Evaluation of Benchmark Programs Vers.2

Deliverable:	D1.2b
Date:	September 1997
Status:	draft
Confidentiality:	public
Responsible:	ECMWF
Revised:	September/17/1997

The **HPF+** Consortium:
AVL GesmbH - Austria
European Centre for Medium-Range Weather Forecasts (ECMWF) - England
Engineering System International SA (ESI) - France
NA Software Ltd (NAS) - England
University of Pavia - Italy
University of Vienna - Austria

Abstract

This report constitutes deliverable D1.2b of the Esprit project "HPF+ -Optimising HPF for Advanced applications" Project No. 21033.

In this report, the second release of kernels from AVL, ECMWF, ESI and NEC/CCRLE are evaluated by criteria including performance, language features, coding effort and maintainability. In assessing performance, HPF+ versions of the kernels are compared with equivalent kernels which have been coded for message passing and other HPF compilers.

The kernels being evaluated reflect the requirements of large industrial applications where non-regular data structures are used, and parallel performance and code maintainability are important issues.

In section 1 a kernel from AVL is evaluated representing a stand-alone implementation of the conjugate gradient equation solver as used in the FIRE application. Section 2 contains an evaluation of three separate kernels from ECMWF, representing different aspects of the weather forecast application IFS. In particular, the programming issues of the irregular data spaces used in the IFS are addressed. Finally, in sections 3 and 4 kernels from ESI and NEC/CCRLE are evaluated. These kernels represent more challenging problems from the ESI PAM-CRASH application, including a contact-impact algorithm where more dynamic communication is required.

The evaluation of the first release of the kernels was reported in D1.2a (December 1996), and was based on experience of the initial release of the Vienna Fortran Compilation System (VFCS). This release of VFCS, primarily provided support for the parallelisation of Fortran 77 programs, although some Fortran 90 features were also supported. It had a number of disadvantages, the most significant being the dependence on a host-node model, the lack of dynamic data structures and the use of files for describing some distributions. With regard to data distributions, only one dimension of an array could be distributed. Also, the use of GEN_BLOCK distributions was particularly restrictive as the block sizes in these distributions had to be specified as constants in data statements.

The second release of VFCS is now called Vienna Fortran Compiler (VFC), supports Fortran 90 with none of the restrictions described above. Another major disadvantage of VFCS was that it required that the number of processors was fixed at compile-time. With VFC programs can be parallelized for an arbitrary number of processors by using NUMBER_OF_PROCESORS(). VFC also provides support for conditional schedule REUSE, perfectly nested INDEPENDENT loops, and extrinsics HPF_LOCAL & HPF_SERIAL. Additionally, a new instrumentation facility is provided, which performs automatic instrumentation of program structures such as loops and subroutines, and in the case of INDEPENDENT loops provides a breakdown of the overhead. This instrumentation facility and the analysis/visualisation provided by Medea is described in section 5.

The main goal for developing VFC has been to build a system that is both general and flexible, and to provide a framework for implementing new HPF language features which are driven by the requirements of the application partners. This evaluation reports that VFC has met this goal. In this release, it has focussed on the extension of the inspector/executor

strategy to deal with multi-dimensional distributions for the parallelisation of irregular independent loops. Furthermore, the support in VFC and MEDEA for instrumentation of such loops has allowed the application partners to obtain a greater understanding of the inspector/executor strategy and the overhead. As reported, the overhead of the inspector can be minimised by use of schedule reuse, however, the performance of the executor carries an additional overhead of indirect addressing. Furthermore, the use of HPF_LOCAL or HPF_SERIAL and array syntax instead of DO loops have resulted in improved performance.

Finally, the application partners report the need the relax the present implementation constraints for INDEPENDENT loops, and for the support of irregular shadows or halos.

Contents

1 AVL

1.1 Introduction

The second version of the AVL kernel contains the entire conjugate gradient solver for the solution of sparse linear equation systems. The equation solver is the most compute intensive part of the FIRE program, which requires up to 70% of the compute time of a typical FIRE calculation. This program part is also representative with respect to the parallelisation of the FIRE code, because all important communication concepts like indirect addressing (which can not be handled under HPF1 and HPF2) and the computation of sum reduction operations can be found there.

The solution is based on a Krylov subspace method which consists of a modified preconditioning and orthomin convergence acceleration. It can be applied to structured and unstructured meshes. The method is designed for iterative sparse matrix computation whereby nonzero matrix elements occur only in the six bands and in the main diagonal.

1.2 HPF+ kernel

The HPF+ kernel consists of a setup phase and a part where the calculations are performed. During the setup phase all required data are read in from an input data file, the data distribution is performed and and the arrays are initialised. The calculation part of the kernel consists of a linear, iterative solver loop, which is performed until the convergence criterion (residual value of less then 1e-10) is satisfied.

For the declaration of the one-dimensional processor array the NUMBER_OF_PROCESSORS statement was used, that allows a dynamic processor allocation at program start.

