CFD ALGORITHMS IN HIGH PERFORMANCE FORTRAN

K.A.Hawick,* E.A.Bogucz,† A.T.Degani,‡ G.C.Fox § G.Robinson,¶
Northeast Parallel Architectures Center,
111 College Place, Syracuse, NY 13244-4100 ‖

Abstract

We evaluate the High-Performance Fortran (HPF) language for expressing and implementing algorithms for Computational Fluid Dynamics (CFD) applications on high performance computing systems. In particular we discuss: implicit methods such as the ADI algorithm, full-matrix methods such as the panel method, and sparse matrix methods such as conjugate gradient. We focus on regular meshes, since these can be efficiently represented by the existing HPF definition. 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

Successful implementations of envisioned multidisciplinary analysis and design for large-scale aerospace systems require High Performance Computing and Communications (HPCC) technology to provide faster computation speeds and larger memory. HPCC is already important for rapid and cost-effective execution of simulation codes in individual disciplines and is even more necessary for the simultaneous execution of the various components of multidiscipline design codes such as computational fluid dynamics (CFD), computational electromagnetics (CEM)[1], thermodynamics and structural models.

Although parallel and distributed machines have shown the promise of fulfilling this objective, the potential of these machines for running large-scale production-oriented CFD codes has not been fully exploited yet. It is expected that such computations will be carried out over a broad range of hardware platforms thus necessitating the use of a programming language that provides portability, ease of maintainance for codes as well as computational efficiency. Until recently, the unavailability of such a language has hindered any comprehensive move toward porting codes from mainframes and traditional vector computers to parallel and distributed computing systems.

High Performance Fortran (HPF)[2] 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[4]. We do not discuss the details of the HPF language here as they are well documented elsewhere[5], 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. In this manner, the code runs fast and can make full use of the larger (distributed) memory.

We have already conducted a preliminary study of the general suitability of the HPF language for CFD[6] using experimental HPF compilation systems developed at Syracuse and Rice, and with the growing availability of HPF compilers on platforms such as Digital's Alphafarm we are able to describe specific coding issues. Unfortunately, we are unable to report performance timing figures since we are working with an early release of a

*Senior Research Scientist; HPF Applications Project Leader; Member, AIAA.
†Associate Director; Associate Professor; Associate Chairman, Department of Mechanical, Aerospace and Manufacturing Engineering, Syracuse University. Member, AIAA.
‡Alex G. Nason Research Fellow, Member, AIAA.
§Director; Professor, Computer Science and Physics, Syracuse University; Member, AIAA.
¶Research Scientist
‖This work sponsored, in part, by ARPA
⁰Copyright ©1995 by A. T. Degani. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

proprietary HPF compiler.

To illustrate the main ideas of data distribution and communication in an HPF code, we implement the ADI algorithm and demonstrate how conflicts between the optimal data decomposition and the computational structure of the algorithm may be resolved. Full matrix algorithms can also be implemented in HPF, and we illustrate these ideas with a panel method code. Sparse matrix methods such as the conjugate gradient method are not trivial to implement efficiently in HPF at present. The difficulty is an algorithmic one rather than a weakness of the HPF language itself. We demonstrate this idea with a conjugate gradient code, where the resulting sparse matrix can be solved iteratively, reducing the linear algebra component to essentially one of matrix-vector multiplies. Practical implementations for large problems require the matrix to be stored as a sparse system, and the resulting indexing into the packed storage scheme is not simple to implement in a scalably efficient manner.

For the purposes of carrying out multi-disciplinary simulations which include CFD, it is generally of prime importance to achieve a given level of numerical accuracy for a given size of system in the shortest possible time. Consequently, there is a tradeoff between rapidly converging numerical algorithms that are inefficient to implement on parallel and distributed systems, and more slowly converging and perhaps less numerically "interesting" algorithms that can be implemented very efficiently[7]. We illustrate these ideas in the implementation of the algorithms considered here.

