The Journal of Supercomputing, 18, 305–339, 2001


# Parallel Sparse Supports for Array Intrinsic Functions of Fortran 90

RONG-GUEY CHANG rgchang@puma.cs.nthu.edu.tw
*Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan*

TYNG-RUEY CHUANG trc@iis.sinica.edu.tw
*Institute of Information Science, Academia Sinica, Taipei, Taiwan*

JENQ KUEN LEE jklee@cs.nthu.edu.tw
*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-dimensional sparse arrays. This way, programmers need not worry about low-level system details when developing sparse applications. Sparse programs can be expressed concisely using array expressions, and parallelized with the help of our library. Our sparse libraries are built for array intrinsics of Fortran 90, and they include an extensive set of array operations such as CSHIFT, EOSHIFT, MATMUL, MERGE, PACK, SUM, RESHAPE, SPREAD, TRANSPOSE, UNPACK, and section moves. Our work, to our best knowledge, is the first work to give sparse and parallel sparse supports for array intrinsics of Fortran 90. In addition, we provide a complete complexity analysis for our sparse implementation. The complexity of our algorithms is in proportion to the number of nonzero elements in the arrays, and that is consistent with the conventional design criteria for sparse algorithms and data structures. Our current testbed is an IBM SP2 workstation cluster. Preliminary experimental results with numerical routines, numerical applications, and data-intensive applications related to OLAP (on-line analytical processing) show that our approach is promising in speeding up sparse matrix computations on both sequential and distributed memory environments if the programs are expressed with Fortran 90 array expressions.

**Keywords:** array intrinsic functions of Fortran 90, non-zero element, parallel sparse computation, parallel sparse support, complexity analysis

## 1. Introduction

An increasing number of programming languages, such as Fortran 90 [1], High Performance Fortran (HPF) [16], APL [7], MATLAB [26], and possibly C++ with built-in array classes, 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 sets have been studied and shown to have

important impacts at program performance [6, 11, 13]. The usages of array intrinsic functions in Fortran 90 also grows recently, due to the availability of Fortran 90 compilers, and the presence of the tutorial books such as *Numerical Recipes in Fortran 90* [25] and *Fortran 90 Handbook* [1]. 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. In this paper, we address this open gap by proposing solutions and presenting an efficient library for parallel sparse computations with Fortran 90 array intrinsic operations.

With 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. Our library uses a two-level design. For each Fortran 90 intrinsic function, 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. In addition, 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.

For the compression schemes and distribution schemes, in our library, we extend conventional two-dimensional compression schemes, such as Compressed Row Storage (CRS) and Compressed Column Storage (CCS) [31], into higher dimensional arrays. As we are interested in providing solutions for sparse array intrinsics of Fortran 90, our compression schemes and distribution schemes need to be applicable to higher-dimensional sparse arrays. Our approach to deal with higher-dimensional arrays is to employ 1-d and 2-d sparse arrays as the bases and compose them for higher-dimensional arrays. For example, we can construct a three-dimensional sparse arrays by constructing two-dimensional CRS structure with each element again a one-dimensional sparse structure. Similarly, to construct a four-dimensional array, 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 sparse libraries are built for array intrinsics of Fortran 90, and they include CSHIFT, EOSHIFT, MATMUL, MERGE, PACK, SUM, RESHAPE, SPREAD, TRANSPOSE, UNPACK, and section moves. These libraries are built for both sequential and parallel environments, and for two-dimensional and higher-dimensional cases. In the experiments, we also apply our higher-dimensional constructions to a class of data-intensive applications related to OLAP (on-line analytical processing) [10, 14]. Our parallel sparse support for array intrinsics of Fortran 90, to our best knowledge, is the first work to provide such support. Previously, other researchers have showed how to annotate sparse notations and distributions for HPF-style languages, but did not provide sparse supports for array

intrinsics of Fortran 90 [29, 31]. In this work, we also provide a complete complexity analysis for our sparse implementation. Our implementation and complexity analysis present an upper bound for this problem. In addition, the complexity of our algorithms is in proportion to the number of non-zero elements in the arrays, and that is consistent with the conventional design criteria for sparse algorithms and data structures. Finally, 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. Our testbed is currently an IBM SP2 workstation cluster. The experiments are done with numerical routines from *Numerical Recipes in Fortran 90* [25], numerical applications [20], and data-intensive applications related to OLAP. They demonstrate the performance benefits of applying our libraries for data parallel sparse computations on both sequential and distributed memory environments. As such, our research work presents an important enhancement for parallel sparse computations with Fortran 90, as currently no vendor Fortran 90 or HPF compilers support the sparse version of array intrinsics of Fortran 90.

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-dimensional arrays. Section 3 presents the key algorithms of two array intrinsic functions with sparse representations to illustrate the idea in the implementations. Section 4 describes the experimental results, and Section 5 discusses performance issues and possible optimizations for our proposed systems. Section 6 surveys related work, and Section 7 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-dimensional arrays.

### *2.1. Motivating examples*

The following Fortran 90 program segment, called banmul, is taken from the book *Numerical Recipes in Fortran 90* [25, p. 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 [25], 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 expect 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 using array indices. Worse, representation details about how the sparse arrays are compressed and distributed also get mixed into the code. The resultant code is less readable and more difficult to be 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 default settings, or by additional interfaces that let programmers dictate the desired compression and distribution schemes. We will later show 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 array intrinsic functions used in Fortran 90. These operations include CSHIFT, EOSHIFT, MATMUL, MERGE, PACK, SUM, RESHAPE, SPREAD, TRANSPOSE, UNPACK, and array section move. 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 [9, p. 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
```

*Table 1.* Array intrinsic functions of Fortran 90 for data parallel programming

| Array Intrinsics | 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 elements of another 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 |

```
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-dimensional compression and distribution schemes*

In recent research effort, such as [3, 8, 9, 31], the distribution and compression schemes being considered for sparse matrices are limited to one-dimensional and two-dimensional 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. Likewise, (Block, *) partitions the matrix by column, while (Block, Block) partitions the matrix both by row and column. For a one-dimensional

*Table 2.* Compression and distribution schemes for two-dimensional arrays

| Compression scheme | Distribution scheme |
|---|---|
| Compressed row storage (CRS) | (Block, *) |
| Compressed column storage (CCS) | (*, Block) |
| Dense representation | (Block, Block) |

array, 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 (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 to represent higher-dimensional arrays. Figure 1 illustrates an example of an three-dimensional sparse array, 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 structure. Similarly, to construct a four-dimensional arrays, one can employ 2-d sparse arrays as the bases and compose them together. The representation will then be a two-dimensional compressed structure with each element in the structure again a two-dimensional compressed structure.

Our construction of the sparse arrays currently are in the form of Fortran 90 programs and libraries. The code fragment below shows a Fortran 90 derived type definition for three-dimensional sparse arrays, of which each array element is a real number. This type describes the compression scheme used in Figure 2.

```
type sparse3d_real
  type(descriptor) :: d
  integer, pointer, dimension(:) :: RO
```

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

*Figure 1.* A sample sparse matrix A(6, 6, 6).

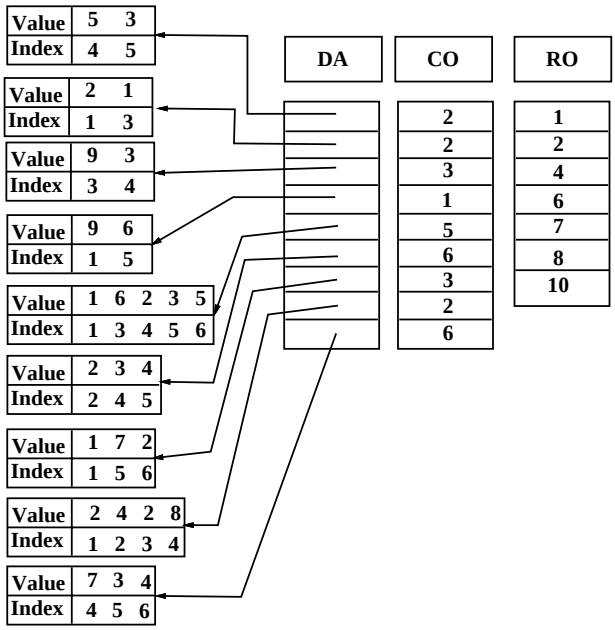


*Figure 2.* A three-dimensional compression scheme for array A of Fig. 1 .

```
  integer, pointer, dimension(:) :: CO
  type(sparse1d_real), pointer, dimension(:) ::DA
end type sparse3d_real
```

Elemental functions such as +, −, ∗, etc., and other transformational array intrinsic functions are implemented and overloaded in our sparse modules. Therefore, the sparse structure of higher-dimensional arrays now can be operated in a way similar to dense arrays.

The code segment below further illustrates 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

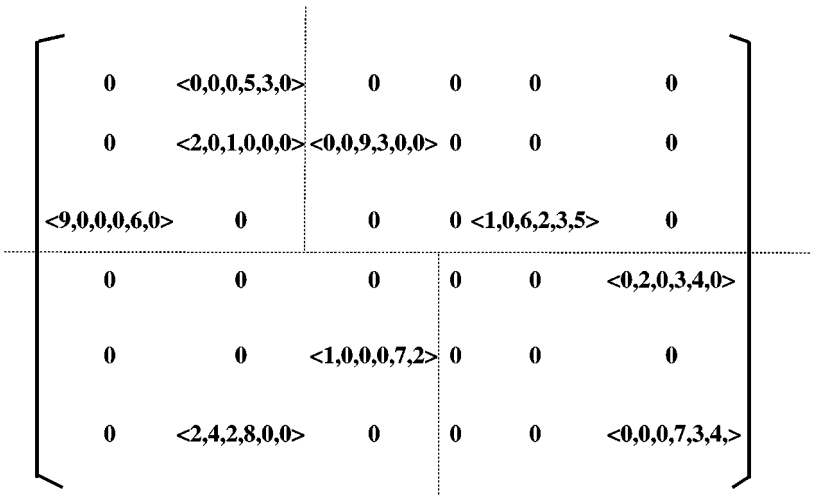


*Figure 3.* A(6, 6, 6) with distribution onto 4 processors.

combining two (conformable) arrays under the control of a mask. However, it now operates over our sparse structures.

