NPAC Technical Report SCCS 691

Conjugate Gradient Algorithms in Fortran 90 and High Performance Fortran

Draft

K.A.Hawick, K.Dincer, G.Robinson, G.C.Fox.
Northeast Parallel Architectures Center,
111 College Place, Syracuse, NY 13244-4100

21 February 1995

Abstract

We evaluate the Fortran-90 and High-Performance Fortran (HPF) languages 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 also consider computational and data storage issues arising from variations of the basic conjugate gradient algorithm as well as surveying typical application problems that require an iterative solution of large matrix-formulated problems. 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.

Introduction

High Performance Fortran (HPF)[13] 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 [3] as a collection of extensions to the Fortran 90 language standard [15]. We do not discuss the details of the HPF language here as they are well documented elsewhere [14], 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 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 [6] and can be solved using direct methods [9] 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. 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 simulation, are based on the solution of large sparse systems of linear equations. We expand on this list in the section on Motivating Applications below. Iterative methods are employed in many of these applications. While the CG method itself is no longer considered state-of-the art in 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 [12] 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.

Conjugate Gradient Algorithms

The classic Conjugate Gradient non-stationary iterative algorithm as defined in [8] 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 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.

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 product 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 SAXPY operations, inner products and matrix-vector multiplication are $\mathcal{O}(n)$, $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, respectively, for vector length n.

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 [8]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 algorithm (BiCGSTAB) 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.

Preconditioners for Conjugate Gradient

The CG algorithm will generally converge to the solution of the system $A.x = b$ in at most n_e iterations, where n_e is the number of distinct eigenvalues of the coefficient matrix A. Therefore, if A has many distinct eigenvalues that 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 here and which will increase the speed of convergence of the CG algorithm. S is chosen such that $A' = S.A.S^T$ has fewer distinct eigenvalues than A and is a nonsingular matrix. 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 PCG algorithm can be expressed as follows:

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

The preconditioner S must be carefully chosen to keep the overhead of solving $M \cdot \vec{z} = \vec{r}$ 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 S is a diagonal matrix with nonzero elements only on the principal diagonal. This S can be easily derived from the principal diagonal of A. If we `ALIGN` $M(i,i)$ with $A(i,i)$ in the processors' memories and if the elements with identical indices reside on the same processor then solving the system $S \cdot \vec{z} = \vec{r}$ is equivalent to dividing each element of r by the corresponding diagonal element of S, 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: S 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 portion of A, respectively. $S = L \cdot L^T$ 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.

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` [8] operations. An efficient compilation system would insert system-optimised run-time library routines for these statements.

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

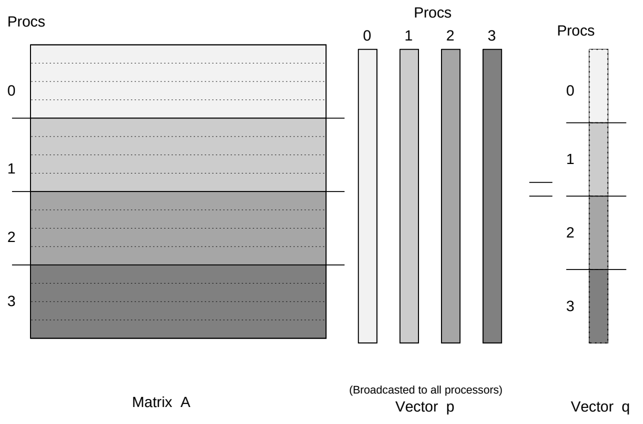

Figure 2: *Communication requirements of (dense) Matrix vector multiplication where A is distributed in a (BLOCK, *) fashion.*

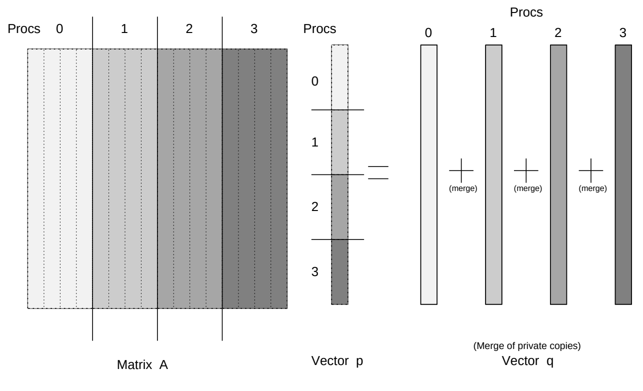

Figure 3: *Communication requirements of (dense) Matrix vector multiplication where A is distributed in a (*, BLOCK) fashion.*

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.