All arrays are defined in a common block in order to have the same structure as in the FIRE program. The distribution of the arrays among the processors depends on the data the arrays are referring to, so that two different cases have to be distinguished: For all arrays with dimension NNINTC (number of internal cells) a block distribution has been implemented, while the arrays with dimension NNCELL (number of internal and external cells) are distributed indirectly by the use of a map-array, which is read in at runtime. In order to minimise the amount of communication within the loops that are going over the internal cells, it is important that all the array elements referring to internal cells are distributed to the same processor for both type of arrays. The map array contains therefore a block distribution for all internal cells, while the external cells are distributed in a way that adjacent external and internal cells are send to the same processor: every pair of internal and external cells NC, NE where LCC(I,NC)=NE is distributed to the same processor.

```
!HPF$ PROCESSORS :: NODES(NUMBER_OF_PROCESSORS())
!HPF$ DYNAMIC :: DIREC1

C----READ MAP ARRAY
      OPEN(20,STATUS='OLD',FILE='map.dat',FORM='FORMATTED')
      READ(20,*) MAP
      CLOSE(20)

!HPF$ REDISTRIBUTE(INDIRECT(MAP)) :: DIREC1
```

The indirect distributed arrays are declared as dynamic and are redistributed after the map-array has been read. The map-array is generated previous to the calculation by a seperate program, and is read during runtime.

The program part inside the iteration loop is performed several hundred times, depending on the input data-set, until convergence is achieved. In this program part indirect data access is necessary, whereby the access pattern does not change from one iteration to the next. The reuse of communication schedules for loops inside the iteration loop becomes very important because the communication pattern for the indirectly accessed elements do not change.

```
!HPF$ INDEPENDENT, ON HOME(DIREC2(NC)), REUSE
        DO 70 NC=NINTCI,NINTCF
          DIREC2(NC)=BP(NC)*DIREC1(NC)
     X              -BS(NC)*DIREC1(LCC(1,NC))
     X              -BW(NC)*DIREC1(LCC(4,NC))
     X              -BL(NC)*DIREC1(LCC(5,NC))
     X              -BN(NC)*DIREC1(LCC(3,NC))
     X              -BE(NC)*DIREC1(LCC(2,NC))
     X              -BH(NC)*DIREC1(LCC(6,NC))
   70 CONTINUE
```

As the time measurements of the kernel have shown, scalability can only be achieved when the REUSE clause is implemented.

The ON HOME clause is used when arrays with different data distribution (BLOCK and INDIRECT) are used within the same loop. In the current version of the benchmark most loops are going over the internal cells, where the data distribution for all arrays is the same. Nevertheless the ON HOME clause is a necessary directive for the compiler, because the data distribution for indirectly distributed arrays is done at runtime.

1.2.1 HPF+ features used

The function NUMBER_OF_PROCESSORS has been used for a dynamic processor specification at program start.

The data distribution has been implemented by the BLOCK and INDIRECT directives, where in particular the INDIRECT distribution is very important for an efficient implementation of the HPF+ kernel. For the indirect data distribution the DYNAMIC and REDISTRIBUTE statements are required.

The INDEPENDENT directive has been used together with with the REUSE, the REDUCTION and the ON HOME clauses. The REUSE clause is very important for an efficient computation of all loops that are containing indirect data access. The computation of the global sums is implemented by the REDUCTION clause. Finally the ON HOME clause was used when arrays with different distribution are accessed within one loop.

1.2.2 Porting Effort

The porting of version 2 of the HPF+ kernel was based on version 1 of the kernel and the conjugate gradient equation solver of the FIRE program. Since the Fortran 77 source was extracted from the FIRE program the porting activities were focused on the parallelisation under HPF+, the implementation of the data structure and the I/O.

The primary interest for a code developer is to compare the porting effort for the HPF+ version to the hand coded message passing version. For the AVL kernel three different program parts can be compared: the effort required for the implementation of the data distribution, for reduction operations and for the handling of the indirect data access. The data distribution can be implemented very easily in HPF+ by using the distribute statement, while for the message passing version the correct distribution of the input data had to be implemented by a separate routine (xread.f). The porting effort for reduction operations is for both kernel versions equally small: in HPF+ the INDEPENDENT REDUCTION statement was used, while for the MPI version reduction operations are implemented by a call to MPI_ALLREDUCE. The main difference between the two versions regarding the porting effort arises from the indirect data access to array elements. In HPF+ the indirect data access can be handled by a simple INDEPENDENT REUSE directive, while for the MPI version two additional routines had to be developed, where send and receive lists are generated and the communication is performed with the help of these lists. Generally the parallelisation effort of the message passing version was significantly higher compared to the HPF+ version.

1.2.3 Message Passing version

With respect to the functionality the message passing (MPI) version and the HPF+ version are identical. The data distribution under MPI is implemented so that array elements are distributed in the same way as it is done in the HPF+ version: all arrays that refer to internal cells are distributed in blocks of equal size and the arrays over internal and external cells are distributed in a way that adjacent internal and external cells are sent to the same processor. The data distribution is done by the routine xread.f, where all necessary input data are read from a file and the distribution of the data to all nodes is done by one node.

For the indirect data access send and receive lists have to be computed after the connectivity array LCC is read and the data are distributed. During the computation these send and receive lists are used for a correct data exchange among the processors. The advantage of the message passing version is that the send and receive lists are computed dynamically during the calculation, which makes the program execution very comfortable: when the program has been compiled and linked it can be used for all different input data sets and for any processor configuration, while for the HPF+ version for every data set the program has to be compiled and linked again. The recompilation for the HPF+ version becomes necessary because a common block change has to be done when different data sets are used. This is due to the fact that all arrays referencing to internal cells are block-distributed. The block distribution depends on the dimension of the arrays that are specified in the common block. In order to obtain load balance the dimension in the common block has to be changed for every data set and the program must be recompiled. The major advantage of the HPF+ version is that the data distribution and the indirect data access is done automatically by the compiler, while for the MPI version two additional routines (cmplcom.f and cmpxchg.f) had to be developed.