2. Alternate Direction Implicit

The Alternate Direction Implicit (ADI) method is an iterative technique commonly used to solve time-dependent nonlinear set of equations. However, in order to highlight the salient algorithmic features of the ADI implementation in HPF, we consider the simple two-dimensional Poisson equation for illustrative purposes. Let

$$\nabla^2\psi = f \quad on \quad \Omega = (0,1)\times(0,1), \tag{1}$$

$$\psi = \psi_g \quad on \quad \partial\Omega, \tag{2}$$

where, for simplicity, we assume Dirichlet boundary conditions on ψ. The domain Ω is discretized into N_x+1 and N_y+1 equal intervals along the x and y directions, respectively, and the grid indices vary as $0 \le i \le N_x+1$ and $0 \le j \le N_y+1$. A finite-difference discretization of equation 1 yields

$$\left\{\frac{\delta_x^2}{\Delta x^2} + \frac{\delta_y^2}{\Delta y^2}\right\}\psi = f, \tag{3}$$

where δ is the central-difference operator. Denoting μ_2 as the averaging operator over twice the mesh interval, the central difference operator, δ, in terms of μ_2 is given by $\delta^2 = 2(-1+\mu_2)$. This identity may be used to express equation 3 in two alternative ways, viz.

$$\left\{-\frac{2}{\Delta x^2} + \frac{\delta_y^2}{\Delta y^2}\right\}\psi = f - \frac{2\mu_{2x}}{\Delta x^2}\psi, \tag{4}$$

$$\left\{\frac{\delta_x^2}{\Delta x^2} - \frac{2}{\Delta y^2}\right\}\psi = f - \frac{2\mu_{2y}}{\Delta y^2}\psi. \tag{5}$$

If we restrict our attention to a second-order accurate formulation, then, for each i, $1 \le i \le N_x$, equation 4 defines a tridiagonal system of equations subject to Dirichlet boundary conditions at $j=0$ and $j=N_y+1$. In the x-sweep of the ADI algorithm, i is varied from 1 to N_x and at each i location, equation 4 is solved to obtain updated values of ψ for all j. Subsequent to the completion of the x-sweep, a y-sweep is initiated by varying j from 1 to N_y, but now, the tridiagonal system in equation 5 is solved subject to Dirichlet boundary conditions at $i=0$ and $i=N_x+1$. An x-sweep followed by a y-sweep completes one ADI iteration. Figure 1 shows the important code fragments for the sequential implementation of the ADI method written in FORTRAN 90. The tridiagonal coefficient matrix along with the right-hand side of equations 4 and 5 are evaluated in the array `C(1:4,:)`. This array is then passed to the subroutine `THOMAS` which uses the well-known Thomas algorithm to solve the tridiagonal system of equations. The variables `BCLFT, BCRHT, BCBOT` and `BCTOP` are the Dirichlet conditions on the four sides of the boundary and are assumed constant. The coefficient matrix for the Poisson equation is constant and need not be formed every iteration; however, for nonlinear problems, the coefficient matrix must be evaluated every iteration and the code structure in figure 1 reflects this general situation. Note that in both the x- and y-sweeps, the updated values of ψ are used immediately in forming the tridiagonal system of equations at the next station.

Prior to discussing the ADI implementation in HPF, we first address the broader issue of data-parallel execution and data dependency. To this end, consider the evaluation of the right-hand side `C(4,:)` in the x-sweep. The presence of `PSI(I-1,1:NY)` within the `DO` loop causes a data dependency which prohibits data-parallel execution. For nonlinear problems, the evaluation of the coefficient matrix may also introduce data dependency. A similar argument also holds for the y-sweep. Clearly, the sequential code in figure 1 must be modified to enable data-parallel execution, and the simplest alternative is to form all the coefficient matrices first before solving any of the tridiagonal system of equations. This modification is shown in the code fragment in figure 2. In effect, the data dependency shown

```
      PROGRAM ADI_SEQUENTIAL
...   declarations, interfaces and initializations ...
      DO ITER = 1,ITERMAX   ! begin iteration loop
      DO I = 1,NX         !x sweep
      C(1,1:NY) = DY2INV
      C(2,1:NY) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,1:NY) = DY2INV
      C(4,1:NY) = F(I,1:NY) - &
                  DX2INV*(PSI(I+1,1:NY)+PSI(I-1,1:NY))
      CALL THOMAS (NY,C,BCBOT,BCTOP,PSI(I,0:NY+1))
      END DO
      DO J = 1,NY     !y sweep
      C(1,1:NX) = DX2INV
      C(2,1:NX) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,1:NX) = DX2INV
      C(4,1:NX) = F(1:NX,J) - &
                  DY2INV*(PSI(1:NX,J+1)+PSI(1:NX,J-1))
      CALL THOMAS (NX,C,BCLFT,BCRHT,PSI(0:NX+1,J))
      END DO
...   check convergence ...
      END DO
      END

      SUBROUTINE THOMAS (NK,C,ZO,ZN,Z)
...   declarations ...
      D(1,0) = 0.0
      D(2,0) = ZO
      DO K=1,NK
      D(1,K) = -C(1,K)/(C(2,K) + C(3,K)*D(1,K-1))
      D(2,K) = (C(4,K)-C(3,K)*D(2,K-1))/ &
               (C(2,K) + C(3,K)*D(1,K-1))
      END DO
      Z(NK+1) = ZN
      DO K=NK,1,-1
      Z(K) = D(1,K)*Z(K+1) + D(2,K)
      END DO
      Z(0) = ZO
      RETURN
      END
```

