Solving the Navier-Stokes equations on a workstation cluster

S.E. Norris S.W. Armfield[*]

(Received 7 August 2000)

Abstract

A Finite Volume CFD code for modelling Natural convection has been parallelised using High Performance Fortran (HPF). A comparison is made between using HPF and message passing libraries (PVM and MPI) in terms of performance and ease of conversion. For the code being used by the authors HPF and MPI give similar speeds, and HPF was considered significantly simpler to program.

[*]Department of Mechanical and Mechatronic Engineering, University of Sydney, NSW 2006, Australia. mailto:s.norris@auckland.ac.nz

[0]See http://anziamj.austms.org.au/V42/CTAC99/Norr for this article and ancillary services, © Austral. Mathematical Soc. 2000. Published 27 Nov 2000.

Contents

1 Introduction

Transient natural convection can be effectively modelled using a finite volume CFD code. However to capture the fine detail of flow structures in 3D natural convection problems places a considerable demand on CPU time and memory size. This demand can be met by using supercomputers, but a low cost alternative is to use a cluster of conventional workstations.

An existing time stepping 3D finite volume CFD code has been parallelised to demonstrate the viability of using a workstation cluster to model 3D flows. In the first part of the paper a test code is used to compare the use of a data parallel language (High Performance Fortran) with message passing libraries. In the second section the conversion of the CFD code from Fortran 77 to High Performance Fortran is discussed.

2 The Test Code

There are two methodologies for programming distributed memory computers—message passing and data parallel languages. In order to compare these two methods, a conjugate gradient solver, which accounts for approximately 90% of the CPU time of the serial version of the CFD code, was implemented using High Performance Fortran (HPF) [6], a data parallel language, and the Fortran interfaces to the MPI [4], [7] and PVM [3] message passing libraries. The codes were run on a cluster of 32 DEC Alpha 500/266's connected with a 100Mb Ethernet switch. The HPF code was compiled using the Digital HPF compiler, whilst the message passing codes were compiled using the Digital Fortran 90 compiler and the LAM MPI library, and the public domain implementation of PVM.

The algorithm for an unpreconditioned conjugate gradient solver [2], [5] is given in Figure 1. The solver was implemented for a 3D finite volume discretisation of Laplace's equation upon a structured mesh with Dirichlet boundary conditions, with the equations and solution field being stored in three dimensional arrays corresponding to the nodes of the mesh. For simplicity the arrays were partitioned across the processors in their third axis.

```
$p^{(0)} = 0$
$r^{(0)} = b - \mathbf{A}x^0$
for $k = 1, 2, \ldots$
    $\rho_{k-1} = r^{(k-1)^T} r^{(k-1)}$
    if $k = 1$
        $\beta_0 = 0$
    else
        $\beta_{k-1} = \frac{\rho_{k-1}}{\rho_{k-2}}$
    $p^{(k)} = r^{(k-1)} + \beta_{k-1} p^{(k-1)}$
    $q^{(k)} = \mathbf{A}p^k$
    $\alpha_k = \frac{\rho_{k-1}}{p^{(k)^T} q^{(k)}}$
    $x^{(k)} = x^{(k-1)} + \alpha_k p^{(k)}$
    $r^{(k)} = r^{(k-1)} - \alpha_k q^{(k)}$
    Check convergence. Continue if necessary
```

FIGURE 1: The unpreconditioned Conjugate Gradient algorithm.

```
INTEGER nx,ny,nz
REAL rho,r(nx,ny,nz)
!hpf$ DISTRIBUTE (*,*,block) :: r

rho = sum(r(2:nx-1,2:ny-1,2:nz-1)**2)
```

FIGURE 2: The HPF implimentation of a dot product operation.

```
INCLUDE 'mpif.h'
INTEGER nx,ny,nz_local,stat
REAL rho,rdum,r(nx,ny,nz_local)

rho = sum(r(2:nx-1,2:ny-1,2:nz_local-1)**2)
call MPI_AllReduce( rho,rdum,1,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,stat)
rho = rdum
```

FIGURE 3: The MPI implimentation of a dot product operation.