1.3 Performance

For the performance evaluation of the HPF+ and the MPI kernel three input data sets of different size have been used as shown in table 1.

Dataset	No. of internal cells	No. of total cells
TJUNC	13845	19061
SWIRL	47312	60934
PENT	108000	124928

Table 1: Data set characteristics

The input data sets, that have been extracted from FIRE applications, contain the coefficients and the source term of the pressure equation and the connectivity array LCC.

For the MPI version the F77 compiler with option -O3 was used, while for the HPF+ version the SUN F90 compiler was used with option -O3 and without optimisation. The performance was measured for the computational part of the kernels (the iteration loop), neglecting the reading of the input data file and the setup phase. The performance obtained for the HPF+ version of the benchmark is shown in table 2.

Dataset	Compiler	1p	2p	4p	8p	16p	32p
TJUNC	f90	76.86	58.05	32.16	23.42	21.98	29.88
	Speedup		1.32	2.39	3.28	3.49	2.57
	f90 -O3	65.01	44.80	36.94	26.10	19.22	27.93
	Speedup		1.45	1.76	2.49	3.38	2.33
SWIRL	f90	276.66	152.64	97.72	78.29	65.02	70.08
	Speedup		1.81	2.83	3.53	4.25	3.95
	f90 -O3	213.25	125.53	83.86	79.85	65.31	77.96
	Speedup		1.69	2.54	2.67	3.26	2.73

Table 2: Performance of HPF+ kernel version (times in seconds)

For the HPF+ kernel it was not possible to use the input data set PENT. The optimisation option -O3 led to faster computation when running on up to four processors. When a higher number of processors was used to non-optimised executable was equally fast in some cases even faster. The first iteration, requires significantly more time than the subsequent loops, which is due to the fact that the first loop iteration contains the inspector execution.

The performance obtained for the MPI Version of the kernel is shown in table 3. In this table, speedup has been calculated with respect to the serial execution time of each version. For the MPI kernel the speedup is close to ideal, as long as the problem size is sufficiently large. When running the TJUNC data set, which consists of 13845 internal cells, on 16

Dataset	Compiler	1p	2p	4p	8p	16p	32p
TJUNC	f77 -O3	1.53	0.77	0.47	0.27	0.24	-
	Speedup		1.98	3.25	5.66	6.37	-
SWIRL	f77 -O3	5.44	2.74	1.38	0.77	0.49	-
	Speedup		1.98	3.94	7.06	11.10	-
PENT	f77 -O3	12.56	6.33	3.15	1.59	0.83	0.56
	Speedup		1.98	3.98	7.89	15.13	22.42

Table 3: Performance of MPI kernel version (times in seconds)

processors, each processor has less than 900 cells to compute, and the communication time is becoming dominant over the calculation time.

1.4 Discussion

The performance of the MPI version is significantly better compared to the HPF+ version, which is mainly caused by additional costs arising from the inspector loop. The inspector does in principle the same as the routine cmplcom.f of the MPI version. This routine is called during the setup phase of the MPI kernel, which is not within the measured part, while the inspector is performed in the first iteration. The execution time for routine cmplcom.f depends on the number of processors used and requires approximately 8 to 10 % of the measured execution time of the kernel. Therefore the timings of the HPF+ and the MPI kernel can not be compared directly.

2 ECMWF

2.1 Introduction

In this section of the report, three version 2 kernels from ECMWF (LG, LM and SL) are evaluated. These kernels represent more complex features of the IFS weather forecast model which is briefly described in the following section. After this, the kernels are each evaluated separately, containing descriptions of the major data structures and language features used in the HPF+ and MPI versions. In this evaluation, the performance of the HPF+ and MPI kernels are compared against a serial version of the same kernel. These kernel versions are all compiled using the same SUN F90 compiler (f90: WorkShop Compilers 4.2 10/22/96 FORTRAN 90 1.2) and optimisation level (-O3), to facilitate a more straightforward comparison of language features. It should be noted here that VFC is a pre-processor that generates F90 source code, including calls to a HPF+ runtime library. Finally, a summary of the main points found during the evaluation and future HPF+ requirements are presented in a final discussion section.

2.1.1 IFS

The Integrated Forecasting System (IFS) of the European Centre for Medium Range Weather forecasts uses a spectral forecast model for the prediction of weather for a period of up to 10 days ahead. This model is highly parallel and has been implemented today on both distributed and shared memory systems. Given a strong desire to protect the scientific code from details of the parallel implementation, a transposition strategy is used. With this approach, the complete data required is redistributed at various stages of a time step so that the arithmetic computations between two consecutive transpositions can be performed without any interprocess communication. Such an approach is feasible because data dependencies in the forecast model exist only in one coordinate direction, this direction being different for each algorithmic component. An overwhelming practical advantage of this technique is that the interprocessor communication is localised in a few routines. The transpositions are executed prior to the appropriate algorithmic stage, so that the computational routines (which constitute the vast bulk of the IFS source code) need have no knowledge of this activity. In terms of HPF, the transpositions are no more than redistributions that would permit more efficient data distributions to be used during an algorithmic stage. However, it is the representation of the data spaces used in the IFS that presents a technical difficulty for HPF as they are all irregular. Grid point space, Fourier space and spectral space cannot be represented by simple BLOCK or CYCLIC distributions without the severe overhead of additional interprocessor communication during the algorithmic stages. Instead GEN_BLOCK and INDIRECT distributions are required for the IFS as explained in the following sections which describe the data spaces in the IFS.

