Efficient Support of Parallel Sparse Computation for Array Intrinsic Functions of Fortran 90*

Rong-Guey Chang
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

Fortran 90 provides a rich set of array intrinsic functions. Each of these array intrinsic functions operates on the elements of multi-dimensional array objects concurrently. They provide a rich source of parallelism and play an increasingly important role in automatic support of data parallel programming. However, there is no such support if these intrinsic functions are applied to sparse data sets.

In this paper, we address this open gap by presenting an efficient library for parallel sparse computations with Fortran 90 array intrinsic operations. Our method provides both compression schemes and distribution schemes on distributed memory environments applicable to higher-dimension sparse arrays. This way, programmers need not worry about these low-level details. Sparse programs can be expressed concisely using array expressions, and parallelized with the help of our library. Our current testbed is an IBM SP2 workstation cluster. Preliminary experimental results show that our approach is promising in speeding up sparse matrix computations on both sequential and distributed memory environments if the computations are expressed with Fortran 90 array expressions.

1 Introduction

An increasing number of programming languages, such as Fortran 90 [1], High Performance Fortran (HPF) [14], APL [6], MATLAB [9], and possibly C++ with built-in array classes [3, 16], provide rich sets of intrinsic array functions and array constructs. These intrinsic array functions and array expressions operate on the elements of multi-dimensional array objects concurrently without requiring iterative statements. They provide a rich source of data parallelism. Optimizations on the array functions operated on dense data set has been studied and shown to have important impacts at program performance [12].

In the case that these intrinsic functions are applied to sparse data sets, however, it remains open how these intrinsic functions should be supported on both sequential and parallel systems. We report in this paper experiments and results on providing sparse library support for Fortran 90 array intrinsics, as well as its parallelization. With

*This paper is submitted to *1998 ACM International Conference on Supercomputing*. 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.

our sparse library support, sparse matrix computation can now be expressed in Fortran 90 using high-level array expressions (just like dense matrix computation has always been), without concerning low-level coding of compression and distribution details. Our approach makes easy both the tasks of sparse programming and its parallelization.

It has been an active research topic on language and compiler supports for sparse matrix computation in Fortran. Previous results mainly concerned with automatic mapping of (individual) array indexing function to the corresponding data access in the underlying sparse representation [2, 22, 21, 23]. Ours starts with high-level array intrinsics and their efficient implementations for sparse data sets. Fortran 90 array intrinsics can be applied to higher-dimension arrays, and our sparse library is designed to support them as well. From this perspective, what we provide to Fortran 90 is similar to sparse matrix routines in MATLAB as described in [10]. However, the difference is that, in MATLAB, arrays are at most two-dimensional and their built-in operations are comparably simpler than in Fortran 90. In addition, our library supports parallelization.

Our library uses a two-level design. For each Fortran 90 intrinsic functions, several low-level routines will be provided, with each requiring the input sparse matrix to be in the specific compression/distribution scheme. Low-level communication routines using the Message Passing Interface (MPI) are used to send and receive parts of the sparse matrix among the processors. The library uses the usual Single Program Multiple Data (SPMD) model. However, high-level sparse array functions are built on top of the low-level routines, and are overloaded to the ordinary (dense) Fortran 90 array intrinsic interfaces. This way, programmers can use the familiar array expressions and need not concern about the low-level details. Special routines are provided for setting up the execution environment and for specifying initial compression and distribution schemes. Our testbed is currently on an IBM SP2 workstation cluster. Preliminary experimental results show that our approach is promising in speeding up sparse matrix computations using array intrinsic functions on both sequential and distributed memory environments.

The remainder of the paper is organized as follows. Section 2 describes our proposed support for sparse and parallel array intrinsic functions applicable to higher-dimension arrays. Next, Section 3 presents the key algorithms of two array intrinsic functions with sparse representations to illustrate the idea in the implementations. Next, Section 4 describes the experimental results, and Section 5 discusses performance issues and possible optimizations for our proposed systems. Finally, Section 6 gives related work and concludes this paper.

2 Sparse Support for Fortran 90 Array Intrinsic Functions

We show in this section several motivating examples that are in need of sparse library support for Fortran 90 array intrinsics. We also outline compression and distribution schemes for higher-dimension arrays.

2.1 Motivating Examples

The following Fortran 90 program segment, called banmul, is taken from the book *Numerical Recipes in Fortran 90* [18, page 1019]. It calculates $b = Ax$, where A is a skewed representation of a banded matrix, and x a vector. (Description of skewed representation and other details can be found in [18], and is omitted here.) This code uses Fortran 90 array intrinsics `eoshift`, `spread`, and a reduction function `sum`.

```
integer, parameter :: row = 1000
real, dimension(row,2*row-1) :: A
real, dimension(row) :: x, b
integer, dimension(2*row-1) :: shift
```

```
b = sum(A*eoshift(spread(x,dim=2,ncopies=2*row-1),&
        dim=1,shift=arth(-row+1,1,2*row-1)),dim=2)
```

If both array A and x are dense, then the above program is expressed in a concise and data parallel style. If HPF data distribution directives are given for arrays `A`, `x`, and `b`, then presumably one can expected improved performance of the program when executed on multiprocessor machines. However, in case when arrays `A` and `b` are sparse, then such concise expression is no longer available in current Fortran 90 language. Often this means that now the same algorithm has to be coded in iterative loop over array indices. Worse, representation details about how the sparse arrays should be compressed and distributed also get mixed into the code. The resultant code is less readable and more difficult to parallelized.

