \times 800] + S[0.0825, 200 \times 800]) = 37041$.

$$Cost(\texttt{Send/Receive}(S[\alpha, n])) = \begin{cases} 0.144\alpha n + 96.579 & \text{integer} \\ 0.144\alpha n + 81.579 & \text{real} \\ 0.261\alpha n + 96.579 & \text{double} \end{cases}$$

The cost models, however, are not yet completely developed. As a consequence, the analyses performed at compile–time for selecting profitable compression/distribution schemes are currently carried out manually.

Still, we are able to carry out some experiments. Figure 2 shows analyses on two sparse 800×800 matrices A and B. Assume four processors (PE's) are available. To calculate $C = A + B$, we show how to distribute the two matrices either by (Block, *) or (*, Block) to balance the computation. Note that both selections are guided by the sparsity structures of the two matrices. We do an analysis to see which scheme is better. We consider two cases. In case 1, *both A and B can be pre–distributed for free*, but we have to choose a scheme for the distribution. It is clear that a (Block, *) distribution results in better performance than (*, Block) because it is more balanced. (The slowest of the four blocks in "A1 + B1 (comm. free)" needs 37041, which is less than the slowest of the four in "A2 + B2 (comm. free)", 41222.) Whether the matrices are represented by CRS or CCS is not significant. In case 2, *A completely resides in PE 1 while B is completely in PE 4*. It turns out that distributing A and B under the (Block, *) takes more time than the (*, Block) scheme, and as a consequence, (*, Block) scheme is the better over–all scheme.

Note that it is nature to use CRS compression scheme when the matrices are distributed by (Block, *) scheme, and use CCS when they are distributed by (*, Block).

Performance measurements from actual experimentations on SP2 also agree with what the cost models have predicted for the above two cases.

5 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 [8] [11]. Another approaches, based on high–level language primitives and structural analyses, however, have been successfully used to parallelize dense matrix computations [4] [9].

We have presented an approach based on high–level primitives for the parallelization of sparse matrix computation. Our approach uses cost models that are specific to target machines 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, *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] [5] [6] [10]. These methods seems to concentrate on specific applications, while ours centers around high–level language primitives, hence, is more general. We are currently applying our method to larger applications to show that the method is effective in practice.

References

[1] S. Abramsky and C. Hankin, eds., *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

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

[3] E. A. Brewer, *High–level optimization via automated statistical modeling*, in Proc. 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming, July 1995, pp. 80–91.

[4] W.-M. Ching and A. Katz, *An experimental APL compiler for a distributed memory parallel machine*, in Proc. Supercomputing '94, Nov. 1994, pp. 59–68.

[5] T.-R. Chuang and W. L. Hwang, *A probabilistic approach to the problem of automatic selection of data representations*, in Proc. 1996 ACM SIGPLAN Int'l Conf. on Functional Programming, May 1996, pp. 190–200.

[6] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, *LogP: A practical model of parallel computation*, Communications of the ACM, 39 (1996), pp. 78–85.

[7] L. De Rose and D. Padua, *A MATLAB to Fortran 90 translator and its effectiveness*, in Proc. 1996 Int'l Conf. on Supercomputing, May 1996, pp. 309–316.

[8] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation*, SIAM J. on Matrix Anal. and Appl., 13 (1992), pp. 333—356.

[9] G.-H. Hwang, J. K. Lee, and D.-C. Ju, *An array operation synthesis scheme to optimize Fortran 90 programs*, in Proc. 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming, July 1995, pp. 112–122.

[10] L. M. Ni, H. Xu, and E. T. Kalns, *Issues in scalable library design for massively parallel computers*, in Proc. Supercomputing '93, Nov. 1993, pp. 181–190.

[11] Y. Saad, *SPARSKIT: A basic tool kit for sparse computations, VERSION 2*, Compuer Science Department, University of Minnesota, June 1994.

[12] M. Ujaldon, E. Zapata, B. M. Chapman, and H. P. Zima, *New data–parallel language features for sparse matrix computations*, in Proc. 9th Int'l Parallel Processing Symp., Apr. 1995, pp. 742–749.