2.1.2 Grid point space

Grid point space is a quasi-regular (so-called reduced) 3 dimensional grid in which the number of grid points per latitude is reduced as the poles are approached. For example, at model resolution T213 each field has 640 grid points on equatorial latitudes and only 18 grid points on latitudes at the north and south poles. The number of grid points for a single model variable at one level in the atmosphere is shown in Table 4.

IFS Spectral Resolution	Grid Points
T63	13280
T106	35718
T213	134028

Table 4: Number of Grid Points for different IFS configurations

With the exception of the semi-Lagrangian (SL) calculations, grid-point computations only involve vertical dependencies. This would permit use of a simple BLOCK distribution for the horizontal (i.e. lat, lon) domain of grid-point space, which is represented in a single array dimension. However, the SL advection computations require access to grid point data in the upwind direction in order to compute departure points. Using a BLOCK distribution for grid-point space would result in a large communication overhead as the number of processors increase. This is due to the processors containing a decreasing number of latitudes and the partitions becoming long and thin. One way to resolve this data dependency would be to minimise the amount of data required from neighbouring processors. This would require a 2 dimensional partitioning of grid point space, which can be represented by use of an INDIRECT distribution.

2.1.3 Fourier space

Fourier space is similar to grid point space in its structure, with the number Fourier coefficients for each latitude reducing towards the poles. At model resolution T213, each field has 213 such coefficients at the equator while only 8 at the poles. Fast Fourier transforms (FFTs) are used to transform grid point space to Fourier space. These require access to all grid points on a latitude, and are independent over latitudes and fields. The distribution of Fourier space can be achieved by using a combination of GEN_BLOCK for complete latitudes and BLOCK for fields. However, it should be noted that the Fourier transforms used in the IFS vectorise over fields, and any distribution of fields (such as BLOCK) would result in shorter vector lengths and a reduction in vector efficiency. Legendre transforms (LTs) are used to transform fourier space to spectral space. These transforms require access to all latitudes for a particular wave number (m), and are independent of each other and of fields. This dependency is a case for another (GEN_BLOCK,BLOCK) distribution of Fourier space, with a transposition to get from the previous distribution suited for FFTs to one suited for LTs.

2.1.4 Spectral space

Spectral space is a 3 dimensional space where two of the dimensions represent a triangular data structure (m,n) where m is the number of zonal wave numbers and n the number of total wave numbers. The third dimension of spectral space represent levels in the atmosphere. Computations in spectral space have different data dependencies in various phases and have either no dependencies (horizonal diffusion calculation), or a dependency over n. For these reasons, spectral wave space would be distributed GEN_BLOCK where for practical and communication efficiency reasons the zonal wave numbers (m) would be distributed as in the Legendre transforms.

2.2 LG kernel

The original (version 1) LG kernel was written to perform IFS transpositions in grid-point space. The LG kernel has now been enhanced to address more of the IFS program issues in grid-point space, namely,

- Indexed access to latitude data structures. This is required as the LTOG and GTOL transpositions need to redistribute grid-point space where each latitude includes some extra space required by the FFT package used at ECMWF. As a result, the transpositions can no longer be expressed as a simple array assignments as seen in version 1 of the LG kernel.
- Addition of computational routine SIMFFT to simulate E-W data dependencies of Fast Fourier Transforms.
- Addition of computational routine SIMPHYS to simulate vertical dependencies of physics calculations.
- Blocking of grid-point space to conserve memory usage in physics calculations and exploit locality of reference.

The serial version of the GTOL transposition (kernel fragment) is coded as,

```
REAL ZGA(NPROMA,NGTO,NBLOCKS)
REAL ZGL(NPROMAG,NGTO)
REAL ZTMP(NGPTOTG)

DO JFLD=1,NGTO
  IB=0
  DO J=1,NGPTOTG,NPROMA
    IBEG=1
    IEND=MIN(NPROMA,NGPTOTG-J+1)
    IB=IB+1
    DO JJ=IBEG,IEND
      ZTMP(J+JJ-1)=ZGA(JJ,JFLD,IB)
    ENDDO
  ENDDO
  DO JGL=1,NDGLG
    DO JL=1,NLOENG(JGL)
      ZGL(NSTABUFL(JGL)+JL,JFLD)=ZTMP(NSTABUFG(JGL)+JL-1)
    ENDDO
  ENDDO
ENDDO
```

where NGT0 is the number of fields to be transposed, NGPTOTG the total number of grid points, NPROMA the block size (or vector length), NDGLG the number of latitudes and NLOENG the number of grid points on each latitude. In the above kernel fragment the blocking of ZGA (by NPROMA) is required to conserve space in physics routines which contain some automatic arrays dimensioned (NPROMA, NFLEV, NFLEV) where NFLEV is the number of model levels (31 today). Without blocking, each of these arrays at the operational T213L31 resolution would be just over 1 Gbyte. With blocking, the space used by these arrays is reduced by the factor 134028/NPROMA or NBLOCKS. The difficulty for HPF is how to distribute ZGA, as the first and third dimensions together represent a field that we would like to distribute INDIRECT. This issue is addressed in the LG kernel by only using the blocked structure in a HPF_LOCAL context, and an INDIRECT distribution (unblocked) outside of this. Another issue (an implementation constraint of VFC) is that the bounds of nested independent loops must be invariant, which would be violated by the (JGL,JL) loops above. The workaround for this constraint is to collapse the (JGL,JL) loops into one (J) loop and use a pre-computed index array (INDL).

