

# GUIDELINES FOR PROGRAMMING IN HIGH PERFORMANCE FORTRAN

by Dave Pruett

NSF CCLI Grant

July 2000

**BACKGROUND:** These comments derive from my initial experiences with programming in High Performance Fortran (HPF) during the summer of 2000. The particular compiler was PGHPF by Portland Group (PG). The platform was a 16 single-processor (400MHz) PC “Beowulf” system with a 100Mb (=12MB=3SPMW) per second SYSCO switch. For “embarrassingly parallel” problems such as matrix-vector and matrix-matrix multiplies, HPF results were stellar—speedups were linear or slightly superlinear. For example, on JMU’s Beowulf system, matrix-vector multiplies in HPF ran at 75-80Mflops serially and at 1.3Gflops on 16 processors. However, for a “real” computational problem, results were not spectacular for our relatively slow switch.

The particular code ported from Fortran90 to HPF was FSL, which simulates the evolution in time and space of a 2D Free-Shear Layer. The algorithm, a modification of the Kim and Moin method (*J. Comput. Phys.*, 1985), of second-order in space and time, had been highly optimized for (serial) efficiency. The particular version exploits disturbance vorticity and streamfunction as the primary variables. Vorticity is advanced in time semi-implicitly. Nonlinear (advection) terms are treated explicitly by the 2nd-order Adams-Bashforth method; linear (diffusion) terms are advanced by the Crank-Nicolson method. Approximate factorization of the 2D spatial diffusion operator results in an efficient ADI-like procedure that exploits multiple tridiagonal solves at each time step to advance the vorticity. The streamfunction is then updated by a Fast Poisson Solver that exploits FFTs. The total operation count per time step is  $O(mn \log_2 m)$ , where the computational mesh is  $m \times n$ . The algorithm is well suited for parallelization over the independent tridiagonal solves and FFTs.

## EXPERIENCE AND RECOMMENDATIONS

- After a month of work, a speedup of 2 was realized on 16 processors relative to the baseline timing of the serial code. Specifically, for a 257 by 144 grid, the serial code ran 256 time steps in 23.4 seconds; the parallel code ran in 11.8. For two reasons, this result is perhaps not as disappointing as it would seem on face value. First, PG’s f90 compiler is blazingly fast when the aggressive optimization compiler flag is used (**-fast**). Apparently the **-fast** option plays games with cache to minimize memory fetches (my guess). Serial time without aggressive compilation was 30.4 seconds. Some routines ran 3 times faster *serially* with **-fast**. Second, of those 11.8 seconds, at least 8 seconds are due to communication costs, which presumably would diminish dramatically with faster communications.
- The compiler options that produced the fastest code were:

**pghpf -Mfreeform -fast -Minfo -Mstats -Mrpm -o fsln \*.hpf**

The run-time options were:

```
./fsln -pghpf -np 16 -stat all < fsl.d
```

Of these compiler flags, **freeform** is necessary for interpreting free-format Fortran90 code, **fast** turns on aggressive optimization, **info** gives parallelization information during compilation, **stats** sets up the run-time statistics, and **rpm** selects the default message-passing substrate. Other message-passing substrates are **pvm** and **mpi**. With **-Mmpi**, the code compiled but produced erroneous results. At the time of this writing, **pvm** was not installed on our system. In parallel mode, **fast** appears to turn on **-Mautopar** for automatic parallelization. All attempts to invoke HPF2.0 via the compiler option **-Mhpf2** resulted in run-time crashes.

