Implementation and Scalability of Fortran 90D Intrinsic Functions on Distributed Memory Machines *

Ishfaq Ahmad  Rajesh Bordawekar  Zeki Bozkus
Alok Choudhary  Geoffrey Fox  Kanchana Parasuram
Ravi Ponnusamy  Sanjay Ranka  Rajeev Thakur

Northeast Parallel Architectures Center
111 College Place
Syracuse, NY 13244-4100

Abstract

We are developing a Fortran 90D compiler, which converts Fortran 90D code into Fortran 77 plus message passing node programs for distributed memory machines. This paper presents the implementation and performance results of Fortran 90D intrinsic functions on Intel iPSC/860 and iPSC/2 hypercubes. An important feature of our implementation is that it is portable and scalable. That is, it can run without modification on a variety of machines such as Intel iPSC/2 and iPSC/860 hypercubes, Intel Touchstone Delta, NCUBE and a network of workstations. This is because the interprocessor communication required in these functions has been implemented using EXPRESS, a portable parallel programming environment developed by Parasoft Corporation.

*This work was sponsored by DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.
1 Introduction

It is widely recognized that massively parallel MIMD distributed memory machines can provide enormous computing power. But this power has not yet been fully harnessed because of the difficulty in programming these machines and the lack of portability between them. Hence it is necessary to have a machine-independent parallel programming model for distributed memory machines.

Fortran 90 is a popular parallel programming language for SIMD machines. With some extensions, it is useful for representing a class of problems called synchronous problems [5]. Fortran 90D, a version of Fortran 90 enhanced with a rich set of data decomposition specifications, is a language designed for this purpose. The data decomposition specifications indicate how arrays should be aligned with respect to one another, both within and across array dimensions, and also how arrays should be distributed among the processors of the parallel machine. These specifications in Fortran 90D are the same as in Fortran 77D [6, 8]. The main advantage of Fortran 90D is that it uses high level data structures explicitly (as arrays) and so the problem architecture is clear and not hidden in values of pointers and DO loop indices [4]. We are developing a Fortran 90D compiler, which converts Fortran 90D code into Fortran 77 plus message passing node programs for a distributed memory machine [11].

Since Fortran 90D is oriented towards scientific applications, many frequently required mathematical functions are provided as part of the language itself so that the user does not have to code them afresh. These are known as intrinsic functions. It is necessary to build a library of intrinsic functions which can be called from the node programs of a distributed memory machine. This paper discusses the implementation and scalability of several of these intrinsic functions. A more detailed discussion can be found in [1]. In order that the intrinsic functions are portable, we have implemented them using EXPRESS, a portable parallel programming environment developed by Parasoft Corporation. [9, 10].

The rest of the paper is organized as follows. Section 2 gives a brief description of EXPRESS. Fortran 90D intrinsic functions are described in Section 3. The implementation of array reduction functions is explained in Section 4. Array manipulation functions are explained in Section 5, array location functions in Section 6 and vector and matrix multiplication functions in Section 7. Finally
conclusions are presented in Section 8.

2 EXPRESS

EXPRESS provides routines for interprocessor communication as well as tools for debugging and performance analysis. A detailed discussion of the functionality of EXPRESS is given in [2]. The most important feature of EXPRESS is that it is portable. Programs written using EXPRESS can be run without any modifications on a number of machines such as NCUBE, Intel iPSC/2 and iPSC/860 hypercubes, Intel Touchstone Delta, transputer arrays, BBN Butterfly and also on networks of workstations. The languages supported by EXPRESS are FORTRAN and C.

One feature of EXPRESS which we have extensively used is a set of function calls collectively known as KXGRID. Their purpose is to take a user specification of a problem domain and perform a mapping to the underlying processor topology. EXPRESS provides a set of conversion routines between the physical processor numbers and their logical positions in the grid. This provides the user with a transparent view of the underlying architecture. Thus, the architecture of the parallel machine is effectively hidden from the user, which makes programming easier and portable. The user has to specify the dimensionality of the logical grid and the number of processors to be assigned to each dimension. EXPRESS then creates a virtual grid and maps the physical processor numbers into coordinates in the grid such that grid based communication can be performed efficiently. The user can then assume that the processors are configured as a grid and use EXPRESS routines to find the physical number of a processor from its location in the grid and vice-versa.

EXPRESS also provides routines for send and receive, vector send and receive, exchange, vector exchange, global reduction operations like sum, product, maximum, minimum etc. and global concatenation, which we have extensively used.