The HPF+ version for the same kernel extract is therefore coded as,

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),INDIRECT(MAPFLD0)) ONTO PROCSAB :: ZGL
      REAL ZGL(NPROMAG,NGT0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZGA
      REAL ZGA(NGPTOTG,NGT0)

!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JFLD=1,NGT0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL(INDL(J),JFLD)=ZGA(J,JFLD)
          ENDDO
        ENDDO
```

As can be seen in the above fragment, the LG kernel uses (GEN_BLOCK,INDIRECT) distributions for ZGL, i.e. more than one array dimension is distributed. This is an enhancement from the version 1 LG kernel where only (GEN_BLOCK,*) could be used.

The use of an INDIRECT distribution to distribute (NGT0) fields over NPROCB processors is required as both 3D variables (i.e. those with a vertical dimension) and 2D variables (surface fields) are included in this dimension. In this case we need to keep the same set of vertical levels for all 3D variables distributed to the same NPROCB processor. This is needed in the IFS fourier space computations, where calculations are performed in fourier space over latitudes and levels. To avoid communication we need variables with the same latitudes and levels in the atmosphere to be in the same processor. However, if we hold 3D and 2D variables in separate arrays we can avoid the use of an INDIRECT distribution and use a more efficient BLOCK distribution for both, as follows,

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK,*) ONTO PROCSAB :: ZGL3
      REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK) ONTO PROCSAB :: ZGL2
```

```
      REAL ZGL2(NPROMAG,NG2D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*,*) ONTO PROCS :: ZGA3
      REAL ZGA3(NGPTOTG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: ZGA2
      REAL ZGA2(NGPTOTG,NG2D0)

!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL2(INDL(J),JVAR)=ZGA2(J,JVAR)
          ENDDO
        ENDDO
```

This is at the expense of doubling the size of such code sections, but acceptable given a potential gain in performance. In the HPF+ kernel we have used HPF_LOCAL as a solution to the memory usage/blocking issue of grid-point space, however, it is also interesting to explore whether HPF can still be used efficiently in physics routines should we choose to accept the memory overhead. A version of SIMPHYS has therefore been coded using HPF+ directives to explore this issue. This version requires the use of IF statements with distributed variables used in the body. However, such loops are currently not parallelised by VFC today, and as a result are left to run serially. Consequently, this variant of the LG kernel has been deferred to the final D1.2c evaluation.

2.2.1 HPF+ features used

The features used in the 3 HPF+ variants of the LG kernel are as follows.

Variant 1.

- grid-point space distribution, unblocked (INDIRECT,*)
- fourier space distribution, 3D and 2D variables (GEN_BLOCK,INDIRECT)
- SIMPHYS and SIMFFT coded as HPF_LOCAL
- GTOL and LTOG transpostions coded as 2 deep nested INDEPENDENT loops
- use of REUSE on INDEPENDENT loops

Variant 2.

- grid-point space distribution, unblocked (INDIRECT,*)
- fourier space distribution, 3D variables (GEN_BLOCK,BLOCK,*)
- fourier space distribution, 2D variables (GEN_BLOCK,BLOCK)
- SIMPHYS and SIMFFT coded as HPF_LOCAL

- GTOL and LTOG transpostions coded as 3 and 2 deep nested INDEPENDENT loops
- use of REUSE on INDEPENDENT loops

Variant 3.

- grid-point space distribution, unblocked (INDIRECT,*)
- fourier space distribution, 3D variables (GEN_BLOCK,BLOCK,*)
- fourier space distribution, 2D variables (GEN_BLOCK,BLOCK)
- SIMPHYS coded with multiple nested INDEPENDENT loops
- SIMFFT coded as PUREST subroutine called within a 3 deep nested INDEPENDENT loop
- GTOL and LTOG transpostions coded as 3 and 2 deep nested INDEPENDENT loops
- use of REUSE on INDEPENDENT loops

2.2.2 Porting Effort

The LG kernel was relatively straightforward to code in HPF+ and certainly easier than the equivalent message passing version. HPF_LOCAL provides a simple and easy to use facility for switching between global and local programming methods and can be used to make HPF more flexible. Porting or writing HPF programs would be made even easier if HPF extensions were exclusively restricted to directives. Extensions such as PURE or EXTRINSIC are not recognised by Fortran 90 compilers and need to be removed before such compilers are used. Another area that could be improved would be the syntax for INDEPENDENT directives for nested loops, as shown in the following example,

```
!HPF$ INDEPENDENT,NEW(J),REDUCTION(SUM),ON HOME(X(:,:,K)),REUSE
      DO K=...
!HPF$ INDEPENDENT,NEW(I),ON HOME(X(:,J,:))
        DO J=...
!HPF$ INDEPENDENT,NEW(TEMP),ON HOME(X(I,:,:))
          DO I=...
            TEMP=X(I,J,K)
            CALL SUB(TEMP)
            SUM=SUM+TEMP
          ENDDO
        ENDDO
      ENDDO
```

In the above example, assume X is distributed and SUB is PUREST. Coding this loop nest is non-trivial for the programmer and potentially prone to error.

