Porting the GENESIS Benchmarks to HPF

Bryan Carpenter and Vladimir Getov

High Performance Computing Centre,
University of Southampton,
Southampton, S09 5NH

DRAFT

Abstract

Selected kernel benchmark and "compact applications" from the Genesis benchmark suite have been converted to HPF. We outline the codes involved, and describe the changes which had to be made in going from sequential or message-passing versions of the codes to parallel HPF programs. Some timings on early HPF platforms are given.

1 Introduction

The GENESIS benchmarks were originally developed in ESPRIT project 2702—GENESIS—with contributions from a number of project partners including Liverpool and Southampton Universities, PALLAS GmbH, the European Centre for Medium Range Weather Forecasting and the Universitat Politechnica de Catalunya. The suite is currently distributed by Southampton High Performance Computing Centre. The benchmarks were designed to evaluate the performance of distributed-memory MIMD computers as applied to a range of scientific and engineering problems.

The codes were written in standard Fortran 77, with explicit message-passing through calls to portable communication libraries. Originally the communication library was PARMACS, a proprietary system distribed by PALLAS GmbH. More recently the codes have been ported to run in the public domain message-passing environment, PVM.

A subset of the benchmarks have been converted to HPF, using Fortran 90 features and eliminating explicit message-passing. The codes we will discuss are

trans1: A communication benchmark involving transposition of a distributed matrix.

pde1: A solver for the Three-dimensional Poisson equation, using red-black relaxation.

fft1: A Fast Fourier Transform.

pde2: A solver for the Two-dimensional Poisson equation, using a recursive "Multigrid" algorithm.

md1: A molecular dynamics program, using the "Particle-in-Cell" method.

Most of the benchmarks had to be extensively rewritten to make them data-parallel. For maximum portability we avoided using features which are not in the HPF official Subset. We also avoided using HPF features which not are part of standard Fortran 90—namely, FORALL. This restriction was needed because at the time the work was done there were no HPF systems available to us. The codes were tested using a standard Fortran 90 system (the NAG compiler). In the absence of FORALL, all explicit parallelism in the converted codes is obtained through Fortran 90 array syntax. Note, however, that the structural changes to the codes would still be needed if FORALL was used.

The main purpose of this article is to describe the changes which were made to the codes to obtain functional HPF programs. We will be particularly interested in things which were difficult to express in HPF, and try to suggest minor changes to the language which would have made the task easier.

Versions of our HPF codes are in release 3.0 of the GENESIS benchmarks. As explained above the original codes were not tested under a full HPF compiler, and there are some bugs in the HPF directives (which, of course, are just treated as comments by standard Fortran 90 compilers). Fully functional versions are included as part of the Southampton HPF release.

2 The codes

2.1 *trans1*

This the simplest of the benchmarks we converted. It transposes a square matrix distributed blockwise on a grid of processors. There is an intrinsic function, TRANSPOSE, in Fortran 90 which does just this. The HPF code is essentially

```
!HPF$ PROCESSORS GRID(SIDE,SIDE)

      DOUBLE PRECISION DATA(MAXD,MAXD)
!HPF$ DISTRIBUTE DATA(BLOCK,BLOCK) ONTO GRID

      ! ...

         DATA = TRANSPOSE(DATA)
```

2.2 *pde1*

This benchmark solves the Poisson equation on a 3-dimensional grid.

The Poisson equation is a partial differential equation (PDE). Similar equations arise in many application areas including Computational Fluid Dynamics, Meteorology, Climate Research and Oil Reservoir simulation. The PDEs are typically discretised on a grid, so that the PDE is represented by a large set of linear or non-linear equations, each coupling values at neighbouring grid points. The Poisson equation is a particularly simple case which arises, for example, in the calculation of electromagnetic fields.

The original message-passing codes use successive over-relaxation (SOR) with Chebyshev acceleration. Our HPF version uses pure red-black relaxation—the simpler algorithm avoids some clutter but retains the features of the original program relevant to parallelisation. The crux of the algorithm is to sweep repeatedly through the lattice replacing the value at each grid site by the average of the values of its neighbours. To avoid bad dependencies and improve convergence the sweep is done in two phases; first all odd sites ("red squares") are updated, then all even sites ("black squares").