3 Fortran 90D Intrinsic Functions

Fortran 90D has several intrinsic functions. We have concentrated on implementing those functions which take arrays as arguments and are non-trivial to implement when the arrays are distributed among
### THE "INFO" ARRAY

<table>
<thead>
<tr>
<th>DIMENSION</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFO</td>
<td>lb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lbo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ubo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distribution Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block size for block-cyclic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nprocs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- lb - lower bound (of local array)
- ub - upper bound (of local array)
- lbo - lower bound with overlap
- ubo - upper bound with overlap

**Distribution Code:**
- 0 - not distributed
- 1 - block
- 2 - cyclic
- 3 - block-cyclic

**nprocs** - number of processors along each dimension of the global array

### LOCAL ARRAY

```
      lb   ub
      |     |
```

**Figure 1:** Array Specifications Passed to Intrinsic Functions
processors. These are the functions whose result depends on more than one element of the array and hence interprocessor communication is needed to implement them. The intrinsic functions that we have implemented fall into four main categories:

- **Array Reduction Functions**: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM.

- **Array Manipulation Functions**: CSHIFT, EOSHIFT, TRANSPOSE.

- **Array Location Functions**: MAXLOC, MINLOC.

- **Vector and Matrix Multiplication Functions**: DOT_PRODUCT, MATMUL.

For each of these functions, we have written Fortran 77 routines which can be called from the node programs of a distributed memory machine. The Fortran 90D compiler will detect calls to intrinsic functions in the Fortran 90D program and replace them with calls to these routines. When an array is passed as an argument to an intrinsic function, it is also necessary to provide some other information such as its size, distribution among the nodes of the distributed memory machine etc. All this information is stored in an array “INFO” and passed as another argument to the intrinsic function. The contents of the “INFO” array are shown in figure 1. Rows 1 and 2 contain the lower and upper bounds of the local array (excluding overlap area) in each dimension. The lower and upper bounds in each dimension including overlap area are stored in rows 3 and 4. The number of elements in each dimension of the array is given in row 5. Row 6 contains information regarding the distribution of the array. If the distribution is block-cyclic, it is also necessary to specify the block size. This is given in row 7. If none of the dimensions have block-cyclic distribution, this row can be ignored. Row 8 specifies the number of processors assigned to each dimension of the array.

We have written separate routines for one- and two-dimensional arrays and also in cases where some of the arguments are optional, because Fortran 77 does not support optional arguments. The compiler has to call the appropriate routine depending on the number of dimensions of the array and the optional arguments supplied.
4 Array Reduction Functions

4.1 MAXVAL and MINVAL

MAXVAL and MINVAL are used to find the maximum and minimum elements in an array. Their Fortran 90D specification is given in figure 2. MAXVAL has been implemented as follows. For a one-dimensional array, each processor calculates the maximum value in the local array and then all processors perform a global maximum operation to find the maximum element among all local arrays. In the case of two-dimensional arrays, if the DIM argument is not specified, the problem is essentially the same as that for a one-dimensional array. If DIM is specified, the result is a one-dimensional array. Each processor determines the maximum value along each row (or column) of the local array. A global maximum operation may or may not be necessary depending on whether the rows (or columns) are distributed or not. If the rows (or columns) of the array are not distributed, the maximum values determined by each processor are the maximum values along the rows (or columns). If the rows (or columns) are distributed, then those processors which share a particular row (or column) of the array, perform a global maximum operation to determine the maximum value along the row (or column). Thus, a separate global operation takes place in every row (or column) of the grid. A global operation can be performed in this case because the EXPRESS routine KXCOMB allows us to specify the list of processors participating in the global operation. It is not necessary that all processors have to participate in the global operation. If the MASK array is specified, then the maximum of only those elements which correspond to the true values of MASK, is calculated. MINVAL has been implemented in the same way as MAXVAL, except that instead of finding the maximum value, the minimum value is determined.

Graphs of speedup versus number of processors for different array sizes are given in figures 3 and 4. We see that the speedup is higher for larger array sizes. This is because as the number of processors increases, the computation time decreases almost linearly, but the communication time increases. For large array sizes, the computation time is much higher than the communication time and hence as the number of processors increases, the speedup increases.
• **Syntax:** MAXVAL(ARRAY, DIM, MASK), MINVAL(ARRAY, DIM, MASK), PRODUCT(ARRAY, DIM, MASK), SUM(ARRAY, DIM, MASK)

• **Optional Arguments:** DIM, MASK

• **Description (MAXVAL):** Determines the maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

• **Description (MINVAL):** Determines the minimum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

• **Description (PRODUCT):** Determines the product of all the elements along dimension DIM corresponding to the true elements of MASK.