A more user friendly syntax would be similar to that of HPF_CRAFT, where a single directive is used for the whole loop nest, and control variables are listed separately from NEW variables, as follows,

```
!HPF$ INDEPENDENT(K,J,I),NEW(TEMP),REDUCTION(SUM),ON HOME(X(I,J,K),REUSE
      DO K=...
        DO J=...
          DO I=...
            TMP=X(I,J,K)
            CALL SUB(TEMP)
```

```
      SUM=SUM+TEMP
    ENDDO
  ENDDO
ENDDO
```

Finally, when using HPF_LOCAL it is somtimes necessary to know which processor a particular process is executing on. The HPF_LOCAL_LIBRARY provides such a function MY_PROCESSOR() for this purpose. As the HPF_LOCAL_LIBRARY and in particular the function MY_PROCESSOR() were not available with the current release of the VFC compiler, a workaround was used, by passing a BLOCK distributed array of length NUMBER_OF_PROCESSORS to the HPF_LOCAL routine. This array was initialised to contain the required processor numbers.

2.2.3 Other HPF compilers

A variant of the LG kernel was coded for the NAS HPFplus compiler V2.0. As this compiler release did not support the INDEPENDENT directive, extrinsic HPF_LOCAL, INDIRECT distributions or GEN_BLOCK distributions, it was decided to start coding from the serial version of the kernel. HPF_SERIAL was used to parallelise SIMPHYS and SIMFFT, by calling these subroutines in a do loop over NBLOCKS (i.e. the number of blocks in grid-point space) and over NGT0 (the number of fields) respectively. BLOCK distribution was used for the corresponding dimensions NBLOCKS and NGT0. Attempts to parallelise the transpositions GTOL and LTOG using a HPF_SERIAL approach were not successful, and these were left to run serially. Parallelisation using the NAS HPFplus compiler was made difficult by the lack of compiler diagnostics to indicate why DO loops did not parallelise. The user is left to inspect the intermediate code (.cmm) or less intelligible generated code (.f90) to see whether parallelisation was successful. It was subsequently found that HPFplus V2.0 does not support the parallelisation of DO loops, instead focusing on the support of array syntax. In contrast, VFC provides very clear diagnostics in this area. Another issue that presented a problem for the NAS HPFplus compiler was the use of Fortran 90 pointers in common blocks. While this construct is accepted by many Fortran 90 compilers (and is valid Fortran 90) it is considered an error by the NAS HPFplus compiler. Note, it is used in the IFS (and the ECMWF kernels) because the early Cray F90 compilers (for the then in-house T3D) did not support modules and further allocatable arrays could not be put in common. An example of a typical use of POINTER in the LG kernel would be,

```
   ...
INTEGER,POINTER :: NLOENG(:)
   ...
COMMON /POIGEM/ ... ,NLOENG, ...
   ...
ALLOCATE( NLOENG(NDGSAG:NDGENG) )
```

The workaround for this problem was to recode such declarations to use the ALLOCATABLE attribute and place these within a MODULE, for example,

```
MODULE MODGEM
```

```
        ...
      INTEGER,ALLOCATABLE :: NLOENG(:)
        ...
    END MODULE MODGEM

       ...
    USE MODGEM
       ...
    ALLOCATE( NLOENG(NDGSAG:NDGENG) )
```

However, the use of allocatable arrays in modules is recognised as being a cleaner practice over pointers in common, and therefore one could view the workaround as being the original code. For this reason, and the fact that the production IFS is currently being converted to using modules, future ECMWF HPF+ kernels will be coded with allocatable arrays in modules and not use pointers in common.

2.2.4 Message Passing kernel

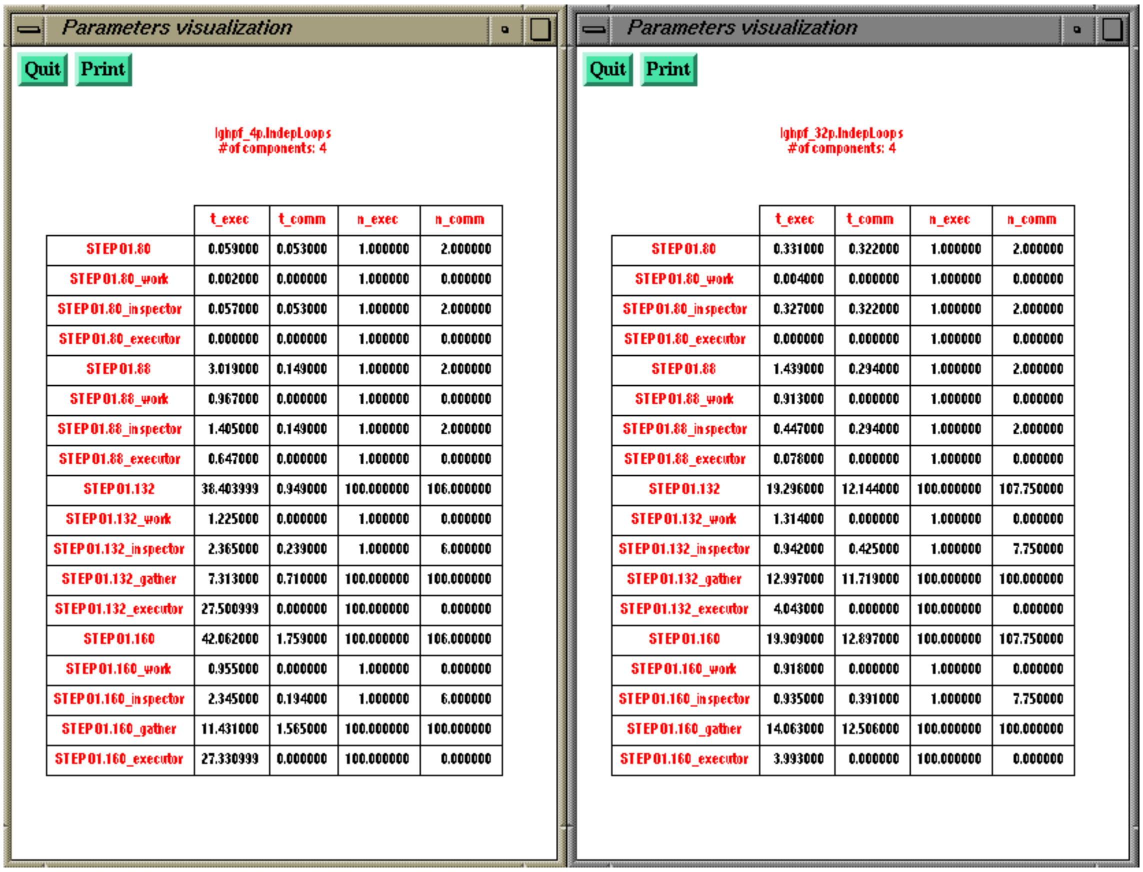
The message passing version of the LG kernel was prepared by adding the SIMFFT and SIMPHYS routines into the version 1 kernel, and including some initialisation code for the data structures used by these routines. As no communication is performed in these portions of the kernel, this update was relatively straightforward. In this kernel the transposition routines TRLTOG and TRGTOL are fairly complex (over 250 lines each), and have been optimised by precomputing their communication schedules at initialisation time. This should be contrasted with the HPF+ versions which are only a few lines each, and where the communication schedules are determined at runtime.

