Implementation of Conjugate Gradient Algorithms in Fortran 90 and HPF and Possible extensions towards HPF-2

K.Dincer, K.A.Hawick, A.Choudhary, G.C.Fox.
Northeast Parallel Architectures Center
Syracuse University
111 College Place, Syracuse, NY 13244-4100
{dincer, hawick, choudhar, gcf}@npac.syr.edu *

October 1994
revised: March 1995

Abstract

We evaluate the High-Performance Fortran (HPF) language for the compact expression and efficient implementation of conjugate gradient iterative matrix-solvers on High Performance Computing and Communications(HPCC) platforms. We discuss the use of intrinsic functions, data distribution directives and explicitly parallel constructs to optimize performance by minimizing communications requirements in a portable manner. We focus on implementations using the existing Fortran 90 and HPF definitions but also discuss issues arising that may influence a revised definition for HPF-2. Some of the codes discussed are available on the World Wide Web at **http://www.npac.syr.edu/hpfa/** alongwith other educational and discussion material related to applications in HPF.

1 Introduction

High Performance Fortran (HPF)[12] is a language definition agreed upon in 1993, and being widely adopted by systems suppliers as a mechanism for users to exploit parallel computation through the data-parallel programming model.

HPF evolved from the experimental Fortran-D system [10] as a collection of extensions to the Fortran 90 language standard [17]. We do not discuss the details of the HPF language here as they are well documented elsewhere [15], but simply note that the central tenet of HPF and data-parallel programming is that program data is distributed amongst the processors' memories in such a way that the "owner computes" rule allows the maximum computation to communications ratio. Language constructs and embedded compiler directives allow the programmer to express to the compiler additional information about how to produce code that maps well to the available parallel or distributed architecture and thus runs fast and can make full use of the larger (distributed) memory.

An excellent review of iterative solvers and some of the general computational issues for their efficient implementation is given in [2]. We focus here on specific implementation issues for the Fortran 90 and the HPF languages.

Consider applications problems that can be formulated in terms of the matrix equation $A\vec{x} = \vec{b}$. The structure of the matrix A is highly dependent on the particular type of application and some applications such as computational electromagnetics give rise to a matrix that is effectively dense [9] and can be solved using direct methods [8] such as Gaussian elimination, whereas others such as computational fluid dynamics [4] generate a matrix that is sparse, having most of its elements identically zero. Conjugate Gradient (CG) and other iterative methods are preferred over simple Gaussian elimination when A is very large and sparse, and where storage space for the full matrix would either be impractical or too slow to access through a secondary memory system. A large number of computationally expensive scientific and engineering applications, e.g. structural analysis, fluid dynamics, aerodynamics, lattice gauge simulation, and circuit simula-

*This work sponsored in part by ARPA

tion, are based on the solution of large sparse systems of linear equations. Iterative methods are employed in many of these applications. While the CG method itself is no longer considered a state-of-the art is terms of its numerical stability and convergence properties, its computational structure is similar to that of methods such as Bi-Conjugate Gradient (BiCG). CG codes have been used in a number of benchmark suites such as PARKBENCH [11] and NAS [1].

We focus on the CG and BiCG methods and it is our intent in this paper to show how HPF makes it simpler to write **portable**, **efficient** and **maintainable** implementations of this class of iterative matrix-solvers.

2 Conjugate Gradient Algorithms

The classic Conjugate Gradient non-stationary iterative algorithm as defined in [7] and references therein can be applied to solve symmetric positive-definite matrix equations. They are preferred over simple Gaussian algorithms because of their faster convergence rate if A is very large and sparse.

Consider the prototype problem $A\vec{x} = \vec{b}$ to be solved for $\vec{x}$ which can be expressed in the form of iterative equations for the solution $\vec{x}$ and residual(gradient) $\vec{r}$:

$$\vec{x}^k = \vec{x}^{k-1} + \alpha^k \vec{p}^k \tag{1}$$
$$\vec{r}^k = \vec{r}^{k-1} - \alpha^k \vec{q}^k \tag{2}$$

where the new value of $\vec{x}$ is a function of its old value, the scalar step size α and the search direction vector $\vec{p}^k$ at the k'th iteration and $\vec{q}^k = A\vec{p}^k$.

The values of $\vec{x}$ are guaranteed to converge in, at most, n iterations, where n is the order of the system, unless the problem is ill-conditioned in which case roundoff errors often prevent the algorithm from furnishing a sufficiently precise solution at the nth step. In well-conditioned problems, the number of iterations necessary for satisfactory convergence of the conjugate gradient method can be much less than the order of the system. Therefore, the iterative procedure is continued until the residual $\vec{r}^k = \vec{b}^k - A\vec{x}^k$ meets some stopping criterion, typically of the form: $\| \vec{r}^k \| \leq \text{tol} \cdot (\| A \| \cdot \| \vec{x}^k \| + \| \vec{b}^k \|)$, where $\| A \|$ denotes some *norm* of A and *tol* is a tolerance level. The CG algorithm uses:

$$\alpha = \left(\vec{r}^k \cdot \vec{r}^k\right) / \left(\vec{p}^k \cdot A\vec{p}^k\right) \tag{3}$$

with the search directions chosen using:

$$\vec{p}^k = \vec{r}^{k-1} + \beta^{k-1}\vec{p}^{k-1} \tag{4}$$

with

$$\beta^{k-1} = (\vec{r}^{k-1} \cdot \vec{r}^{k-1})/(\vec{p}^{k-2} \cdot A\vec{p}^{k-2}) \tag{5}$$

which ensures that the search directions form an A-orthogonal system.

2.1 Computational Structure

The non-preconditioned CG algorithm is summarised as:

$\vec{p} = \vec{r} = \vec{b};\ \vec{x} = 0;\ \vec{q} = A\vec{p}$
$\rho = \vec{r} \cdot \vec{r};\ \alpha = \rho/(\vec{p} \cdot \vec{q})$
$\vec{x} = \vec{x} + \alpha\vec{p};\ \vec{r} = \vec{r} - \alpha\vec{q}$
DO $k = 2$, Niter
  $\rho_0 = \rho;\ \rho = \vec{r} \cdot \vec{r};\ \beta = \rho/\rho_0$
  $\vec{p} = \vec{r} + \beta\vec{p};\ \vec{q} = A \cdot \vec{p}$
  $\alpha = \rho/\vec{p} \cdot \vec{q}$
  $\vec{x} = \vec{x} + \alpha\vec{p};\ \vec{r} = \vec{r} - \alpha\vec{q}$
  IF(stop_criterion)exit
ENDDO

---

for the initial "guessed" solution vector $\vec{x}^0 = 0$.