• **Description (SUM):** Determines the sum of all the elements along dimension DIM corresponding to the true elements of MASK.

• **Arguments:**

1. ARRAY: must be of type integer or real. It must not be scalar.

2. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of ARRAY.

3. MASK (optional): must be of type logical and must be conformable with ARRAY.

The result is of the same type and type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise the result is an array of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Figure 2: FORTRAN 90D specification for MAXVAL, MINVAL, PRODUCT and SUM
Figure 3: MAXVAL on iPSC/860 for one-dimensional array

Figure 4: MAXVAL on iPSC/2 for one-dimensional array
4.2 ALL, ANY and COUNT

The Fortran 90D specifications for ALL, ANY and COUNT are given in figure 5. ALL has been implemented as follows. For a one-dimensional array, each processor performs an AND operation on the local array and generates a local result. Then all processors perform a global AND operation using the EXPRESS routine KXCOMB to find the global AND of the local variables; the result of the global operation is left in each participating processor. In the case of two-dimensional arrays, if the DIM argument is not specified, the problem is essentially the same as that for a one-dimensional array. If DIM is specified, the result is a one-dimensional array. Each processor performs the AND operation along each row (if reduction dimension is 2) or column (if reduction dimension is 1) of the local array. The result of this local computation is that each processor generates a vector of TRUE and FALSE values. Depending upon the reduction dimension and processor grid configuration, a global operation may be required to take a global AND across the vectors generated by each processor. The result of this global operation is itself a vector which is left in each participating processor. ANY and COUNT have been implemented in a similar manner.

4.3 SUM and PRODUCT

4.4 SUM

The Fortran 90D specifications for SUM and PRODUCT are given in figure 2. SUM is implemented in the same way as MAXVAL, except that each processor finds the sum of all elements in the local array and then all processors perform a global sum operation. PRODUCT is also implemented in the same way as MAXVAL, except that each processor finds the product of all elements in the local array and then all processors perform a global product operation.

5 Array Manipulation Functions

5.1 CSHIFT and EOSHIFT

The Fortran 90D specification for CSHIFT is given in figure 6. The algorithm for implementing
• **Syntax**: ALL(MASK, DIM), ANY(MASK, DIM), COUNT(MASK, DIM)

• **Optional Arguments**: DIM

• **Description (ALL)**: Determines whether all values are true in MASK along dimension DIM.

• **Description (ANY)**: Determines whether any value is true in MASK along dimension DIM.

• **Description (COUNT)**: Count the number of true elements of MASK along dimension DIM.

  1. MASK: must be of type logical and must be conformable with ARRAY.

  2. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where $n$ is the rank of ARRAY.

For ALL and ANY, the result is of type logical with the same kind of type parameters as MASK. For COUNT, the result type is integer. It is scalar if DIM is absent or ARRAY has rank one; otherwise the result is an array of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, \text{ } d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Figure 5: FORTRAN 90D Specification for ALL, ANY and COUNT
- **Syntax**: `CSHIFT(ARRAY, SHIFT, DIM)`

- **Optional Arguments**: MASK

- **Description**: Performs a circular shift on an array expression of rank one or performs a circular shift on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.

- **Arguments**:

  1. **ARRAY**: may be of any type. It must not be scalar.

  2. **SHIFT**: must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank \((d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)\) where \((d_1, d_2, ..., d_n)\) is the shape of the array.

  3. **DIM** (optional): must be scalar and of type integer with a value in the range \(1 \leq DIM \leq n\), where \(n\) is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

The result is of the same type and type parameter as ARRAY and has the shape of ARRAY.

*Figure 6: FORTRAN 90D Specification for CSHIFT*
• Syntax: EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

• Optional Arguments: BOUNDARY, MASK

• Description: Performs an end-off shift on an array expression of rank one or perform an end-off shift on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements are shifted out at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.

• Arguments:

1. ARRAY: may be of any type. It must not be scalar.

2. SHIFT: must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank \((d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)\) where \((d_1, d_2, ..., d_n)\) is the shape of the array.