Figure 1: Sequential ADI solver for the two-dimensional Poisson equation

within the DO loops in figure 1 is removed by decomposing each DO loop into two: *(i)* in the first loop, the coefficient matrix is formed using the values of PSI from the previous sweep, and *(ii)* in the second loop, which is not shown in figure 2, a set of tridiagonal system of equations are solved. It may be noted that C needs to be promoted to a three-dimensional array which increases memory requirements — a common feature of data-parallel constructs. Although data-parallel execution has been enabled by the modifications shown in figure 2, it should be noted that the convergence characteristics of the ADI algorithm have also been altered and this issue is discussed subsequently.

An important aspect of writing code in HPF is determining the optimal data distribution among the processors. Consider the thomas algorithm in figure 1. The two loops in K correspond to forward and backward substitution and use recursive relationships which necessarily introduce data dependency. On the other hand, if the coefficient matrix and the right-hand sides are evaluated as suggested in figure 2, the resulting set of the tridiagonal system of equations may be solved in any order. Thus, the order of execution in the x-sweep is independent of I, and, for the y-sweep, is independent of J. It then follows that the optimal data distribution for the x- and y-sweeps is as shown in figure 3 where, for illustrative purposes, a machine with four processors is assumed. The ADI method is a simple example that serves to illustrate that the optimal data distribution for efficient execution is, in general, different for various sections of the code. One possible solution to this problem is to redistribute the data back and forth between the two layouts shown in figure 3, and this is considered next in the implementation of the ADI method in HPF.

```
      DO I = 1,NX     !x sweep
      C(1,I,1:NY) = DY2INV
      C(2,I,1:NY) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,I,1:NY) = DY2INV
      C(4,I,1:NY) = F(I,1:NY) - &
                    DX2INV*(PSI(I+1,1:NY)+PSI(I-1,1:NY))
      END DO
...   solve a set of tridiagonal system of equations
      DO J = 1,NY       !y sweep
      C(1,J,1:NX) = DX2INV
      C(2,J,1:NX) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,J,1:NX) = DX2INV
      C(4,J,1:NX) = F(1:NX,J) - &
                    DY2INV*(PSI(1:NX,J+1)+PSI(1:NX,J-1))
      END DO
...   solve a set of tridiagonal system of equations
```

Figure 2: Modification of x- and y-sweeps to enable data-parallel execution

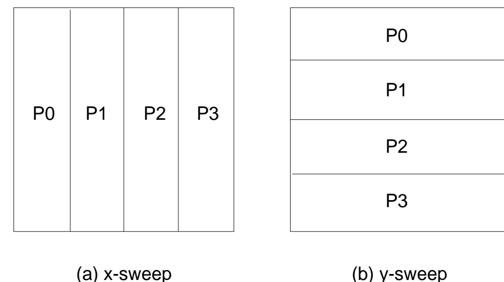

Figure 3: Optimal data distribution for the x- and y-sweeps of the ADI method.

Figure 4 shows a fragment of the HPF code, where the directives are specified by '!HPF$'. The processors P are arranged in a one-dimensional array and the number of processors, NUMPROCS, is specified in a module

```
      PROGRAM ADI_DATA_PARALLEL_HPF
!     this version is valid only for NX=NY
      INTEGER :: NMAX = MAX(NX,NY)
...   declarations, interfaces and initializations ...

!HPF$ PROCESSORS P(NUMPROCS)
!HPF$ TEMPLATE   TEMPPSI(0:NMAX+1,0:NMAX+1)
!HPF$ TEMPLATE   TEMPC(4,0:NMAX+1,0:NMAX+1)
!HPF$ DISTRIBUTE TEMPPSI(BLOCK,*)
!HPF$ DISTRIBUTE TEMPC(*,BLOCK,*)
!HPF$ ALIGN with TEMPPSI :: PSI,F,FT
!HPF$ ALIGN with TEMPC   :: C

      FT = TRANSPOSE(F)      !transpose F and store in FT

      DO ITER = 1,ITERMAX    ! begin iteration loop
      DO I = 1,NX            !x sweep
      C(1,I,1:NY) = DY2INV
      C(2,I,1:NY) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,I,1:NY) = DY2INV
      C(4,I,1:NY) = F(I,1:NY) - &
                    DX2INV*(PSI(I+1,1:NY)+PSI(I-1,1:NY))
      END DO
      CALL THOMAS (NX,NY,C,BCBOT,BCTOP,PSI)

      PSI = TRANSPOSE(PSI)   !transpose after x-sweep

      DO J = 1,NY            !y sweep
      C(1,J,1:NX) = DX2INV
      C(2,J,1:NX) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,J,1:NX) = DX2INV
      C(4,J,1:NX) = FT(J,1:NX) - &
                    DY2INV*(PSI(J+1,1:NX)+PSI(J-1,1:NX))
      END DO
      CALL THOMAS (NY,NX,C,BCLFT,BCRHT,PSI)

      PSI = TRANSPOSE(PSI)    !transpose after y-sweep

...   check convergence ...
      END DO
      END
```

