Exploiting High Performance Fortran for Computational Fluid Dynamics

K. A. Hawick, G. C. Fox and H. W. Yau

Northeast Parallel Architectures Center
Syracuse University, 111 College Place,
Syracuse, NY 13244-4100, USA.
Tel. 315-443-3933, Fax. 315-443-1973,
Email: hawick@npac.syr.edu

Abstract. We discuss the High Performance Fortran data parallel programming language as an aid to software engineering and as a tool for exploiting High Performance Computing systems for computational fluid dynamics applications. 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. In particular we use an implicit method such as the ADI algorithm to illustrate the major issues. We focus on regular mesh problems, since these can be efficiently represented by the existing HPF definition, but also discuss issues arising from the use of irregular meshes that are influencing 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

Successful implementations of future computational fluid dynamics (CFD) codes for large-scale aerospace systems require High Performance Computing and Networking (HPCN) [1] technology to provide faster computation speeds and larger memory. 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)[7] 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

[1] Known as High Performance Computing and Communications (HPCC) in the USA.

evolved from the experimental Fortran-D system [2] as a collection of extensions to the Fortran 90 language standard [10]. Many ideas were also absorbed from the Vienna Fortran System [3]. We do not discuss the details of the HPF language here as they are well documented elsewhere[8]. HPF 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. We have already conducted a study of the general suitability of the HPF language for CFD[1] 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 and IBM's SP2 we are now able to describe specific coding issues.

We employ an ADI algorithm applied to a steady flow problem for illustrative purposes. There are some conflicts between the optimal data decomposition and the computational structure of the algorithm. We show how the data `DISTRIBUTE` and data `ALIGN` directives of HPF can be used to resolve this conflict. Full matrix algorithms can also be implemented in HPF, although at the time of writing, the optimal algorithms for full matrix solution by LU factorisation, for example, are only available as message-passing based library software such as ScaLAPACK[4]. We are currently investigating how scalable algorithms such as these can either be expressed as HPF code directly, or as HPF-invocable run-time libraries. 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, however[6].

For CFD simulations, 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 [5].

2 CFD using HPF

There are many formulations for fluid dynamics equations that give rise to a computationally manageable form. The example system we use here involves the Poisson and vorticity transport equations that arise from the unsteady Navier Stokes equations for incompressible flows, where the velocity has components (u, v), ψ is the stream-function and ζ the vorticity [11]. These equations can be expressed as Poisson equations in spatial terms with the time dependence separated:

$$\frac{\partial \zeta}{\partial t} = \frac{1}{\mathrm{Re}} \nabla^2 \zeta + \left(\frac{\partial(\psi)}{\partial y} \frac{\partial(\zeta)}{\partial x} - \frac{\partial(\psi)}{\partial x} \frac{\partial(\zeta)}{\partial y} \right) \tag{1}$$

and

$$\frac{\partial \psi}{\partial t} = \nabla^2 \psi - \zeta \tag{2}$$

coupled together.

A common approach to this problem is to use a split operator technique such as the Alternating Direction Implicit (ADI) method [9]. This technique assumes that the five star stencil operator $\mathcal{L}$ for the finite difference scheme can be split into two line operators, such that for the simpler problem of equation (2) we have:

$$\mathcal{L}_x \psi = -\psi_{i+1,j} + 2\psi_{i,j} - \psi_{i-1,j} \tag{3}$$

$$\mathcal{L}_y \psi = -\psi_{i,j+1} + 2\psi_{i,j} - \psi_{i,j-1} \tag{4}$$

for two dimensions (or three operators in 3d). A very similar treatment can be carried out for equation (1), but we shall continue with the simpler problem to illustrate the points we wish to make. The computation for the ADI scheme requires the solution of two (in 2d) or three (in 3d) matrix equations for one (two) intermediate finite difference step(s). Labeling the iterations by n, this can be expressed as:

$$\frac{\psi^{n+1} - \psi^n}{\triangle t/2} = -\frac{\mathcal{L}_x \psi^{n+1} + \mathcal{L}_y \psi^n}{h^2} - \zeta \tag{5}$$

$$\frac{\psi^{n+2} - \psi^{n+1}}{\triangle t/2} = -\frac{\mathcal{L}_x \psi^{n+1} + \mathcal{L}_y \psi^{n+2}}{h^2} - \zeta \tag{6}$$

where, h and $\triangle t$ are the finite differences in space and time respectively. Note that we have introduced an artificial time variable to construct an iterative algorithm using $\frac{\triangle \psi}{\triangle t} = \nabla^2 + \zeta = 0$. In this formulation, we expect $\triangle \psi$ to tend to zero (within some convergence criterion) after a certain number of iterations. This is a somewhat artificial construct, but we are only using this formulation to illustrate the computational structure and issues of implementation. Expressing this in matrix notation:

$$(\mathbf{L}_x + \frac{2h^2}{\triangle t}\mathbf{I}) \cdot \psi^{n+1} = (\frac{2h^2}{\triangle t}\mathbf{I} - \mathbf{L}_y) \cdot \psi^n - h^2 \zeta \tag{7}$$

$$(\mathbf{L}_y + \frac{2h^2}{\triangle t}\mathbf{I}) \cdot \psi^{n+2} = (\frac{2h^2}{\triangle t}\mathbf{I} - \mathbf{L}_x) \cdot \psi^{n+1} - h^2 \zeta \tag{8}$$

where $\mathbf{I}$ is the identity matrix. We solve equation (7) for ψ^{n+1}, given ψ^n and equation (8) for ψ^{n+2}. This cycle of solving both matrix equations constitutes a complete iteration of the ADI algorithm. For this scheme, the matrices on the left hand sides of equations (7) and (8) are tridiagonal and we can represent these by three vectors $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Note that although we visualize ψ as a scalar field in 2d, for a 2d problem, and thus we imagine indexing its finite difference representation by (i, j) for the (x, y) co-ordinates, from the point of view of matrix equations (7) and (8), ψ is just a long vector that happens to be partioned by a nested pair of indices (i, j). We can therefore write a Fortran 77 code to illustrate the tridiagonal solver for matrix equation (7) as shown in figure 1.

```
   REAL PSI(M1,M2), RHS(M1,M2), TMP(M1,M2)
   BETA = B
   PSI(1,1) = RHS(1,1) / BETA
   DO 30 J = 1,M2
     DO 10 I = 2,M1
       TMP(I,J) = C / BETA
       BETA = B - A * TMP(I,J)
       PSI(I,J) = ( RHS(I,J) - A * PSI(I-1,J) ) / BETA
10   CONTINUE
     DO 30 I = M1 -1, 1, -1
       PSI(I,J) = PSI(I,J) - TMP(I+1,J) * PSI(I+1,J)
20   CONTINUE
30 CONTINUE
```

Fig. 1. Tridiagonal solver for ADI Poisson equation in Fortran 77.

This has two loops in I, one for decomposition and forward substitution, and one for backsubstitution. These loops contain a dependency on the previous values of I, but are independent of J. An identical code fragment can be used for equation (8) but with I and J `DO loop` variables interchanged. Note that the three vectors a, b, c of the tridiagonal matrix can be written here as constants for the simple linear case shown. They have the following values for the ADI solution to the Poisson equation: $A = C = -1$, $B = 2 + \frac{2h^2}{\Delta t}$. Now consider how this may be decomposed under the data parallel programming model, so that the ψ data array can be distributed across the memory of a number of processors in a parallel or distributed computing system.

In figure 2 the two data decompositions are shown for equations (7) and (8) where for simplicity, $M1 = M2 = 16$ is the number of finite difference points in the mesh for ψ and $P = 4$ is the number of processors in the parallel or distributed architecture. These would normally be set by Fortran `PARAMETER` statements. The optimal decomposition for one of the equations is to give each processor a "strip" of the data. This can be accomplished using the fragment of HPF code in figure 3.