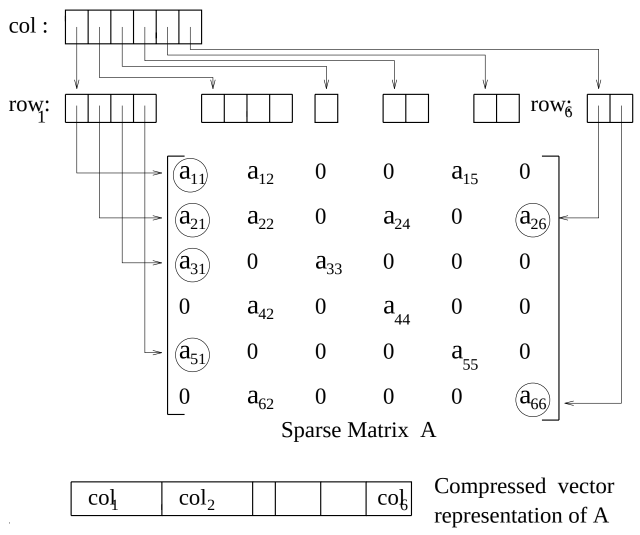
col :
row1 :
row6 :
a_{11} a_{12} 0 0 a_{15} 0
a_{21} a_{22} 0 a_{24} 0 a_{26}
a_{31} 0 a_{33} 0 0 0
0 a_{42} 0 a_{44} 0 0
a_{51} 0 0 0 a_{55} 0
0 a_{62} 0 0 0 a_{66}
Sparse Matrix A
col1 col2 col6
Compressed vector representation of A

Figure 4: *Compressed Sparse Column(CSC) representation of sparse matrix A.*

The Compressed Sparse Column (CSC) storage scheme, shown in figure 4, 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 5 illustrates how BLAS level library routines such as SAXPY and SDOT can be employed for a sparsely stored system.

Each iteration of the CG algorithm in figure 5 performs three main computations: the vector-vector operations, inner product (here shown using BLAS routines) and the matrix-vector multiplication, shown explicitly in figure 6.

In any parallel implementation that distributes the vectors and matrix A across processors' memories, the inner-products 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.

```
      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 5: *Fortran 77 version of sparse storage CG (CSC format).*

If all vectors are distributed identically among the processors, vector-vector operations such as SAXPY require no data communication in a parallel implementation since the vector elements with the same indices are involved in a given arithmetic operation and thus are locally available on each processor. Using P processors, each of these steps is performed in time $\mathcal{O}(n/P)$ on any architecture.

A parallel implementation of the inner product

```
        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 6: *Sparse matrix-vector multiply in Fortran 77 (CSC format).*

(SDOT) with P processors, takes $\mathcal{O}(n/P)+t_s \cdot logP$ time on the hypercube architecture, where t_s 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.

The computation and communication cost of the matrix-vector multiplication step depends critically on the structure of the sparse matrix A. This is discussed below.

Motivating Applications

Many scientific, engineering and other application areas generate matrices with a variety of structures but share a common feature in that the majority of entries are identically zero. Some of these matrices possess a regularity which is often exploited in alternative iterative schemes avoiding the need for the assembly, storage and solution of the matrix. A compressed storage scheme can prove more efficient by saving both memory and reducing needless computation. The exact storage scheme used is often tailored to key features of the matrix and several of these are described elsewhere in this paper. All reduced storage schemes share one common feature, the use of pointers and lists to identify the matrix row, column and value. This implies that matrix operations involve indirect addressing as shown in both the code fragments shown in figures 5 and 6.

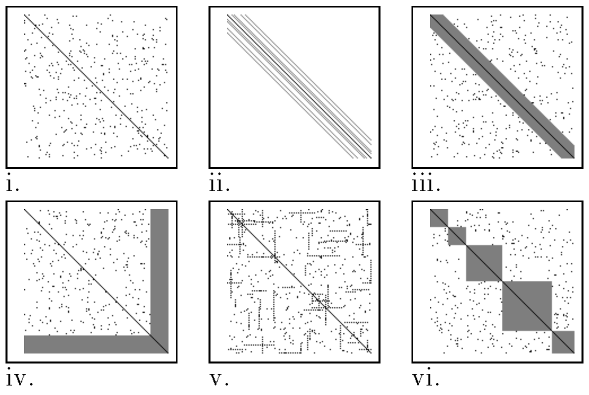

Figure 7: Typical sparse matrices from various applications.