Implementation of this algorithm requires storage for four vectors:, $\vec{x}$, $\vec{r}$, $\vec{p}$ and $\vec{q}$ as well as the matrix A and working scalars α and β.

Notice that the work per iteration is modest, amounting to a single matrix-vector multiplication for $A \cdot \vec{p}$, two inner products $\vec{p}^k \cdot \vec{p}^k$ and $\vec{r}^k \cdot \vec{r}^k$, and several simple $\alpha\vec{x} + \vec{y}$ (SAXPY) operations, where α is scalar, and $\vec{x}$ and $\vec{y}$ are vectors. The number of multiplications and additions required for matrix-vector multiplication, inner products and SAXPY operations are $\mathcal{O}(n^2)$, $\mathcal{O}(n)$, and $\mathcal{O}(n)$, respectively, for vector length n.

2.2 Other CG Algorithms

The **Bi-Conjugate Gradient** (BiCG) method can be applied to non-symmetric matrices, for which the residual vectors employed by CG cannot be made orthogonal with short recurrences. More complex algorithms such as GMRES make use of longer recurrences (which require greater storage). The BiCG [2] algorithm employs an alternative approach of using **two** mutually orthogonal sequences of residuals. This requires three extra vectors to be stored, and different choices of α and β, but otherwise the computational structure of the algorithm is similar to CG. It can be implemented using the

same BLAS-level [7]operations as CG. BiCG does however require **two** matrix-vector multiply operations one of which uses the matrix transpose A^T, and therefore any storage distribution optimisations made on the basis of row access vs. column access will be negated with the use of BiCG.

The **Conjugate Gradient Squared** (CGS) algorithm avoids using A^T operations but also requires additional vectors of storage over the basic CG. CGS can be built using the operations and data distributions we describe here, but can have some undesirable numerical properties such as actual divergence or irregular rates of convergence and so is not discussed further here.

The **Stabilized BiCG** (BiCGSTAB) algorithm also uses two matrix vector operations but avoids using A^T and therefore can be optimized using the data distribution ideas we discuss here. It does however involve **four** inner products, so will have a greater demand for an efficient intrinsic for this than basic CG.

2.3 Preconditioners for Conjugate Gradient

The CG algorithm will generally converge to the solution of the system $A.\vec{x} = \vec{b}$ in at most n_e iterations, where n_e is the number of distinct eigenvalues of the coefficient matrix A. Thus in cases where A has many distinct eigenvalues and those eigenvalues vary widely in magnitude, the CG algorithm may require a large number of iterations before converging. A preconditioner S for A can be added to any of the algorithms described above and which will increase the speed of convergence of the CG algorithm. A nonsingular matrix S is chosen such that $A' = S.A.S^T$ has fewer distinct eigenvalues than A. The CG algorithm is then used to solve $A'.x' = b'$, where $x' = (S^T)^{-1}.x$ and $b' = S.b$. This is described in detail in [2].

The preconditioning may cause efficiency tradeoffs if the preconditioner matrix requires a different data distribution pattern to the main iterative solver. However, this overhead is compensated by a reduction in the number of iterations required to achieve acceptable performance and therefore in the total wall clock time for problem completion.

There are certain problems with applying the CG algorithm directly to the system $A'.x' = b'$. Unless S is a diagonal matrix, the sparsity pattern of A is not preserved in A'. Moreover, the matrix multiplications involved in computing A' can be expensive. In practical implementations of the **preconditioned conjugate gradient** (PCG) algorithm, it is formulated such that it works with the original matrix A but maintains the same convergence rate as that for the system $A'.x' = b'$.

The following PCG algorithm is one representative example of those optimised algorithms that avoid computing A' and works with A by using a preconditioner matrix $T = (S^T S)^{-1}$:

$\vec{r} = \vec{b}$; $\vec{x} = 0$
Solve the system $T \cdot \vec{z} = \vec{r}$
$\gamma = \vec{r} \cdot \vec{z}$
$\vec{p} = \vec{z}$
$\vec{q} = A\vec{p}$
$\rho = \vec{r} \cdot \vec{r}$; $\alpha = \gamma/(\vec{p} \cdot \vec{q})$
$\vec{x} = \vec{x} + \alpha\vec{p}$; $\vec{r} = \vec{r} - \alpha\vec{q}$
DO $k = 2$, Niter
    Solve the system $T \cdot \vec{z} = \vec{r}$
    $\gamma = \vec{r} \cdot \vec{z}$
    $\gamma_0 = \gamma$; $\vec{p} = \vec{z} + \gamma \cdot \vec{p}/\gamma_0$
    $\rho_0 = \rho$; $\rho = \vec{r} \cdot \vec{r}$
    $\vec{q} = A \cdot \vec{p}$
    $\alpha = \gamma/\vec{p} \cdot \vec{q}$
    $\vec{x} = \vec{x} + \alpha\vec{p}$; $\vec{r} = \vec{r} - \alpha\vec{q}$
    IF(stop_criterion)exit
ENDDO

The preconditioner T must be carefully chosen to keep the overhead of solving $T \cdot \vec{z} = \vec{r}$ in each iteration inexpensive compared to solving the original system of equations. We describe below two parallelizable preconditioning methods which do not involve a significant computation and communication overhead:

Diagonal preconditioning: The preconditioner matrix T is a diagonal matrix with nonzero elements only on the principal diagonal. This T can be easily derived from the principal diagonal of $A(n,n)$. If we `ALIGN` $T(i,i)$ with $A(i,i)$ in the processors' memories then the elements with identical indices reside on the same processor and solving the system $T \cdot \vec{z} = \vec{r}$ is equivalent to dividing each element of r by the corresponding diagonal element of T, and this operation does not require any communication. The iteration can be performed in $\mathcal{O}(n/P)$ time on a P processor system.

Incomplete Cholesky(IC) preconditioning: T is based on an IC factorization of A that factorizes it as the product of two triangular matrices (i.e. $L \cdot L^T$). The locations of the nonzero elements in L^T and locations of nonzero elements in L correspond with the nonzero elements in the upper and lower triangular portions of A, respectively. $T = L \cdot L^T$ is used as the preconditioner, and the matrix vector system in the algorithm is solved in two steps:

1. $L \cdot \vec{u} = \vec{r}$

2. $L^T \cdot \vec{z} = \vec{u}$

Solution of these triangular systems of equations take $\mathcal{O}(\sqrt{n})$ time regardless of the number of processors used. The other operations in an iteration of the IC preconditioned CG algorithm take the same amount of time as in the diagonal preconditioned CG algorithm.