The situation changes if sparse versions of the intrinsic functions (`eoshift`, `spread`, and `sum`) are available, and representation decisions of sparse matrices are hidden behind a high-level data type interface. The sparse version, written in Fortran 90 as well, now looks like the following.

```
integer, parameter :: row = 1000
type(sparse2d_real) :: A
type(sparse1d_real) :: x, b
integer, dimension(2*row-1) :: shift

call bound(A, row, 2*row-1)
call bound(x, row)
call bound(b, row)

b = sum(A*eoshift(spread(x,dim=2,ncopies=2*row-1),&
        dim=1,shift=arth(-row+1,1,2*row-1)),dim=2)
```

This is possible because Fortran 90 is a very high-level language. It allows, among others, user-defined data type, pointer access, function and operator overloading, and modular program development where implementation and interface are separated. In the above code, we do not specify how the sparse matrices are compressed and distributed. These decisions could be handled by a default settings, or by additional interfaces that let programmers dictate the desired compression and distribution schemes. We will later shown in section 2.2 how such schemes can be expressed by the programmers.

Our libraries are built for both sequential and parallel environments. Table 1 shows a list of sparse intrinsic functions used in Fortran 90. These operations include CSHIFT, EOSHIFT, MATMUL, MERGE, PACK, SUM, RESHAPE, SPREAD, TRANSPOSE, UNPACK, and section moves. We show below another usage of Fortran 90 array intrinsic functions in a conjugate gradient solver. This code segment is translated from MATLAB code in [10, page 354, Fig. 8]. In the following code, if array `A` is sparse, then we can replace the dense-matrix/dense-vector multiplication (`Ap = MATMUL(A, p)`) by a spare-matrix/dense-vector version to gain better performance.

```
integer, parameter :: row =479, column = 479
double precision, dimension(row) :: x, b, r, p, Ap
double precision, dimension(row,column) :: A
double precision :: norm_r, norm_b, rtr, rtrold, beta, alpha, tol
```

Array Intrinsic Function	Functionality
CSHIFT	Circular shift of the elements of the argument array
DOT_PRODUCT	Calculate the dot product of two vectors
EOSHIFT	End-off shift of the elements of the argument array
MATMUL	Matrix Multiplication
MERGE	Combines two (conformable) arrays under the control of a mask
PACK	Pack an array
reduction	Reduce array dimensions with an operator
RESHAPE	Construct an array of a specified shape from the elements of a given array
SPREAD	Replicate an array by adding one dimension
section move	Perform data movement on a region of the array
TRANSPOSE	Matrix transposition
UNPACK	Unpack an array

Table 1: Array Intrinsic Functions in Fortran 90 for Data Parallel Programming

```
beta = 0.0
tol = 0.1
do
    norm_r = norm(r,size(r))
    norm_b = norm(b,size(b))
    if (norm_r .le. tol*norm_b) exit
    p = r + beta*p
    Ap = MATMUL(A,p)
    alpha = rtr/DOT_PRODUCT(p,Ap)
    x = x + alpha*p
    r = r - alpha*Ap
    rtrold = rtr
    rtr = DOT_PRODUCT(r,r)
    beta = rtr/rtrold
end do
```

2.2 Higher-Dimension Compression and Distribution Schemes

In recent research effort, such as [23, 7, 4], the distribution and compressed schemes being considered for sparse matrices are limited to one dimension and two dimension arrays. We consider here schemes for higher-dimensional arrays as well. The distribution schemes currently being considered are general block partitions based on number of non-zeroes. The compression schemes being considered are Compressed Row Storage (CRS), Compressed Column Storage (CCS), and dense representations. Table 2 lists the compression and distribution schemes for sparse matrices. Note that by (Block, *) scheme, we mean distributing the sparse matrix by row in a two-dimensional array. 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 non-zeros among processors, as long as it achieves over-all balanced between computation and communication. Likewise, (*, Block) partitions the matrix by column, while (Block, Block) partitions the matrix both by row and column. For a one-dimensional array,

Compression Scheme
Compressed Row Storage (CRS)
Compressed Column Storage (CCS)
Dense Representation

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

Table 2: Compression/Distribution Schemes for Two-dimensional Arrays

$$
\begin{bmatrix}
\mathbf{0} & \langle 0,0,0,5,3,0 \rangle & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \langle 2,0,1,0,0,0 \rangle & \langle 0,0,9,3,0,0 \rangle & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\langle 9,0,0,0,6,0 \rangle & \mathbf{0} & \mathbf{0} & \mathbf{0} & \langle 1,0,6,2,3,5 \rangle & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \langle 0,2,0,3,4,0 \rangle \\
\mathbf{0} & \mathbf{0} & \langle 1,0,0,0,7,2 \rangle & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \langle 2,4,2,8,0,0 \rangle & \mathbf{0} & \mathbf{0} & \mathbf{0} & \langle 0,0,0,7,3,4, \rangle
\end{bmatrix}
$$

Figure 1: Sample Sparse Matrix A(6,6,6)

its compression can be either in dense representation, or by an array of (index, value) pairs of the non-zero array elements.