Figure 4: ADI implementation in HPF

(not shown). Two templates, TEMPPSI and TEMPC are defined and distributed to conform with the data layout in figure 3(a). The asterisk indicates that the data elements of that dimension are all mapped onto the same processor. The arrays PSI, F and C are then ALIGNed with the appropriate templates.

The data redistribution required between two successive alternate direction sweeps may be best accomplished by the HPF directive REDISTRIBUTE. One of the main objective in undertaking the present study was to write HPF codes which compiled and executed using current state-of-the-art compilers. Unfortunately, at present, the REDISTRIBUTE directive has not been implemented in the compiler used here, and a less appealing alternative, viz. the intrinsic function TRANSPOSE was utilized. In this case, modifications are required in the code shown in figure 4 to treat situations in which $N_x \neq N_y$. In light of the fact that such changes are not necessary with the HPF directive REDISTRIBUTE, we restrict our attention henceforth to the case where $N_x = N_y$.

The x-sweep is immediately followed by a TRANSPOSE of PSI. In this fashion, the distribution of the transpose PSI(J,I) according to the data layout in figure 3(a) is equivalent to the distribution of PSI(I,J) in figure 3(b). Thus the stage is set for the data-parallel execution of the y-sweep. In comparing the DO loop in the y-sweep in figures 2 and 4, note that the arrays on the right-hand side in the HPF implementation are transposes of those in shown in figure 2. The forcing function of the Poisson equation, represented by the array F, needs to be transposed only once, and, in the implementation here, the transpose is stored in the array FT. Note that in both the x- and y-sweeps, communication is required along the processor boundaries in order to evaluate the right-hand side in C(4,:,:). Finally, the subroutine THOMAS (not shown) is modified to include an outer DO loop which steps along the direction of the sweep.

We now return to the issue of comparing the convergence characteristics of the data-parallel and sequential implementations. Since the former does not use the updated values within a sweep, it may be expected that its convergence rate is inferior to that of the latter. Indeed, for $N_x = N_y$ and the specific equation considered here, it may be confirmed that the data-parallel implementation requires twice as many iterations as the sequential algorithm. The degradation in convergence rate, however, will generally depend on the equation being solved. We now seek to modify the data-parallel implementation to improve the convergence rate by incorporating the oft-used 'two-color' scheme. This method is based on the observation that in the x-sweep, the solution PSI(I,1:NY) for odd (even) I depends only on the values of PSI at the adjacent even (odd) I locations. Thus, if I is restricted to be either odd or even, the solution at these I locations may be executed in data-parallel mode. In this manner, the updated values at the odd locations are used immediately to evaluate the solution at the even locations and vice-versa. A similar scheme may also be adopted for the y-sweep. The modified x- and y-sweeps are shown in figure 5. With the above modification, the increase in the number of iterations to obtain a converged solution reduces from a factor of 2 for the version shown in figure 4 to 4/3. Moreover, this modification does not introduce any additional communication or other overhead. An added benefit of this implementation is that with some additional book-keeping, the storage requirements for C and D may be reduced by half.

