NPAC Technical Report SCCS 709

Fortran 90 and High Performance Fortran for Dense Matrix-formulated Applications

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

May 19, 1995

Abstract

We discuss the use of the Fortran 90 (F90) and High Performance Fortran (HPF) computer programming languages for use in effectively-dense matrix-formulated applications. We provide a brief introduction to the main features of the languages and present a computational fluid dynamics application which uses the panel method as an example which can expressed in both F90 and HPF to evaluate the languages.

We consider issues such as computational performance and efficiency on High Performance Computing and Communications systems as well as other important matters such as ease of expression and code maintainance and code extension.

Further details regarding HPF standards, tutorials and example programs can be found on the World Wide Web at http://www.npac.syr.edu/hpfa.

Introduction

There is a class of problems in computational science and engineering which require formulation in full matrix form and which are generally solved as dense matrices either because they *are* dense or because the sparsity can not be easily exploited. Several problems in computational electromagnetics (CEM) [5] and computational fluid dynamics (CFD) [15] are what we term "effectively dense" [4]. We focus on the panel method [14] as employed in CFD as an example of such a problem that can be used to evaluate the Fortran 90[13] and High Performance Fortran languages [10].

Panel methods are widely used in the aerospace and automotive industry and are effectively boundary-element methods for computational fluid dynamics problems. These methods employ the surface of the body over which fluid is flowing to be used as the computational domain rather than using the whole region in which the body is embedded. This is not only computationally more efficient than a finite difference method for example but also allows more complicated body shapes to be studied than would be tractable if the body were embedded in a regular mesh.

Of particular interest in the aerospace industry is the flow past a streamlined isolated body such as an aerofoil and the pressure distribution around such a body that gives rise to lift forces on the body. For explanatory purposes we consider the simpler example of an elliptical body.

The panel method is so named due to the subdivision of the body surface into a number of contiguous "panels". Each panel has associated with it a source density, the strengths of which are determined as an intermediate part of the solution method.

Consider figure 1 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 embedded in a uniform stream of velocity U_0 parallel to the x-axis, then the distribution of N source panels produce a potential:

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

where $\vec{r}_k = (x_k, y_k)$ is the position of each panel's control point; $\vec{r}_{k,j} = \sqrt{(x_k - x_j)^2 + (y_k - y_j)^2}$ is the distance between two panels; and $w_k \int ds_k$ is the source strength of the k'th panel.

The source densities must be chosen so that the

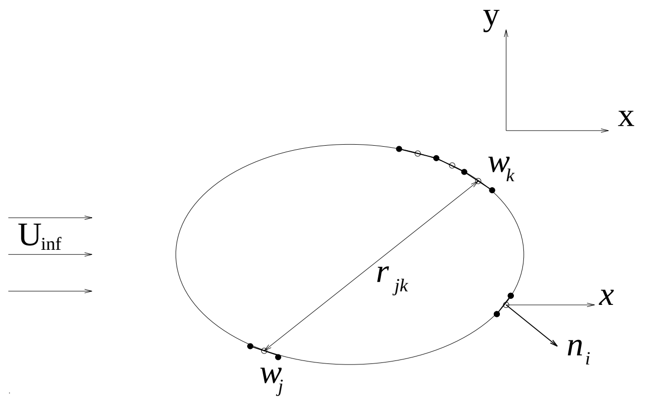

Figure 1: Ellipse in uniform incident fluid flow

boundary condition of zero (normal) flow through the body surface is satisfied. This can be expressed as:

$$v_n = \frac{\partial \Phi}{\partial n_k} = -U_0 \sin \alpha_k + \frac{1}{2\pi} \sum_{j=1}^{N} w_j \int \frac{\partial}{\partial n_k} (\ln \vec{r}_{k,j}) ds_j \equiv 0 \tag{2}$$

where α_k is the angle between the panel and the x-axis. This generates a system of linear equations $A \cdot \vec{w} = \vec{b}$ with each component of A given by:

$$A_{k,j} = \frac{\delta_{k,j}}{2} + \frac{1}{2\pi} \int \frac{\partial}{\partial n_k} (\ln r_{k,j}) ds_j \tag{3}$$

and the right hand side vector is simply $b_k = U_0 \sin \alpha_k$, and $\vec{w}$ the vector of unknown source densities.