We further extend two-dimensional compression schemes (such as CRS and CCS) into higher dimensional arrays. Our approach to dealing with higher-dimensional arrays is to employ 1-d and 2-d sparse arrays as the bases and compose them for higher-dimension arrays. Figure 1 illustrates an example of an three-dimensional sparse structure, A(6,6,6), and Figure 2 shows the compression scheme used. The compressed structure is a two-dimensional CRS structure with each element again a one-dimensional sparse structures. Similarly, to construct a four-dimensional arrays, one can employ 2-d sparse arrays as the bases and compose them together. The compressed structure will then be a two-dimensional compressed structure with each element again a two-dimensional sparse structure.

Our construction of the sparse arrays currently are in the form of Fortran 90 programs and libraries. The code fragment below shows a type descriptor for the three-dimensional sparse structure, of which each array element is a real number.

```
type sparse3d_real
  private
  type(descriptor) :: d
  integer,             pointer, dimension (:) :: RO
  integer,             pointer, dimension (:) :: CO
  type(sparse1d_real), pointer, dimension (:) :: value
end type sparse3d_real
```

Element-wise functions such as '+', '-', '*', etc., and Fortran 90 array intrinsic functions listed in Table 1 are

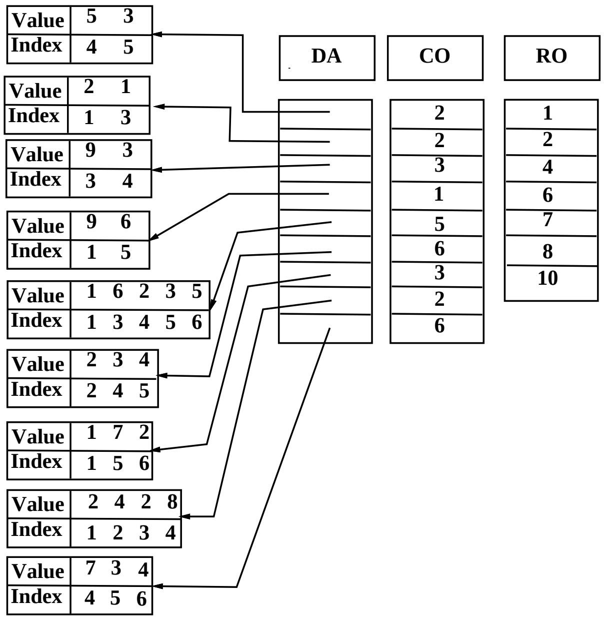

Figure 2: Three Dimensional Compression Schemes

implemented and overloaded in our sparse module. Therefore, the sparse structure of higher-dimensional arrays now can be operated in a way similar to dense arrays. The code segment below illustrate the parallel computation of the sparse structures using array intrinsics and element-wise functions.

```
program main
 use sparse
 type (sparse3d_real) :: a, b, c, d
 type (sparse3d_bool) :: y

 call bound(b, 6,6,6)
 call bound(c, 6,6,6)
 call bound(d, 6,6,6)
 call bound(y, 6,6,6)
 . . .
 d = c + merge(a, b, y)
 end program main
```

In the code above, the bound function sets the shape of the sparse structure. The merge operator is an overloaded function of the Fortran 90 intrinsic function for combining two (conformable) arrays under the control of a mask. However, it now operates over our sparse structures.

So far, we have shown how the multi-dimensional arrays can be compressed and programmed in Fortran 90. In the case when the program is executed on distributed memory environments, we need to further deal with the issues of data distribution. With two dimensional structures, we have (Block, *), (*, Block), and (Block, Block) distributions. For higher-dimensional arrays, the distribution can be done dimension-wise. Figure 3 and 4 give examples to illustrate the distribution schemes. The distribution is arranged as (block, block, *) in the program. We then show the code fragment below with compression and distribution schemes on parallel environments.

$$
\begin{bmatrix}
0 & \langle 0,0,0,5,3,0\rangle & 0 & 0 & 0 & 0 \\
0 & \langle 2,0,1,0,0,0\rangle & \langle 0,0,9,3,0,0\rangle & 0 & 0 & 0 \\
\langle 9,0,0,0,6,0\rangle & 0 & 0 & 0 & \langle 1,0,6,2,3,5\rangle & 0 \\
0 & 0 & 0 & 0 & 0 & \langle 0,2,0,3,4,0\rangle \\
0 & 0 & \langle 1,0,0,0,7,2\rangle & 0 & 0 & 0 \\
0 & \langle 2,4,2,8,0,0\rangle & 0 & 0 & 0 & \langle 0,0,0,7,3,4,\rangle
\end{bmatrix}
$$