3 A Fortran 90 Implementation

The non-preconditioned CG algorithm for a **dense** system can be expressed using Fortran 90 intrinsic functions as shown in figure 1, where, for illustrative purposes, we have provided the full array-section notation for each vector or matrix reference even though these are not necessary when the entities have been declared of exactly dimension n.

```
REAL, dimension(1:n) :: x, b, p, q, r
REAL :: alpha, rho, rho0
REAL, dimension(1:n,1:n) :: A

x(1:n) = 0.0                ! An initial guess
r(1:n) = b(1:n) - MATMUL( A(1:n,1:n), x(1:n) )
p(1:n) = r(1:n)
rho = DOT_PRODUCT( r(1:n), r(1:n) )
q(1:n) = MATMUL( A(1:n,1:n), p(1:n) )
alpha = rho / DOT_PRODUCT( p(1:n) * q(1:n) )
x(1:n) = x(1:n) + alpha * p(1:n)    !saxpy
r(1:n) = r(1:n) - alpha * q(1:n)    !saxpy
DO k = 2, Niter
  rho0 = rho
  rho = DOT_PRODUCT( r(1:n), r(1:n) )
  p(1:n) = r(1:n) + ( rho/rho0 ) * p(1:n)
  q(1:n) = MATMUL( A(1:n,1:n), p(1:n) )
  alpha = rho / DOT_PRODUCT( p(1:n) * q(1:n) )
  x(1:n) = x(1:n) + alpha * p(1:n)  !saxpy
  r(1:n) = r(1:n) - alpha * q(1:n)  !saxpy
  IF ( stop_criterion ) EXIT
END DO
```

Figure 1: *CG Fortran 90 version of dense storage CG.*

Note, this is highly artificial, since CG finds its main use for **sparse** systems, which would not be stored using full matrices and vectors as indicated. We simply use this code to express the full algorithm, and note that the Fortran 90 intrinsic `DOT_PRODUCT` and array-section notation allows compact expression of `SAXPY` and `SDOT` [7] operations. An efficient compilation system would insert system-optimised run-time library routines for these statements.

4 Sparse Matrix Representations

It is efficient in storage to represent an $n \times n$ dense matrix as an $n \times n$ Fortran array. However, if the matrix is sparse, a majority of the matrix elements are zero and they need not be stored explicitly. Furthermore, for some very large application problems it would be simply impractical to store the matrix as a dense array either because of the prohibitive cost of enough primary memory, or because of the slow access speed of a secondary storage medium. It is therefore customary to store only the nonzero entries and to keep track of their locations in the matrix. Special storage schemes not only save storage but also yield computational savings. Since the locations of the nonzero elements in the matrix are known explicitly, unnecessary multiplications and additions with zero are avoided. A number of sparse storage schemes are described in [2], some of which can exploit additional information about the sparsity structure of the matrix. We only consider here the compressed row and compressed column schemes which can store **any** sparse matrix.

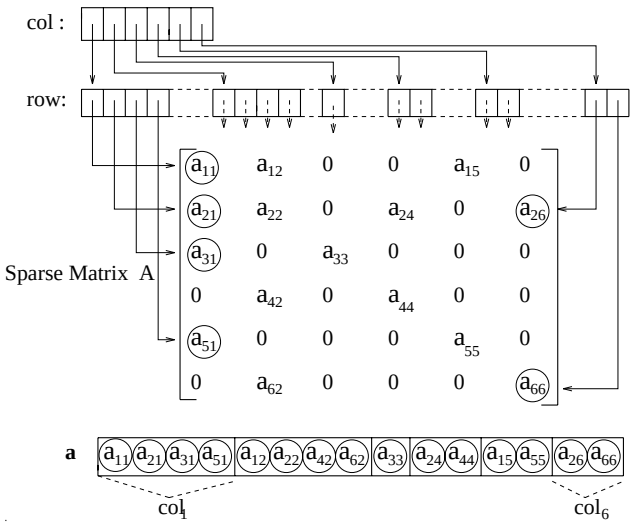

Figure 2: *Compressed Sparse Column(CSC) representation of sparse matrix A.*

The **Compressed Sparse Column** (CSC) storage scheme, shown in figure 2, uses the following three arrays to store an $n \times n$ sparse matrix with nz non-zero entries:

- `a(nz)` containing the nonzero elements stored in the order of their columns from 1 to n.
- `row(nz)` that stores the row numbers of each nonzero element.

- `col(n+1)` whose jth entry points to the first entry of the j'th column in `A` and `row`.

A related scheme is the **Compressed Sparse Row** (CSR) format, in which the roles of rows and columns are reversed.

The serial Fortran 77 code fragment in figure 3 illustrates how BLAS level library routines such as SAXPY and SDOT can be employed for a sparsely stored system.

```
    INTEGER row(nz), col(n+1)
    REAL a(nz), x(n), b(n), r(n), p(n), q(n)
    REAL SDOT

    DO i = 1, n
      x(i) = 0.0
      r(i) = b(i)
      p(i) = b(i)
    END DO
    rho = SDOT(n, r, r)
    CALL SAYPX(p, r, beta, n)
    CALL MATVEC(n, a, row, col, p, q, nz)
    alpha = rho / SDOT(n, p, q)
    CALL SAXPY(x, p, alpha, n)
    CALL SAXPY(r, q, -alpha, n)

    DO n = 2, Niter
      rho0 = rho
      rho = SDOT(n, r, r)
      BETA = RHO / RHO0
      CALL SAYPX(p, r, beta, n)
      CALL MATVEC(n, a, row, col, p, q, nz)
      ALPHA = RHO / SDOT(n, p, q)
      CALL SAXPY(x, p, alpha, n)
      CALL SAXPY(r, q, -alpha, n)
      IF( stop_criterion ) GOTO 300
    END DO
300 CONTINUE
```

Figure 3: *Fortran 77 version of sparse storage CG (CSC format).*

Each iteration of the CG algorithm in figure 3 performs three main computations: the vector-vector operations, inner product (here shown using BLAS routines) and the matrix-vector multiplication, shown explicitly in figure 4.