This illustrates how the programmer may impart extra information about his target system to the compiler, to obtain an efficient executable code for that system. These directives are implemented using the Fortran 90 comment symbol (an exclamation mark) so that a non-HPF compiler would ignore the directives as comments. The `PROCESSORS` directive is used here to hint to the compiler that the processors in the system should be treated as connected in a one dimensional

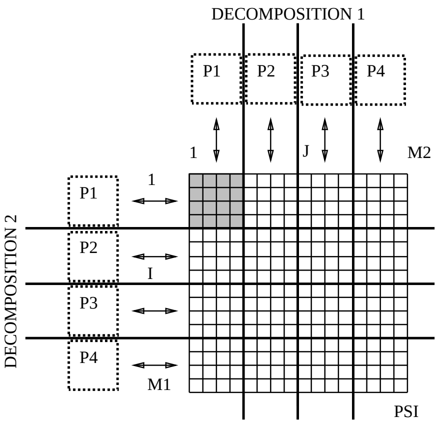

Fig. 2. Optimal Data Decompositions for Matrix equations in ADI Solver.

```
!HPF$ PROCESSORS PROC(P)
!HPF$ TEMPLATE(M1,M2)
!HPF$ DISTRIBUTE TEMPLATE(*,BLOCK) ONTO PROC
      REAL PSI(M1,M2), RHS(M1,M2), TMP(M1,M2)
!HPF$ ALIGN WITH TEMPLATE :: PSI, RHS, TMP
      BETA = B
      PSI(1,1) = RHS(1,1) / BETA
      FORALL J=1:M2
        DO I = 2,M1
          TMP(I,J) = C / BETA
          BETA = B - A * TMP(I,J)
          PSI(I,J) = ( RHS(I,J) - A * PSI(I-1,J) ) / BETA
        END DO
        DO I = M1 -1, 1, -1
          PSI(I,J) = PSI(I,J) - TMP(I+1,J) * PSI(I+1,J)
        END DO
      END FORALL
```

Fig. 3. Tridiagonal solver for ADI Poisson equation in HPF.

vector. Furthermore, a `TEMPLATE` mapping directive sets up the data distribution onto the memory of those processors. In this case an asterisk denotes ordinary Fortran serial order for dimension one, and a `BLOCK` distribution in dimension 2 as indicated in decomposition 1 of figure 2. Aside from using `END DO` statements instead of labeled `CONTINUE` statements, the `DO loops` over I are unchanged from the Fortran 77 case. The J loop is now a `FORALL` statement, which is an indication to the compiler that the code for each J value may be done in **any** order as far as the computation is concerned, and that the compiler should decide to use the owning processors to carry out the calculations associated with their own range of J values. Specifically, as shown, processor 1 will carry out calculations for $J = 1, 2, ..4$, processor 2 will compute for $J = 5, 6, ..8$, etc.

An alternative implementation is to leave the Fortran 77 code unchanged entirely, but use the **INDEPENDENT** directive of HPF applied to the outermost loop as an assertion to the compiler that "iterations" over that loop may be executed in parallel. See [7, 8].