TABLE 1: Code sizes (in number of lines of code) for dot product and matrix-multiply operations, and a conjugate gradient solver.

Method	Dot Product	Matrix-Vector Multiply	Conjugate Gradient Solver
HPF	4	6	22
MPI	6	15	36
PVM	15	16	54

```
INCLUDE 'fpvm3.h'
INTEGER :: nx,ny,nz_local,stat,bufid,tid_parent,mtag=15
REAL rho,r(nx,ny,nz_local)
CHARACTER*(*) group

rho = sum(r(2:nx-1,2:ny-1,2:nz_local-1)**2)
call PvmfReduce( PvmSum,rho,1,Real4,mtag,group,0,stat )

call PvmfParent( tid_parent )
if (tid_parent.eq.PvmNoParent) then
   call PvmfInitSend( PvmDataRaw,bufid )
   call PvmfPack( Real4,rho,1,1,stat )
   call PvmfBCast( group,mtag,stat )
else
   call PvmfRecv( tid_parent,mtag,bufid )
   call PvmfUnpack( Real4,rho,1,1,stat )
endif
```

Figure 4: The PVM implimentation of a dot product operation.

```
INTEGER nx,ny,nz,i,j,k,l,noit
REAL, DIMENSION(nx,ny,nz) :: s,p,ae,aw,an,as,at,ab,ap
!hpf$ DISTRIBUTE (*,*,block) :: s,p,ae,aw,an,as,at,ab,ap

do l = 1,noit

  forall(i=2:nx-1,j=2:ny-1,k=2:nz-1)                         &
      s(i,j,k) = ae(i,j,k)*p(i+1,j,k) + aw(i,j,k)*p(i-1,j,k) &
               + an(i,j,k)*p(i,j+1,k) + as(i,j,k)*p(i,j-1,k) &
               + at(i,j,k)*p(i,j,k+1) + ab(i,j,k)*p(i,j,k-1) &
               + ap(i,j,k)*p(i,j,k)

enddo
```

Figure 5: The HPF implementation of a matrix vector multiply in an iterative loop.

```
INCLUDE 'mpif.h'
INTEGER :: nx,ny,nz_local,i,j,k,l,noit,id_previous, &
           id_next,stat,req(4),mtag = 15
REAL, DIMENSION(nx,ny,nz_local) :: s,p,ae,aw,an,as,at,ab,ap
call MPI_RECV_INIT( p(1,1,nz_local),nx*ny,MPI_REAL,id_next,mtag, &
                    MPI_COMM_WORLD,req(3),stat )
call MPI_RECV_INIT( p(1,1,1),nx*ny,MPI_REAL,id_previous,mtag, &
                    MPI_COMM_WORLD,req(4),stat )
call MPI_SEND_INIT( p(1,1,2),nx*ny,MPI_REAL,id_previous,mtag, &
                    MPI_COMM_WORLD,req(1),stat )
call MPI_SEND_INIT( p(1,1,nz_local-1),nx*ny,MPI_REAL,id_next,mtag, &
                    MPI_COMM_WORLD,req(2),stat )
do l = 1,noit
   call MPI_STARTALL( 4,req,stat )
   call MPI_WAITALL( 4,req,status,stat )
   forall(i=2:nx-1,j=2:ny-1,k=2:nz_local-1)                    &
        s(i,j,k) = ae(i,j,k)*p(i+1,j,k) + aw(i,j,k)*p(i-1,j,k) &
                 + an(i,j,k)*p(i,j+1,k) + as(i,j,k)*p(i,j-1,k) &
                 + at(i,j,k)*p(i,j,k+1) + ab(i,j,k)*p(i,j,k-1) &
                 + ap(i,j,k)*p(i,j,k)
enddo
do i = 1,4
   call MPI_REQUEST_FREE( req(i),stat )
enddo
```