5 HPF Implementation

The data-parallel programming model upon which HPF is based requires some well-defined mapping of the data onto processors' memory to achieve a good computational load balance and thus an efficient use of the parallel architecture. This is not trivial for the sparse storage

```
      q = 0.0
      DO j = 1, n
         pj = p(j)
         DO 10 k = col(j) , col(j+1) - 1
            q( row(k) ) = q( row(k) ) + a(k) * pj
         END DO
      END DO
```

Figure 4: *Sparse matrix-vector multiply in Fortran 77 (CSC format).*

schemes that we will elaborate upon later.

In this section we assume that the vectors are represented as n-element one-dimensional arrays, and the arbitrarily sparse matrix A is either represented as an n by n two-dimensional matrix when a dense storage format is used, and as a (row, col, a) trio when a sparse storage format is employed.

In any parallel implementation that distributes the vectors and matrix A across processors' memories, the inner-products (SDOTs) and sparse matrix vector multiplication require data communication. However, the data distributions can be arranged so that all of the other operations will be performed only on **local** data. For each operation type we will show the optimum data distribution patterns for obtaining best performance, and how the operation can be represented in HPF.

Vector-vector operations

If all vectors are distributed identically among the processors, vector-vector operations such as SAXPY require no data communication since the vector elements with the same indices are involved in a given arithmetic operation and thus are locally available on each processor. Similarly, the element-wise multiplications in the inner-product operations can be performed locally without any communication overhead. However, the inner-product involves a communication phase to add up the partial local results from each processor.

Similarly, the vectors used in vector operations are aligned and distributed in HPF as follows in order to minimize the communication requirements.

```
!HPF$ ALIGN (:) WITH p(:) :: q, r, x
!HPF$ DISTRIBUTE p(BLOCK)
```

Vector p is chosen as the target of the ultimate alignment thus the distribution of p determines the distribution of all other vectors aligned with it. Whenever its distribution is changed, the others are also automatically redistributed.

Using N_P processors, SAXPY operations can be performed in $\mathcal{O}(n/N_P)$ time on any architecture. On the other hand, the inner products take $\mathcal{O}(n/N_P)$ time for the local phase, but the communication or merge phase changes according to the network architecture type. For example on a hypercube architecture it is done in $t_{start-up} \cdot logN_P$ time, where $t_{start-up}$ is the start-up time. If the reduction intrinsic functions are well supported by hardware reduction operations then the communication time for the inner-product calculations does not dominate.

HPF readily supports the inner product operations by an intrinsic function, called `DOT_PRODUCT()`. SAXPY operations are easily performed using HPF's parallel array assignments.

Matrix-vector multiplication

We consider the multiplication of an $n \times n$ arbitrarily sparse matrix A with an $n \times 1$ vector p that gives another $n \times 1$ vector q. As in the dense matrix vector multiplication, each row of matrix A must be multiplied with the vector p. The computation and data communication costs vary depending on the distribution of the matrix A and vectors p and q. We will keep the distributions of vectors as defined above, and concentrate on the two different partitioning scenarios for the sparse matrix A and their associated costs. Then, we will generalize the results drawn from these scenarios to the cases where a sparse storage format for matrix A is used.

Scenario 1: Row-wise partitioning

In the first scenario, we would like to partition the rows of the sparse matrix A among the processors in an even manner. We can do this by aligning the first dimension of A with p. When the p vector is distributed, A's first dimension will be distributed in an aligned manner (Figure 5.)

```
!HPF$ ALIGN A(:, *) WITH p(:)
```

Since the nonzero elements are at random positions in A, a row can have a nonzero entry in **any** column. This requires the entire vector p to be accessible by each row so that any of its nonzero entries can be multiplied with the corresponding element of the vector. As the vector p is partitioned among the processors, this would require an all-to-all broadcast of the local vector elements. This all-to-all broadcast of messages containing n/N_P vector elements among N_P processors, takes $t_{start-up} \cdot logN_P + t_{comm} \cdot n/N_P$ time if a tree-like broadcasting mechanism is used. Here $t_{start-up}$ is the start-up time, and t_{comm} is the transfer time per byte.

After the local computation phase, each processor has the corresponding block of n/N_P elements of the resulting vector which is assigned to that processor originally. Hence, no communication is needed to rearrange the distribution of the results.

If A is stored using the CSR format then the sparse matrix A is represented by the trio of (row, col, a). In order to keep the locality in accessing the elements of individual rows, the HPF's `BLOCK` distribution is appropriate to partition all those vectors. To ensure that the $(n+1)$'th element of row is placed in the last processor, we explicitly specify the block size in the directive.

```
$HPF$ DISTRIBUTE row(BLOCK( (n+NP-1)/NP ))

$HPF$ ALIGN a(:) WITH col(:)
$HPF$ DISTRIBUTE col(BLOCK)
```

When the CSR format is used for storing the sparse matrix, the following HPF code fragment can be applied for the matrix-vector multiplication:

```
q = 0.0
FORALL( j = 1:n )
  DO k = row(j), row(j+1)-1
     q(j) = q(j) + a(k) * p( col(k) )
  ENDDO
ENDFORALL
```

where the `FORALL` expresses parallelism across the j-loop. This works because $A(i,j) = A(j,i)$ for the case of CG where A **must** be symmetric. The operation runs in row order, finishing up with one element of q at each iteration and iterations are independent of each other.

Similar to the dense storage format any row-based sparse storage format like CSR will incur the same broadcast overhead when a partitioning like shown in figure 5 is used. In addition, there is an additional overhead not found in dense storage format. Since the index set of the FORALL in the outer loop is partitioned among the processors, a processor that is responsible from a specific row may not have all the actual data elements (i.e., col and a) on that row. Therefore, additional communication is needed to bring in those missing elements.