The velocity field $\vec{v}(\vec{r})$ can be obtained from the solution source vector as follows:

$$\begin{aligned} u(x,y) &= \frac{1}{2\pi} \sum_{j=1}^{N} w_j \int \frac{x - x_j}{(x-x_j)^2 + (y-y_j)^2} ds_j \\ v(x,y) &= \frac{1}{2\pi} \sum_{j=1}^{N} w_j \int \frac{y - y_j}{(x-x_j)^2 + (y-y_j)^2} ds_j \end{aligned} \tag{4}$$

The complete velocity field solution is $\vec{V} = \vec{U_0} + \vec{v}$. Furthermore, the surface pressure distribution P can be obtained from the Bernoulli equation [15], thus:

$$C_P = \frac{2(P - P_0)}{\rho U_0^2} = 1 - (\frac{\vec{V}}{U_0})^2 \tag{5}$$

Matrix Solution Method

Here the dense matrix obtained from the assembly process is solved by LU decomposition [6]. One implementation is illustrated in the F90 code and consists of three distinct actions:

- converting the matrix into a diagonal form where the lower part is zero. Here the row used to eliminate lower diagonal parts is chosen to improve numerical stability, a process known as pivoting.
- forward elimination where the transformations and multiplications performed on the matrix are repeated on the RHS vector.
- taking this form and removing the above diagonal terms by back substitution.

Solution of this diagonal matrix is now straightforward. The conversion to the diagonal form requires order n^3 operations and is the most numerically intensive part of the computation, the latter stages of forward elimination and backsubstitution taking order n^2 operations.

The preceding operations are dominated by memory access operations rather than floating point operations. The entire operation possesses a considerable data dependence which equates to a serial dependence, there is little freedom in the ordering of operations. Often data is used in groups or blocks to try and minimise transfers [8, 7, 12] which are particularly important in modern hierarchical architectures or distributed memory architectures. Also the above formulation of the algorithm can make use of optimised BLAS subroutines or indeed the algorithm could simply use a prepackaged ScaLAPACK solver.

HPF Implementation

HPF is derived from the experimental Fortran D language [2] and has been described as implementing a data-parallel, owner-computes model. Whilst this does describe the behaviour in simple cases it should be realised that there are significant differences between HPF and earlier data parallel languages. HPF has many features designed to support the arrangement and rearrangement of data and computations so that the locality of data is exploited in as efficient and portable manner as possible. HPF directives should allow the compiler to produce efficient and correct code for a variety of architectures and processor numbers, the only requirement being recompilation of source code. A greater discussion of the philosophy of HPF and the language itself can be found in HPF standards [10] and books [11].

The key features of the panel methods computation and the various aspects of HPF will be discussed below.

Data Layout

The two most important parts of the panel method code are the application of the physical equations and the assembly and solution of the resulting matrix. Note that different sections of the code may suit different layouts and that there may be considerable interaction between sections. Due to the owner computes rule of HPF it is best to use arrays of exact size when known, otherwise there may be alignment or loadbalance problems later, although for some problems we can foresee the use of padded arrays to ensure alignment. ALLOCATE can be used to create work arrays as needed. The DISTRIBUTE or ALIGN directives should be used to ensure that variables used in the same expression share data layouts. In all but the most trivial cases different data distributions will interact. Communication as determined by the compiler can occur or TRANSPOSE or REDISTRIBUTE and REALIGN directives can be employed. The high communication cost of such a global data rearrangement should only be undertaken when converting *to* a lower or *from* a high communication cost distribution. It is a compromise between the cost of data rearrangement and the cost of performing data communications within the subroutines.

Expressing Parallelism

It should be noted that FORALL is not just a parallel DO loop. Statements within a FORALL are executed independently and in no fixed order. A full description and explanation is given in [11]. The INDEPENDENT directive can be used to provide additional reassurances to the compiler that the activities within the FORALL can be executed without need to consider interactions. Examples of FORALL usage in place of DO loops be illustrated in sections of the panel code. In addition to the FORALL statement the array syntax notation of F90 is used and here the execution order of any array syntax statement cannot be assumed.