Sparse matrices can vary considerably, ranging from completely unstructured sparsity , figure 7 i) to a highly banded and regular structure, figure 7 ii). Such regular banded matrices typically arise from discrete equation sets based upon regular grids and can be found in laminar fluid flow, structural mechanics and other engineering and scientific applications. Some applications can generate matrices where the majority of entries are close to the diagonal but there also being significant entries outside of this area. For example the iterative solution of nonlinear equation sets can produce such matrices, as shown in figure 7 iii). Note that the values of these terms many change considerably during the computation. For irregular or refined grids each grid point may be connected to a different number of neighbours and a general sparse matrix is often generated, as shown in figure 7 i). Although each node is connected to its neighbours in physical space these need not be adjacent within the matrix.

Disciplines other than those traditionally associated with the solution of large matrices have started to address problems through numerical computation. The problems of power distribution and computer networking generate matrices with a bordered structure, as shown in figure 7 iv) where dense regions represent centralised or localised distribution or connectivity features. Chemical engineering and financial modeling generate matrices with some apparent structure showing the close coupling of key processes, (figure 7 v).

Recently, considerable interest has been displayed in the literature for solving more complex sets of equations with correspondingly complex matrix structures. Codes solving coupled equation sets can generate blocked diagonal structures such as those in figure 7 vi). These matrices can be derived from high order coupled equation sets for a single process or from many distinct models interacting as in a multiblock code. Here each variable can produce a dense matrix stating its own relationship with the off diagonal terms representing dependencies on other variables. In conventional approaches terms outside the shaded areas would have been linearised to the right hand side of the equation before the solution procedure begins.

Note that the form and structure of the matrix is often determined by the storage of the parent data. Sometimes the structure of the matrix can be exploited to simplify the storage scheme selected and to improve the stability and reduce the effort required for solution.

Exploiting Sparsity

It is not a trivial matter to exploit the sparsity of the matrices that arise in many applications. There are a number of tradeoffs that must be considered, including simplicity of storage scheme versus memory requirements as well as the problems of data reorganisation for various stages of the calculation compared with the high memory costs of a dense storage scheme. These include: schemes such as the general compressed row and compressed column schemes that do not rely on any inherent matrix structure; the Yale scheme [10] which is suitable for very sparse matrices; and others which exploit some symmetry or structure in the non zero elements [2]. We focus on the general compressed row or column schemes here.

Figures 2 and 3 show how matrix row and columns can be decomposed across processors' memories. The data decomposition for the entire program will be a compromise between the demands placed by each section or dictated by a single dominant section. The use of the `TRANSPOSE` or the `REDISTRIBUTE` and `REALIGN` directives may be too expensive (in terms of communications time) to consider their repeated use at each iteration of the solver algorithm. Figures 2 and 3 show a matrix distributed according to the `BLOCK` directive. Note that this simply assigns the memory allocated to the matrix and vector in a regular manner. However the matrix may not possess an equal number of terms in each row (or column) and such a decomposition may result in severe load imbalance. The replication and broadcast of the entire vector is not in fact necessary, since only those entries actually referenced need be transferred between processors. This will reduce the communication volume leading to improved parallel performance. However since the matrices are sparse, references to off processor data may be suboptimal, with many small message exchanged. The cost and complexity of attempting an optimised domain decomposition of the problem is acknowledged in many cases be a non-trivial task. If the structure of the matrix is known in advance or clearly defined some degree of restructuring can improve matters but in general the owner-computes model of HPF does not allow adequate freedom for such problems to be expressed efficiently without considerable effort. This is further compounded if the structure of the matrix depends greatly on the input datasets and it is not possible to optimise a the code at compile time. Alternatives have been proposed [7] [5] and these are still being debated by the HPF Forum [13].

Sparse Matrix-vector Multiply

In this section we consider the multiplication of an $n \times n$ arbitrarily sparse matrix A with an $n \times 1$ vector p. As in the dense matrix vector multiplication, each row of matrix A must be multiplied with the vector x. The computation and data communication costs varies depending on the distribution of the matrix A and vectors p and q. We, here, will describe two different data distribution scenarios and show the associated costs of each scenario.