Additional tuning of the ADI implementation in HPF is possible. One alternative is to execute the x- and y-sweeps m times before redistributing the data. Al-

```
....
      DO ICOLOR = 0,1
      DO I = 1+ICOLOR,NX,2     !x sweep
      C(1,I,1:NY) = DY2INV
      C(2,I,1:NY) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,I,1:NY) = DY2INV
      C(4,I,1:NY) = F(I,1:NY) - &
                    DX2INV*(PSI(I+1,1:NY)+PSI(I-1,1:NY))
      END DO
      CALL THOMAS (ICOLOR,NX,NY,C,BCBOT,BCTOP,PSI)
      END DO
....
      DO ICOLOR = 0,1
      DO J = 1+ICOLOR,NY,2 !y sweep
      C(1,J,1:NX) = DX2INV
      C(2,J,1:NX) = -2.0_FPNUM*(DX2INV+DY2INV)
      C(3,J,1:NX) = DX2INV
      C(4,J,1:NX) = FT(J,1:NX) - &
                    DY2INV*(PSI(J+1,1:NX)+PSI(J-1,1:NX))
      END DO
      CALL THOMAS (ICOLOR,NY,NX,C,BCLFT,BCRHT,PSI)
      END DO
...
```

Figure 5: Modification of x- and y-sweeps to enable data-parallel execution

though the convergence rate will degrade with increasing m, the cost of the overhead in redistributing data is amortized over a larger number of sweeps. The optimum value of m, which results in a converged solution in the minimum wall clock time, will depend on the equation being solved and the hardware and software characteristics of the machine on which the code is executed.

3. Panel Methods

Panel methods are effectively boundary element methods for Computational Fluid Dynamics problems. These methods employ the surface of the body as the computational domain rather than the entire flow region in which the body is immersed. This is not only computationally more efficient than a finite-difference method, for example, but it also allows more complicated body shapes to be studied that otherwise may not be tractable if the flow domain is discretized by a regular mesh.

Consider figure 6 which shows panels around an ellipse in a uniform incident velocity flow. Each k'th panel is centred around a control point at $\vec{r}_k$ and has a source density w_k. If the body is immersed in a uniform stream of velocity U_{inf} parallel to the x-axis, then the distribution of N source panels produces a potential given by[8]

$$\Phi(\vec{r}_k) = U_0 x_k + \frac{1}{2\pi}\sum_{j=1}^{N} w_j \int \ln |\vec{r}|_{k,j} ds_j, \qquad (6)$$

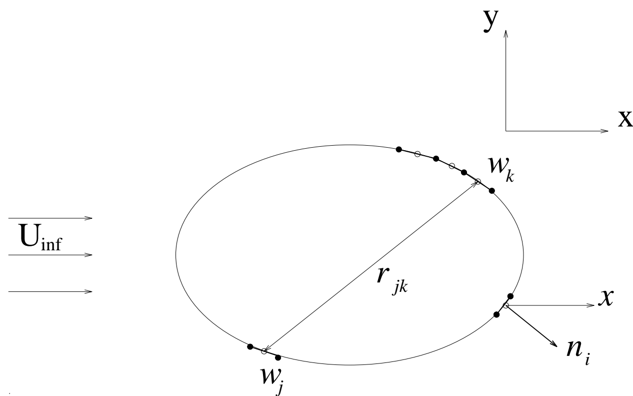

Figure 6: Flying ellipse

where $\vec{r}_k = (x_k, y_k)$ is the position of each panel's control point, $|\vec{r}|_{k,j}$ is the distance between two panels, and $w_k \int ds_k$ is the source strength of the k'th panel. The source densities are determined by applying the boundary condition of zero normal flow through the body surface, i.e.,

$$v_n = \frac{\partial \Phi}{\partial n_k} = 0. \qquad (7)$$

This generates a system of linear equations $A \cdot \vec{w} = \vec{b}$ with each component of A given by

$$A_{k,j} = \frac{\delta_{k,j}}{2} + \frac{1}{2\pi}\int \frac{\partial}{\partial n_k}(\ln r_{k,j}) ds_j, \qquad (8)$$

and the right hand side vector is simply $b_k = U_0 \sin \alpha_k$, where α_k is the angle between the panel and the x-axis. Once the vector of source densities is determined, the velocity field may be obtained from the potential given in equation 6. Although the calculation of the inviscid flow field past a body using the panel method consists of several steps, we consider here only the numerically intensive solution procedure for the dense matrix equation $A \cdot \vec{w} = \vec{b}$.

Consider first the actions being performed upon the matrix and the right-hand side as part of the LU solution procedure. The relative advantages and disadvantages of two possible arrangements, viz. *(i)* row distribution, and *(ii)* column distribution are discussed below. In the row distribution, the matrix rows are distributed between processors either according to BLOCK or CYCLIC structures as seen in figure 7. The determination of the pivot requires a distributed global test and the subsequent broadcast of the results. The pivot row is then exchanged and broadcasted so that it can be used in the elimination process. Note that if the rows are distributed in BLOCK structure, the load-balance is poor since the elimination of the final rows involves only a subset of the available processors

as indicated in figure 7. Despite reduced broadcast communication cost due to the reduction in the number of processors involved in the computation and the reduced computational load due to the shortening of the rows, there is still a significant load imbalance. This can be improved by using a CYCLIC distribution where alternate rows are on different processors. Here the computation is load-balanced until the number of rows remaining is less than the number of processors. A CYCLIC distribution also ensures that at each stage, the computation is load-balanced finely in contrast to the BLOCK distribution where each processor is expected to perform the elimination operation until the current row is part of the allocated set.