- For code development and debugging, turn on **-Mprof=func** and turn off **-fast**; an apparent compiler bug will generate erroneous results if these options are on simultaneously. The option **-Mprof=func** initializes the timing of individual subroutines for later profiling, with little computational overhead. In contrast **-Mprof=lines** times every single executable statement (line), at enormous computational overhead (but useful for optimizing individual subroutines).
- Inherently serial code can actually run many times *slower* in parallel mode than in serial mode. For example, the initialization routine INITIAL.hpf of FSL.hpf required 12 seconds on 16 processors, but less than a second on 1 processor! For long production runs, the cost of initialization is amortized, so one can choose to ignore this problem. On the other hand, it's a nuisance during code development. Fortunately, there is a fix. Inherently serial subroutines can be declared as extrinsics and compiled separately with **-Mf90**. They then run at the serial rate. For example

```
EXTRINSIC (F77_SERIAL) SUBROUTINE initial (...)
```

Any routine so declared requires an explicit INTERFACE block in the calling routine. A further complication arose. Some but not all vectors defined inside the SERIAL EXTRINSIC did not copy properly when DISTRIBUTED in the main program. This appears to be another compiler error. Initializing the problem vectors in the main program solved the problem.

- Interestingly, no explicit FORALL statements or INDEPENDENT compiler directives were necessary to produce parallel code. This fact was discovered by trial and error. The best results were obtained simply by using Fortran90 vector syntax and SAXPY (scalar  $a$  times vector  $X$  + vector  $Y$ ) operations wherever possible. For example, the linear-combination algorithm for a matrix-vector multiply yields

```
REAL, DIMENSION(n) :: b  
REAL, DIMENSION(m) :: c
```

```

REAL, DIMENSION(m,n) :: A
!HPF$ DISTRIBUTE (BLOCK) :: c
!HPF$ ALIGN (:,*) WITH c(:) :: A
  c = 0.0
  DO j = 1,n
    c(1:m) = c(1:m) + b(j) * A(1:m,j)
  END DO

```

With either the **-Mautopar** or the **-fast** compiler option, the compiler automatically generated FORALLs, as indicated in the compilation log. The (BLOCK) designation distributes subvectors across processors. Without explicit FORALLs, the same code can run on any Fortran90 compiler.

- Thus, the current HPF code exploits only 3 of the many HPF compiler directives: EXTRINSIC (F77\_SERIAL), !HPF\$ DISTRIBUTE, and !HPF\$ ALIGN, the latter two of which “farm out” the data across the processors. One need use only DISTRIBUTEs; however, liberal use of ALIGNs results in greater versatility if one wishes to experiment with different distributions (for example, BLOCK vs. CYCLIC). A DISTRIBUTE is perhaps best thought of simply as a copy. If the data distribution in a subroutine matches that of the calling program or routine, then no re-copy is necessary. If not, a time consuming COPY\_IN operation is required for INTENT(IN) variables, a COPY\_OUT for INTENT(OUT) variables, and both for INTENT(INOUT).
- For reasons mysterious (to me), it is faster to use temporary arrays to re-copy (manually redistribute) data prior to entry and/or subsequent to exit of a subroutine rather than to rely on the built-in COPY\_IN and/or COPY\_OUT HPF intrinsics during the execution of the subroutine. It was noticed that explicit array copies from (BLOCK,\*) to (\*,BLOCK) run at about twice the maximum serial data transfer rate, so some parallel communication capability is being exploited. Perhaps this explains why explicit copies run faster than the default intrinsics.
- For the reason above, a “flat” program structure is recommended. That is, avoid subroutines that call subroutines that call subroutines, and so on. A flat structure allows all the re-copying of data (and communication costs) to be localized and confined to the main program.
- Although HPF allows for REDISTRIBUTE and REALIGN, which are executable versions of the non-executable DISTRIBUTE and ALIGN directives, these operations are *very* slow in a relative sense and are to be avoided. It appears that explicit array copies (e.g., TMP\_ARRAY = ARRAY) are relatively fast, the COPY\_IN and COPY\_OUT intrinsics (used when redistribution is necessary in a subroutine) are a little slower, and REDISTRIBUTE/REALIGN are very slow.
- Unfortunately, at the time this document was written no option exists with PGHPF to save intermediate code with MPI calls, which would allow tweaking of the code by explicit message passing.

## REMAINING POINTS OF CONFUSION

- Explicit array copies (i.e., `TMP_ARRAY = ARRAY`) take different times at different execution points in the main program. All 2D arrays are the same size. Why the different times?