The kernel of the HPF code is

```
      INTEGER NX, NY, NZ
      DOUBLE PRECISION U(NX,NY,NZ), F(NX,NY,NZ)
      LOGICAL RED(NX,NY,NZ)

!HPF$ INHERIT RED
!HPF$ ALIGN U(I,J,K) WITH *RED(I,J,K)
!HPF$ ALIGN F(I,J,K) WITH *RED(I,J,K)
!HPF$ DISTRIBUTE RED *(BLOCK,BLOCK,BLOCK)

      ! ...

        WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
          U(2:NX-1,2:NY-1,2:NZ-1) =                                 &
     &        FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+                  &
     &        U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+        &
     &        U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+        &
     &        U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
        ELSEWHERE
          U(2:NX-1,2:NY-1,2:NZ-1) =                                 &
     &        FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+                  &
     &        U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+        &
     &        U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+        &
     &        U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
        END WHERE
```

This HPF version is reasonably concise, but would be more readable if we used FORALL instead of array syntax. The parallel sweep is equivalent to

```
      FORALL (X = 2:NX-1, Y = 2:NY-1, Z = 2:NZ-1, RED(X, Y, Z))        &
  &     U(X,Y,Z) = FACTOR * (HSQ * F(X, Y, Z) +                         &
  &         U(X-1, Y, Z) + U(X+1, Y, Z) +                               &
  &         U(X, Y-1, Z) + U(X, Y+1, Z) +                               &
  &         U(X, Y, Z-1) + U(X, Y, Z+1))

      FORALL (X = 2:NX-1, Y = 2:NY-1, Z = 2:NZ-1, .NOT. RED(X, Y, Z)) &
  &     U(X,Y,Z) = FACTOR * (HSQ * F(X, Y, Z) +                         &
  &         U(X-1, Y, Z) + U(X+1, Y, Z) +                               &
  &         U(X, Y-1, Z) + U(X, Y+1, Z) +                               &
  &         U(X, Y, Z-1) + U(X, Y, Z+1))
```

`RED` is a logical array which is `.TRUE.` for even sites. It is used as a mask in the WHERE or FORALL constructs. The array `F` is the "forcing term"—it might be an electric charge distribution.

In any case, the program converts easily to HPF, and, when high quality optimising compilers for HPF become available, we can anticipate that they should have few problems generating fast code for this kind of simple, regular problem.

2.3 *fft1*

The one-dimensional FFT benchmark is a popular test with relatively high level of parallelism, but low level of vectorisation and a hypercube type communication. The GENESIS code follows the classical vectorised implementation (scheme A) of the 1-dimensional FFT algorithm. The kernel of this benchmark is the subroutine which implements the FFT butterfly. It is convenient to consider the butterfly calculation as a two-dimensional matrix of $\log N$ layers with $5N$ floating point operations in each of them. The layers represent the outer loop of the calculations, while each layer contains a variable number of inner loops. This number doubles from one layer to the next, beginning with one inner loop in the first layer. Thus, the length of the inner loops varies and for some of them an HPF compiler will have to insert communications before the execution of the next leyer. The essential fragment from the HPF code follows:

```
      DOUBLE PRECISION FR(0:NTOTAL-1), FI(0:NTOTAL-1)
      DOUBLE PRECISION WR(0:NTOTAL/2-1), WI(0:NTOTAL/2-1)
      DOUBLE PRECISION G(0:NTOTAL-1,0:1)

!HPF$ PROCESSORS LINE(NPROC)

!HPF$ DISTRIBUTE FR *(BLOCK) ONTO LINE
!HPF$ ALIGN FI(I) WITH *FR(I)

!HPF$ DISTRIBUTE G(BLOCK, *) ONTO LINE

      STEP = NTOTAL
```

```
DO L = 1, LOGN
   IMAX = STEP/2

   DO M = 0, NTOTAL/STEP-1
      K = M*STEP
      J = K+IMAX
      JMAX = J+IMAX-1

      G(J:JMAX,0) = WR(M)*FR(J:JMAX) - WI(M)*FI(J:JMAX)
      G(J:JMAX,1) = WR(M)*FI(J:JMAX) + WI(M)*FR(J:JMAX)
      FR(J:JMAX) = FR(K:J-1) - G(J:JMAX,0)
      FI(J:JMAX) = FI(K:J-1) - G(J:JMAX,1)
      FR(K:J-1) = FR(K:J-1) + G(J:JMAX,0)
      FI(K:J-1) = FI(K:J-1) + G(J:JMAX,1)
   ENDDO

   STEP = IMAX
ENDDO
```

This code is a good test for the ability of HPF compilers to generate communication structures with a hypercube connectivity. Further simplification of the indexation employed in the above fragment will definitely make this task much easier.