Figure 6: The MPI implementation of a matrix vector multiply in an iterative loop.

```
INCLUDE 'fpvm3.h'
INTEGER :: nx,ny,nz_local,i,j,k,l,noit,id_previous,id_next, &
           stat,ibuf,mtag = 15
REAL, DIMENSION(nx,ny,nz_local) :: s,p,ae,aw,an,as,at,ab,ap
do l = 1,noit
   ! Left shift
   call PvmfInitSend( PvmRaw,ibuf )
   call PvmfPack( Real4,p(1,1,2),nx*ny,1,stat )
   call PvmfSend( id_previous,mtag,stat )
   call PvmfRecv( id_next,mtag,ibuf )
   call PvmfUnpack( Real4,p(1,1,nz_local),nx*ny,1,stat )
   ! Right shift
   call PvmfInitSend( PvmRaw,ibuf )
   call PvmfPack( Real4,p(1,1,nz_local-1),nx*ny,1,stat )
   call PvmfSend( id_next,mtag,stat )
   call PvmfRecv( id_previous,mtag,ibuf )
   call PvmfUnpack( Real4,p(1,1,1),nx*ny,1,stat )
   forall(i=2:nx-1,j=2:ny-1,k=2:nz_local-1)                     &
        s(i,j,k) = ae(i,j,k)*p(i+1,j,k) + aw(i,j,k)*p(i-1,j,k) &
                 + an(i,j,k)*p(i,j+1,k) + as(i,j,k)*p(i,j-1,k) &
                 + at(i,j,k)*p(i,j,k+1) + ab(i,j,k)*p(i,j,k-1) &
                 + ap(i,j,k)*p(i,j,k)
enddo
```

Figure 7: The PVM implementation of a matrix vector multiply in an iterative loop.

For each iteration of the solver there are three SAXPYs, two dot products, a matrix vector-multiply, and a reduction operation to determine convergence. The SAXPYs are trivially parallel, but the dot product operations require a global reduction/broadcast across all processors, and the matrix-vector multiply requires a boundary swapping of data between neighbouring processors.

The implementation of the dot product using HPF, MPI and PVM is shown in Figures 2, 3 and 4 respectively. Whilst the dot product is a simple operation to code with both HPF and MPI, the PVM version is more complicated, with a broadcast needed after the reduction, to return the final value back to all processors.

In Fortran 90 the matrix-vector multiply can be implemented in a number of ways; with `do` loops (as would be natural in Fortran 77), using the `cshift` or `eoshift` intrinsics, using a `forall` block, or using Fortran 90 array operations. The speed of different implementations can vary by a factor of two, and whilst the `cshift` intrinsic is the slowest on most platforms, on some (the Thinking Machines CM-5 being an example) it is the fastest. For the DEC Alpha using Digital Fortran the forall and Fortran 90 array forms of the operation are fastest for both serial Fortran 90 code and code parallelised using HPF, the forall form being shown in Figure 5.

With the message passing codes the matrix-vector operation can be implemented in the same forms as the HPF code, with the additional step of a swap operation between neighbouring processors to share elements at the array boundaries. The code is simplest when written with a dummy element at the boundaries that are used in these swaps. For the MPI code the neighbour swap operation can be implemented in a number of ways–the code here uses a persistent communication request, or a "half channel", with the communication pattern being initialised with the `MPI_*_INIT` calls, and then executed with the `MPI_STARTALL` and `MPI_WAITALL` calls. The message passing versions of the matrix-vector multiply are given in Figures 6 and 7.

The code sizes for the above code fragments and the complete conjugate gradient solvers are given in Table 1. The message passing codes are in all cases larger than the HPF codes. Extra care must be taken when writing message passing codes to prevent deadlocks caused by mismatched send and receive operations. Code must also be written to partition the data across the nodes and to initialise the multiple processes—tasks which are done transparently by the HPF codes.

3 Speed of the Test Code

For most modern architectures the speed of array operations is strongly dependent on the array size, with memory banking, cache limits, start up time of vector processors, and data partitioning all causing different behaviour on different machines. For this reason the solver was tested for a range of array sizes, typically in the range 10^3 to 120^3.

Program speeds were calculated on an unloaded cluster using wallclock time, in an effort to account for the communication overhead. For each array size enough iterations were done to ensure an overall runtime of several seconds, with the conjugate gradient solver being restarted after every 100 iterations to ensure that the operations weren't being performed on arrays of zeros. Wallclock time was measured with the C `gettimeofday` call, which gives a resolution in milliseconds with Digital Unix.