So far, we have shown how sparse, 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.

Figures 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 specified distribution schemes for parallel environments.

```
use sparse
type(sparse3d_real):: c, t
type(sparse2d_real):: u, v, b
integer:: i
...
call distribution(c, Block, Block, Whole)
call distribution(t, Block, Block, Whole)
call distribution(u, Block, Block)
call distribution(v, Block, Block)
call distribution(b, Block, Block)
...
t = eoshift(c, dim=3, 1)
u = t(:,:,1) + filter(v)
i = maxval(b + u)
```

The above code also illustrates one application aspect of higher-dimensional arrays. Consider a two-dimensional data coming with the time domain in the third

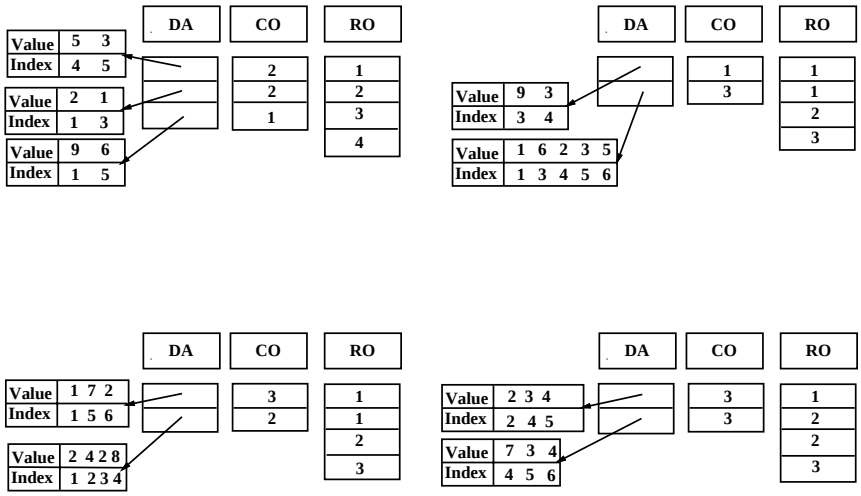


*Figure 4.* Three-dimensional compression scheme of A with (Block, Block, *) distribution scheme.

dimension [28]. The data set coming in can be very huge and sparse. At each time step, the data set has to be shifted 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-dimensional array can be made explicitly as well, as shown by the following code segment.

```
use sparse
type(sparse3d_real):: A, B
...
call set_dim_struct(A, (/2,1/), (/1,2,3/))
call set_dim_struct(B, (/1,2/), (/3,1,2/))
```

The statement set_dim_struct above sets the dimension order. The first call says array `A` is 2-d $\times$ 1-d, and the dimension order is 1, 2, and 3, which is shown in Figure 1. The second call says array `B` is 1-d $\times$ 2-d, and the dimension order is 3, 1, and 2. That is, the top-level 1-d sparse array is a sequence of non-zero planes along the third dimension of array `B`. Elements in the 1-d array are 2-d sparse arrays; each corresponds to a non-zero plane of the first and second dimension. This structure matches the shifting dimension of the `eoshift` function in statement `t = eoshift(c, dim=3, 1)`, and is the ideal compression scheme for sparse arrays `c` and `t`. This mechanism allows flexible and efficient manipulations of large sparse objects, as frequently found in data-mining applications [10, 14]. We can have one- or two-dimensional sparse array to index each object from the top. The object itself can again be one- or two-dimensional sparse objects. As the compressed arrays are composed with 1-d or 2-d 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 performance.

## 3. Algorithms for sparse intrinsic functions

In this section, we illustrate the main idea in implementing the sparse intrinsic functions of Fortran 90 on both sequential and parallel environments. Due to space limitation, we present only the algorithms for CSHIFT, SPREAD, and RESHAPE operations. We give full complexity analysis of our sparse implementations of Fortran 90 array intrinsics in Appendix A.

### *3.1. CSHIFT*

The call *CSHIFT(ARRAY, DIM, SHIFT)* will perform a circular shift on array *ARRAY* along the dimension *DIM*, with the shifting amount specified by *SHIFT*. Figures 5 and 6 illustrate the sparse algorithm for CSHIFT operations 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

```
Input:1. ARRAY is a n by m sparse array.
       2. DIM is the shifting direction.
       3. shape(SHIFT) = (s1, s2, ..., sn) is the shift vector.
Begin_Of_Algorithm
  If compressed by CRS and the shift direction is in row direction, then
    For each row do
      Scan this row to find the element with the starting column index
        after the shifting.
      Use the found element as the leader in the row to perform the shifting.
    End do
  Else if compressed by CCS and shift direction is in column direction, then
    For each column do
      Scan this column to find the element with the starting row index
        after the shifting.
      Use the found element as the leader in the column to perform the shifting.
    End do
  Else /* The compressed scheme is not the same as the shifting direction. */
       /* WLOG, assume CRS and the shifting is in the column direction. */
    Begin
      Find the leading column elements in all rows.
      Perform column shifting from left to right by using leader information
        from each row.
    End
End_Of_Algorithm
```

*Figure 5.* Sparse algorithm for CSHIFT(ARRAY, DIM, SHIFT) operation.

```
Input : 1. ARRAY is a n by m sparse array.
        2. DIM is the shifting direction.
        3. shape(SHIFT) = (s1, s2, ..., sn) is the shift vector.
Begin_Of_Algorithm
  For each processor do
  Begin
    Calculate the Send_Set to other processors according to the shifting
      direction and vector.
    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 remote elements, and combine them with local elements
      according to the shifting direction and vector.
  End
End_Of_Algorithm
```

*Figure 6.* Parallel sparse algorithm for CSHIFT(ARRAY, DIM, SHIFT) operation.

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 direction 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 the shifting under the matched compressed scheme and shifting direction. We then transform the compressed scheme back to the original compressed scheme after the shifting. This method provides good abstraction with step-by-step high-level procedures but is less efficient. The second method is outlined in Figure 5. Again, assuming the compressed scheme is CRS, and the requested shifting direction is along the column, we then first find the leading element in each row. The shifting can be done starting from the first column in a synchronized left-to-right manner using the leading element information. The parallel algorithm is outlined in Figure 6. There, elements in Send_Set need to be calculated, sent, received, and merged with local elements at other processors.

The above result is for one or two dimensional arrays. For higher-dimensional arrays, we use the compression 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-dimensional 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. The operation can then be performed in a way similar to the shifting operation in a 2-d sparse arrays.

### 3.2. SPREAD

The call *SPREAD(SOURCE, DIM, NCOPIES)* will duplicate, in *NCOPIES* times, the subarray along dimension *DIM* of the *SOURCE* array. Therefore, the spread function can extend a $r$-dimensional array into a $(r + 1)$-dimensional array along a specific dimension. In order to present the spread operation on sparse arrays clearly, we give the following two examples to show how it works. The array shown in Figure 7 is used as an example. As described in earlier sections, 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-dimensional arrays. Figure 7 illustrates an example of an three-dimensional sparse array, B(3, 3, 3) and its compressed structure. The compressed structure is a two-dimensional CCS structure with each element again a one-dimensional sparse structure.

In our algorithm for SPREAD operation, we follow the chain of the array basis to find the specified dimension. There are four possible cases to handle in the process. First, if the specified dimension is the first dimension, we add a new 1-d compressed basis with the size of NCOPIES, and each element of the new basis is a duplication of the source array. Figure 8 illustrates a SPREAD operation on the example array B for such a case. Second, the specified dimension can be found between two sparse array bases. For example, since array B in Figure 7 is a 2-d $\times$ 1-d array, if the specified dimension for SPREAD operation is dimension three, it falls into this category. In such a case, we add a new 1-d compressed basis of size NCOPIES between these two bases. Further more, each element of the new basis is a duplication of the original 1-d array. Third, the specified dimension can be found in the second dimension of a 2-d basis. For example, if the specified dimension for SPREAD operation on array B is dimension two, it falls into this category. In such a case, more care has to be taken. Figures 9 and 10 illustrate our solutions for such a case. In our solution, we first split the 2-d compressed structure into a 1-d $\times$ 1-d sparse structure. We then perform the SPREAD operation between these two bases

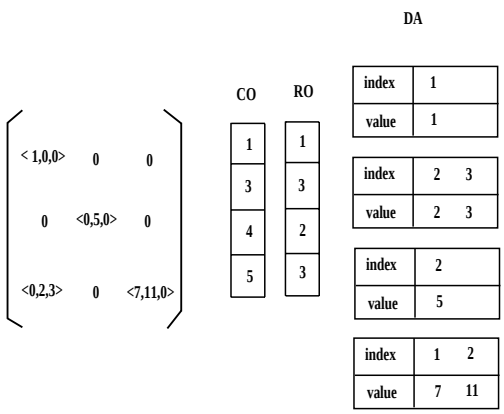


*Figure 7.* Array B(3, 3, 3) and its compressed structure: A 2-d $\times$ 1-d structure. The 2-d structure is compressed using CCS.

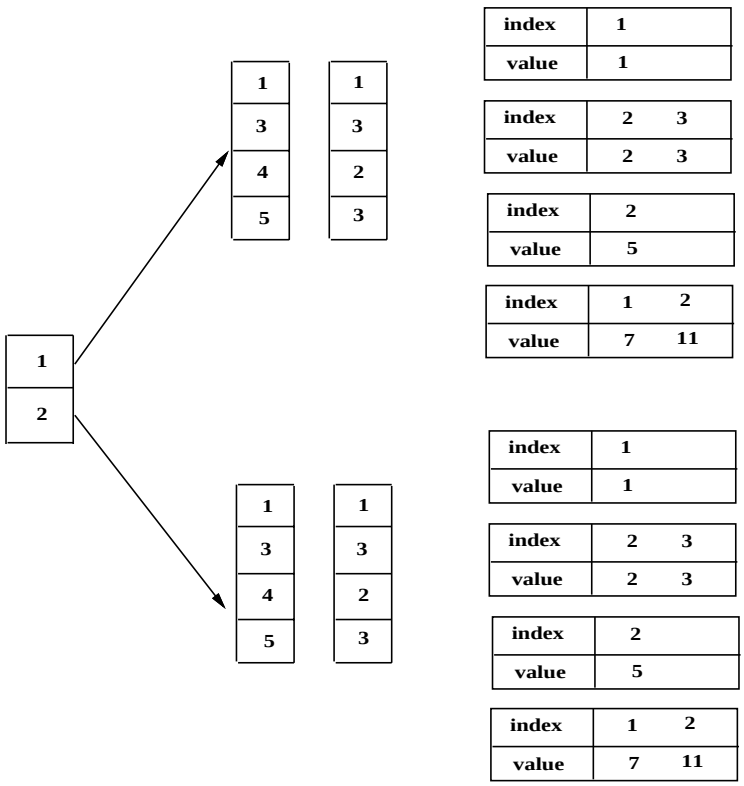


*Figure 8.* The resultant array of SPREAD(B, 1, 2).