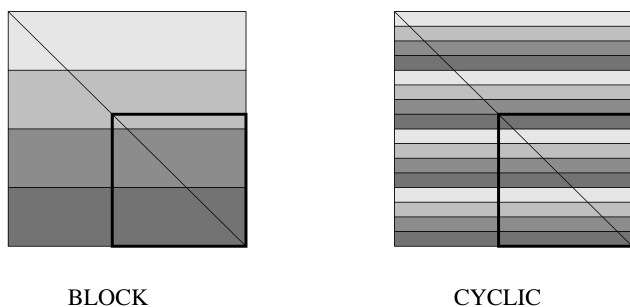

Figure 7: Row distribution

In the column distribution, the matrix columns are distributed as in figure 8. The global test to determine the pivot location is restricted to a single processor but the results must be broadcasted. The elimination process requires only the broadcast of a multiplying factor since all data for the elimination occur within columns. The same arguments regarding load-balancing and the relative merits of BLOCK and CYCLIC distributions apply here also and are illustrated in figure 8.

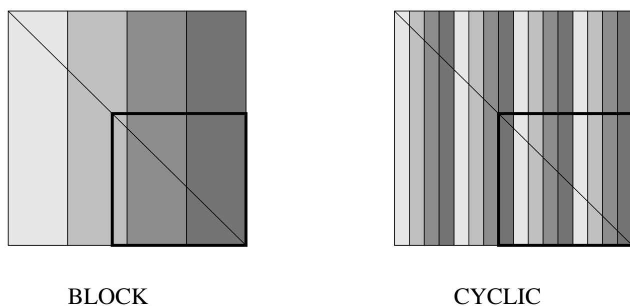

Figure 8: Column distribution

The difference between row and column distributions can be summarized as follows. The row distribution features a distributed global test for the pivot, whereas, for column distribution, the global test is poorly balanced but also requires no communication. The row decomposition requires the broadcast of a partial matrix row in comparison to the broadcast of a multiplication factor in the column decomposition. The choice between these different structures may depend on the typical matrix size, number of processors and relative communication costs.

The forward elimination stage of the solver in the Fortran 77 parent code is serial in nature as indicated in figure 9. Here the right-hand side is mapped according to the elimination stored within the lookup tables. The parallelism of the actual multiplication can be exploited, but this is a small fragment of the total work involved. The use of a distributed list also causes difficulties since the exchange of the entries in the right-hand side subsequent to the pivoting operation in the matrix must be performed in order. Note that the

```
! forward elimination:
      nm = n -1
      DO k=1,nm
        kp = k+1
        l = jpvt(k)
        s = rhs(l)
        rhs(l) = rhs(k)
        rhs(k) = s
        DO i=kp,n
          rhs(i) = rhs(i) + a(i,k) * s
        ENDDO
      ENDDO
```

Figure 9: Use of list to sort RHS in forward elimination.

inner loop over i can be expressed as a FORALL and is INDEPENDENT requiring a broadcast of the multiplication factor s. This, however, represents a very small and poorly load-balanced section of the algorithm.

The back substitution phase can be considered equivalent to the factorization in figure 10; however, there is no pivoting since only one row can perform the required elimination. For all distributions, this section is poorly load-balanced and generates a low ratio of computation to communication. The degree of parallelism could be increased, but this would require writing complex code or explicit knowledge of the data decomposition that cannot be easily generalized.

For both the forward elimination and back substitution, an alternative data distribuion could be considered. In figure 11, we show different possible data layouts for the back substitution phase. Note that a CYCLIC row distribution provides the best load-balancing for both matrix and right-hand side vector operations. The column distributions are poorly load-

```
f90 code
! back substitution:
      do ka=1,nm
        km = n - ka
        k = km + 1
        rhs(k) = rhs(k) / a(k,k)
        s = - rhs(k)
        do i=1,km
          rhs(i) = rhs(i) + a(i,k) * s
        enddo
      enddo
      rhs(1) = rhs(1) / a(1,1)
```

Figure 10: F90 code for back substitution.

balanced since a single processor is required to work on the entire right-hand side vector of all stages of the back substitution. It is also possible to mix vector and matrix distributions; indeed, the cost of performing a REDISTRIBUTE on the matrix may be too high, but performing a REDISTRIBUTE on the right-hand side vector may accrue a communication saving. The low level of computation alongwith the global nature and small size of the messages to be exchanged suggest that this would be highly dependent on the problem size and features of the target architecture.