Since the data is quite "spiky" it has been smoothed in the following graphs to aid readability. The speedups were calculated from the smoothed data, since otherwise spurious non-linear speedups could occur on some mesh sizes due to the poor performance of the single processor version of the code. It should be noted that run times for a particular array size typically varied between runs by less than 1%, so the variability of the speeds is real and not experimental error.

The speed for a single DEC Alpha 500/233 is shown in Figure 8, as both raw and smoothed data, using the version of the code written using the forall construct for the matrix-vector multiply. The graph is typical for a modern RISC machine, with the speed increasing as the array sizes are increased, until the arrays can no longer fit into the processor's cache. When the size of the cache is exceeded the speed drops off and is limited by the speed of memory accesses from the main system memory. The large variation in speeds from one array size to the next is also typical, with the extremely slow speeds for arrays of 16^3, 32^3, 64^3 and 96^3 being a result of the memory banking scheme breaking down for these sizes.

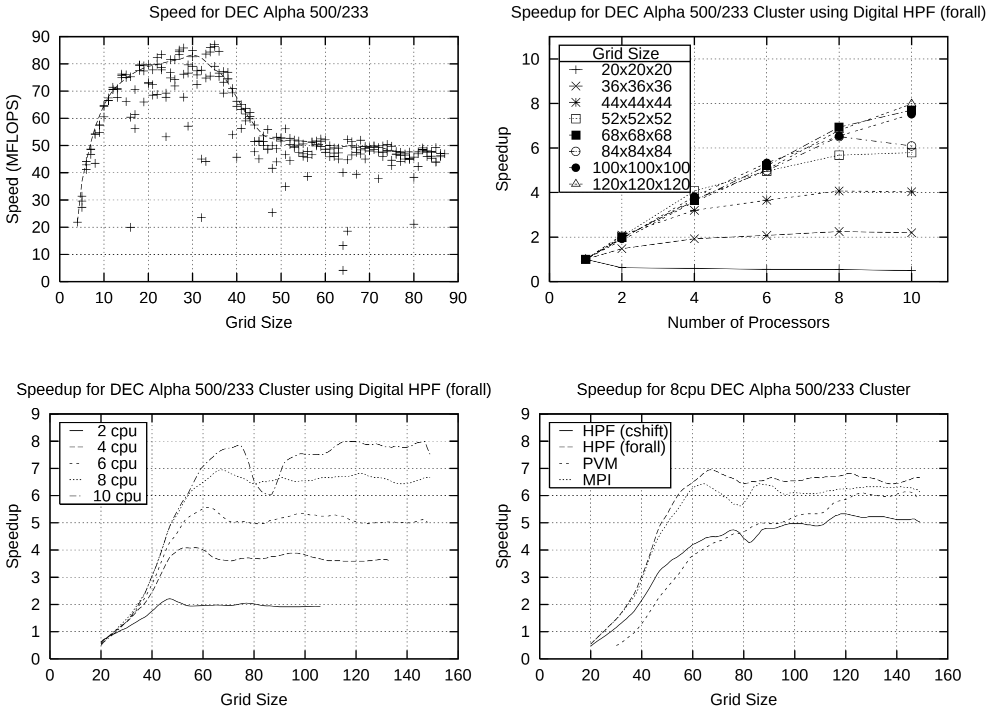

FIGURE 8: Speed of a single processor DEC Alpha 500/233, and speedup for a DEC Alpha 500/233 Cluster.

Also shown in Figure 8 is the speedup for the parallel versions of the code, which all showed an initial increase in speedup with increasing array size, with the speedup levelling off as the array reached a size of 40^3 to 70^3, for cases of 2 to 10 processors respectively. The two and four processor HPF codes showed a small region of superlinear speedup, where the data for the parallel code fitted in cache, whilst the serial version exceeded the cache size. Otherwise any gains in caching are offset by the increased relative communication cost for small arrays, and for large arrays the speed is largely constrained by memory access speeds.