Figure 3: Sample Sparse Matrix A(6,6,6) with Distribution on 4 Processors

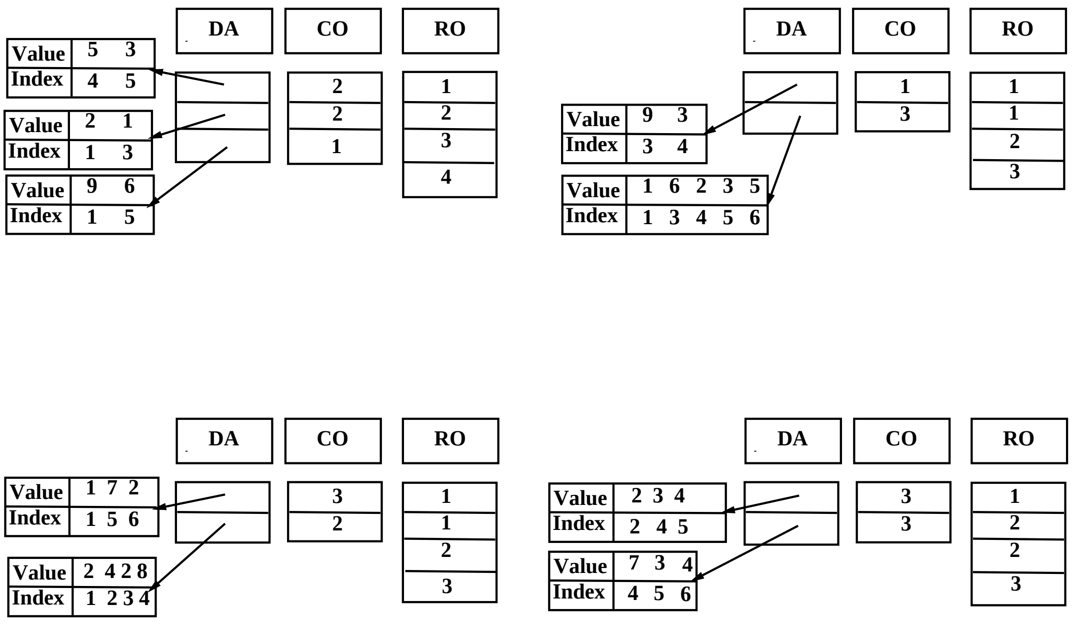

Figure 4: Three Dimensional Compression Schemes with (Block,Block,*) Distributions

```
program main
use sparse
type (sparse3d_real) :: c, t
type (sparse2d_real) :: v, u, b
type (sparse3d_real) :: y
integer :: i
call bound(b, 64,64,64)
call bound(c, 64,64,64)
call bound(t, 64,64,64)
call bound(v, 64,64)
call distribution(a, 'Block', 'Block', 'Whole')
call distribution(b, 'Block', 'Block', 'Whole')
call distribution(c, 'Block', 'Block', 'Whole')
call distribution(t, 'Block', 'Block', 'Whole')
. . .
t = eoshift (c, dim=1, 3)
u  = t(:,:,64) + filter(v)
i = maxval(b + u)
. . .
end program main
```

The above code also illustrates one aspect of the application to the compressed scheme for higher-dimensional arrays. Consider a two-dimensional data[20] coming with the time domain in the third dimension. The data set coming in can be very huge and sparse. At each time step, the data set then has to be shifting one distance away. New data can then be filtered and brought in. Sparse array intrinsic functions are important for this type of computations. Finally, the dimension order and the bases to be composed of a multi-dimension array can be made explicitly as well. We have an example shown below.

```
program main
use sparse
type (sparse3d_real) :: a,b
call set_dim_struct(a,/1,2/,/1,2,3/)
call set_dim_struct(b,/1,2/,/3,1,2/)
. . .
end program main
```

The statement set_dim_struct above sets the dimension order. It says array "a" is 1D $\times$ 2D, and the dimension order is 1, 2, and 3. Furthermore, it says array "b" is 1D $\times$ 2D, and the dimension order is 3, 1, and 2. The 1D array is constructed by the third dimension of the input array, and the 2D array is constructed by the first and second dimension of the input array. The mechanism allows manipulations of the large sparse objects. For example, consider the dataset used in the application of data-mining[11, 13], we can have one or two dimensional space to index each object. The object itself can again be two or three dimensional sparse objects. As the compressed arrays are composed with 1d or 2d array as the basis, the index in the higher-level can have quick access speed. The manipulation of the index domain allows flexibility to improve system performances.

3 Algorithms for Sparse Intrinsic Functions

In this section, we illustrate the main idea in implementing the sparse intrinsic functions for Fortran 90 on both sequential and parallel environments. Due to space limitations, we will present only the algorithms for CSHIFT and RESHAPE operations to illustrate the key idea in the process. Figure 5 and 6 first illustrate the the key idea of sparse and compressed algorithms for CSHIFT operation on sequential and distributed-memory parallel environments, respectively. The algorithm first checks if the compressed scheme is in the same direction as the shifting direction. For example, if the compressed scheme is CRS, and the requested shifting is along the row direction, we will have a match between compression scheme and shifting direction. If there is a match, the algorithm goes ahead to do shifting along the compressed dimension. In the case that the compressed dimension is not the same as the shifting direction, we have two ways to solve the problem. For example, if the compressed scheme is CRS, and the requested shifting is along the column, it will fall into this category. The first way to solve the problem is to first do a compression scheme transformation, and then do a shifting under matched compressed scheme and shifting directions. We can then transform the compressed scheme back to the original compressed scheme after the shifting if necessary. This method provides good abstractions with a step-by-step procedure. The second method is also outlined in Figure 5. For example, if the compressed scheme is CRS, and the requested shifting is along the column, we will first find the leading elements in each row. The shifting can be done from the first column and from left to right with the leading element informations. The parallel algorithm is then outlined in Figure 6. There, Send_Set needs to be calculated and local elements can then be merged with remote elements.