Coding Issues

In this section the various issues raised for each major part of the panel code will be discussed. Since the panel code is similar to many other scientific and engineering codes the solution of the matrix dominates the computation cost and will be addressed in the greatest depth.

Matrix Solution.

Here we must consider the actions being performed upon the matrix and the right hand side (RHS) as part of the LU solution procedure. The relative advantages and disadvantages of two possible arrangements will be discussed below.

Row distribution.

In the case of a row distribution the matrix rows are distributed between processors either according to BLOCK or CYCLIC structures as shown in figure 2.

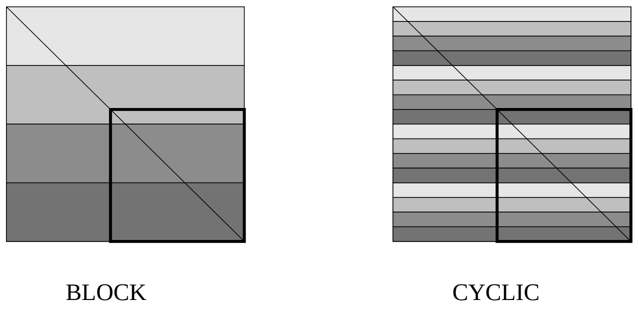

Figure 2: Row Distribution

The determination of the pivot requires a distributed global test and the broadcast of the results. The pivot row is then exchanged and broadcast so it can be used in the elimination process. Note if the rows are distributed in BLOCK structure the loadbalance is poor. Here the subset of the matrix referenced at an intermediate stage of the diagonalisation is identified by the smaller square in figure 2 showing how some processors are idle. Whilst this reduction in the number of processors involved in the computation may be rewarded in reduced broadcast communication costs and the computational demand lessens as each row becomes shorter due to the elimination process there is still a significant load imbalance. This can be improved by using a CYCLIC distribution where alternate rows are on different processors. A CYCLIC distribution also ensures that at each stage the computation is loadbalanced 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.

Column Distribution

In the case of the column distribution, the matrix columns are distributed across processors as shown in figure 3.

The global test to determine the pivot row location is restricted to a single processor but the results must still be broadcast. The exchange of the pivot is an entirely local process. The elimination process requires only the broadcast of a multiplying factor since

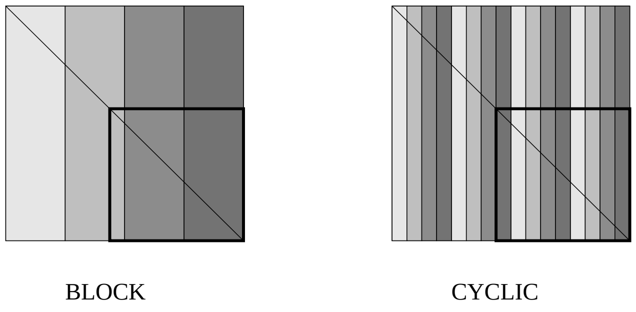

Figure 3: Column distribution

all data for the elimination occurs within columns. The same arguments regarding loadbalancing and the relative merits of BLOCK and CYCLIC distributions discussed for the case of row distribution apply here also and are illustrated in figure 3.

Comparison of Distributions

The difference between row and column distributions can be summarised as follows. The row distribution features a distributed global test for the pivot row whereas column distribution the global test is poorly balanced not being distributed, and 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 many depend on the typical matrix size, number of processors and relative communication costs of the architecture and software.

The demands of the decomposition here are quite different to those involved with any code requiring the multiplication of the matrix and vector. A description of the issues arising in this case can be found in [9].

Forward Elimination and Back Substitution.

The factorisation of the matrix into upper diagonal form is the expensive part of the algorithm requiring of order n^3 operations. However in any parallel code any feature which does not parallelise well will degrade performance. The forward elimination and back substitution stages can be such processes.