3. DIM (optional): must be of the same type and type parameter as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either a scalar or of rank \(n - 1\) and of shape \((d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)\). BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar values shown.

<table>
<thead>
<tr>
<th>Type of ARRAY</th>
<th>Value of BOUNDARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Real</td>
<td>0.0</td>
</tr>
<tr>
<td>Complex</td>
<td>((0.0, 0.0))</td>
</tr>
<tr>
<td>Logical</td>
<td>false</td>
</tr>
<tr>
<td>Character (len)</td>
<td>len blanks</td>
</tr>
</tbody>
</table>

4. DIM (optional): must be scalar and of type integer with a value in the range \(1 \leq \text{DIM} \leq n\), where \(n\) is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

The result is of the same type and type parameter as ARRAY and has the shape of ARRAY.

Figure 7: FORTRAN 90D Specification for EOSHIFT
CSHIFT consists of two parts: one for sending messages and one for receiving messages. Within each part, there are separate parts for positive and negative shift factor. If the shift factor is positive, the array is shifted to the right and elements coming off the edge of the rightmost processor are shifted to the beginning of the array in the leftmost processor on the grid. On the other hand, if the shift factor is negative, the array is shifted to the left and elements coming off the edge of the leftmost processor are shifted to the end of the array in the rightmost processor on the grid. The shift factor can be greater than the size of the local array. Each processor needs to communicate with at most two processors. Messages are packed and local arrays are shifted to make room for the incoming messages. A temporary array is used to save the elements coming off the edges. This array needs to be used for performing a circular shift if the number of processors is 1. For 2 dimensional CSHIFT, the shift can be performed along dimension 1 or dimension 2. The shift factor is specified by a shift vector whose elements can have different values. Furthermore, the values of that vector can be positive or negative. A combination of positive and negative values is allowed as long as the same dimension is used. For 2 dimensional CSHIFT, two algorithms have been written, one for each dimension. Since these algorithms assume that the value of shift in each element of the shift vector is less than the size of the local array, the algorithms for 2d cshift cannot directly be called if the amount of shift is greater than the local array size. In that case, a front end routine has been written which invokes these algorithms a number of times if required.

The Fortran 90D specification for EOSHIFT is given in figure 6. EOSHIFT has been implemented in a similar manner as CSHIFT.

5.2 Transpose

TRANSPOSE performs the transpose of a matrix. The Fortran 90D specification for Transpose is given in figure 8. We have implemented TRANSPOSE assuming that the matrix is distributed as (BLOCK,BLOCK) and the number of processors along the rows is a multiple of the number of processors along the columns.

The algorithm used for TRANSPOSE is given in figure 9. Let A be the local array corresponding
• **Syntax:** TRANSPOSE(MATRIX)

• **Description:** Transpose of an array of rank two.

• **Argument:** MATRIX may be of any type but must have rank two.

The result is an array of the same type and type parameters as MATRIX and with rank two and shape \((n, m)\) where \((m, n)\) is the shape of MATRIX.

Figure 8: FORTRAN 90D SPECIFICATION FOR TRANSPOSE

to the matrix whose transpose is to be determined and let \(A\)\_trans be the local array corresponding to the transpose of the matrix. A *virtual grid* is defined so that the number of *virtual processors* along both dimensions of the grid is the same. The virtual dimensions of the matrix \(A\) belonging to this *virtual node* are defined on the basis of this virtual grid. The transpose of the resultant matrix for each virtual processor is called \(A\)\_trans\_virtual and it is used to calculate \(A\)\_trans as follows:

Each node calculates, in sequence, \(A\)\_trans\_virtual for each virtual sub-block of the matrix \(A\). The coordinates of the *destination virtual processor* are then determined, from which we find the *real coordinates* of the destination node. This is used to find the actual processor number of the destination. For example, the first block in node \(p\) may have virtual coordinates \((i, j)\). This implies that the \(A\)\_trans\_virtual calculated for this block has to be transmitted to virtual processor \((j, i)\), which may be located in node \(q\). So node \(p\) does a non-blocking write that sends \(A\)\_trans\_virtual to node \(q\) with virtual\_dest\_coord(2) being used as a unique type identifier that can be recognized at the receiving end. All \(A\)\_trans\_virtuals are sent off sequentially in a loop.

Another loop then receives all the \(A\)\_trans\_virtual blocks that were sent by other nodes and puts each block at the appropriate place in the transpose matrix \(A\)\_trans. This overlapping of computation and communication was found to give the best performance.