Scenario 2: Column-wise partitioning

If a dense storage format is used to represent A, then the second dimension of A should be aligned with the p and q vectors. When vector p is distributed, columns of A are automatically partitioned among the processors (figure 6). The HPF directive for this purpose is:

```
!HPF$ ALIGN A(*, :) WITH p(:)
```

As the vector p is already aligned with the columns of A, performing the element-wise multiplication will not require any interprocessor communication. However, since each processor will have a partition of the final vector q, each time some other processor produces a result corresponding to an element that is owned by another processor, it has to communicate this value to the owner of it. Since the owner may also update the same element, this operation will cause an interprocessor dependency. Therefore the matrix-vector operation can not be performed in parallel and the following serial code is used:

```
q = 0.0
DO j = 1, n
   pj = p(j)
   DO i = 1, n
      q(i) = q(i) + A(i, j) * pj
   END DO
END DO
```

If we used the message-passing SPMD model, then each processor would have a private copy of the vector q which would be used to gather the partial results locally, and a merge operation would be employed at the end to obtain the final product (q vector) of the matrix-vector multiplication. We could simulate the same thing using two dimensional temporary local vectors in place of vector q in each processor. At the end of the outer loop we use the HPF `SUM` intinsic to generate the final vector.

```
      paramater (NP = NUMBER_OF_PROCESSORS())
      REAL, dimension(1:n) :: q
      REAL, dimension(1:n, 1:NP) :: q_private

!HPF$ ALIGN q_private(*, :) WITH q(:)
!HPF$ ALIGN q(:) WITH p(:)
!HPF$ DISTRIBUTE p(BLOCK)
      ...
      ...
      q_private = 0.0
      BS = n/NP
      FORALL (l = 1 : NP)
        DO (j = (l-1)*BS+1 , l*BS)
          pj = p(j)
          DO i = 1, n
             q_private(i, l) = q_private(i, l)
                             + A(i, j) * pj
          END DO
        END DO
      END FORALL
      q = SUM(q_private, DIM = 2)
```

If the matrix A is stored in CSC Format then the following distribution and alignment directives and serial code fragment arises for the matrix-vector multiply ($A \cdot \vec{p} = \vec{q}$):

```
$HPF$ DISTRIBUTE col(BLOCK( (n+NP-1)/NP ))

$HPF$ ALIGN a(:) WITH row(:)
$HPF$ DISTRIBUTE row(BLOCK)

     ...
     q = 0.0
     DO j = 1, n
        pj = p(j)
        DO k = col(j), col(j+1)-1
           q(row(k)) = q(row(k)) + a(k)*pj
        ENDDO
     ENDDO
```

This operates in Fortran column-major order where each i-iteration gives a partial sum at several elements of q. As in the dense case, there are dependencies between j-iterations and no parallel loop execution is possible. This part can also be parallelized by using a two dimensional local array as described as above.

```
      REAL, dimension(1:n, 1:NP) :: q_private
      ...
      ...
      q_private = 0.0
      BS = n/NP
      FORALL (l = 1 : NP)
        DO (j = (l-1)*BS+1 , l*BS)
          DO k = col(j), col(j+1)-1
            q_private(row(k),l)=q_private(row(k),l)
                              + a(k)*pj
          END DO
        END DO
      END FORALL
      q = SUM(q_private, DIM = 2)
```

Computation and Communication Costs

For simplicity, assume that the average number of nonzero elements per row in A is m_z, and the total number of nonzero elements in the entire matrix is $n_z = m_z \times n$.

It may be desirable to control the number of non zero elements stored on each processor if there is some identifiable structure to the sparse matrix that would otherwise lead to a load imbalance. Generally this would require a data mapping that forces processors to perform the same number of scalar multiplications and additions while multiplying the matrix with a vector. This however requires that $A(i, i)$ and $p(i)$ no longer

necessarily be assigned to the same processor which requires communication before the required multiplication.

Assuming that the average number of non zero elements m_z is representative of all rows or columns, each processor performs an average of $n \times n/P$ multiplications and additions if a dense storage format is used or $m_z \times n/P$ multiplications and additions if a sparse storage format is used in the computation phase.

The communication time for Scenario 2 is the same as the communication time for the global broadcast used in Scenario 1. Hence, it is not possible to reduce the communication time if the matrix is partitioned into regular stripes either in a row-wise or column-wise fashion.

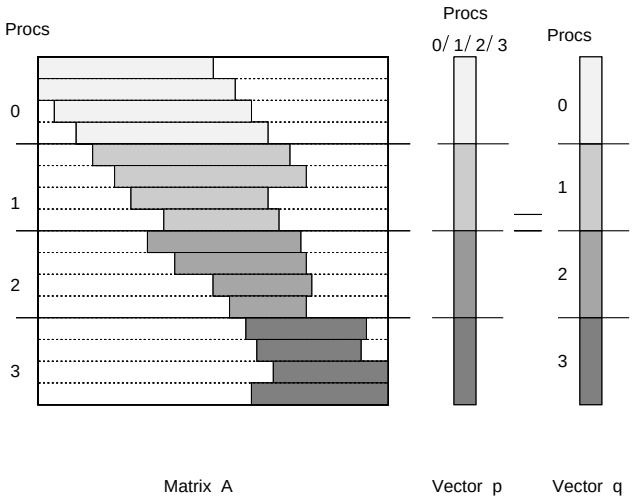

Figure 5: *Matrix vector multiplication where A is distributed in a (BLOCK, *) fashion, and vectors are distributed in a (BLOCK) fashion.*

The full HPF code for the CG algorithm for CSR format is given in figure 7.

6 Proposed HPF Extensions

We propose two kinds of extensions to the current HPF definition that will make writing the above mentioned algorithms easier and will enhance load balance to support CG codes.

The first one specifically addresses the CG codes which uses the CSC format to store the sparse matrices. As seen above, in the current HPF definition it is not easy to express this loop in a parallel fashion although an explicit message-passing program is able to do that.

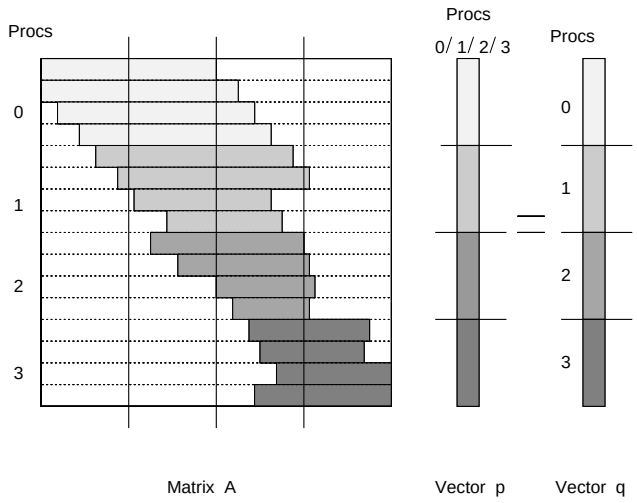

Figure 6: *Matrix vector multiplication where A is distributed in a (*, BLOCK) fashion, and vectors are distributed in a (BLOCK) fashion.*

We propose a new way to eliminate the existing dependencies caused by the many-to-one assignments and partition the resulting parallel loops in an elegant way.

The second type of extensions are related to the cases where the load imbalance may become an important issue due to the sparsity of the data structures. We propose ways to partition the sparse matrix in a manner that will allow the compiler not to disturb the logical structure of the matrix. That is rows and columns may be identified as indivisable entities while the distribution is performed.