For each cluster size, the HPF (forall) and MPI implementations gave similar performance, with the HPF (cshift) and PVM codes being 5–50% slower. The PVM code is slower than MPI for smaller array sizes, but as the array sizes are increased the speed difference between the two codes decreases. This was thought to be due to the high start-up overhead of the PVM implementation when compared with MPI. The comparative slowness of the HPF (cshift) code when compared to the HPF (forall) implementation is due to the cshift operations creating temporary arrays. This increases the memory used in an operation that is limited by memory bandwidth, thus slowing the speed of the operation.

The similarity of the speeds of the HPF (forall) and MPI codes means that the choice between using a message passing library or a data parallel language cannot be made in terms of performance, and must instead be made in terms of ease of programming and the codes data structures. A single structured mesh obviously fits easily within the HPF model, whilst a block decomposition of a complex domain might be more easily coded using a message passing model.

4 The CFD Code

The code to be converted was a 3D finite volume time stepping code that used a structured Cartesian mesh [1]. The diffusion terms are discretised using an implicit Crank-Nicholson time stepping scheme with second order differencing in space, with the advection terms using an explicit Adams-Bashford time stepping scheme with a third order upwind spatial discretisation. At each time step the temperature and momentum equations are calculated, and then a pressure correction equation is used to force continuity. In the serial code the pressure equations were initially solved using an ADI iterative method. Since this wouldn't easily parallelise it was converted to a Jacobi preconditioned Conjugate gradient solver. This step alone resulted in a 5 times speed up of the code. To improve the robustness of the pressure correction step a BiCGSTAB solver was substituted for the Conjugate gradient solver, with a marginal improvement in speed.

The code was then converted to HPF, with a rewriting of the operations into Fortran 90 array operations and the addition of HPF directives. Run times for the code running on the DEC Alpha cluster are shown in Figure 9 along with the speedups for different mesh and cluster sizes.

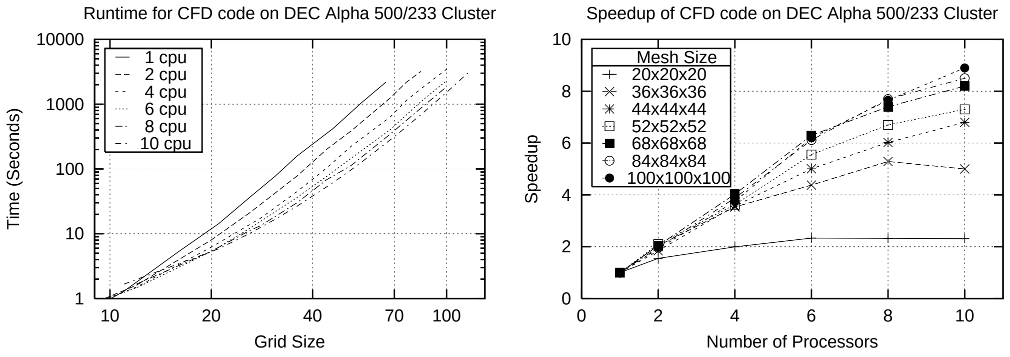

Figure 9: Runtime and speedup for CFD code on DEC Alpha 500/233 Cluster. The code was modelling transient natural convection in a cubic cavity, with $\Delta t/\Delta x$ being kept constant to ensure a uniform Courant number for all mesh sizes.

As can be seen the speedup characteristics are broadly similar to those of the test programs, and the runtime increases as the fifth power for moderate to large problems.

5 Conclusion

A CFD code has been converted to run on a workstation cluster by rewriting it into High Performance Fortran (HPF). The use of HPF shows no disadvantages in terms of speed when compared to message passing systems, but eases the programming of the code. Good speedup is shown for a cluster of 2–10 workstations.

References

[1] S. W. Armfield and R. Janssen. Direct simulation of travelling wave instability in steady state convection. *Heat and Fluid Flow*, 17:539–546, 1996. C1072

[2] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems*. SIAM, Philadelphia, 1994. C1060

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1994. C1060

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994. C1060

[5] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–437, 1952. C1060

[6] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr, and M.E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, 1994. C1060

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, Massachusetts, second edition, 1998. C1060