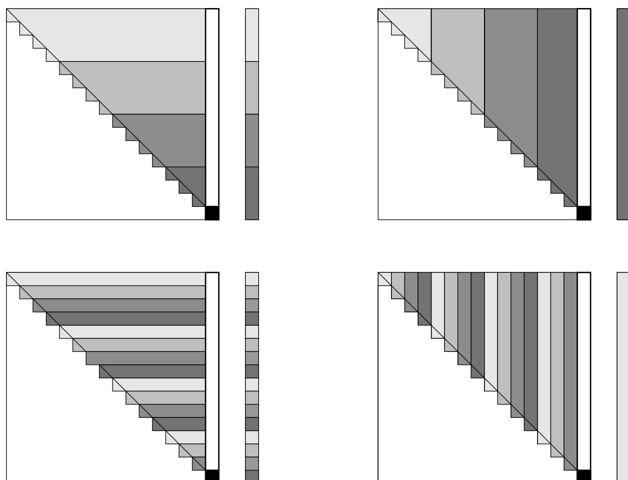

Figure 11: Substitution inline with elimination

4. Conjugate Gradient Methods

The classic Conjugate Gradient non-stationary iterative algorithm[9] 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 the matrix 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) $\vec{r}$:

$$\vec{x}^k = \vec{x}^{k-1} + \alpha^k \vec{p}^k \tag{9}$$

$$\vec{r}^k = \vec{r}^{k-1} - \alpha^k \vec{q}^k \tag{10}$$

where the new value of $\vec{x}$ is a function of its old value, α is the scalar step size, $\vec{p}^k$ is the search direction vector 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: $\| \vec{r}^k \| \leq \epsilon \cdot (\| A \| \cdot \| \vec{x}^k \| + \| \vec{b}^k \|)$, where $\| A \|$ denotes some *norm* of A and ϵ is a tolerance level. The CG algorithm uses

$$\alpha = \left(\vec{r}^k \cdot \vec{r}^k\right) / \left(\vec{p}^k \cdot A\vec{p}^k\right), \tag{11}$$

with the search directions chosen according to

$$\vec{p}^k = \vec{r}^{k-1} + \beta^{k-1}\vec{p}^{k-1} \tag{12}$$

with

$$\beta^{k-1} = (\vec{r}^{k-1} \cdot \vec{r}^{k-1})/(\vec{p}^{k-2} \cdot A\vec{p}^{k-2}) \tag{13}$$

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

The non-preconditioned CG algorithm is summarized 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, viz. $\vec{x}$, $\vec{r}$, $\vec{p}$ and $\vec{q}$ as well as the matrix A and working scalars α and β. Note that the work per iteration is modest, amounting to a single matrix-vector

product for $A \cdot \vec{p}$, two inner products $\vec{p}^k \cdot \vec{q}^k$ 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 a vector of length n.

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. 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 known[10] 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.

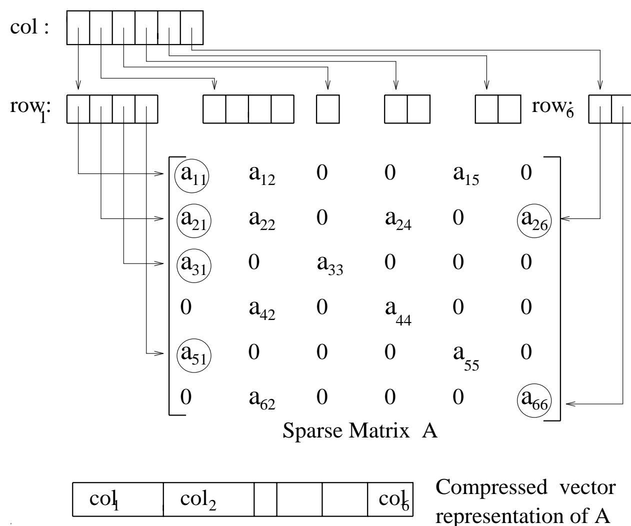

Figure 12: Compressed Sparse Column(CSC) representation of sparse matrix A.