2.4 *pde2*

Like *pde1*, this program solves a Poisson equation, this time in two dimensions. The main difference is that it uses a more sophisticated algorithm—a Multigrid scheme.

The differential equation is discretised on a grid again, then a sequence of auxilliary grids is introduced. The first has twice the spacing of the original grid (and proportionately fewer points), the second four times the spacing, and so on, up to a top level "grid" with a single non-boundary point. As the solution is updated, error terms are propagated between the different grids. The general idea is that the coarser grids give a more global view of the of the solution, so that corrections are propagated rapidly to all points, while the finer grids take care of details on shorter length scales.

The algorithm is quite complex and we can only illustrate fragments here. One of the kernel routines simply performs sweeps of red-black relaxation on a fixed grid, similar to *pde1*. More novel is the for transferring information between grids. Implemented in terms of message-passing program the communication pattern here is quite intricate. Implemented in HPF it is straightforward. For example, the procedure for correcting a fine grid `UF` from a coarse grid `UC` is

```
SUBROUTINE INT2A (NPC, NPF, UC, UF)
IMPLICIT NONE
INTEGER NPC, NPF
```

```
      DOUBLE PRECISION UC (NPC, NPC), UF (NPF, NPF)
CHPF$ DISTRIBUTE UC * (BLOCK, BLOCK)
CHPF$ DISTRIBUTE UF * (BLOCK, BLOCK)
C
C     COARSE-TO-FINE TRANSFER (BILINEAR INTERPOLATION) AND CORRECTION.
C     (THIS MODIFIES ODD SITES ONLY OF UF)
C
        INTEGER NC

        NC = NPC - 1
        UF (2 : 2 * NC : 2, 3 : 2 * NC - 1 : 2) =
     1      UF (2 : 2 * NC : 2, 3 : 2 * NC - 1 : 2) +
     2          0.5D0 * (UC (1 : NC, 2 : NC) + UC (2 : NC + 1, 2 : NC))
        UF (3 : 2 * NC - 1 : 2, 2 : 2 * NC : 2) =
     1      UF (3 : 2 * NC - 1 : 2, 2 : 2 * NC : 2) +
     2          0.5D0 * (UC (2 : NC, 1 : NC) + UC (2 : NC, 2 : NC + 1))
      END
```

the grids are illustrated in figure 1 for the case `NF`$=5$, `NC`$=3$. The expression

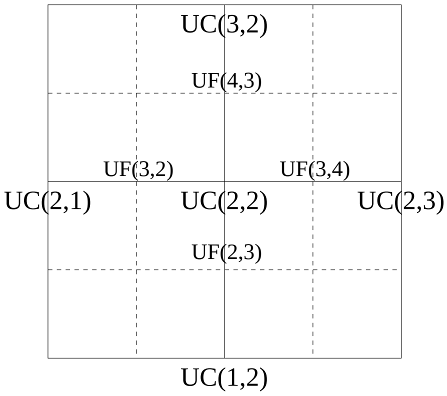

Figure 1: The coarse and fine grids in subroutine INT2A.

involving `UC` on the RHS of the first assignment is the average of pairs of sites connected by vertical links of the coarse grid. The updated point on the fine grid is the one sitting in the middle of this link. The second assignment deals similarly with sites of the fine grid lying on horizontal links of the coarse grid.

(As the comment indicates, only odd sites of the fine grid need to be updated by this particular routine, and these are just the sites lying on the links of the coarse grid.)

The program contains handful of routines performing similar transfers between grids, some for interpolation to finer grids, some for restriction of the error term to coarser grids. All can be expressed straightforwardly with array assignments or FORALL.

The main problem encountered in translating the program to HPF was the difficulty in describing a stack of grids of different sizes. This could probably be achieved in full HPF (which includes full Fortran 90) by creating a vector of derived data types, each containing a pointer to an array. The arrays would be dynamcally allocated to the required size, and individually distributed blockwise. Unfortunately Subset HPF doesn't have the dynamic facilities needed for this, and we had to fall back on statically declaring a fixed set of arrays of size 3, 5, 9, 17, ... up to some fixed maximum problem size.

2.5 *md1*

md1 is a molecular dynamics program simulating a Lennard-Jones fluid, i.e. a fluid where the molecules interact through a Lennard-Jones force law. This force law is short-range in nature, so that only a few nearby molecules have a direct influence on the movement of any given molecule. This fact is exploited by using the particle-in-cell method. Space is divided into a set of small cells whose size is approximately equal to the range of the force between the molecules. Only molecules in neighbouring cells can interact with one another. This is a major simplification because it means that the number of interactions which have to be taken into account is now proportional to the total number of molecules times the average number of molecules in a group of adjacent cells. Without using the particle-in-cell method the number of interactions would be proportional to the square of the total number of molecules.