For the simplicity of the discussion, 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 severe 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 $x(i)$ no longer necessarily be assigned to the same processor which requires communication before the required multiplication.

Row-wise partitioning

In this scenario, the sparse matrix A is partitioned row-wise among the processors in an even manner. The vectors p and q are aligned with the rows of the matrix A in all the processors. This distribution is shown in figure 8 and can be expressed in HPF as follows:

```
!HPF$ DISTRIBUTE A(BLOCK, *)
!HPF$ DISTRIBUTE p(BLOCK)
!HPF$ DISTRIBUTE q(BLOCK)
```

Since the nonzero elements are at random positions in A, a row can have a nonzero entry at **any** column. This requires the entire vector p to be accessible to 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 obligates an all-to-all broadcast of the local vector elements. This all-to-all broadcast of messages containing n/P vector elements among P processors, takes $t_{start-up} \cdot logP + t_{comm} \cdot n/P$ 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.

Column-wise partitioning

The matrix A may be partitioned in a column-wise fashion amongst the processors such that each processor gets n/P columns. Vectors are partitioned amongst the processors uniformly. This corresponds to the following distribution directives in HPF:

```
!HPF$ DISTRIBUTE A(*, BLOCK)
!HPF$ DISTRIBUTE p(BLOCK)
!HPF$ DISTRIBUTE q(BLOCK)
```

where only the distribution of the matrix itself is different from that for row-wise partitioning.

As illustrated in figure 9, the vector p is already aligned with the rows of A, and hence performing the multiplication will not require any interprocessor communication. However, since each processor will have a partial product vector q at the end of the operation, these partial vectors should be merged into one final vector. A global summation operation has to be performed with messages of size n/P where each processor sends its own portion of the partial vector to the owner of that portion according to the distribution directives given. This scenario is easily generalized to the CSC format that we will use later.

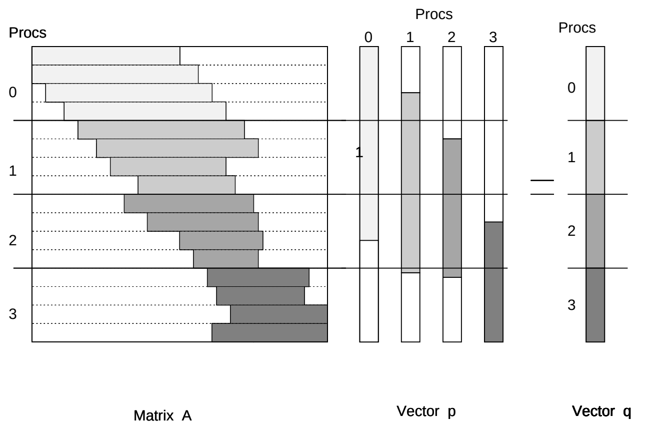

Figure 8: *Communication requirements of Matrix vector multiplication where A is distributed in a (BLOCK, *) fashion.*

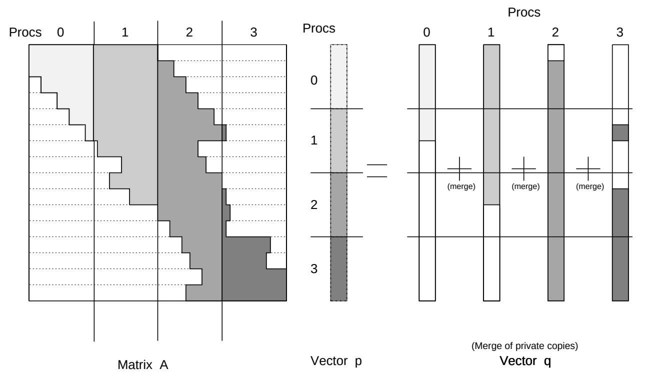

Figure 9: *Communication requirements of Matrix vector multiplication where A is distributed in a (*, BLOCK) fashion.*

Computation Costs

In the computation phase, 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.

After the computation phase, each processor has the corresponding block of 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.

The communication time for column-wise partitioning is the same as the communication time for the global broadcast used in row-wise partitioning. It is not possible to reduce the communication time whether the matrix be partitioned into regular stripes either in a row-wise or column-wise fashion.