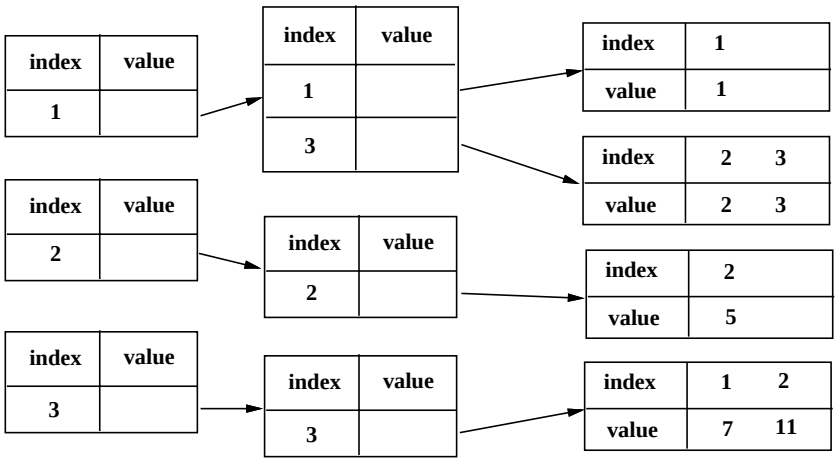


*Figure 9.* Decomposition of the 2-d compressed structure in B into a 1-d × 1-d structure. B is now a compressed 1-d × 1-d × 1-d structure.

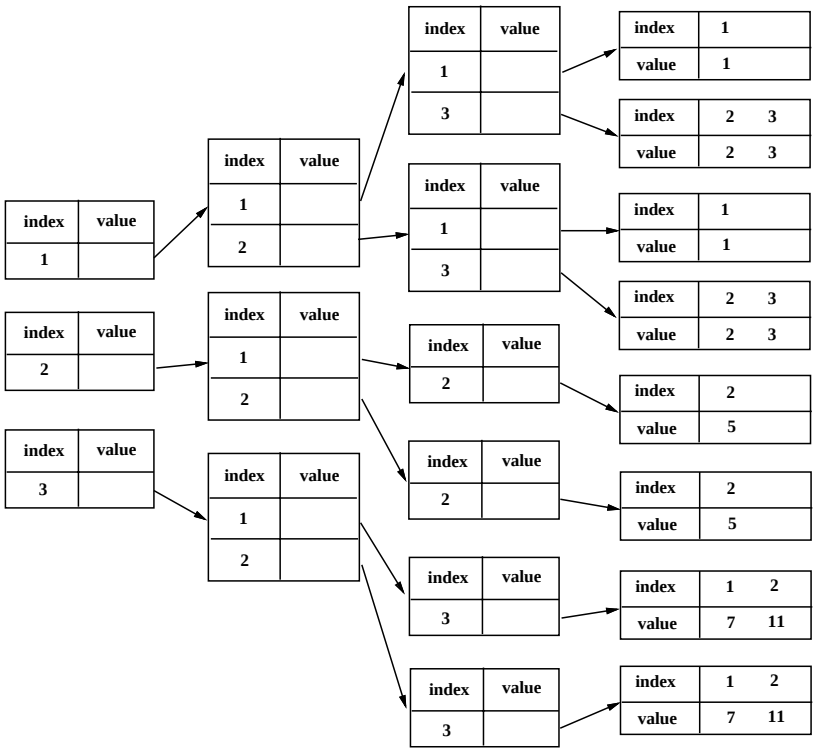


*Figure 10.* The resultant array of SPREAD(B, 2, 2), assuming a 2-d to (1-d × 1-d) decomposition has been performed first.

in a way similar to the handling of the second case above. Finally, if the specified dimension is the final dimension of the source array, we then replicate the element in the last basis which can be found by following the chain of the array bases. Each element of the last basis is now a vector of size NCOPIES.

Figure 13 gives a summary algorithm for the above procedure. The parallel and distributed version can be performed similarly to spread the array on distributed memory environments. Figure 11 shows the spread operation for array A of Figures 3 and 4 in the first dimension, while Figure 12 shows the spread operation on A along the third dimension. Each processor simply spread the elements it owns in parallel.

### *3.3. RESHAPE*

The call *RESHAPE(SOURCE, SHAPE, PAD, ORDER)* will re-arrange the *SOURCE* array into a new shape as specified by the rank one array *SHAPE*. Array *ORDER* specifies the dimension order of which the elements are retrieved form the *SOURCE* array. Elements in array *PAD* are used to fill up the target array if necessary. The following is our sparse implementation of the RESHAPE operation. Suppose we

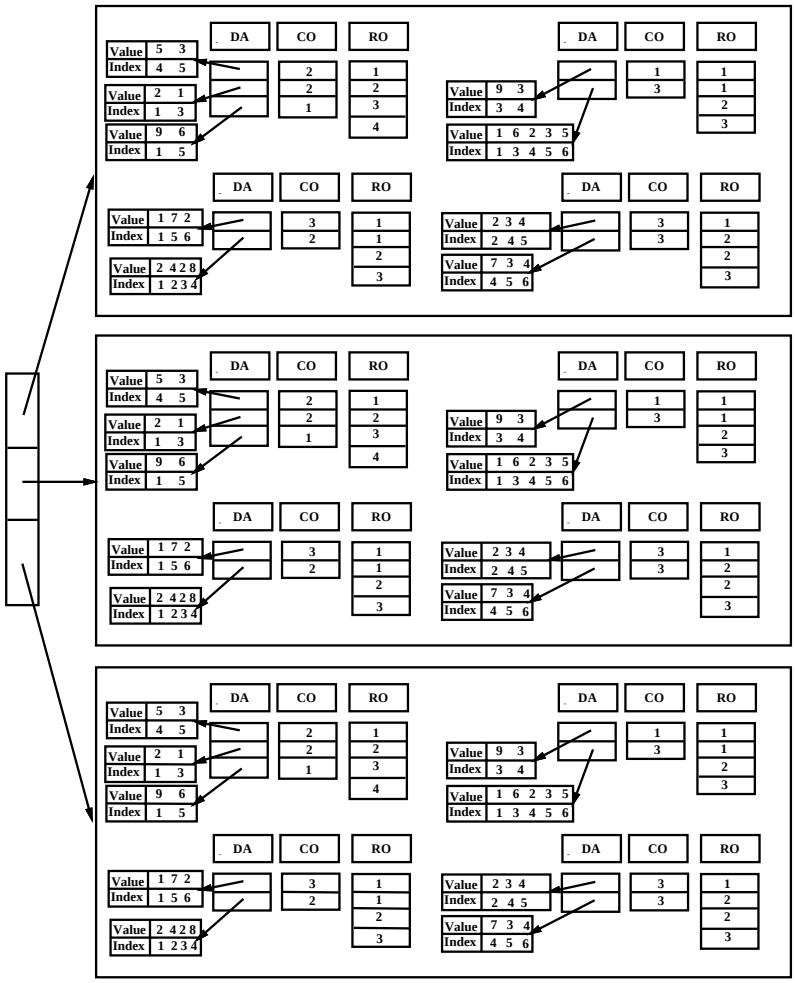


*Figure 11.* Parallel SPREAD operation on array A (as described in Figures 3 and 4), along the first dimension.

want to reshape an source array of $n$ dimensions into a target array of $m$ dimensions. Assume that the shape of the source array is $(p_1, p_2, \ldots, p_n)$ and the target shape, as specified by array *SHAPE* is $(q_1, q_2, \ldots, q_m)$. Our method is to map the source array to a one-dimensional array first, then move the data into the target shape. In the parallel environment, we again need to calculate the sending set and receiving set of the arrays elements that needs inter-processor communication. The mapping is illustrated below.

$$(p_1, p_2, \ldots, p_n)\text{-shaped Array} \rightarrow \text{1-d Array} \rightarrow (q_1, q_2, \ldots, q_m)\text{-shaped Array}$$

The key equations in the above transformation is the formula for translating the indices of nonzero elements between the source and target arrays.

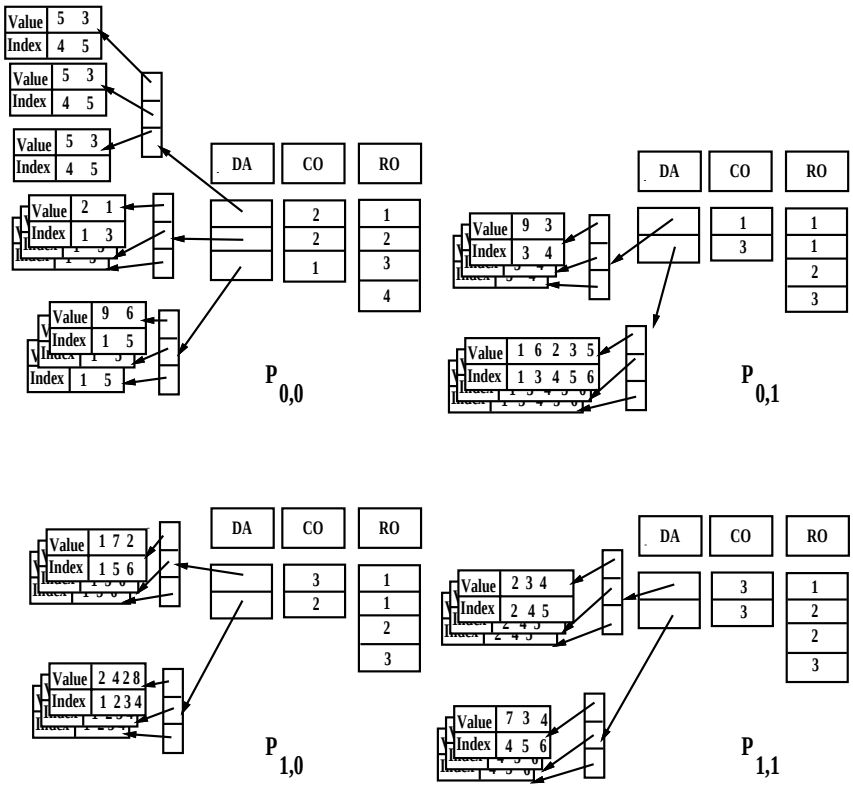


*Figure 12.* Parallel SPREAD operation on array A (as described in Figures 3 and 4), along the third dimension.