6.1 Private variables and Reductions

In HPF, the `DO` loops have sequential semantics. Single- or multi-statement `FORALL` and `INDEPENDENT FORALL` or `INDEPENDENT DO`'s are provided for expressing the parallelism in loops. In the case of CG codes where the A matrix is represented using CSC format, the main obstacle that prevents us from parallelizing both loops of the sparse matrix-vector multiply is that in the inner loop, the $row(k)$ values are **not** unique and so many left hand sides accumulate into a single right hand side in a many-to-one fashion which introduces a dependency in the inner loop that even prevents us parallelizing the outer loop.

The matrix-vector multiplication loops can not be expressed in paralllel using neither the `FORALL` construct nor the `INDEPENDENT DO` construct. The option of using a `FORALL` is eliminated because its semantics require that all the right-hand sides should be computed before

```
      REAL, dimension(1:nz) :: a
      INTEGER, dimension(1:nz) :: col
      INTEGER, dimension(1:n+1)  :: row
      REAL, dimension(1:n) :: x, r, p, q

!HPF$ PROCESSORS :: PROCS(NP)
!HPF$ ALIGN (:) WITH p(:) :: q, r, x, b
!HPF$ DISTRIBUTE p(BLOCK)
!HPF$ DISTRIBUTE row(CYCLIC((n+NP-1)/np)
!HPF$ ALIGN a(:) WITH col(:)
!HPF$ DISTRIBUTE col(BLOCK)

      (usual initialisation of variables)

      DO k=1,Niter
        rho0 = rho
        rho = DOT_PRODUCT(r, r)  ! sdot
        beta = rho / rho0
        p = beta * p + r    ! saypx

        q = 0.0             ! sparse mat-vect multiply
        FORALL( j=1:n )
          DO i = row(j), row(j+1)-1
            q(j) = q(j) + a(i) * p(col(i))
          END DO
        END FORALL

        alpha = rho / DOT_PRODUCT(p, q)
        x = x + alpha * p   ! saxpy
        r = r - alpha * q   ! saxpy
        IF ( stop_criterion ) EXIT
      END DO
```

Figure 7: *HPF version of sparse storage CG (CSR format).*

an assignment to the left-hand sides be done. An accumulation operation like we would like to express is not allowed within the `FORALL` body. At the same time, the write-after-write dependency violates Bernstein's conditions [3], and eliminates the possibility of using an `INDEPENDENT DO`.

Since the array q is causing the dependency, a naive solution would be to extend the language definition to let arrays as well as variables be declared by the `NEW` clause. However, even though, it accepted arrays this semantics would require q's value to be forgotten at the end of each iteration which would not suit our purposes. We need something that will be associated with processors and will survive until it is merged at the loop termination.

If we could eliminate the dependency in the inner loop by using *private arrays*, we could express the outer loop in a parallel fashion. Hence, we propose a new mechanism which we call `PRIVATE` abstraction to allow the program to fork copies of a data structure that are private to each processor. Private variables are different from the ones declared using the HPF `NEW` in loops because they will stay alive until the end of the private region as opposed to *new variables* that stays alive until the end of the loop iteration that it is defined. The private variables are merged into a global single copy again (`WITH MERGE` option) (Figure 8) or discarded completely (`WITH DISCARD` option) at the end of the loop(private region.)

In practice, this can be implemented in HPF by assuming N_P virtual processors and by allocating storage for N_P temporary vectors each of length n. The loop is then executed in parallel where each iteration of the outer loop is assigned to a specific processor and the operation of each processor is truly independent of each other. A runtime library function similar to Fortran 90 `SUM` intrinsic reduction function can provide the necessary merging of these temporary values into a single vector outside the loop. This is somewhat unsatisfactory, due to the potentially unnecessary storage requirements, particularly if $n \gg N_P$, and our proposed HPF extension would relieve the programmer of a lot of the cumbersome temporary storage allocation and alignments.

Using two-dimensional arrays as shown in the previous section seems to be favorable at first considering that it eliminates the allocation/deallocation costs of vectors at each loop entry/exit. However, keeping large vectors in each processor's memory permanently is costly especially if both n and N_P are very big and this kind of loops are executed just a few times in the lifetime of the program.

```
      q = 0.0
!EXT$ ITERATION j ON PROCESSOR(j/np),  &
!EXT$   PRIVATE(q(n)) WITH MERGE(+),   &
!EXT$   NEW(pj, k), PRIVATE(q(n))
      DO j = 1, n
        pj = p(j)
        DO k = col(j) , col(j+1)-1
           q(row(k)) = q(row(k)) + A(k)*pj
        END DO
      END DO
C --  private copies of q() are merged to
C --  a global q at the termination of outer loop.
```

Once the privatization is established, the loop can be parallelized. Most HPF compilers uses the well-known *owners compute rule* where an iteration is assigned to the processor which owns the left-hand-side (lhs) array element that is assigned to in that iteration. As the array q is accessed through a level of indirection, the value of its index (i.e.$row(k)$) can be known

only at run-time. *Inspector-executor* mechanisms [14] which are costly in nature should be employed for the determination of the owner of the lhs. However, in our case, a much simpler mechanism can be used. We propose using a `ON PROCESSOR(f(i))` construct which will map iteration i onto processor $f(i)$. In this way we can specify the iteration mapping at compile-time without any runtime overhead. A similar mechanism was used in the implementation of the Kali and Vienna Fortran compilers [13, 5]. Actually, in some cases as above we are obligated to specify the iteration mapping while using the private abstraction, because the lhs arrays have been privatized and they have no specific owner. Of course, if private arrays are used only on the rhs (possibly with a `DISCARD` option), then using the `ON` clause is just an option. For those cases, private mechanism helps the compiler to prefetch the future data before it is needed and without necessitating expensive inspector loops.