The graphs of speedup versus number of processors for different array sizes are given in figures 10 and 11. The graphs of speedup versus size of matrix for different grid configurations are given in
figures 12 and 13. We see that the speedup is less than 10 for 32 processors, because the transpose operation is basically a communication operation and it is difficult to get any significant speedup.

6 Array Location Functions

6.1 MAXLOC and MINLOC

Figure 14 gives the Fortran 90D specification for MAXLOC and MINLOC. The implementation of MAXLOC is similar to that of MAXVAL, except that certain additional steps need to be taken to ensure that if there is more than one element with the maximum value, then the location of the first element of the array (in column major order) having the maximum value, is returned.

Each processor determines the first element of the local array having the maximum value among the elements in the local array. On the basis of the array decomposition information, it calculates the global coordinates of this element to determine the location of the element in the global array. All processors then perform a global operation to determine the maximum element and if it is not unique, the one which appears first in the global array. The address returned by MAXLOC is the global address of the maximum element. The time taken for MAXLOC will be is more than for MAXVAL because of the additional local to global index conversion. MINLOC is implemented similar to MAXLOC.

7 Vector and Matrix Multiplication Functions

7.1 DOT_PRODUCT

The Fortran 90D specification for DOT_PRODUCT is given in figure 15. We have implemented this assuming that both the vectors have identical distributions. The problem then reduces to finding the dot product of the local vectors followed by a global sum operation. If the two vectors have different distributions, we first convert one of the vectors into the same distribution as the other and then perform the dot product. Graphs of speedup versus number of processors for different vector sizes are given in figure 16. We see that the speedup is higher for larger vector sizes because the computation time is much higher than the communication time.
Begin

Define a VIRTUAL GRID so that the number of virtual processors
along both dimensions of the array is the same;

Calculate size of matrix A belonging to a virtual node;

/\* Send off all virtual blocks */

fac_v = \# of processors along dim 1/ \# of processors along dim 2

For i = 0 to (fac_v - 1)

Begin

Calculate A_trans_virtual, the transpose of the i-th virtual sub-block;
Calculate virtual address, virtual_coord(2), of this block;
Calculate virtual address, virtual_dest_coord(2), of the destination of
this block;
Calculate real coordinates of the destination, and from that, the
processor number of the destination;
Send this virtual block to the destination (real) processor;
End

/\* Receive all virtual blocks */

For i = 0 to (fac_v - 1)

Begin

Calculate virtual address, virtual_coord(2), of this block;
Calculate virtual address, virtual_source_coord(2), of the source from
which to receive this block;
Calculate real coordinates of the source, and from that, the
processor number of the source;
Receive the i-th virtual block from the source node and assign it to
the appropriate slot in A_trans;
End

return(A_trans);

End

Figure 9: Algorithm for TRANSPOSE
Figure 10: TRANSPOSE on iPSC/860

Figure 11: TRANSPOSE on iPSC/2
Figure 12: TRANSPOSE on iPSC/860, for different grid sizes

Figure 13: TRANSPOSE on iPSC/2, for different grid sizes
- **Syntax**: MAXLOC(ARRAY, MASK), MINLOC(ARRAY, MASK)

- **Optional Arguments**: MASK

- **Description (MAXLOC)**: Determines the location of the first element of ARRAY having the maximum value of the elements identified by MASK.

- **Description (MINLOC)**: Determines the location of the first element of ARRAY having the minimum value of the elements identified by MASK.

- **Arguments**:

  1. ARRAY: must be of type integer or real. It must not be scalar.

  2. MASK (optional): must be of type logical and must be conformable with ARRAY.

The result is of type integer. It is an array of rank one and of size equal to the rank of ARRAY. If there is more than one element with maximum (or minimum) value, it returns the location of the first element of the array in column major order, which has the maximum (or minimum) value.

**Figure 14: FORTRAN 90D Specification for MAXLOC and MINLOC**

---

- **Syntax**: DOT_PRODUCT(VECTOR_A, VECTOR_B)

- **Description**: Performs dot product of numeric or logical vectors.

- **Arguments**:

  1. VECTOR_A: must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.

  2. VECTOR_B: must be of the same type and size as VECTOR_A.

The result is scalar.