Let $A$ be the source array and $B$ be the target array. Furthermore, array element $A(s_1, s_2, \ldots, s_n)$ is reshaped as the array element $B(t_1, t_2, \ldots, t_n)$. We first calculate the total order, $I$, of each non-zero element $A(s_1, s_2, \ldots, s_n)$ using the following *order* function.

$$order(s_1, s_2, \ldots, s_n) = (s_1 - 1) \prod_{k=2}^{n} p_k + (s_2 - 1) \prod_{k=3}^{n} p_k + \cdots + s_n = I \qquad [1]$$

The next equation translates this total order into the index of the corresponding element in the target array.

$$B\left(\left\lceil \frac{I}{\prod_{k=2}^{m} q_k} \right\rceil, \mathrm{mod}\left(\left\lceil \frac{I}{\prod_{k=3}^{m} q_k} \right\rceil, q_2\right), \ldots, \right.$$
$$\left. \mathrm{mod}\left(\left\lceil \frac{I}{\prod_{k=i+1}^{m} q_k} \right\rceil, q_i\right), \ldots, \mathrm{mod}(I, q_m)\right). \qquad [2]$$

Once we have the indices of all the non-zero elements in $B$, we can then represent the sparse array $B$ in any compressed structure. Figure 14 gives the sparse algorithm for the RESHAPE operation. It also considers the case for distributed memory environments.

```
Input : 1. SOURCE may be scalar or array of any type.
        2. DIM is the specified dimension.
        3. NCOPIES ≥ 0 is the number of copies.
Begin_Of_Algorithm
  If the specified dimension is the first dimension, then {
    Add a new 1-d compressed basis with the size of NCOPIES.
    For each section of the new compressed basis do
      Duplicate the original array as the elements of 1-d basis. }
  Else if the specified dimension is between two bases, V1 and V2, then {
    Add a new 1-d compressed basis of size NCOPIES between the two basis.
    Make each element of the new basis a duplication of the original array,
      leading by the V2 basis. }
  Else if the specified dimension is the second dimension of a 2-d basis, then {
    Decompose the 2-d compressed structure into 1-d× 1-d compressed structure.
    Perform a SPREAD operation between the two bases as in the case above. }
  Else if the specified dimension is the final dimension of the source array, then {
    Replicate, NCOPIES times, each element in the last basis of the source array.
    Make each element of the last basis a new vector of above replications. }
End_Of_Algorithm
```

*Figure 13.* Sparse algorithm for SPREAD(SOURCE, DIM, NCOPIES) operation.

```
Input: 1. SOURCE may be of any type. It must be array valued.
       2. SHAPE must be a rank one array with size n < 8.
       3. PAD must be of the same type as SOURCE.
       4. ORDER is a permutation of (1,2,···,n).
Begin_Of_Algorithm
  Map the source array into a 1-d compressed array.
    { Calculate mapping indices of the source array using formula (1) in
        this section, as well as calculate Send_Set and Receiving_Set.
      Perform communication to send the Send_Sets to other processors, and
        receive remote elements from other processors.
      Combine the receiving elements with local elements according to
        mapping index.
    }
  Map the 1-d intermediate array into the target array with shape SHAPE.
    { Calculate mapping indices of the target array using formula (2).
      Perform communication for Send_Set and Receiving_Set.
      Combine the receiving elements with local elements according to
        the new indices.
    }
  If PAD is present, then append the non-zero elements in PAD into
    the target array.
End_Of_Algorithm
```

*Figure 14.* Sparse algorithm for RESHAPE(SOURCE, SHAPE, PAD, ORDER) operation.

## 4. Experimental results

In this section, we report preliminary performance results with sparse support of Fortran 90 array intrinsic functions. We will show three set of experiments to demonstrate the performances of our system. In the first set of our experiment we show the performance results for numerical routines. We show results from three test cases: sparse conjugate gradient solver (cgsolver), banded matrix multiplication (banmul), and factorial by using gamma functions. The code fragments with original Fortran 90 dense notations are shown in Figure 15. The cgsolver code segment is translated from the MATLAB code in [9, p. 354, Fig. 8], and the banmul and factorial routines are taken from [25, p. 1019 and 1088]. 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 $479 \times 479$ sparse matrix with non-zero ratio of 0.03. It is taken from [9, p. 352, Fig. 4]. In banmul, the spread, eoshift, and sum (reduction) array intrinsics are changed from the dense versions to our 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 factorial code, array intrinsic functions `pack` and `unpack` are replaced with the sparse versions. Both test matrices `n` and `mask` are $400 \times 400$ sparse matrices with non-zero ratio of 0.01. The results are shown in Table 3. All the times measured are in seconds. The results show that the performance of sparse versions are better than the dense versions. The sparse versions also exhibit improved performance when executed on multiple processors.

In the second set of our experiment, we test our support for parallel sparse intrinsics of Fortran 90 for application programs. The sparse library is currently being used in a joint work with Power Mechanics Department, National Tsing-Hua university, to develop scalable methods concerning the parallelization of pressure correction method on unstructured grid and large sparse systems [20]. In our initial experiment, we again use our the sparse conjugate gradient solver described earlier, but now the data set is taken from unstructured grid applications being developed at the Power Mechanics Department. In the unstructured fluid codes, an edge-based, cell-centered, finite-volume numerical procedure is employed to simulate incompressible flows. The computation is described as follows. First, the recursive graph bisection (RGB) algorithm is adopted to partition the unstructured grids on a distributed memory parallel machine (an IBM SP2). Then, the finite volume scheme performs stencil pattern (SIMPLE) computation. After the stencil pattern computation, the system needs to solve a linear system. Figure 16 shows the mapping of the unstructured grid into an adjacent matrix results in a large sparse matrix. A sparse conjugate gradient solver using our compressed schemes and intrinsic libraries are employed to solve the sparse linear systems. The performance results on 16-node IBM SP2 machines are shown in Table 4. All the times measured are in seconds. The problem size of the unstructured grid in the experiment is $128 \times 128$.

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 factorial:

```
integer, dimension(:,:) :: n
real, dimension(size(n)) :: factln_v
logical, dimension(size(n)) :: mask
integer, parameter :: TMAX = 100
real, dimension(TMAX), save :: a

mask = (n >= TMAX)
factln_v = unpack(gammln(pack(n,mask)+1.0),mask,0.0)
where (.not. mask) factln_v = a(n+1)
```

*Figure 15.* Three test cases

*Table 3.* Performance results of the three test cases

| No. of proc. | F90 dense code | With sparse support |
|---|---|---|
| **Performance result of cgsolver** | | |
| 1 | 161.5 | 41.6 |
| 2 | — | 24.1 |
| 4 | — | 15.6 |
| 8 | — | 12.1 |
| **Performance result of banmul** | | |
| 1 | 1.48 | 0.66 |
| 2 | — | 0.38 |
| 4 | — | 0.22 |
| 8 | — | 0.14 |
| **Performance result of factorial codes** | | |
| 1 | 3.57 | $3.80 \times 10^{-2}$ |
| 2 | — | $1.92 \times 10^{-2}$ |
| 4 | — | $9.63 \times 10^{-3}$ |
| 8 | — | $5.00 \times 10^{-3}$ |

Finally, we conduct experiments to demonstrate the performances of the sparse intrinsics beyond two-dimensional arrays. In the experiment, we use our compressed arrays to represent the data cube for a three-dimensional group-by operations related to OLAP (on-line analytical processing) [10, 14]. The code segment in Figure 18 is for a sequence of sum functions operated on sparse arrays ABC, AB, AC, BC, A, B, and C. We also have ABC with a nonzero ratio of 0.01. In group-by operation of OLAP, the relationships among these sparse arrays can be represented as a lattice in Figure 17. These are mainly higher-dimensional data cubes, and often they are large in sizes and very sparse. Our work can help the code segment in Figure 18 for parallel sparse computations. Original Fortran 90 dense array

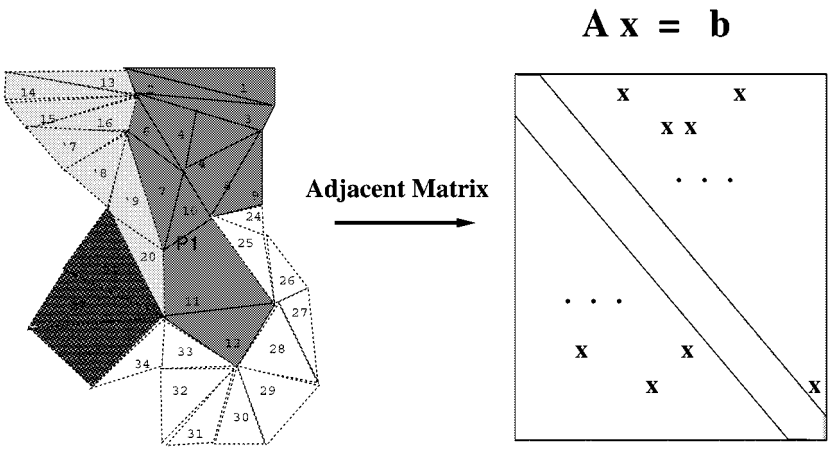


*Figure 16.* Mapping a unstructured grid into an adjacent matrix.