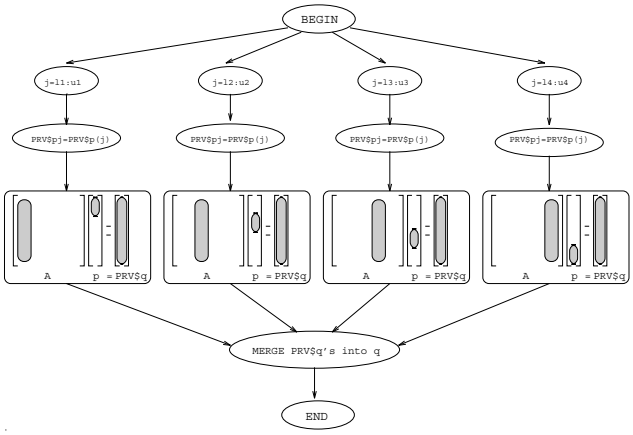

Figure 8: *Illustrating the use of private abstraction on parallel loops.*

Cost Analysis

In terms of the implementation cost of this PRIVATE mechanism, it is cheap and easy to implement in terms of storage and computation time. Here we will compare a serial implementation with one using the private abstraction in a limited memory environment.

Assuming p, q and row, col are distributed in a block fashion, and a, row and col were adjusted in a suitable fashion that would not require interprocessor requirement (this will be shown later), the sequential code will be executed in

$$\mathcal{O}(m_z).T_{func} + (m_z/N_P).T_{comm}.\log N_P$$

time.

On the other hand, the parallel version will take

$$\mathcal{O}(m_z/N_P).T_{func} + 2 * ((m_z/N_P).T_{comm}).\log N_P$$

time.

Here T_{func} is the time required to make the local computation in the loop, N_P is the number of processors, T_{comm} is the time to transfer one byte of data to another processor. We assume that a tree like broadcast/accumulation mechanism is used.

6.2 Compressed Sparse Block Distributions

Consider how the sparse data may be blocked prior to distribution. We discuss two sparse block distributions: one of them is *regular* or *uniform* which is used in cases where the number of elements across rows or columns of the sparse matrix is approximately the same and the other one uses a load-balancing heuristic and distributes and aligns related data structures accordingly since the number of elements across rows or columns varies a lot.

6.2.1 Regular (Uniform) Sparse Block Distributions

The uniform or regular sparse block distribution can be used in cases where each sparse matrix row(or column) is known to have approximately the **same** number of elements, therefore there is an approximate load balance. In such a case, it is sufficient to distribute A and row (or col) so that each corresponding row (or column) is stored in its entirety in only **one** processor. The HPF regular block distributions divide the data array in an even fashion without paying attention to whether the division point is at the middle of a column or not. It is sufficient to adjust the partition to reduce communication among intra-column elements.

Since in typical CG applications the number of nonzero elements and the structure of the matrix is not known until runtime, compiler cannot determine the layout patterns for row (or col) and A at compile-time. Therefore, these data structures are initially distributed using HPF's regular distribution primitives. In the case of CSC format, we use the following initial distribution statements:

```
!HPF$ PROCESSORS :: PROC(NP)
!HPF$ DISTRIBUTE col(BLOCK((N+NP-1)/NP))
!HPF$ DYNAMIC, ALIGN a(:) WITH row(:)
!HPF$ DYNAMIC, DISTRIBUTE row(BLOCK)
```

The `DYNAMIC` keyword warns the compiler that this distribution is temporary, actual data distribution is dependent on the runtime data. Distributed array descriptors (DAD) for the dynamically distributed arrays are generated at runtime. DADs contain information about the portions of the arrays residing on each processor. The compiler uses this hint to generate communication calls and to distribute corresponding loop iterations.

We now introduce the concept of *indivisable entities* within larger data structures. An *indivisable entity (atom)* is a logical abstraction consisting of a chunk of elements enclosed within two border elements, and it cannot be divided among processors during the data distribution process. It should completely belong to one **single** processor. The following directive is used to inform the compiler on the logical grouping of subdata within a larger data structure.

```
!EXT$ INDIVISABLE row(ATOM:i) :: col(i:i+1)
```

The above directive specifies that atomic entity i of *row* is encapsulated by the elements i and $i+1$ of the indirection array *col*.

The `REDISTRIBUTE` directive indicates that the data is available for use in the partitioning of the data arrays. The user is responsible for putting the `REDISTRIBUTE` directive in the proper place to improve the performance. Given the concept of atoms, redistribution can be made, depending on the runtime data, in an elegant manner.

```
!EXT$ REDISTRIBUTE row(ATOM: BLOCK)
```

This directive ensures that the elements of the *row* vector are distributed in a similar fashion to the regular HPF `BLOCK` distribution, yet the atoms instead of individual elements are used as the basis in the distribution. This ensures that elements of an atom is not divided among two or more processors. We could use an `(ATOM: CYCLIC)` distribution in a similar way. Since we still keep the continuity of the column (or row) elements, the compiler avoids generating a full distribution map of the size of the target arrays. A small array in the size of the number of processors keeps the cut-off points, and it is replicated over all processors.

Another possibility may be extending the definition of HPF `ALIGN` to permit the alignment of atoms of one array with the elements of another. For example, if atoms of *row* array are aligned with the elements of *col* array:

```
!HPF$ ALIGN row(ATOM:i) WITH col(i)
```

then any change in the distribution of the *col* array is spontaneously followed by a corresponding change the distribution pattern of the atoms (i.e., individual columns) pointed to by the *col* array.

6.2.2 Irregular Sparse Block Distributions

In some types of problems, the structure of the sparse matrix is completely irregular - or in fact has some problem specific structure that is identifiable to a human but not to a compiler. For example, this might arise from a very irregular grid model in which some grid points may have many neighbours, while others have very few. In those cases, neither the HPF regular block distributions nor the above proposed uniform distributions will allow a good load balance.

As in the regular case, the arrays are distributed initially using HPF regular distribution directives. Indivisable entities are defined in a suitable way.

```
!HPF$ PROCESSORS :: PROCS(NP)

!HPF$ ALIGN (:) WITH p(:) :: q, r, x, b
!HPF$ DISTRIBUTE p(BLOCK)

!HPF$ DYNAMIC, DISTRIBUTE row(CYCLIC((n+NP-1)/np)
!HPF$ DYNAMIC, ALIGN a(:) WITH col(:)
!HPF$ DYNAMIC, DISTRIBUTE col(BLOCK)

!EXT$ INDIVISABLE ROW(ATOM: i) :: COL(i:i+1)
!EXT$ INDIVISABLE A(ATOM: i) :: COL(i:i+1)
```

Sparse storage format specification

Another alternative for informing the compiler that there is a sparse matrix represented by a sparse matrix storage scheme is to use an explicit directive:

```
!HPF$ SPARSE_MATRIX (CSR) :: smA(row, col, a)
```

This directive gives two clues to the compiler:
1) which sparse storage representation format is used.
2) what are the three vectors (possibly, in pointing order) representing the sparse matrix, named *smA*.
A sparse matrix definition puts a tight binding between the members of this trio, whenever any one's distribution is changed, the other two should be aligned accordingly. Furthermore, if an element of *row* is to be accessed, most probably the elements it points to in *col* and *a* will be also accessed, therefore compiler should generate code for bringing them into memory if they are not local. In short, the compiler can exploit the locality rule by knowing the relation among the members of the trio.