The equations of motion for the molecules are integrated by a straightforward Verlet algorithm. For purposes of updating positions and velocities it is easiest to use one-dimensional arrays (vectors) subscripted by molecule number. The three-dimensional particle-in-cell representation is only used during the calculation of the forces on the molecules. So there are three main components in the kernel code

- Updates of position and velocity, working on vectors of particles.
- Calculation of forces, working on arrays of cells.
- Code to convert between the vector and array-of-cells representations of data.

In more detail an update has the following phases

1. Start from a vector of molecule positions, indexed by the molecule's identifier.
2. Assign the molecules to cells on the basis of their position. This gives a array of cells with some number of molecules in each cell.
3. From this cellular array of positions, compute the cellular array of forces on on the molecules.

4. Scatter this array of forces is to a vector of forces indexed by molecular identifiers.

5. Use the vector of forces to update the vector of positions and finally the vector of positions.

The original sequential code uses a linked list to implement the cell arrays. This wasn't feasible in a straightforward data-parallel implementation. Instead we place a static bound on the number of particles in a cell, and add an extra dimension of this extent to the three dimensional array. A separate three-dimensional array contains a count of the number of particles in each cell. Note that placing a static bound (8 in our case) on the number of particles in a cell is physically acceptable for this problem, because there is a limit on how many molecules can be squeezed into a given volume. It does mean that some memory is wasted, and some unused values are communicated unnecessarily when regular shifts on the data arrays are executed.

Most of the computation is in phase 3. This phase was straightforward but a little tedious to transcribe to array syntax. The nearest neighbour communications can be implemented straightforwardly by CSHIFT. The transcription was made laborious by the fact that the *local* computations are quite complex. Using array syntax this meant that a large number of temporary vectors had to be declared, and many WHERE statements are needed. The code would have been greatly simplified (and almost certainly more efficient) if we had allowed ourselves to call procedures inside a FORALL (or "elementally" on arrays). *Full* HPF does allow use of *pure* procedures inside FORALL; we avoided using them because we wished to maintain maximum portability by restricting to Subset HPF.

The communication in phases 2 and 4 is less trivial. Phase 4 can be expressed using vector subscripts:

```
NATMCV = RESHAPE (NATMCL, (/NCELLS/))
DO I = 1, MAXCL
  IDV = RESHAPE (IDL (I, :, :, :), (/NCELLS/))
  FXV = RESHAPE (FXL (I, :, :, :), (/NCELLS/))
  FYV = RESHAPE (FYL (I, :, :, :), (/NCELLS/))
  FZV = RESHAPE (FZL (I, :, :, :), (/NCELLS/))
  WHERE (I .LE. NATMCV)
    FX (IDV) = FXV
    FY (IDV) = FYV
    FZ (IDV) = FZV
  ENDWHERE
END DO
```

`NATMCL` is a three-dimensional array containing the number of molecules in each cell. `MAXCL` is the bound on the number of particles in a cell. `IDL` is a cell array containing the identifiers of the molecules in each cell, and `FXL`, `FYL` and `FZL` contain the force components. The vectors `FX`, `FY` and `FZ` are the resulting vectors of force components, subscripted by molecule identity. We had to reshape the subscripts because vector subscripts must be one-dimensional arrays. Using nested FORALL constructs would allow the less obscure code:

```
        FORALL (X = 1 : MX, Y = 1 : MY, Z = 1 : MZ)
          FORALL (I = 1 : NATMC (X, Y, Z))
    FX(IDL (I, X, Y, Z)) =  FXL(I, X, Y, Z)
    FY(IDL (I, X, Y, Z)) =  FYL(I, X, Y, Z)
    FZ(IDL (I, X, Y, Z)) =  FZL(I, X, Y, Z)
          END FORALL
        END FORALL
```

In any case, although the patterns of vector subscripting are fairly complex there is no difficulty of principle in expressing this particular scatter operation.

Phase 2 is more problematic. There doesn't seem to be an efficient way to implement it in parallel using HPF. There are various convoluted ways, which involve using either parallel sorting routines or multiple applications of combining scatter intrinsics. In practise we have resorted to sequential code. What is really needed is a new combining scatter intrinsic—a "binning scatter"—which returns an array of lists directly.