*Table 4.* Performance results for the sparse conjugate solver using the data set from a unstructured grid solver concerning the parallelization of pressure correction method on unstructured grid

| No. of proc. | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Execution time (seconds) with sparse support | 2.15 | 1.18 | 0.68 | 0.47 | 0.41 |

intrinsics in the code can be replaced with the corresponding sparse routines from our library. Performance results with our sparse library on 8-node IBM SP2 shows significant improvement over dense code (see Table 5). Note that it is quite frequent in OLAP computation that one will deal with higher-dimensional large data sets. These data sets are often sparse, and they demand efficient implementations of group-by operators specific to their sparse representation. This demand can be satisfied by using high-level Fortran 90 array expressions with support from our sparse library.

## 5. Optimization issues and discussions

In this section, we discuss 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 this paper, we address the issues to provide parallel sparse supports for array intrinsics of Fortran 90. Our library uses a two-level design. In the low-level routines, it requires the input sparse matrix to be specified with compression/distribution schemes for each function. Low-level communication routines use the Message Pass-

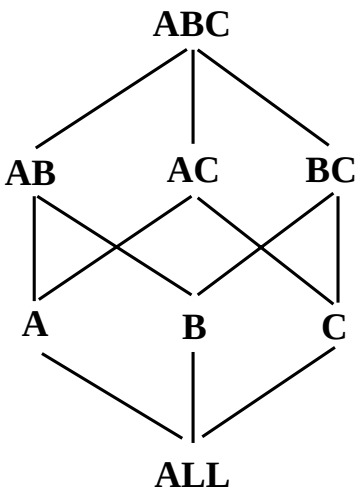


*Figure 17.* 3-d lattice for a group-by operation from OLAP.

```
integer, dimension(100,100,100) :: ABC
integer, dimension(100,100) :: AB
integer, dimension(100,100) :: AC
integer, dimension(100,100) :: BC
integer, dimension(100) :: A
integer, dimension(100) :: B
integer, dimension(100) ::  C
integer :: ALL

AB = SUM(ABC,dim=3)
AC = SUM(ABC,dim=2)
BC = SUM(ABC,dim=1)
A  = SUM(AB,dim=2)
B  = SUM(AB,dim=1)
C  = SUM(AC,dim=1)
ALL = SUM(A)
```

*Figure 18.* Code segment of group-by operator.

ing Interface (MPI) for communications of the sparse matrix among the processors. The library uses MPI and SPMD programming model. In the high-level representation, sparse array functions are overloaded for array intrinsic interfaces so that programmers need not concern about the low-level details. There exist quite a few optimization issues in transforming the high-level representations to low-level routines. As evidenced by the significant amount of work with compiler optimizations for dense arrays on distributed environments [6, 15, 18, 19, 23, 24], it will be hard for us to give all the sparse optimization techniques and all the details required in one work here. Therefore, in this paper, we have focused ourselves on our overall design, sparse library for array intrinsics, complexity analysis for sparse array intrinsics, higher-dimensional extensions for sparse arrays, and preliminary experimental results. For the optimization parts, we will only give the key idea we have been working on with related references below.

In transforming the high-level representations to low-level routines, if compression and distribution schemes of sparse arrays are specified by programmers, the transformation from high-level to low-level libraries is more a syntactic transformation. Optimization issues do occur with the assignments of compression and distribution schemes for temporary arrays of which the assignments cannot be done by programmers. In addition, if we wish to automatically supply the compression

*Table 5.* Performance results on 8-node IBM SP-2 for 3-d group-by with our sparse cube representations and intrinsic functions

| No. of proc. | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Execution time (seconds) with sparse support | 0.295 | 0.157 | 0.101 | 0.081 |

and distribution schemes for sparse arrays for programmers in our high-level representations, we will need to solve the classical assignment problems on alignments and distributions [6, 15, 18, 19, 23, 24], but now on sparse arrays and with additional compression scheme assignments. A key difference in assignment problems between sparse arrays and dense arrays is that the selections of distributions of sparse arrays are mainly based on the sparsity of arrays, while the selections of distributions of dense arrays are mainly based on the index domain of arrays. In our research work, we assume that the sparsity of source arrays can be profiled to obtain initial information. The sparsity for the rest of the arrays (including temporary arrays) operated with sparse array intrinsics can be estimated with a probability inference scheme developed earlier in [3, 5, 8]. Once sparsity of all arrays are estimated, we can estimate the communication cost and access cost for accessing the sparse arrays with assigned distribution and compression schemes by assuming uniform distribution of sparse elements or other given distributions of sparsity. With this cost model, we can use a two-phase assignments. In the first phase, we assign the compression schemes for sparse arrays, and then in the second phase, we assign the distribution schemes for the sparse arrays on distributed memory environments. This basically extends the dense distribution assignments for a sparse version [6, 15, 18, 19, 23, 24]. Details of our work can be seen in [5].

Another important optimization issue with sparse library design is the need of adaptive representation. 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 non-zeroes increases with the iteration. This may affect how the matrices is represented in order to get good performance. At some stage during the iteration, the matrix may have sufficient number of non-zeroes to require a dense representation. The way the assignment of the distribution and compression schemes may change as well to improve load-balancing.

We now discussed the issues on how the current framework can be extended to other languages. In this paper, we support the sparse versions of the array intrinsics for Fortran 90. For languages such as C++ and Java, one can use this approach and similarly design a sparse matrix class library to support high-level parallel sparse computation (see, e.g., [3]). However, because there is no language level constructs in C++ and Java for array intrinsics and expressions, the exact extent of array operations and functions that should be incorporated into the parallel sparse matrix library becomes an interesting design issue in itself. A simple design is to build the library like the one we have done for Fortran 90, and to add more functionality when the need arises. For high-level application languages like APL and Matlab, they often have their own array notations as well as efficient toolboxes for matrix operations. One can implement a sparse version of the toolbox such that it can interoperate with the original dense version [9]. However, it becomes complicated once one wants to implement a parallel version of the sparse toolbox, as the work will certainly involve the run-time system which may not have a standard way to interact with the MPI communication library or other user-defined C libraries.

Finally, a naive translation from translate high-level representations to low-level sparse routines for fat expressions in the RHS of a program statement will require extra temporary sparse array variables for the right hand side. (One for each of the array operation in the RHS and then one 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 manner. We can further eliminate the temporary array for array sections by a single loop over the RHS 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 [11, 12, 13]. 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 stay the same, hence may not be directly reusable. There is still an open research issue in performance tradeoffs between using efficient representations and using reusable representations for temporary sparse arrays.

## 6. Related work and future directions

### *6.1. Related work*

In this paper, we propose solutions to provide parallel sparse supports for array intrinsics of Fortran 90 as well as its parallelization. Our support works for two-dimensional arrays and higher-dimensional arrays. Previous research work considered the distribution and compressed schemes related to one dimension and two dimension arrays [3, 8, 29, 30, 31]. The work in [31] pioneered the research direction to annotate sparse notations and distributions for HPF-style languages. In our research work, we built an extension applicable to higher-dimensional arrays and with array intrinsics for sparse data parallel computations. 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-dimensional arrays. The work in [2] addressed the sparse compiler issues. They mainly deal with shared memory environment, while we work on distributed memory environments. In addition, the work in [2] was limited to Fortran 77 syntax and two-dimensional arrays, and did not address the issues of Fortran 90 array intrinsics.

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, 17], or by programmers using portable sparse libraries [9, 27]. Approaches based on high-level language primitives and structural analyses have been successfully used to parallelize dense matrix computations [7, 11]. Several recent works have used statistical information about the applications or the target machines to help select implementation strategies [21]. These methods seem to concentrate on specific applications, while ours center

around efficient support of array intrinsic functions for sparse data sets [3, 8]. Runtime compilation techniques and their applications to sparse computations can be found at [32]. In transforming the high-level representations of our sparse library to low-level SPMD routines, we can perform optimizations to supply compression and distribution schemes of sparse arrays in an automatic way [5]. In our related work, we assume that the sparsity of source arrays can be profiled to obtain initial information. The sparsity for the rest of the arrays (including temporary arrays) operated with sparse array intrinsics can be estimated with a probability inference scheme developed earlier in [3, 5, 8]. Once sparsity of all arrays are estimated, we can estimate the communication cost and access cost for accessing the sparse arrays with assigned distribution compression schemes by assuming uniform distribution of sparse elements or other given distributions of sparsity [5].

### *6.2. Future research directions*

In this paper, we address the issues of providing parallel sparse supports for array intrinsics of Fortran 90, both on sequential environments and on distributed memory parallel environments. An interesting aspect of the problem, but not addressed in the current work, is the support of parallel array intrinsics of Fortran 90 on shared memory parallel environments. With the emerging of the standard like OpenMP [22], it will be interesting to see how the current work can be efficiently integrated into OpenMP platforms of shared memory parallel systems.

Next, the efficient support of sparse array intrinsics of Fortran 90 on sequential environments is an important research issue in itself. More aggressive optimization techniques, especially for advanced architectures such as Intel IA-64 that support speculative execution, for sparse computation should be both interesting and challenging for further explorations.

Finally, there are domain specific languages that provide high-level array primitives that go beyond array intrinsics of Fortran 90. The group-by operator in the domain of OLAP is just an example. Often these high level array primitives are applied to sparse data sets as well, and they present special optimization opportunities on both sequential and parallel execution environments. One can always implement a customized library for a fixed set of these high level primitives so that they operate efficiently on sparse data sets. It remains an open research issue on how to support, in a systematic manner, an entire class of new primitives based on efficient implementations of existing, low-level, sparse array primitives.

## 7. Conclusion

We have presented 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-dimensional sparse arrays. This way, programmers need not worry about low-level system details. Sparse programs can be expressed concisely using array

expressions, and parallelized with the help of our library. Our sparse libraries are built for array intrinsics of Fortran 90, and they include an extensive set of array operations such as CSHIFT, EOSHIFT, MATMUL, MERGE, PACK, SUM, RESHAPE, SPREAD, TRANSPOSE, UNPACK, and section moves. These libraries are built for both sequential and parallel environments, and for two-dimensional and higher-dimensional cases. In addition, we provided a complete complexity analysis for our sparse implementation. The complexity of our algorithms is in proportion to the number of non-zero elements in the arrays, and that is consistent with the conventional design criteria for sparse algorithms and data structures. 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.