It is important to note that this analysis assumes that the average number of non zero elements m_z is representative of all rows or columns. In practice, this is often not the case and individual rows or columns may have significant variations causing a load imbalance.

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 sparse storage schemes.

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

```
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
```

In this case the use of indirect addressing on the write operation within the row summation of $\vec{q}$ causes the compiler to generate serial or sequential code. However a directive could be used if it was known that there were no duplicate entries in any one segment of the loop. Such strategies have often been used successfully on vector machines although This considerable care on the part of the programmer and significant reordering of the datasets are required.

If however A is stored in CSR Format then the following HPF code fragment can be applied:

```
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. This works in row order, finishing up with one element of q at each iteration and iterations are independent of each other.

The HPF code for the CG algorithm for CSR format can be expressed as in figure 10.

```
      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$ DISTRIBUTE (BLOCK) :: q, p, r, x
!HPF$ DISTRIBUTE A(BLOCK)
!HPF$ DISTRIBUTE col(BLOCK)
!HPF$ DISTRIBUTE row(CYCLIC((n+1)/np)

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

Roundoff and Reduction Operations

While the above analysis has considered the computational costs there are also some numerical issues to be addressed. Rounding error can cause significant problems if there are large differences in the sign and magnitude of the individual contributions to the dot product. In particular many scientific computations generate nearly equal and opposite terms which, depending on the exact order of addition, can cause smaller, but important, terms to be obscured.

If we consider the impact of rounding error on the two algorithms then in row-wise partitioning the computation on P processors is performed in exactly the same order as on a single processor. However the in the column partitioning scheme only part of each result vector is computed on each processor.

If this were to be assembled as a partial sum and these totaled during the merging of private copies numerical differences could occur in equation 3. The sum of partial sums

$$\sum_{j=1}^{P}\sum_{i_P=1}^{N/P} \neq \sum_{i=1}^{N} \qquad (6)$$

is not necessarily equal to the full sum, in **finite arithmetic.**

This could be at least made into a deterministic (repeatable) deficiency by exchanging the entire partial vector and performing the summation on these sets in a rigid predetermined order. Since this only alters the volume of data to be communicated it will only influence parallel performance if the data volume is considerable. In the case of a dense matrix it is equivalent to performing a transpose of the matrix from column to row storage. For sparse matrices, the storage scheme can be used to reduce the total data exchange to the number of non-zero terms within the matrix. It is problematic in using a high level construct such as the INTRINSIC function DOT_PRODUCT and other numeric reduction operations, that the user has no strict way of controlling the order in which operations are carried out.

We note that is therefore unrealistic to expect arbitrary precision reproduction on an arbitrary processor configuration, without some sacrificed performance. In practice, it is hoped that the problem data will be relatively insensitive to such considerations.

Conclusions

We have discussed conjugate gradient algorithms, and have shown that the key issue for an efficient implementation is a matrix-vector multiply routine. This in turn must be able to exploit the sparse data storage scheme employed by the rest of the application code.

The advantages of HPF include 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.

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.

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 incorporated 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.

We also repeat the general observation [11] that implementations of numerically intensive applications on parallel architectures often encounter a tradeoff between the most rapidly converging (in terms of numerical analysis) algorithm which do not parallelize well), and less numerically advanced algorithms which, because they *can* be parallelized may produce the desired result in a faster absolute time.

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] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., and Ranka, S., “Fortran 90D/HPF compiler for distributed-memory MIMD computers: design, implementation, and performance results,” *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.351.

[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., Mortisch, H., and Zima, H., “Dynamic data distributions in Vienna Fortran,” *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.284.

[6] 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.

[7] Dincer, K., Hawick, K.A., Choudhary, A., Fox, G.C., “High Performance Fortran and Possible Extensions to support Conjugate Gradient Algorithms” NPAC Technical Note SCCS-639, October 1994.

[8] 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.

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

[10] Eisenstat, S.C., Elman, H.C., Schultz, M.H., Sherman, A.H., “The Yale Sparse Matrix Package”, Yale Technical Report DCS/RR-265, Department of Computer Science, Yale University, April 1983.

[11] Hawick, K.A., and Wallace, D.J., “High Performance Computing for Numerical Applications”, Keynote address, *Proceedings of Workshop on Computational Mechanics in UK*, Association for Computational Mechanics in Engineering, Swansea, January 1993.

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

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

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

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