Our example unfortunately illustrates that a different decomposition is optimal for the two matrix equations. One solution to this problem is to apply a data **TRANSPOSE** after each equation is solved. This will leave the data in the optimal decomposition for the next equation. The **TRANSPOSE** function for a 2d array is provided as an **INTRINSIC** function in Fortran 90 and HPF. This at least abstractifies the problem to one for the system suppliers — namely to provide an optimized **TRANSPOSE** function which can make use of decomposition information known at compile time. An alternative to the **TRANSPOSE** intrinsic, is to invoke a data **REDISTRIBUTE** directive after solution of each matrix equation has been solved. This also allows the code to be trivially extended to three dimensions, despite the fact that the **TRANSPOSE** intrinsic is for 2d matrices only. The **REDISTRIBUTE** invocation for our 2d code is shown in figure 4.

```
!HPF$ PROCESSORS PROC(P)
!HPF$ TEMPLATE(M1,M2)
!HPF$ DYNAMIC, DISTRIBUTE(*,BLOCK) ONTO PROC :: TEMPLATE
      REAL PSI(M1,M2), RHS(M1,M2), TMP(M1,M2)
!HPF$ DYNAMIC, ALIGN WITH TEMPLATE :: PSI, RHS, TMP
      DO ITER,1,NITER

!  ... solve 1st Matrix eqn. using code above ...
!  ...
!HPF$ REDISTRIBUTE TEMPLATE(BLOCK,*) ONTO PROC

!  ... solve 2nd Matrix eqn. with I and J loops inverted.
!  ...
!HPF$ REDISTRIBUTE TEMPLATE(*,BLOCK) ONTO PROC

      IF( convergence test ) EXIT
      END DO
```

Fig. 4. HPF code structure for full ADI Poisson solver in 2d.

This code also illustrates the Fortran 90 **EXIT** statement, which may be used to exit the `DO loop` once some convergence criterion has been met. Note that the **DYNAMIC** attribute is necessary for the data items which may be redistributed.

A disadvantage of a parallel code is that additional data storage is required.

Although we have written the Fortran 77 code in a similar manner to our HPF code using full matrices to store RHS and TMP, these could be reduced to scalars for a serial code. It is one of the tradeoffs in programming parallel and distributed computing systems, that individual processors need their own private workspace to allow separate parts of the computation to proceed in parallel. Fortunately, such systems generally have access to much larger amounts of memory than do serial systems. Furthermore, it is possible to reuse storage space between parts of a full application code using the dynamics memory `ALLOCATE` and `DEALLOCATE` statements in Fortran 90 and HPF. Dynamically allocatable data can also be `DISTRIBUTE`d and `REDISTRIBUTE`d.

In this description we have shown how HPF allows the programmer to control the data distribution amongst processors and the parallel constructs that are thus enabled. It should be noted that the `TRANSPOSE` and `REDISTRIBUTE` constructs are somewhat expensive in communications requirements and therefore make considerable demands on the underlying message-passing system of the HPF run-time library. Nevertheless we feel this abstraction of the algorithm's communications into the systems-code makes a substantial contribution to application code maintainability.

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 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.

References

1. 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,
2. 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.
3. Chapman, B., Mehrotra, P., Mortisch, H., and Zima, H., "Dynamic data distributions in Vienna Fortran," *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.284.
4. Choi,J., Dongarra, J.J., Pozo, R., and Walker, D.W., "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers", In Proc. of the Fourth Symposium on the Frontiers of Massively Parallel Computation, PP 120-127. IEEE Computer Society Press, 1992.

5. 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.
6. Hawick, K.A., Dincer, K., Choudary, A., Fox, G.C., "Conjugate Gradient Algorithms Implemented in High Performance Fortran", NPAC Technical Report, SCCS 639, October 1994.
7. High Performance Fortran Forum (HPFF), "High Performance Fortran Language Specification," *Scientific Programming*, vol.2 no.1, July 1993. Also available by anonymous ftp from ftp.npac.syr.edu (`cd /HPFF`).
8. Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, G.L., Zosel, M.E., "The High Performance Fortran Handbook", MIT Press 1994.
9. Leca, P., Mane, L., "A 3-D Algorithm on Distributed Memory Multiprocessors", in "Parallel Computational Fluid Dynamics", Simon, Horst D. (Editor), MIT Press 1992, PP149-165.
10. Metcalf, M., Reid, J., "Fortran 90 Explained", Oxford, 1990.
11. Tritton, D.,J., "Physical Fluid Dynamics", Oxford Science Publications, 1987.