## Acknowledgments

The authors express their gratitude to the anonymous reviewers for their valuable suggestions and comments. In addition, we would like to acknowledge the National Center for High-Performance Computing, Taiwan, for providing access to IBM SP-2 machines.

## Appendix

### *A. A complexity analysis of our sparse implementations of Fortran 90 array intrinsics*

In this section, we give a complexity analysis for our sparse implementation of Fortran 90 array intrinsics, for both 2-d and higher-dimensional cases. The complexity analysis presents an upper bound of the computational cost for each of the sparse implementations of the array intrinsics. We show that, except for de-generate cases, the computation cost of our algorithm is in proportion to the number of non-zero elements in the arrays, which is consistent with the conventional design criteria for sparse algorithms and data structures.

Before we give the complexity analysis for each sparse array intrinsic, we first introduce some notations, as well as some design considerations. First, we only consider rectangular arrays. Without loss of generality, we also assume a $k$-dimensional array has extent $n$ at each of its dimensions (i.e., each dimension has $n$ elements).

Second, for higher-dimensional arrays (beyond 2-d), the complexity of the algorithm will also include the cost of traversing the intermediate bases in the hierarchy of the composed sparse arrays, in addition to the cost of processing non-zero elements. For example, for each 2-d compressed plane (either in CCS or CRS compression scheme) of a higher-dimensional structure, there will be at least $O(n)$ cost for processing and storing the indices of the random-access dimension. Therefore, for a 2-d $\times$ 2-d $\times$ 2-d sparse arrays, we will have at least $O(n^3)$ cost, which may

*Table 6.* Complexity of sparse array intrinsics in Fortran 90 (for 2-d and sequential cases)

| Array intrinsics | Complexity |
|---|---|
| `cshift`(*Array, Shift, Dim*) | $O(\text{nnz}(Array) + n)$ |
| `eoshift`(*A, Shift, Boundary, Dim*) | $O(\text{nnz}(A) + n + Shift \times \text{nnz}(Boundary))$ |
| `merge`(*Tsource, Fsource, Mask*) | $O(\text{nnz}(Tsource) + \text{nnz}(Fsource) + \text{nnz}(Mask) + n)$ |
| `spread`(*Source, Dim, Ncopies*) | $O(\text{nnz}(Source) + n + Ncopies \times (\text{nnz}(Source) + n))$ |
| `pack`(*Array, Mask*) | $O(\text{nnz}(Array) + \text{nnz}(Mask) + n)$ |
| `unpack`(*Vector, Mask, Field*) | $O(\text{nnz}(Vector) + \text{nnz}(Mask) + \text{nnz}(Field) + n)$ |
| `sum`(*Array, Dim, Mask*) | $O(\text{nnz}(Array) + \text{nnz}(Mask) + n)$ |
| `reshape`(*Source, Shape, Order*) | $O(\text{nnz}(Source) + n)$ |

be larger than the number of non-zero elements, for array intrinsics that need to access all array elements. To control this overhead, we use $\phi(A)$ to represent the maximal cost of any sparse array construction, which will be linear to the number of non-zeroes. The definition of $\phi(A)$ is the following. Let $A$ be a $k$-dimensional array, and the extent of each dimension be $n$, then function $\phi(A)$ is defined as

$$\phi(A) = n^r$$

where $r$ is the maximum integer such that $2 * r \leq k$, $n^r \leq \text{nnz}(A)$, and $\text{nnz}(A)$ is the total number of non-zero elements in $A$. That is, in the construction of sparse structure for the $k$-dimensional arrays, only $r$ 2-d planes will be used. The remaining dimensions will be implemented as compressed 1-d vectors, whose cost is linear to the extent of the dimension. This way, we can make sure that the complexity of algorithm is related to the total number of non-zero elements in the sparse array.

Third, in the parallel implementations for distributed memory environments, we divide the cost into three essential components. They are the size of the communication set, the cost for packing elements in the sending site, and the cost for receiving elements and local computations in the receiving site. The cost for each component is described below. Let $A$ be an array, then let $\text{nnz}(A_i)$ represent the set of non-zero elements of $A$ residing on processor $i$ in the distributed memory environment. We use notation $\max_i(\text{nnz}(A_i))$ to represent the largest set of non-zero elements of array $A$ that resides in the processors. The communication set is the total amount of communication needed for the specified operations. The cost for packing element in the sending site is bound by $\max_i(\text{nnz}(A_i))$, as that is the maximum amount of elements each processor can send. In the complexity analysis, we also assume the owner-computing rule is used.

Note that the distribution of sparse arrays is not guided by an even partition of the its index range, but rather by an even partition based on non-zero structure in the matrix. In our complexity analysis, we assume that the de-generate cases do not happen, and each processor receives the even amount of non-zero elements. To partition index range of arrays among the processors such that the non-zeroes are evenly distributed, one can use a form of inspector processing [8, 32]. As the inspection phase often takes less cost than the actual computation and communication phase, we do not include the complexity of inspection in our analysis.

Tables 6 and 7 list the complexity of our sequential and parallel implementations of Fortran 90 array intrinsics on 2-d arrays. Tables 8 and 9 list the complexity

*Table 7.* Complexity of sparse array intrinsics in Fortran 90 (for 2-d and parallel cases)

| Array intrinsics | Complexity | | | Precondition |
|---|---|---|---|---|
| | Pack & send | Receive & local combine | Communi-cation set | |
| `cshift` $(A, S, D)$ | — | $O(\max_i(\text{nnz}(A_i)) + n)$ | — | shift direc. = distri. direc. |
| `cshift` $(A, S, D)$ | $O(\max_i(\text{nnz}(A_i)) + n)$ | $O(\text{nnz}(A)/p + n)$ | $O(\text{nnz}(A))$ | shift direc. $\neq$ distri. direc. |
| `eoshift` $(A, S, B, D)$ | — | $O(\max_i(\text{nnz}(A_i)) + n + S \times \max_i(\text{nnz}(B_i))$ | — | shift direc. = distri. direc. |
| `eoshift` $(A, S, B, D)$ | $O(\max_i(\text{nnz}(A_i)) + n + \max_i(\text{nnz}(B_i)))$ | $O(\text{nnz}(A)/p + n + S \times (\text{nnz}(B)/p))$ | $O(\text{nnz}(A))$ | shift direc. $\neq$ distri. direc. |
| `merge` $(T, F, M)$ | $O(\max_i(\text{nnz}(M_i)) + \max_i(\text{nnz}(T_i)) + \max_i(\text{nnz}(F_i)) + n)$ | $O((\text{nnz}(M) + \text{nnz}(T) + \text{nnz}(F))/p + n)$ | $O(\text{nnz}(M) + \text{nnz}(T) + \text{nnz}(F))$ | — |
| `spread` $(S, D, NS)$ | — | $O(\max_i(\text{nnz}(S_i)) + n + NS \times (\max_i(\text{nnz}(S_i)) + n))$ | — | no decompo. |
| `spread` $(S, D, NS)$ | $O(\max_i(\text{nnz}(S_i)) + n)$ | $O(NS \times (\max_i(\text{nnz}(S_i)) + n))$ | $O(\text{nnz}(S))$ | decompo. |
| `pack` $(A, M)$ | $O(\max_i(\text{nnz}(A_i)) + \max_i(\text{nnz}(M_i)))$ | $O((\text{nnz}(A) + \text{nnz}(M))/p + n)$ | $O(\text{nnz}(A) + \text{nnz}(M))$ | — |
| `unpack` $(V, M, F)$ | $O(\max_i(\text{nnz}(V_i)) + \max_i(\text{nnz}(M_i)) + \max_i(\text{nnz}(F_i)))$ | $O((\text{nnz}(V) + \text{nnz}(M) + \text{nnz}(F))/p + n)$ | $O(\text{nnz}(V) + \text{nnz}(M) + \text{nnz}(F))$ | — |
| `sum` $(A, D, M)$ | — | $O(\max_i(\text{nnz}(A_i)) + \max_i(\text{nnz}(M_i)) + n)$ | — | specified dim. distri. by * |
| `sum` $(A, D, M)$ | $O(\max_i(\text{nnz}(A_i)) + \max_i(\text{nnz}(M_i)) + n)$ | $O((\text{nnz}(A) + \text{nnz}(M))/p + n)$ | $O(\text{nnz}(A) + \text{nnz}(M))$ | specified dim. distri. |
| `reshape` $(S, Sh, O)$ | $O(\max_i(\text{nnz}(S_i)) + n + \text{nnz}(S)/p)$ | $O(\text{nnz}(S)/p + n)$ | $O(\text{nnz}(S))$ | — |

of the intrinsics for higher-dimensional arrays. Each of the following subsections introduces the functionality of one array intrinsic function, as well as gives a brief complexity analysis of its sparse implementations.

### *A.1.* `cshift`*(Array, Shift, Dim)*

The `cshift` operation moves all the elements in the source array. The complexity of our sparse algorithm is proportional to the number of non-zero elements in the source array.