The *forward elimination* stage of the solver in the F90 code is serial in nature, figure 4. Here the RHS is mapped according to the elimination sequence stored within lookup tables. The parallelism of the actual multiplication can be exploited but is a small fragment of the total work involved. Use of a distributed list also causes problems since the exchange of the entries in the RHS follows the pivoting operation in the matrix and must be performed in sequence.

```
! 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 4: *Use of list to sort RHS in forward elimination.*

Note that the inner loop over i can be expressed as a FORALL construct and is INDEPENDENT requiring a broadcast of the multiplication factor s. This however represents a very small and poorly loadbalanced section of the algorithm.

Since the elimination is performed on both the matrix and the vector RHS, a simple modification to incorporate the vector as an additional column within the matrix can be made. A common parallel optimisation is of performing two or more identical computations inline for the cost of one communication event. The RHS data layout will mirror that of the matrix rows and this may require some remapping for later stages of the code.

The *back substitution* phase can be considered as an equivalent to the factorisation, figure 5. There is no pivoting since only one row can perform the necessary elimination. For all distributions this section is poorly loadbalanced and generates a low ratio of computation to communication. The degree of parallelism could be increased but this would involve complex coding or explicit knowledge of the data decomposition or processor number.

Note again that the inner loop over i can be expressed as a FORALL and is INDEPENDENT requiring a broadcast of the multiplication factor s and as with the forward elimination is a small and poorly loadbalanced workunit. The data transfers involved would be improved by selecting a different data decomposition to that in the factorisation stage. Figure 6 shows possible decomposition of both vector and matrix.

```
! 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 5: *F90 code for back substitution.*

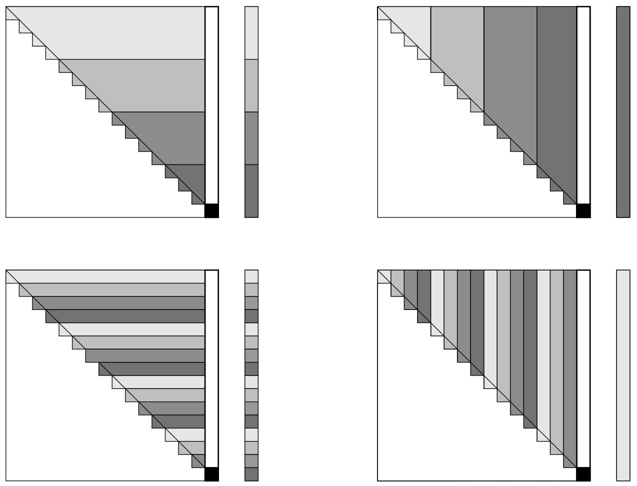

Figure 6: Substitution inline with elimination

Matrix Assembly

The assembly of the matrix simply of following the procedure outlined in the introduction to panel methods. Here the majority of data references are local however there are two patterns of reference which are difficult to express in HPF. As described in the introduction, panel methods involve the interaction of all panels and produce a dense matrix. The production of these terms results in a loop similar to that shown in figure 7, where some code has been omitted for clarity.

Many options exist for the distributions of *xc, yc*. The preferred distribution of the two dimensional arrays will determine the HPF loop order to reduce communication and that some re-expression of the structure will be required. The need to compute *xd* for all values of *i* and *j* will always place a communication cost on this section, either through communication during the computation or in a replication of one array depending on which loop if converted to a FORALL. It is assumed that the second dimension of *fn* and *ft* is distributed so

```
! F90 version

      DO k=1,n
        DO j=1,n
          IF( k .ne. j)THEN
            xd = xc(k) - xc(j)
            yd = yc(k) - yc(j)
            ........
            ukj = qt * ci(j) - qn * si(j)
            vkj = qt * si(j) + qn * ci(j)
            fn(k,j) = - ukj * si(k) + vkj * ci(k)
            ft(k,j) =   ukj * ci(k) + vkj * si(k)
          ENDIF

        ENDDO
      ENDDO


! HPF version

      DO k=1,n

      FORALL (j=1:n, k/=j)
        xd(j) = xc(k) - xc(j)
        yd(j) = yc(k) - yc(j)
      ENDFORALL

      FORALL (j=1:n, k/=j)
        ........
        ukj(j) = qt(j) * ci(j) - qn(j) * si(j)
        vkj(j) = qt(j) * si(j) + qn(j) * ci(j)
      ENDFORALL

      FORALL (j=1:n, k/=j)
        fn(k,j) = - ukj(j) * si(k) + vkj(j) * ci(k)
        ft(k,j) =   ukj(j) * ci(k) + vkj(j) * si(k)
      ENDFORALL

      ENDDO
```

Figure 7: *Part of matrix assembly code*

FORALLs are constructed to use this dimension. The loops could easily have been inverted in the first dimension is distributed. In HPF if no DISTRIBUTE or ALIGN directive is used an array is assumed to be replicated.