The above result is for one or two dimensional arrays. For higher-dimensional arrays, we used the compressed scheme proposed in Section 2.2. As mentioned earlier, our approach to dealing with higher-dimensional arrays is to employ 1-d and 2-d sparse arrays as the bases and compose them for higher-dimension arrays. For example, a four-dimensional arrays can be composed by 2-d sparse arrays. The compressed structure will then be a two-dimensional compressed structure with each element again a two-dimensional sparse structure. In the case that a CSHIFT operation is operated on a higher-dimensional arrays (such as 4-d sparse arrays), we will first need to find out the plane required for the shifting operation. It then can be done similar to the shifting operation in a 2-d sparse arrays.

Similarly, the RESHAPE operation can be done as follows. Suppose we want to reshape an source array of p dimensions into a target array of d dimensions. Assume that the shape of the source array is $(m_1, m_2, ..., m_p)$ and the target shape is $(d_1, d_2, ..., d_q)$. Our method is to map the array to one-dimensional array first, and then move the data into the target shape. In the parallel environment, we again need to calculate the sending set and receiving set. The mapping is illustrated below.

$$(m_1, m_2, ..., m_p) \rightarrow 1d\ Array \rightarrow (d_1, d_2, ..., d_q)$$

Due to space limitations, we only presented the key idea for CSHIFT and RESHAPE operations here. The detailed algorithms for all the sparse and parallel intrinsic functions will be available in [5].

4 Experimental Results

In this section, we report preliminary performance results from experiments with sparse support of Fortran 90 array intrinsic functions. We show results from three test cases: conjugate gradients solver (cgsolver), banded matrix multiplication (banmul), and red-black Gauss-Seidel relaxation (relaxation). They are shown in Figure 7.

```
Input : 1. ARRAY is of size n by m with CRS form (RO1,CO1,DA1)
        2. shape(SHIFT) = (s_1, s_2, ..., s_n)
        3. DIM = 2
Begin_Of_Algorithm
        if the compressed scheme is with CRS and shift direction is in row direction
          each row do
                Scan this row to find the element with the smallest column index after the shifting.
                Use the found element as the leader in the row to do shifting and the rest of the elements in the row follow.
          end do
        else if the compressed scheme is with CCS and shift direction is in column direction
          each column do
                Scan this column to find the element with the smallest row index after the shifting.
                Use the found element as the leader in the column to do shifting and the rest of the elements in the column follow.
          end do
        else /* the compressed scheme is not the same as the shifting directions */
            /* WLOG, Assuming CRS used and shifting is in column direction */
            begin
                To find the leader in each row.
                To perform column shifting from left to right by using leader information in each row
            end
End_Of_Algorithm
```

Figure 5: Compressed Sparse Algorithm for CSHIFT Operations

```
Input : 1. ARRAY is of size n by m with CRS form (RO1,CO1,DA1)
        2. shape(SHIFT) = (s_1, s_2, ..., s_n)
        3. DIM = 2
Begin_Of_Algorithm
        Each processor do
        begin
          Calculate the Send_Set to other processors due to shifting operations
          Calculate the set of elements remaining local after the shifting.
          Perform communication to send the Send_Set to other processors, and receive remote elements from other processors.
          Unpack the receiving elements, and combine them with local remaining elements according to SHIFT operations.
        end_each_processor_do
End_Of_Algorithm
```

Figure 6: Parallel Sparse Algorithms for CSHIFT Operations

Code segment of cgsolver:

```
integer, parameter :: row =479, column = 479
double precision, dimension(row) :: x, b, r, p, Ap
double precision, dimension(row,column) :: A
double precision :: norm_r, norm_b, rtr, rtrold, beta, alpha, tol

beta = 0.0
tol = 0.1
do
    norm_r = norm(r,size(r))
    norm_b = norm(b,size(b))
    if (norm_r .le. tol*norm_b) exit
    p = r + beta*p
    Ap = MATMUL(A,p)
    alpha = rtr/DOT_PRODUCT(p,Ap)
    x = x + alpha*p
    r = r - alpha*Ap
    rtrold = rtr
    rtr = DOT_PRODUCT(r,r)
    beta = rtr/rtrold
end do
```

Code segment of banmul:

```
integer, parameter :: row = 1000
real, dimension(row,2*row-1) :: A
real, dimension(row) :: x, b
integer, dimension(2*row-1) :: shift

b = sum(A*eoshift(spread(x,dim=2,ncopies=2*row-1),&
        dim=1,shift=arth(-row+1,1,2*row-1)),dim=2)
```