**Figure 15: FORTRAN 90D SPECIFICATION FOR DOT_PRODUCT**

18
7.2 MATMUL

The Fortran 90D specification for MATMUL is given in figure 17. We have implemented the communication efficient parallel matrix multiplication algorithms proposed by Fox et al [7] and Berntsen [3]. The former method requires a particular multiplicand sub-matrix to be broadcast to all the processors which are in the same row of the processor configuration, followed by multiplication and neighbor communication along the columns. In the latter method, initially matrices are redistributed such that only neighbor communication is necessary in subsequent steps. Theoretically, this algorithm reduces the asymptotic communication cost of $2(N^2/P^{1/2})\beta$ of the first algorithm to $3(N^2/P^{2/3})\beta$, where $N$ is the matrix size, $P$ the number of processors and $\beta$ the communication cost per word. The two algorithms are given in figures 18 and 19.

If the processor configuration is not a perfect square, but one of the dimensions is a multiple of the other, the sub-matrices will not be square. In this case, virtual processors are created to get square sub-matrices. FORTRAN follows column major order to store arrays. The matrix multiplication kernel is a modified version of the conventional kernel to suit the column major ordering. The matrices are
• **Syntax:** MATMUL(MATRIX_A, MATRIX_B)

• **Description:** Performs matrix multiplication of numeric or logical matrices.

• **Arguments:**

1. MATRIX_A: must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.

2. MATRIX_B: must be of the same type as MATRIX_A and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

   If MATRIX_A has shape \((n, m)\) and MATRIX_B has shape \((m, k)\), the result has shape \((n, k)\). If MATRIX_A has shape \((m)\) and MATRIX_B has shape \((m, k)\), the result has shape \((k)\). If MATRIX_A has shape \((n, m)\) and MATRIX_B has shape \((m)\), the result has shape \((n)\).

   **Figure 17:** FORTRAN 90D SPECIFICATION FOR MATMUL
C perform matrix multiplication $C = AB$
C $A$ and $B$ are of size $N \times N$
C $T$ - a temporary matrix
  subroutine MATMUL($\hat{A}, \hat{B}, \hat{C}, \hat{T}$)
C initialize matrix $\hat{C}$
  $\hat{C} \leftarrow 0$
  do $i = 1, \sqrt{N} - 1$
C broadcast matrix $A$ along rows
  call bcast($\hat{A}, \hat{T}$)
  $\hat{C} \leftarrow \hat{C} + \hat{T}\hat{B}$
C roll matrix $\hat{B}$ upwards
  call roll($\hat{B}$)
  end do
end

Figure 18: Matrix multiplication Algorithm I

of double precision floating point numbers and the times in seconds. The current implementation assumes that both matrices are block decomposed and the number of processors allocated along a dimension of the array must be a multiple of the other. The graphs of speedup versus number of processors are given in figures 20 and 21. The speedup increases almost linearly with the number of processors. In some cases the speedup is superlinear because of cache misses in the single processor case. The communication cost is marginally less for the second algorithm for some of the processor configurations. The communication time varies for different configuration of processors for the same total number of processors because the buffering requirement varies for sending and receiving messages.
C perform matrix multiplication C = AB
C A and B are of size N x N
    subroutine MATMUL(\(\hat{A}\), \(\hat{B}\), \(\hat{C}\), \(\hat{T}\))
C initialize matrix \(\hat{C}\)
    \(\hat{C} \leftarrow 0\)
C rearrange matrix \(\hat{A}\) left
    call rearrangeA(\(\hat{A}\))
C rearrange matrix \(\hat{B}\) upwards
    call rearrangeB(\(\hat{B}\))
    do i = 1, \(\sqrt{N} - 1\)
C broadcast matrix A along rows
    \(\hat{C} \leftarrow \hat{C} + \hat{A}\hat{B}\)
C roll matrix \(\hat{A}\) left
    call roll_left(\(\hat{A}\))
C roll matrix \(\hat{B}\) upwards
    call roll_up(\(\hat{B}\))
end do
end

Figure 19: Matrix multiplication Algorithm II
Figure 20: MATMUL on iPSC/860

Figure 21: MATMUL on iPSC/2
8 Conclusion

We have implemented several Fortran 90D intrinsic functions so that they can be called from the node programs of a distributed memory machine. We have implemented them on Intel iPSC/2 and iPSC/860 hypercubes and the results indicate that our implementations are scalable. They are also portable and can be run on a variety of other machines such as Intel Touchstone Delta, NCUBE and network of workstations. We have developed routines for one and two dimensional arrays and we are in the process of extending them for upto seven dimensional arrays.

References