Input of Geometry and Output of Results

For this example the geometry of the test problem is defined within the code, rather than from an input file. It is interesting to note that the use of a lateral symmetry results in poor loadbalance and that a more efficient solution is to compute the geometry on all nodes ignoring the potential for reduced computation since this will require communication later.

The solution of the matrix is only part of the computation. To obtain velocities at the desired points further computation is required. Here I/O is involved since a list of x,y pairs must be read and then the contribution from each panel summed. This is a simple process and the matrix solution provides these terms and a FORALL can be constructed. The summation process can use an intrinsic function. The value for each x,y coordinate pair must be replicated since final computation involves similar operations to the matrix assembly. The design of optimised input/output, [1] is outside the scope of this paper and we do not discuss it further.

Data Rearrangements

For both forward elimination and back substitution an alternative data distribution could be considered. In figure 6 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 RHS vector operations. The column distributions are poorly load-balanced since a single processor is required to work on the entire RHS 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 by performing a REDISTRIBUTE on the RHS vector a communication saving may be obtained. The low level of computation combined with the global nature and small size of the messages to be exchanged suggests this would be highly dependent on both problem size and upon features of the target architecture.

Higher Dimensionality of Decomposition.

Only distributions of one dimension of the matrix have been discussed. The difficulty of this is that if two dimensions of the matrix are distributed the associated vectors cannot be aligned with all vector parts. The vector could be replicated across all processors but the cost of maintaining such updated replicants can be high. In this case much of the data needed for maintenance of the vector is broadcast as part of the pivot elimination operation. The loadbalancing of the algorithm for higher dimensionality is considerably poorer than with a single dimension, figure 8.

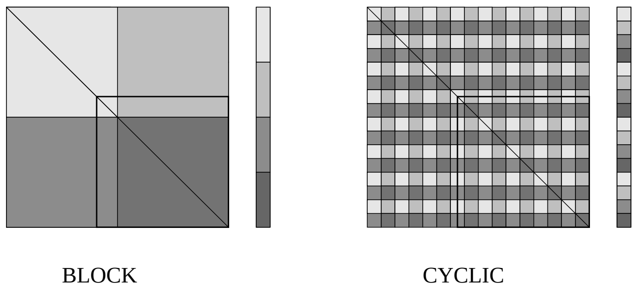

Figure 8: Higher dimension distributions

Discussion

The greatest improvement in performance can be obtained by removing any serial code. Some parts of any algorithm may require recoding or the selection of an alternative algorithm. LU factorisation possesses a sequential dataflow, the elimination of rows at any one stage can be performed in parallel but these eliminations must be performed in a strict sequence. Here we shall discuss several examples encountered in the coding of the panel algorithm.

Determination of Pivot

In Fortran the use of a serial DO loop is a natural method to determine the values and locations of extreme bounding values or to determine summations. However in HPF such a DO loop would be processed serially and no advantage is taken of the parallelism, the MAXLOC intrinsic can be used. These two code fragments are shown in figure 9. In this example, data has been copied to a temporary work vector so that the ABS function can be applied. Here the variable *atmp* would be aligned with *aa* so that no data transfer would occur. Perhaps the list of intrinsic functions should be expanded so that ABSMAXLOC is included, or a more general calling interface so that any PURE function can be used in a reduction operation rather than just MAX and MIN. MAXLOC is a generalisation of the F90 intrinsic. There are other intrinsics described in the HPF standard [10] and book [11].

In the particular case of the pivot operation it is interesting to note that the actual exchange of rows is to improve the numerical stability and accuracy of the computation. In some cases the actual pivoting might not be necessary and could save a considerable parallel overhead in terms of the requirement for global reduction operations.

Other uses of such global reductions can be found in the code for example, where a loop is used to de-

```
f90 code

        iset=ks
        DO i=kp,n
          IF( abs( aa(i,k)) > abs(aa(iset,k))) iset = i
        ENDDO


HPF code

!
        atmp=0.0d0
        atmp(kp:n)= aa(kp:kn,k)
        atmp(:)= abs(atmp(:))
!
        iset = maxloc(atmp, dim=1)
```

Figure 9: *example of intrinsic replacing serial DO loop*

termine if some error condition has occurred and a flag set. Again these loops will be processed serially unless expressed as a FORALL or a reduction operator is employed.