Load balancing sparse partitioners

It is possible to specify a load-balancing heuristic that is applied to the A, *row* and *col* arrays to cluster the rows in a way that can be distributed among the processors in an almost even-load fashion. This could map sparse columns onto processors in a balanced way if the compiler applies the heuristic to the kernel arrays first, and redistributes the elements of dependent vectors accordingly later. Several partitioners may be described from simple ones that just take care of the load distribution in the matrix vector multiplication to more elaborate ones that balances the number of columns (rows) distributed to each processor as well as the number of elements to ensure the load is also balanced in the other vector operations.

Extended syntax for expressing the redistribution of smA using a special partitioner in an HPF way might be:

```
!EXT$ REDISTRIBUTE smA
            USING CG_BALANCED_PARTITIONER_1
```

The compiler generates code for calling necessary partitioners to determine the new data distribution and arranging all dependent vectors accordingly.

A similar mechanism has been proposed in the Vienna Fortran compilation system [6] whereby an indirect mapping is constructed and passed through the HPF `DISTRIBUTE` or `REDISTRIBUTE` directives. It remains to be seen whether this can be effectively implemented on present generation architectures.

There are two approaches in the localization of *col* array. In the first one, the index values of the *row* (and a) array(s) are localized by the compiler by using a complicated runtime function. In the second case, as a by-product of the partitioner, a new *col* array is generated corresponding to the original vector whose contents have been localized to point at the local *row* array. It can be directly used without further manipulation. This second approach is possible since it is accessed in a read-only manner in the CG codes.

7 Conclusions

We have illustrated some of the issues arising from the use of HPF for expressing conjugate gradient algorithms. The advantages are the potential for faster computation on parallel and distributed computers, and additional code portability and ease of maintainance by comparison with message-passing implementations. Disadvantages (in common with any parallel implementation) over serial implementations are additional temporary data-storage requirements of parallel algorithms.

We have identified how existing features in HPF allow efficient expression and implementation of some of the components of conjugate gradient algorithms. We have also highlighted where possible extensions to HPF will allow a compilation system to produce even more memory-efficient and compute-efficient executable code.

Current HPF distribution directives only allow arrays to be distributed according to regular structures such as BLOCK and CYCLIC. Whilst this is adequate for dense or regularly structured problems it does not provide the necessary flexibility for the efficient storage and manipulation of arbitrarily sparse matrices. We also propose extensions for the iteration mapping of the loops employed by CG codes.

Although we have described the limitations of the current HPF-1 definition and the basic requirements for the further development of HPF-2, we have not attempted to discuss how these should be implemented within the compiler itself through directives, intrinsic functions or some other mechanism. Instead we have indicated in general terms that the provision of *some* additional flexibility to cope with irregular problems such as those described within this paper is essential if HPF is to be widely adopted in place of existing message passing technologies.

References

[1] Bailey, D., Barton, J., Lasinski, T. and Simon, H., Editors, "The NAS Parallel Benchmarks", NASA Ames, NASA Technical Memorandum 103863, July 1993.

[2] Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato,. J., Dongarra, J.J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H.A. "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", SIAM, 1994.

[3] Bernstein, A.J., "Analysis of Programs for Parallel Processing," IEEE Transactions on Computers, 15(5), October 1966.

[4] Bogucz, E.A., Fox, G.C., Haupt, T., Hawick, K.A., Ranka, S., "Preliminary Evaluation of High-Performance Fortran as a Language for Computational Fluid Dynamics," *Paper AIAA-94-2262* presented at 25th AIAA Fluid Dynamics Conference, Colorado Springs, CO, 20-23 June 1994.

[5] Chapman, B., Mehrotra, P., and Zima, H., "Programming in Vienna Fortran," *Scientific Programming*, 1(1):31-50, Fall 1992.

[6] Chapman, B., Mehrotra, P., Mortisch, H., and Zima, H., "Dynamic data distributions in Vienna Fortran," In *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.284.

[7] Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A., "Solving Linear Systems on Vector and Shared Memory Computers", SIAM, 1991.

[8] Duff, I.S., Erisman, A.M., Reid, J.K., "Direct Methods for Sparse Matrices", Clarendon Press, Oxford 1986.

[9] Cheng, Gang., Hawick, Kenneth A., Mortensen, Gerald, Fox, Geoffrey C., "Distributed Computational Electromagnetics Systems", to appear in Proc. of the 7th SIAM conference on Parallel Processing for Scientific Computing, Feb. 15-17, 1995.

[10] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng, Wu, M., Fortran D Language Specification, Technical Report CRPC-TR90079, Center for Research on Parallel Computation, December 1990.

[11] Hockney, R.W., and Berry, M., (Editors) PARKBENCH Committee Report-1, "Public International Benchmarks for Parallel Computers", February 1994.

[12] High Performance Fortran Forum (HPFF), "High Performance Fortran Language Specification," *Scientific Programming*, vol.2 no.1, July 1993.

[13] Koelbel, C. and Mehrotra, C., "Compiling Global Name-Space Parallel Loops for Distributed Execution", IEEE Trans.of Par.and Dist.Systems, 2(4):440-451, October 1991.

[14] Koelbel, C.H., Mehrotra, P., Saltz, J., and Berryman, H., "Parallel Loops on Distributed Machines," in Proc. of the Fifth Distributed Memory Computing Conference, 1990.

[15] Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, G.L., Zosel, M.E., "The High Performance Fortran Handbook", MIT Press 1994.

[16] Kumar, V., Grama, A., Gupta, A., and Karypis, G., "Introduction to Parallel Computing: Design and Analysis of Algorithms," Benjamin/Cummings, 1994.

[17] Metcalf, M., Reid, J., "Fortran 90 Explained", Oxford, 1990.

[18] Ponnusamy, R., "Runtime and Compilation Methods for Irregular Computations on Distributed Memory Multiprocessors", Ph.D. Dissertation, Syracuse Un., May 1994.

[19] Ponnusamy, R., Saltz, J. and Choudhary, A., "Runtime Compilation Techiques for Data Partitioning and Communication Schedule Reuse", Technical Report, UMIACS-93-32, University of Maryland, April 1993.