Code segment of relaxation:

```
integer, parameter :: row = 400, column = 400
double precision, dimension(row,column) :: u, rhs
integer :: n
double precision :: h, h2

h = 1.0/(n-1)
h2 = h*h
u(2:n-1:2,2:n-1:2) = 0.25*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
      u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-h2*rhs(2:n-1:2,2:n-1:2))
u(3:n-2:2,3:n-2:2) = 0.25*(u(4:n-1:2,3:n-2:2)+u(2:n-3:2,3:n-2:2)+&
      u(3:n-2:2,4:n-1:2)+u(3:n-2:2,2:n-3:2)-h2*rhs(3:n-2:2,3:n-2:2))
u(3:n-2:2,2:n-1:2) = 0.25*(u(4:n-1:2,2:n-1:2)+u(2:n-3:2,3:n-1:2)+&
      u(3:n-2:2,3:n:2)+u(3:n-2:2,2:n-2:2)-h2*rhs(3:n-2:2,2:n-1:2))
u(2:n-1:2,3:n-2:2) = 0.25*(u(3:n:2,3:n-2:2)+u(1:n-2:2,3:n-2:2)+&
      u(2:n-1:2,4:n-1:2)+u(2:n-1:2,2:n-3:2)-h2*rhs(2:n-1:2,3:n-2:2))
```

Figure 7: Three test cases

The cgsolver code segment is translated from the MATLAB code in [10, page 354, Fig. 8], and the banmul and relaxation code segments are taken from [18, page 1019 and 1338].

In the experiments, original Fortran 90 array intrinsics in the code are replaced with the corresponding routines from our sparse library. Performance of the resultant sparse version is then compared with the original dense version. Furthermore, the sparse version is executed on an 8-node IBM SP2 cluster to measure its performance speedup. In cgsolver, we replace the dense-matrix/dense-vector multiplication (`Ap = MATMUL(A,p)`) to a sparse-matrix/dense-vector version (`A` is sparse while `p` and `Ap` dense). The test matrix `A` is a 479×479 sparse matrix with non-zero ratio of 0.03. It is taken from [10, page 352, Fig. 4]. In banmul, the spread and eoshift array intrinsics are changed from the dense versions to sparse versions. The test matrix `A` has a non-zero ratio of 0.0005, and the test vector `x` has a non-zero ratio of 0.1. In relation, all array section moves and array additions are replaced with the sparse versions. Both the test matrices `u` and `rhs` each has a non-zero ratio of 0.25. The results are shown in Table 3. All the time measured are in seconds. The results show that the performance of sparse versions are better than the dense versions except for the relaxation example. The sparse versions also exhibit improved performance when executed on multiple processors. The performance of the sparse relaxation code is bad because of a naive translation of the dense code to sparse code where temporary sparse arrays are used to store intermediate results. We will later discuss in section 5 optimization techniques that are needed for elimination of such temporary arrays.

Focus is now directed to Table 4 which shows the preliminary performance and scalability of each individual sparse routine for various Fortran 90 array intrinsic functions. The experiment is conducted assuming the sparse matrix has been partitioned and distributed on the processors before the sparse routines are invoked, and the target and source arrays are properly aligned. The experiment is conducted to evaluate the basic performance characteristics of each individual intrinsic. Note that in our earlier experiments with those three applications, the compression schemes might have to be transformed from one shape into another between different phase of computations with intrinsic functions. For example, in the banmul application, after the spread operation, the compression scheme is transformed to work with the eoshift operation. Table 4 represents the performance for each individual operator when data are properly aligned, and compression scheme conversions are not needed between different computation phases. The test done in Table 4 uses the same sparse matrix used for the cgsolver.

5 Optimization Issues and Discussions

We outline in this section several optimization issues in the design and implementation of an intrinsics-based sparse library. When applicable, we also briefly describe how techniques originated from implementing (dense) array languages and libraries can be used to approach these issues. For issues unique to sparse optimizations, we discuss directions for solving the problems.

In the following discussion, we will use *representation scheme* as a general term to denote both compression scheme and distribution scheme for sparse matrices. In general, compression schemes concern representations of a sparse matrix (or a partition of it) in a single processor, while distribution schemes concern the ways a sparse matrix is partitioned among the processors. A good compression scheme aims at both economical memory usage and fast data access, while a good distribution scheme balances between computation load and communication cost.

From a programming perspective, a sparse matrix differs from a dense matrix in that representations for a dense matrix are often dictated by the programming languages or their implementations, while those of a sparse matrix need not, and better not, to be. For example, matrices are column-majored in Fortran and row-major in C/C++. On the contrary, sparse matrices, when viewed as instances of abstract data types, need not have fixed compression

Performance result of cgsolver:

no. of proc.	F90 dense code	with sparse support
1	161.5	41.6
2	-	24.1
4	-	15.6
8	-	12.1

Performance result of banmul:

no. of proc.	F90 dense code	with sparse support
1	1.48	0.66
2	-	0.38
4	-	0.22
8	-	0.14

Performance result of relaxation:

no. of proc.	F90 dense code	with sparse support
1	0.19	1.32
2	-	0.74
4	-	0.50
8	-	0.33

Table 3: Performance results of the three test cases.

no. of proc.	1	2	4	8
cshift	1.04×10^{-2}	5.35×10^{-3}	2.75×10^{-3}	1.42×10^{-3}
eoshift	1.76×10^{-1}	8.80×10^{-2}	4.50×10^{-2}	2.30×10^{-2}
matmul	1.20×10^{-1}	6.00×10^{-2}	3.11×10^{-2}	1.60×10^{-2}
merge	2.55×10^{-2}	1.30×10^{-2}	6.63×10^{-3}	3.43×10^{-3}
section move	1.00×10^{-1}	5.01×10^{-2}	2.61×10^{-2}	1.48×10^{-2}
spread	2.51×10^{-3}	1.33×10^{-3}	7.08×10^{-4}	3.94×10^{-4}
sum	7.38×10^{-2}	3.74×10^{-2}	1.90×10^{-2}	9.01×10^{-3}

Table 4: Performance of sparse intrinsic functions

schemes. Another example is distribution schemes of dense matrices in High Performance Fortran. They are fixed at compile-time by directives. This will not work well for sparse matrices because good distributions necessarily will depend on non-zero structures of the sparse matrices which usually are not available until run-time.