In the parallel version for distributed memory environments, there will be no communication cost if data distribution matches the specified shifting direction. For example, if the shifting direction is the row-dimension, and data is distributed by row among processors, there will be no communication cost. Otherwise, communication

*Table 8.* Complexity of sparse array intrinsics in Fortran 90 (for higher-dimensional and sequential cases)

| Array intrinsic function | Complexity |
|---|---|
| cshift$(A, S, D)$ | $O(\text{nnz}(A) + \phi(A))$ |
| eoshift$(A, S, B, D)$ | $O(\text{nnz}(A) + \phi(A) + S \times \text{nnz}(B))$ |
| merge$(T, F, M)$ | $O(\text{nnz}(M) + \text{nnz}(T) + \text{nnz}(F) + \max(\phi(M), \phi(T), \phi(F)))$ |
| spread$(S, D, NS)$ | $O(\text{nnz}(S) + \phi(S) + NS \times (\text{nnz}(S) + \phi(S)))$ |
| pack$(A, M)$ | $O(\text{nnz}(A) + \text{nnz}(M) + \max(\phi(A), \phi(M)))$ |
| unpack$(V, M, F)$ | $O(\text{nnz}(V) + \text{nnz}(M) + \text{nnz}(F) + \phi(M) + \phi(F))$ |
| sum$(A, D, M)$ | $O(\text{nnz}(A) + \text{nnz}(M) + \phi(A) + \phi(M))$ |
| reshape$(S, Sh, O)$ | $O(\text{nnz}(S) + \phi(S))$ |

is needed. As mentioned earlier, we divide the cost into three essential components: the size of the communication set, the cost for packing elements in the sending site, and the cost for receiving elements and local computations in the receiving site. In this case, the communication set in the worst case is $O(\text{nnz}(A))$, as in the extreme case where one has one row at each processor, and the elements are shifted along the column direction. The complexity for packing and sending elements in the sending site is $O(\max_i(\text{nnz}(A_i)) + n)$. We need to add the overhead of $O(n)$ for traversing the local 2-d compressed array at each sending site. Next, we consider

*Table 9.* Complexity of sparse array intrinsics in Fortran 90 (for higher-dimensional and parallel cases)

| | Complexity | | |
|---|---|---|---|
| Array intrinsics | Pack & send | Receive & local combine | Communication set |
| cshift $(A, S, D)$ | $O(\max_i(\text{nnz}(A_i)) + \phi(A))$ | $O(\text{nnz}(A)/p + \phi(A))$ | $O(\text{nnz}(A))$ |
| eoshift $(A, S, B, D)$ | $O(\max_i(\text{nnz}(A_i)) + \phi(A) + \max_i(\text{nnz}(B_i)))$ | $O(\text{nnz}(A)/p + \phi(A) + S \times (\text{nnz}(B)/p))$ | $O(\text{nnz}(A) + \text{nnz}(B))$ |
| merge $(T, F, M)$ | $O(\max_i(\text{nnz}(M_i)) + \max_i(\text{nnz}(T_i)) + \max_i(\text{nnz}(F_i)) + \max(\phi(T), \phi(F), \phi(M)))$ | $O((\text{nnz}(T) + \text{nnz}(F) + \text{nnz}(M))/p + \max(\phi(T), \phi(F), \phi(M)))$ | $O(\text{nnz}(T) + \text{nnz}(F) + \text{nnz}(M))$ |
| spread $(S, D, NS)$ | $O(\max_i(\text{nnz}(S_i)) + \phi(S))$ | $O(\text{nnz}(S)/p + NS \times (\max_i(\text{nnz}(S_i)) + \phi(S)))$ | $O(\text{nnz}(S))$ |
| pack $(A, M)$ | $O(\max_i(\text{nnz}(A_i)) + \max_i(\text{nnz}(M_i))) + \max(\phi(A), \phi(M)))$ | $O((\text{nnz}(A) + \text{nnz}(M))/p + \max(\phi(A), \phi(M)))$ | $O(\text{nnz}(A) + \text{nnz}(M))$ |
| unpack $(V, M, F)$ | $O(\max_i(\text{nnz}(V_i)) + \max_i(\text{nnz}(M_i)) + \max_i(\text{nnz}(F_i)) + \phi(M) + \phi(F))$ | $O((\text{nnz}(V) + \text{nnz}(M) + \text{nnz}(F))/p + \phi(M) + \phi(F)) \text{nnz}(F))$ | $O(\text{nnz}(V) + \text{nnz}(M) + \text{nnz}(F))$ |
| sum $(A, D, M)$ | $O(\max_i(\text{nnz}(A_i)) + \max_i(\text{nnz}(M_i)) + \phi(A) + \phi(M))$ | $O((\text{nnz}(A) + \text{nnz}(M))/p + \phi(A) + \phi(M))$ | $O(\text{nnz}(A) + \text{nnz}(M))$ |
| reshape $(S, Sh, P, O)$ | $O(\max_i(\text{nnz}(S_i)) + \phi(S) + \text{nnz}(S)/p)$ | $O(\text{nnz}(S)/p + \phi(S))$ | $O(\text{nnz}(S))$ |

the cost in the receiving site. As mentioned earlier, the data partitioning is done based on an even distribution of the non-zero elements among processors, and the owner-compute rule is used. Without considering de-generate case, and assuming each processor getting an even partition of non-zero elements for the target array, then the cost at the receiving site is $O(\text{nnz}(A)/p + n)$. Tables 6 and 7 summarize the complexity results, for `cshift` and other intrinsics, for the 2-d cases.

For higher dimension arrays, the complexity of the algorithm is proportional to the number of non-zero elements in the source array, with the additional cost of traversing the intermediate nodes of the composed sparse structure. For each 2-d compressed plane, as mentioned previously, we will need at least $O(n)$ traversal cost. Therefore, for a 2-d $\times$ 2-d $\times$ 2-d array, we will need $O(n^3)$ cost. To control this overhead, we use function $\phi(A)$ described above to represent the cost of a sparse array construction that is linear to the number of non-zero elements. The traversal overhead is now $\phi(A)$ instead of $O(n^3)$ in this six-dimensional case. Tables 8 and 9 summarize the complexities of `cshift` and other array intrinsics for higher-dimensional arrays, each for the sequential and parallel cases.

### *A.2.* `eoshift`*(Array, Shift, Boundary, Dim)*

The `eoshift` operation is similar to the `cshift` operation except that vacant positions in the target array are filled with elements from array *Boundary*, or with zero elements if not specified, instead of elements circulated from the source array. Let's consider the sequential cases first. If array *Boundary* is absent, `eoshift` is similar to `cshift` except that zero elements will be filled. Therefore, it has the complexity of $O(\text{nnz}(Array) + n)$. For the case that array *Boundary* is present, then the number of elements to be shifted from *Boundary* to the target array is at most the product of shifting distance *Shift* and $\text{nnz}(Boundary)$. Tables 6 and 8 show the complexity results for the sequential cases, for both 2-d and higher-dimensional arrays.

In the parallel version for distributed memory environments, we first assume array $A$ and *Boundary* has the same distribution. For 2-d arrays, and following the discussion for the `cshift` intrinsic, we can see that the worst-case communication set has size $O(\text{nnz}(A) + \text{nnz}(Boundary))$. The complexity for sending elements is $O(\max_i(\text{nnz}(A_i)) + n + \max_i(\text{nnz}(Boundary_i)))$, as one processor can only send as many elements as it owns. We need to add the overhead of $O(n)$ for traversing a 2-d compressed array. Next, we consider the cost in the receiving site, as well as the cost to combine remote elements with local elements. Here we assume that array $A$ and *Boundary* use the same distributions among processors, and the cost in the receiving site will be $O((\text{nnz}(A) + \text{nnz}(Boundary))/p + n)$. For higher dimensional arrays, the complexity of the algorithm is similar to 2-d cases, but with the additional cost of traversing the intermediate nodes of the composed sparse structure, which is $\phi(A)$. In the case that array $A$ and *Boundary* do not have the same distribution for non-zero elements, the complexity for receiving site is then $O(\text{nnz}(Array)/p + n + Shift \times (\text{nnz}(Boundary)/p))$ for 2-d arrays and $O(\text{nnz}(Array)/p + \phi(Array) + Shift \times (\text{nnz}(Boundary)/p))$ for higher-dimensional arrays. Tables 7 and Table 9 give the complexity for the parallel versions.

### *A.3.* `merge`*(T source, F source, Mask)*

The `merge` operation works as follows. The element of the target array comes from *T source* if the corresponding element at array *Mask* is of true value, otherwise it comes from *F source*. Thus, `merge` function will scan all non-zero elements at the three arrays. Then we have the complexity as $O(\text{nnz}(Tsource) + \text{nnz}(Fsource) + \text{nnz}(Mask) + n)$ as shown in Table 6 for the 2-d, sequential, case.

In the parallel version, the worst-case communication set results from different distribution schemes for *TSOURCE*, *FSOURCE*, and *MASK*. The complexity is then $O(\text{nnz}(Tsource) + \text{nnz}(Fsource) + \text{nnz}(Mask) + n)$. For computation phase, the complexity of at the sending site is $O(\max_i(\text{nnz}(Mask_i)) + \max_i(\text{nnz}(T_i)) + \max_i(\text{nnz}(F_i)) + n)$, and the cost at receiving site for local computation is $O((\text{nnz}(Tsource) + \text{nnz}(Fsource) + \text{nnz}(Mask))/p + n)$. The result is listed in Table 7.

For higher dimensional arrays, the complexity of the algorithm is proportional to the number of non-zero elements, with the additional cost of traversing the intermediate nodes of the three composed sparse structures. Tables 8 and 9 give the results for sequential and parallel complexity for higher dimension arrays, respectively.

### *A.4.* `spread`*(Source, Dim, Ncopies)*

Function `spread` duplicates, in *NCOPIES* times, the subarray along dimension *DIM* of the *SOURCE* array. Therefore, the spread function can extend a $r$-dimensional array into a $(r + 1)$-dimensional array along a specific dimension.

The sparse algorithm for spread operation was given earlier in Section 3.2. The complexity for sequential and 2-d case is described as follows. In our algorithm for SPREAD operation, we follow the chain of the array basis to find the specified dimension. There are two cases. If the specified dimension is found in the second dimension of a 2-d basis, we first decompose the 2-d compressed array into a 1-d $\times$ 1-d sparse compressed structure. We then perform SPREAD operations between these two bases to duplicate the desired sub-arrays. If the decomposition is not needed, the complexity is $O(Ncopies \times (\text{nnz}(Source) + n))$, otherwise it takes extra $O(\text{nnz}(Source) + n)$ cost for the decomposition. Table 6 gives the complexity for the sequential version of the 2-d case.

Similarly, Table 7 gives the complexity for parallel version of 2-d cases. If the decomposition is not needed, there is no communication cost needed as well. If the decomposition is needed, the extra communication cost is needed to perform decomposition; then we perform the spread operations. The complexity for higher dimensional arrays also has similar properties, as it includes the decomposition cost and the cost in traversing the structure.