Numerical Accuracy and Intrinsic Functions

Methods which involve a significant amount of numerical computation on a small number of variables can often exhibit sensitivity to rounding error. There are two examples in the panel code, the use of MAXLOC to determine the pivot and then perform an elimination and the use of SUM in computing the velocity at desired points.

The distributed nature of the MAXLOC intrinsics can lead to different solutions being obtained. Here the actual value of the pivot exchanged is identical, but if multiple entries exist in any one column with the same absolute numerical value the selected pivot will depend on the order of processing. This can cause numerical differences since there are considerable dependencies on dataflow in direct solution methods like GE and LU.

In the case of the SUM intrinsics if the result were to be assembled as a partial sum and these summed during the merging of private copies numerical differences could occur. The sum of partial sums

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

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

Both these and other reduction operations 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. It will only influence parallel performance if the data volume is considerable. It is problematic in using a high level construct such as the INTRINSIC functions 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 it 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.

FORALL and DO Loops

As has been noted the FORALL loop construct is not a parallel DO loop. For example, in figure 10 the pivot interchange and elimination is shown in F90, HPF and FORALL and array syntax. The outer loop must be performed in sequence since it modifies data references in the next iteration. The inner loops can be performed independently and replaced by a FORALL or an array syntax expression. In HPF, FORALL loops will be analysed to ensure correct execution in parallel. The INDEPENDENT directive many be required to express true independence in particular if indirect addressing is used.

Both the FORALL and array syntax expression provide an elegant formulation of the expressions involved often being equivalent to the suffix notation in the algebra. This may produce clearer and less error prone code in some cases and will allow the compilers to generate more efficient code.

Temporary Workspace Vectors

Many sections of the code can easily be expressed in array notation, figure 10. Often where a DO loop contains a scalar temporary this must be promoted to a vector temporary and aligned with the source data to allow the use of a FORALL loop and the independent parallel execution. For the exchange of the pivot a two dimensional array is used which is aligned with the source matrix. If a one dimension temporary variable were used here a considerable overhead in data mapping could occur. Also this two dimensional array is only required in row

```
F90 code

! interchange and eliminate by columns:
      DO j=kp,n
        s = a(l,j)
        a(l,j) = a(k,j)
        a(k,j) = s

        DO i=kp,n
          a(i,j) = a(i,j) + a(i,k) * s
        ENDDO
      ENDDO

HPF code using FORALL

! interchange and eliminate by columns:
      FORALL (j=kp:kf)
        s(l(ks),j) = aa(l(ks),j)
        aa(l(ks),j) = aa(ks,j)
        aa(ks,j) = s(l(ks),j)
      ENDFORALL
!
      FORALL (i=kp:kf )
        aa(kp:kf,i) = aa(kp:kf,i)
     &                + aa(kp:kf,k) * aa(k,i)
      ENDFORALL

HPF code using array syntax

! interchange and eliminate by columns:
      s(l(ks),kp:kf) = aa(l(ks),kp:kf)
      aa(l(ks),kp:kf) = aa(ks,kp:kf)
      aa(ks,kp:kf) = s(l(ks),kp:kf)
!
      aa(kp:kf,kp:kf) = aa(kp:kf,kp:kf)
     &                 + aa(kp:kf,k) * aa(k,kp:kf)
```

Figure 10: *Comparison of FORALL and DO loop usage.*

decomposition. If the matrix is distributed in column order only a one dimensional array is required.

The copy operation is somewhat costly since it also implies communication as the two rows being exchanged need not be on the same processor. Pointers could be used to reduce data transfer and a mask function to determine which rows are actually below the diagonal on each sweep. A final rearrangement of the rows could be performed at the end of the factorisation process. The pivot row exchange is a small fraction of the total communication since the pivot row will be broadcast in order to eliminate the column below the diagonal.