The **Compressed Sparse Column** (CSC) storage scheme, shown in figure 12, uses the following three arrays to store an $n \times n$ sparse matrix with nz non-zero entries: *(i)* `A(nz)` containing the nonzero elements stored in the order of their columns from 1 to n, *(ii)* `row(nz)` that stores the row numbers of each nonzero element, and *(iii)* `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 13 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 13

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

performs three main computations: the vector-vector operations, inner product (here shown using BLAS routines) and the matrix-vector multiplication, shown explicitly in figure 14.

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

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.

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 (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.

It now remains to discuss 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 a 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. Here, we will describe two different data distribution schemes and show the associated costs of each.

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. 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 $p(i)$ are no longer necessarily assigned to the same processor, and thus requires communication before the multiplication.

In the first scheme, 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 15 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 in **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.

In the second scheme, the matrix A is 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 16, 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 scheme is easily generalized to the CSC format.

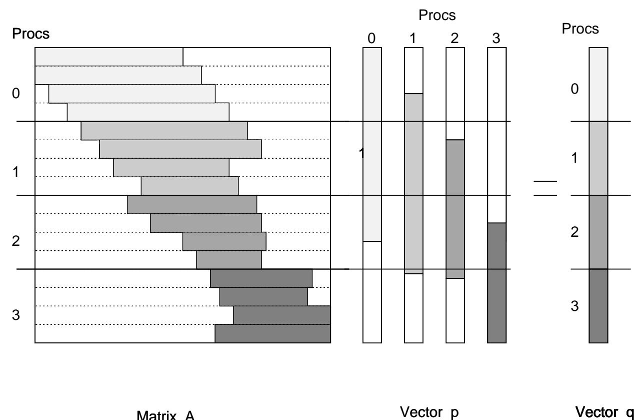

Figure 15: Communication requirements of Matrix vector multiplication where A is distributed in a (BLOCK, *) fashion.

In the computation phase, each processor performs an average of $m_z \times n/P$ 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

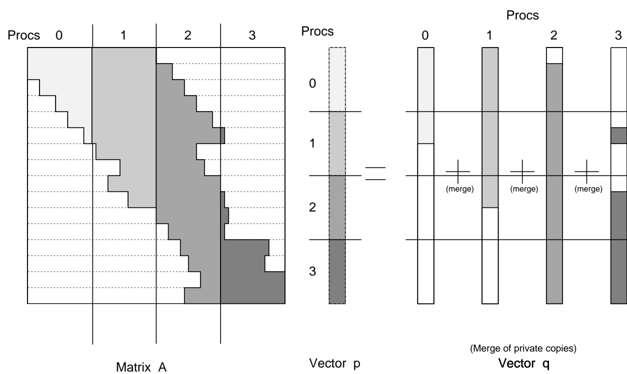

Figure 16: Communication requirements of Matrix vector multiplication where A is distributed in a (*, BLOCK) fashion.

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. 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. Clearly, 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 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 the iterations are independent of one another.

The HPF code for the CG algorithm for CSR format can be expressed as in figure 17.

```
      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 17: HPF version of sparse storage CG (CSR format).

A more extensive discussion of the Conjugate Gra-

dient method and High Performance Fortran may be found elsewhere[11,12].

5. Conclusions

We have illustrated some of the issues arising from the use of HPF for expressing algorithms in CFD applications. The advantages are the potential for faster computation on parallel and distributed computers, and additional code portability and ease of maintainence 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.

The basic concepts of HPF have been demonstrated through examples which are characteristic of current scientific and engineering codes. The removal of serial features from sections of code has been as important as adding parallelism, and the relative merits of alternative decompositions have been compared. The actual choice between the different decompositions and the remapping of data between the different stages of the algorithm will, in general, depend upon the problem sizes being considered and the performance of the `TRANSPOSE` intrinsic function (or the `REDISTRIBUTE` directive) for particular machines.

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.

Finally, we repeat the general observation 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.

It is a pleasure to thank T.Haupt, K.Dincer and S.Ranka for useful discussions regarding the work reported here.

References

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

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

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

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

[6] 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.

[7] 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.

[8] Fletcher, C. A. J., "Computation Techniques for Fluid Dynamics", Vol. II, Springer-Verlag, 1991.

[9] 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.

[10] 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.

[11] Hawick, K. A., Dincer, K., Robinson, G. and Fox, G. C., "Conjugate Gradient Algorithms in Fortran 90 and High Performance Fortran", Northeast Parallel Architectures Center Report No. SCCS-691, Syracuse University, 1995.

[12] Dincer, K., Hawick, K. A., Choudhary, A. and Fox, G. C., "High Performance Fortran and Possible Extensions to Support Conjugate Gradient Algorithms", Northeast Parallel Architectures Center Report No. SCCS-703, Syracuse University, 1995.