### *A.5.* `pack`*(Array, Mask)*

The intrinsic function `pack` groups elements in *Array* into a rank one array under the control of *Mask*. Let us consider the 2-d case first. First, if the compressed schemes

of *Array* and *Mask* are different, then a compression scheme conversion must be performed. Then a traversal with non-zero elements of these two arrays can then be performed to pack the elements into rank one target array. In the worst case, the complexity of our algorithm includes both the conversion and traversal costs, which are $O(\text{nnz}(Array) + \text{nnz}(Mask) + n)$.

In the parallel version, the costs for the three essential components are the following. First, the communication set as a whole is at most proportional to the summation of the non-zero elements of *Array* and *Mask*. For receiving elements and local computation, the cost is $O((\text{nnz}(Array) + \text{nnz}(Mask))/p + n)$. The cost at the sending site is $O(\max_i(\text{nnz}(Array_i)) + \max_i(\text{nnz}(Mask_i)) + n)$. Tables 6 and 7 summarize the complexity results for the 2-d cases.

For higher-dimensional arrays, the complexity of the algorithm is the number of non-zero elements with the additional cost of traversing the intermediate nodes of the composed sparse structure. Similarly, Tables 8 and 9 give the complexity for sequential and parallel complexity for higher-dimensional arrays, respectively.

### *A.6.* `unpack`*(Vector, Mask, Field)*

The `unpack` function works as follows. It moves elements from the rank-one array *Vector* to form an array with the shape of *Field* under the control of *Mask*. The elements in *Field* will fill the corresponding positions in the target array if the corresponding elements in *Mask* are false.

In the sequential case, the complexity of this operation is proportional to the number of non-zero elements in the three arrays. That is $O(\text{nnz}(Vector) + \text{nnz}(Mask) + \text{nnz}(Field) + n)$ for the 2-d case, and $O(\text{nnz}(Vector) + \text{nnz}(Mask) + \text{nnz}(Field) + \phi(Vector) + \phi(Mask) + \phi(Field))$ for the higher-dimensional case.

Next, we consider the cost for the parallel implementation. We first consider the 2-d case. The communication set is $O(\text{nnz}(Vector) + \text{nnz}(Mask) + \text{nnz}(Field))$, the cost for packing the sending elements is $O(\max_i(\text{nnz}(Vector_i)) + \max_i(\text{nnz}(Mask_i)) + \max_i(\text{nnz}(Field_i)) + n)$, and the cost at the receiving site to receive elements and performing local computation is $O((\text{nnz}(Vector) + \text{nnz}(Mask) + \text{nnz}(Field))/p + n)$. Table 7 summarize the results for 2-d case. Higher-dimensional result is shown in Table 9.

### *A.7.* `sum`*(Array, Dim, Mask)*

We use the summation function, `sum`, as the representative reduction operations in Fortran 90. The analysis here can be applied to sparse implementations of other reduction operations. For sequential cases, the complexity is $O(\text{nnz}(Array) + \text{nnz}(Mask) + n)$ for the 2-d case, as all the non-zero elements of these two arrays might be accessed in the worst case. Similarly, the complexity is $O(\text{nnz}(Array) + \text{nnz}(Mask) + \phi(Array) + \phi(Mask))$ for higher-dimensional case.

Next, we consider the cost for the parallel version on distributed memory environments for 2-d arrays. There are two cases. In the case that the elements in the reduction dimension are not distributed among processors, there

will be no communication needed. The complexity is $O(\max_i(\text{nnz}(Array_i)) + \max_i(\text{nnz}(Mask_i)) + n)$. In the case that the elements in the reduction dimension are distributed over the processors, we need to consider the communication cost. The communication set is $O(\text{nnz}(Array) + \text{nnz}(Mask))$, the cost for packing step in sending site is $O(\max_i(\text{nnz}(Array_i)) + \max_i(\text{nnz}(Mask_i)) + n)$, and the cost at receiving site to receive elements and perform local computation is $O((\text{nnz}(Array) + \text{nnz}(Mask))/p + n)$. Tables 7 and 9 summarize the results for 2-d and higher-dimensional cases.

### *A.8.* `reshape`*(Source, Shape, Order)*

The detailed algorithm for the `reshape` function was described earlier in Section 3. In short, we reshape an source array of $n$ dimensions into a target array of $m$ dimensions by first mapping the array to a one-dimensional array, then moving the data into the target array with the specified shape. Let's first consider the sequential cases. The complexity for each of these two intermediate reshape operations is in proportion to the number of non-zero elements of the source arrays. The complexity is $O(\text{nnz}(Source) + n)$ for the 2-d case, and $O(\text{nnz}(Source) + \phi(Source))$ for the higher-dimensional case.

Next, we consider the cost for the parallel version. For the 2-d case, in the first intermediate reshape operation, the communication set is $O(\text{nnz}(Source))$, the cost for packing step in the sending site is $O(\max_i(\text{nnz}(Source_i)) + n)$, and the cost at the receiving site to receive elements and performing local computation is $O(\text{nnz}(Source)/p + n)$. In the second intermediate reshape operation, the communication set is $O(\text{nnz}(Source))$, the cost for packing step in the sending site is $O(\text{nnz}(Source)/p + n)$ because after the first reshape the elements are distributed evenly among the processors. The cost for the receiving site to receive elements and perform local computation is $O(\text{nnz}(S)/p + n)$. The total cost is the summation of that of the two intermediate reshape operations. Table 7 gives the summary result for the 2-d case, and Table 9 gives the summary result for the higher-dimensional case.

## Notes

1. A preliminary version of this work appeared in *Proceedings of the 1998 ACM International Conference on Supercomputing*, July 1998 [4].
2. This work is supported in part by National Science Council of Taiwan under grant nos. NSC 88-2213-E-001-007, NSC 89-2213-E-001-005, and NSC 89-2213-E-007-049. It is also supported by a MOE/NSC program for promoting academic excellence of universities under grant no. 89-E-FA0414.

## References

1. J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook*, Intertext Publications/McGraw–Hill Inc., 1992.

2. A. J. C. Bik and H. A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7:109–126, 1996.
3. R.-G. Chang, C.-W. Chen, T.-R. Chuang, and J. K. Lee. Towards automatic supports of parallel sparse computation in Java with continuous compilation. *Concurrency: Practice and Experience*, 9:1101–1111, 1997.
4. R.-G. Chang, T.-R. Chuang, and J. K. Lee. Efficient support of parallel sparse computation for array intrinsic functions of Fortran 90. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, pp. 45–52, 1998.
5. R.-G. Chang, T.-R. Chuang, and J. K. Lee. Compiler optimizations for parallel sparse programs with array intrinsics of Fortran 90. In *Proceedings of the 1999 International Conference on Parallel Processing*, Aizu–Wakamatsu City, Japan, pp. 103–110, 1999.
6. S. Chatterjee, J. Gilbert, R. Schreiber, and S. H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, pp. 16–28, 1993.
7. W.-M. Ching and A. Katz. An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of Supercomputing*, Washington, D.C., pp. 59–68, 1994.
8. T.-R. Chuang, R.-G. Chang, and J. K. Lee. Sampling and analytical techniques for data distribution of parallel sparse computation. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, 8 pp., 1997.
9. J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13:333—356, 1992.
10. S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. In *Proceedings of 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, Orlando, Fl., pp. 548–555, 1998.
11. G.-H. Hwang, J. K. Lee, and D. C. Ju. An array operation synthesis scheme to optimize Fortran 90 programs. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Santa Barbara, Calif., pp. 112–122, 1995.
12. G.-H. Hwang, J. K. Lee, and D. C. Ju. Array operation synthesis to optimize HPF programs. In *Proceedings of the 1996 International Conference on Parallel Processing*, Bloomingdale, Ill., Vol. 3, pp. 1–8, 1996.
13. G.-H. Hwang, J. K. Lee, and D. C. Ju. A function-composition approach to synthesize Fortran 90 array operations. *Journal of Parallel and Distributed Computing*, 54:1–47, 1998.
14. M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: a new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, Orlando, Fl., pp. 573–579, 1998.
15. K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.
16. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.
17. V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel sparse code for user-defined data structures. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, 8 pp. 1997.
18. J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.
19. J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines, *IEEE Transactions on Parallel and Distributed Systems*, 2:361–376, 1991.
20. T. M. Lin, Y.-C. Chuang, J. K. Lee, K. L. Wu, and C. A. Lin. Parallelisation of pressure correction method on unstructured grids, In *Parallel Computational Fluid Dynamics: Development and Applications of Parallel Technology*, pp. 451–458, 1998.
21. L. M. Ni, H. Xu, and E. T. Kalns. Issues in scalable library design for massively parallel computers. In *Proceedings of Supercomputing*, Portland, Ore., pp. 181–190, 1993.
22. OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*, Version 1.1, 1999.
23. M. Philippsen Automatic alignment of array data and process to reduce communication time on

DMPPs. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, Calif., pp. 112–122, 1995.
24. M. Philippsen and M. U. Mock. Data and process alignment in Modula-2*. In *Automatic Parallelization: New Approaches*, pp. 177–191. Verlag Vieweg, 1994.
25. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing*. Cambridge University Press, 1996.
26. L. D. Rose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, pp. 309–316, 1996.
27. Y. Saad. SPARSKIT: a basic tool kit for sparse computations (Version 2). Technical report, Computer Science Department, University of Minnesota, USA, 1994.
28. S. D. Stearns and R. A. David. *Signal Processing Algorithms Using Fortran and C*. Prentice-Hall, 1993.
29. M. Ujaldon, S. D. Sharma, J. Saltz, and E. L. Zapata. Run–time techniques for parallelizing sparse matrix problems. In *Parallel Algorithms for Irregularly Structured Problems: 2nd International Workshop*, Lyon, France, 1995. Lecture Notes in Computer Science, Vol. 980, pp. 43–57. Springer-Verlag, 1995.
30. M. Ujaldon and E. Zapata. Efficient resolution of sparse indirections in data-parallel compilers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, pp. 117–126, 1995.
31. M. Ujaldon, E. Zapata, B. M. Chapman, and H. P. Zima. New data–parallel language features for sparse matrix computations. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, Calif., pp. 742–749, 1995.