As a consequence, the design of a sparse library can be very flexible, but its efficient implementation is considerably harder since much less representational information is known statically. One can always choose fixed representation schemes for all sparse matrices, just like one does for dense matrices in Fortran and C/C++. However, this often results in bad performance. Our sparse library for Fortran 90 array intrinsics uses a two-level design. We implement a collection of low-level sparse libraries with each library using one fixed representation scheme. These low-level routines are efficient so long as their input matrices are of the right representations. They are tedious to use because arguments about sparse representation have to be given when they are invoked. On top of the low-level libraries, a high-level library is constructed such that decisions about sparse representation are hidden from the programmers. The low-level library is best utilized by a sparse compiler where the representation selections are dictated by the result of program analyses; while the high-level library is best used by application programmers where representation selections are handled by the library itself without help from the compiler. Many optimization issues thus mainly concern the inference and utilization of representation schemes of sparse matrices, such that suitable low-level routines can be selected for their perspective intrinsic operations.

5.1 Selection and Conversion of Representation Schemes

Many array intrinsics demand specific representations of their input arguments for efficient computation. As an example, let us consider the following statement that uses the intrinsic function `MATMUL` for matrix multiplication.

```
A = MATMUL(B, C)
```

An efficient computation of `MATMUL` will require matrix B to be compressed using the CRS scheme and C using the CCS scheme. The resultant matrix A can be either in CRS or CCS scheme depending on how the nested loop of sparse dot products is arranged. Similar requirement for distribution schemes also arises in the implementation of `MATMUL`.

To optimize performance, one needs to choose suitable representation schemes for sparse matrices, and to perform conversions of schemes if necessary. To use the low-level sparse array intrinsic library, one needs to supply matrices in specific schemes, and and must provide conversions whenever needed. However, to get performance, this often requires global understanding of the computation cost involved in a multi-statement block. For example, in the following two-statement block

```
A = MATMUL(B, C) + MATMUL(C, B)      ! statement 1.
A = A + D                            ! statement 2.
```

one can observe that

- from statement 2, matrices A and D should use the same representation scheme; and if the representation of D is already fixed, then
- the right hand side of statement 1 should produce a representation for A that agrees with D; and
- one can convert the representations of B or C (or both) to get better performance of the two `MATMUL` function calls.

Assuming matrix D is already in CRS scheme, then the following low-level invocation sequences should be a good way to get performance:

```
! First assert that B is in CRS scheme and C in CCS scheme, then
!   call MATMUL_CRS_CRS_CCS, which will produce result in CRS scheme given
!   the two arguments already in CRS and CCS schemes respectively.
call MATMUL_CRS_CRS_CCS (A, B, C)

! Convert the representation schemes of B and C, and produce the result of
!   the second multiplication in a temporary matrix A1.
call CRStoCCS (B, B1)
call CCStoCRS (C, C1)
call MATMUL_CRS_CRS_CCS(A1, C1, B1)

! Use AddSet_CRS to add a matrix to another matrices; both are in CRS scheme.
call AddSet_CRS(A, A1)
call AddSet_CRS(A, D)
```

There are other ways to translate the two-statement block to low-level invocation sequences using other compression schemes for the matrices. The goal is to choose one translation that exhibits good performance. One can use the following naive strategy

> For each array intrinsic function in the block, choose one of its low-level sparse routines that is most efficient, and convert the arguments to the required representations, then call the routine. If there are several low-level routines that are equally suitable, choose one that will require the least cost of converting the argument representations.

Other more complicated strategies for representation selection/conversion are possible. For example, one can use global analysis with the help of a cost model (more in section 5.3).

5.2 Adaptive Representation Schemes

Recall that our sparse library is a two-level design where high-level routines are built on top of low-level but efficient routines. In the interface of high-level routines, decisions about sparse matrix representations are hidden from the programmers. The programmers just use Fortran 90 array expressions to perform sparse matrix computations. But as shown previously, the low-level routines demand specific representation schemes for the input arguments. Hence, the implementations of the high-level routines must select, and convert if necessary, representation schemes for its input arguments and pass them to the low-level routines. The problem is that during different stages of execution, various representation schemes may be assigned to a single sparse matrix variable. This require a level of indirection in the implementation of a sparse matrix such that different representations can be freely associated and dissociated with the matrix variable when are needed.

Adaptive representation scheme is also needed because the sparsity of a sparse matrix, hence its representation, may change during execution. For example, if we iterate over $A, A^2, A^3, \ldots, A^n$, until it converges to calculate the transitive closure of a sparse matrix A, then the number of no-zeroes increases with the iteration. This may affect how the matrices is represented in order to get good performance. For example, at some stage during the iteration, the matrix may have sufficient number of non-zeroes to require a dense representation. The way the matrix is partitioned may change as well to improve load-balancing.