An Alternative Solution Procedure

Since the operation count for the LU factorisation is of order (n^3) there are alternative methods which require similar computation costs but offer increased parallelism. The LU factorisation process presents a range of granularities and the forward elimination and back substitution process is essentially poorly loadbalanced with a high communication to computation ratio. An alternative formulation, Gaussian Elimination (GE) provides a higher degree of parallelism avoiding the poor performance of the forward elimination and backsubstitution steps.

Gaussian Elimination

For GE the floating point operation count is twice that of LU but the memory access and data distribution requirements are roughly equivalent. This suggests that since our solver will be limited by data transfer rather than floating point performance we can use GE to eliminate the serial coding difficulties of LU.

The first stage is shown in figure 11. The operations performed are identical in nature except that the elimination in GE is performed over all rows rather than just those below the diagonal as in LU, the pivot row is still selected from below the diagonal since only a member of this set can perform the desired elimination.

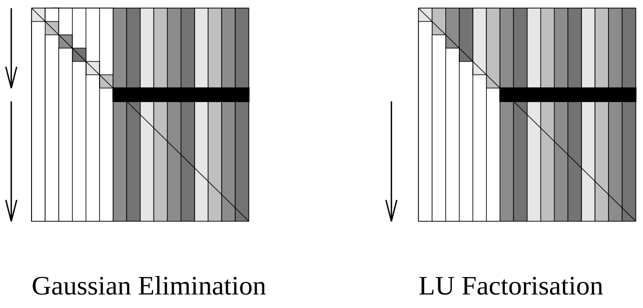

Figure 11: Comparison of the GE and LU solution procedures

Advanced Parallelism

In several parts of the code there are actions for which there are no optimal alignments or best fitted data distributions. One example is seen in the equation assembly phase where the interaction between panels is considered. The DO loop structure here can cause many small messages to be exchanged. Here options are to global broadcast and replicate one of the two arrays or to perform the operation in a blocked sequence. This is often used in message passing or cache based systems to improve performance. Some of these points, and others, are part of the HPFF discussion forum [10] and are also discussed in [3].

Conclusions

In this paper we have shown how a simple application involving the solution of a dense matrix can be expressed in HPF. The basic concepts of HPF have been demonstrated through several 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*. The relative merits of alternative decompositions have been compared and we conclude that the coding changes made between the two data decompositions are minor. The choice between the different decompositions and the remapping of data between the different stages of the algorithm will depend upon the problem sizes being considered and the performance of the TRANSPOSE intrinsic function and the REDISTRIBUTE directive for particular machines. At present HPF cannot express the flexible degree of parallelism obtained with message passing implementations of dense matrix algorithms, such are implemented in the ScaLAPACK library and perhaps it is necessary for an HPF runtime library to be developed to obtain high levels of performance – particularly for the case of relatively small problems on large numbers of processors, where the optimal decomposition is more dependent upon problem size.

Acknowledgments

One of us (KAH) wishes to acknowledge valuable discussions on the work reported here with E.A.Bogucz.

References

[1] Bordawekar, R., Thakur, R., Ponnusamy, R., Choudhary, A., "Runtime Support for Parallel I/O in PASSION", NPAC Technical Report SCCS-670, 1995.

[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] Cheng, G., Fox, G.C., and Hawick, K.A., "A Scalable Paradigm for Effectively-Dense Matrix Formulated Applications" Proc. HPCN 1994, Munich, Germany April 1994. Volume 2, PP202.

[5] 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.

[6] 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.

[7] Duff, I.S., "Tutorial on ," *Supercomputing '93*, Portland, OR, 1993.

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

[9] Hawick, K.A., Dincer, K., Robinson, G., Fox, G.C., "Conjugate Gradient Algorithms in Fortran 90 and High Performance Fortran", *NPAC Technical Report SCCS 691*,Northeast Parallel Architectures Center, Syracuse, NY 13244-4100.

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

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

[12] G. von Lazewski, M. Parashar, A.G. Mohamed, G.C. Fox, "On the parallelization of blocked LU factorization algorithms on distributed memory architectures" *Supercomputing 92*, Minneapolis, Nov. 1992.

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

[14] Moran, J., "An Introduction to Theoretical and Computational Aerodynamics", Pub. Wiley, 1984.

[15] Tritton, D.,J., "Physical Fluid Dynamics", Oxford Science Publications, 1987.