2.2.5 Performance

The LG kernel is characterised by two phases of compute (SIMFFT and SIMPHYS) where no communication is required, and two phases of communication, which are the GTOL and LTOG transpositions. Performance is summarised in Table 5 for three problem sizes, T63, T106 and T213, representing a small, medium and large problem, against an increasing number of processors. For each of the problem sizes, performance is shown for the serial kernel, message-passing kernel, two HPF+ variants and in the case of T63, also a NAS HPFplus variant. In this table, timing is presented in millesecs (m/s) per time-step and speed-up (s/u) is the performance attained relative to the serial version of the kernel. The performance of the message-passing (MPI) kernel shows near perfect scaling on up to 32 processors with a speed-up of 29 to 31. The HPF+ kernel variants do not perform quite as well as the MPI version, however, the performance is quite respectable with variant 1 showing a speed-up of 22 to 25 on 32 processors. Analysis of timers inserted in the code show that the difference in this performance can be attributed to the transpositions which run approximately 4 (1 proc) to 14 (32 procs) times slower than the corresponding code in the MPI versions. This is with the use of schedule reuse. The tracing facility of VFC can be used to investigate why the transposition loops perform as they do. Figure 1 shows detailed timing information displayed by the visualisation module of Medea for the HPF+ variant 1, and in particular,

Problem	Version		1p	4p	16p	32p
T63	SERIAL	ms	15,742			
		s/u	1.00			
	MPI	ms	16,631	4,025	1,024	541
		s/u	0.95	3.91	15.37	29.10
	HPF+ variant 1	ms	18,975	5,102	1,240	709
		s/u	0.83	3.09	12.70	22.20
	HPF+ variant 2	ms	19,003	4,770	1,268	743
		s/u	0.83	3.30	12.41	21.19
	NAS HPFplus variant	ms	104,709	30,474	12,858	9,116
		s/u	0.15	0.52	1.22	1.73
T106	SERIAL	ms	42,718			
		s/u	1.00			
	MPI	ms	42,788	10,653	2,768	1,391
		s/u	1.00	4.01	15.43	30.71
	HPF+ variant 1	ms	51,333	12,712	3,386	1,729
		s/u	0.83	3.36	12.62	24.71
	HPF+ variant 2	ms	51,353	12,935	3,397	1,781
		s/u	0.83	3.30	12.58	23.99
T213	SERIAL	ms	164,908			
		s/u	1.00			
	MPI	ms	not enough	42,258	10,723	5,262
		s/u	memory	3.90	15.38	31.34
	HPF+ variant 1	ms	not enough	49,472	13,155	6,637
		s/u	memory	3.33	12.54	24.85
	HPF+ variant 2	ms	not enough	49,825	12,797	6,552
		s/u	memory	3.31	12.89	25.17

Table 5: Comparison of run times (in millesecs per time step) and speedups for the versions of the LG kernel for different numbers of processors on a QSW CS2



	t_exec	t_comm	n_exec	n_comm
STEP01.80	0.059000	0.053000	1.000000	2.000000
STEP01.80_work	0.002000	0.000000	1.000000	0.000000
STEP01.80_inspector	0.057000	0.053000	1.000000	2.000000
STEP01.80_executor	0.000000	0.000000	1.000000	0.000000
STEP01.88	3.019000	0.149000	1.000000	2.000000
STEP01.88_work	0.967000	0.000000	1.000000	0.000000
STEP01.88_inspector	1.405000	0.149000	1.000000	2.000000
STEP01.88_executor	0.647000	0.000000	1.000000	0.000000
STEP01.132	38.403999	0.949000	100.000000	106.000000
STEP01.132_work	1.225000	0.000000	1.000000	0.000000
STEP01.132_inspector	2.365000	0.239000	1.000000	6.000000
STEP01.132_gather	7.313000	0.710000	100.000000	100.000000
STEP01.132_executor	27.500999	0.000000	100.000000	0.000000
STEP01.160	42.062000	1.759000	100.000000	106.000000
STEP01.160_work	0.955000	0.000000	1.000000	0.000000
STEP01.160_inspector	2.345000	0.194000	1.000000	6.000000
STEP01.160_gather	11.431000	1.565000	100.000000	100.000000
STEP01.160_executor	27.330999	0.000000	100.000000	0.000000