5.3 Cost Modeling

The exact performance of a low-level sparse routine depends on many factors: number of the non-zeroes in the matrices, representation schemes that are used, and performance characteristics of the underlying execution environment (computation speed vs. communication bandwidth). In order to help select suitable representation scheme for sparse matrices, one need to be able to estimate the relative performance of low-level routines, given estimations of the sparsity structures of the matrices involved.

Cost functions of various low-level sparse routines can be constructed in a pre-processing stage from performance measurements of those routines on specific machine using sample matrices of various sparsity and representation. At run-time, when selecting representation schemes to use in a low-level routine, cost functions of those routines (one for each choice of representation) can be evaluated according to the current sparsity of the input matrices. The routine with the smallest cost will be selected as the representation. Detailed description of this approach can be found in our earlier work[7].

5.4 Storage Reuse and Aggregate Data Access

Look at the following Fortran 90 array statement, which is a part of the relaxation example as described in Section 4.

```
u(2:n-1:2,2:n-1:2) = 0.25*(u(3:n:2,2:n-1:2)+u(1:n-2:2,2:n-1:2)+&
          u(2:n-1:2,3:n:2)+u(2:n-1:2,1:n-2:2)-h2*rhs(2:n-1:2,2:n-1:2))
```

A naive translation to low-level sparse routines will require 6 temporary sparse array variables for the right hand side (one for each of the five array sections, and the sixth as the accumulator to store the result of the entire right hand side.) An efficient translation will require only two temporary arrays where one is reused to store the current array section; then it is added to the other one, the accumulator, in a successive matter. We can further eliminate the temporary array for array sections by a single loop over the 5 array sections, and put the result one element at a time into the accumulator temporary array.

This problem is related to the problem of array operation synthesis, where intermediate arrays during the evaluation a Fortran 90 array expressions are eliminated to reduce store and load instructions [12]. Techniques previously developed to eliminate temporary dense arrays can be applied to eliminate temporary sparse arrays as well. The sparse problem differs from the dense problem in that representations and sizes of the temporary sparse arrays may not be the same, hence may not be directly reusable. We are currently investigating performance tradeoffs between using efficient representations and reusable representations for temporary sparse arrays.

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 [2, 15], or by programmers using portable sparse libraries [10, 19]. Approaches based on high-level language primitives and structural analyses have been successfully used to parallelize dense matrix computations [6, 12].

Several recent works have used statistical information about the applications or the target machines to help select implementation strategies [8, 17]. These methods seem to concentrate on specific applications, while ours center around efficient support of array intrinsic functions for sparse data sets [4, 7]. Runtime compilation techniques and their applications to sparse computation can be found at [24].

Our work here addressed an open gap by presenting an efficient library for parallel sparse computations with Fortran 90 array intrinsic operations. Our method provided both compression schemes and distribution schemes on distributed memory environments. In addition, we addressed the issues with the compression and distribution schemes for higher-dimension sparse arrays. Previously, research work [4, 7, 21, 22, 23] only considered the distribution and compressed schemes limited to one dimension and two dimension arrays. Preliminary experimental results show that our approach is promising in speeding up sparse matrix computations on both sequential and distributed memory environments if the computations are expressed with Fortran 90 array expressions.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brain T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. Intertext Publications/McGraw–Hill Inc., 1992.

[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] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific programming*, 2(3), Fall 1993.

[4] Rong–Guey Chang, Cheng–Wei Chen, Tyng–Ruey Chuang, and Jenq Kuen Lee. Towards automatic supports of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience*, 9(11):1101–1111, November 1997. This special journal issue is dedicated to the *1997 ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, Nevada, USA, June 1997.

[5] Rong–Guey Chang, Tyng–Ruey Chuang, and Jenq Kuen Lee. Algorithms for Parallel and Sparse Array Intrinsic Functions in Fortran 90. Technical Notes, in preparation, Tsing-Hua Univ., Taiwan, 1998.

[6] 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.

[7] 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.

[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] Luiz De Rose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 309–316. Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[10] 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.

[11] Sanjay Goil and Alok Choudhary. High Performance OLAP and Data Mining on Parallel Computers, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[12] 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.

[13] Mahesh V. Joshi, George Karypis, and Vipin Kumar. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets, In Proceedings of IPPS/SPDP '98, Orlando, April, 1998.

[14] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation Series. The MIT Press, 1994.

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

[16] Jenq Kuen Lee and D. Gannon. Objected-oriented parallel programming: Experiments and results. In *Proceedings of Supercomputing '91.* New Mexico, USA, 1991.

[17] 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.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in Fortran 90: The Art of* Parallel *Scientific Computing.* Cambridge University Press, 1996.

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

[20] Samuel D. Stearns, Ruth A. David. Signal processing algorithms using Fortran and C Prentice-Hall, 1993.

[21] M. Ujaldon, S. D. Sharma, J. Saltz, and E. L. Zapata. Run–time techniques for parallelizing sparse matrix problems. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems: 2nd International Workshop*, pages 43–57. Lyon, France, September 1995. Lecture Notes in Computer Science, Volume 980, Springer–Verlag.

[22] Manuel Ujaldon and Emilio Zapata. Efficient resolution of sparse indirections in data–parallel compilers. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 117–126. Barcelona, Spain, July 1985. ACM Press.

[23] 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.

[24] 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.