	t_exec	t_comm	n_exec	n_comm
STEP01.80	0.331000	0.322000	1.000000	2.000000
STEP01.80_work	0.004000	0.000000	1.000000	0.000000
STEP01.80_inspector	0.327000	0.322000	1.000000	2.000000
STEP01.80_executor	0.000000	0.000000	1.000000	0.000000
STEP01.88	1.439000	0.294000	1.000000	2.000000
STEP01.88_work	0.913000	0.000000	1.000000	0.000000
STEP01.88_inspector	0.447000	0.294000	1.000000	2.000000
STEP01.88_executor	0.078000	0.000000	1.000000	0.000000
STEP01.132	19.296000	12.144000	100.000000	107.750000
STEP01.132_work	1.314000	0.000000	1.000000	0.000000
STEP01.132_inspector	0.942000	0.425000	1.000000	7.750000
STEP01.132_gather	12.997000	11.719000	100.000000	100.000000
STEP01.132_executor	4.043000	0.000000	100.000000	0.000000
STEP01.160	19.909000	12.897000	100.000000	107.750000
STEP01.160_work	0.918000	0.000000	1.000000	0.000000
STEP01.160_inspector	0.935000	0.391000	1.000000	7.750000
STEP01.160_gather	14.063000	12.506000	100.000000	100.000000
STEP01.160_executor	3.993000	0.000000	100.000000	0.000000

Figure 1: Independent loop and communication statistics displayed by Medea for the LG HPF+ kernel executing on 4 and 32 processors for 100 time steps at T63 resolution.

code section		1p	4p	16p	32p
GTOL	without REUSE	10,561	3,865	2,775	2,435
	with REUSE	1,444	591	114	119
SIMFFT	HPF_LOCAL	1,917	477	100	54
LTOG	without REUSE	10,548	3,645	2,393	2,031
	with REUSE	1,601	547	156	97
SIMPHYS	HPF_LOCAL	13,730	3,425	854	427
TOTAL	without REUSE	36,758	11,412	6,122	4,947
	with REUSE	18,692	5,040	1,224	697
	improvement factor	2.0	2.3	5.0	7.1

Table 6: Analysis of performance for LG HPF+ (variant 1) kernel for T63 problem on different numbers of processors on a QSW CS2 (times in millesecs)

the GTOL transposition (starting at stepo1 line 132) and the LTOG transposition (starting at stepo1 line 160). From this information it can be seen that although the processor count has increased by a factor of 8, the time for the transpositions have only reduced by a factor of 2. Without instrumentation this is closer to a factor of 4. Also it can be seen that the poor scaling is mainly due to the gather phase for the transposition loops.

The benefit of schedule reuse can be clearly seen in table 6. On a single processor the transpositions run 6 times faster with schedule REUSE, while on 32 processors they run about 20 times faster. The overall benefit of schedule reuse on the total time is an improvement of a factor of 7 on 32 processors.

The summary of performance shows that the two HPF+ variants achieve more or less the same performance. This suggests that there is little benefit in adopting a coding approach of separating 3D variables from 2D variables, so as to use a simpler (BLOCK) distribution over the fields dimension.

As for the NAS HPFplus version, this performed several times slower than the serial code when run on one processor. This is probably due to the amount of indirect addressing in the kernel, which is expected to be addressed in a future compiler release. Aside from this, some scaling was observed, although this is limited by the transpositions running sequentially.

Finally, the compilation time was measured for the serial, MPI and HPF+ versions of the LG kernel and is shown in table 7. As can be seen the compilation time for the HPF+ version is an order of magnitude longer than for the serial and MPI versions. Of this time, it should be noted that the VFC pre-processor time is only 11 seconds and the excessive compilation time is due to the SUN f90 compiler (version 1.2) compiling the VFC generated source. When instrumentation is selected (i.e. -mci) a further increase in compilation time is observed.

LG kernel version	compiler	user	system	wall
SERIAL	f90	105	39	224
MPI	f90	416	103	598
HPF+	vfc + f90	3,351	71	3,569
HPF+	vfc -mci + f90	3,676	179	3,984

Table 7: Compilation times for the different versions of the LG kernel on a QSW CS2 (times in seconds)

2.3 LM kernel

The LM kernel was developed to test IFS transpositions in fourier space. In these transpositions, fourier data is redistributed between a latitude distribution (required for fourier transforms) and a wave distribution (required for legendre transforms). For example, the serial code fragment for an LTOM transposition is coded as follows,

```
      REAL ZL0(NGT0,NFTOT2G)
      REAL ZM (NGT0,NFTOT2G)

      DO JGL=1,NDGLG
        DO JM=0,NMENG(JGL)
          IRL=NPNTGTB0G(JM,JGL)*2+1
          IIL=IRL+1
          IRM=NPNTGTB1G(JM,JGL)*2+1
          IIM=IRM+1
          DO JF=1,NGT0
            ZM(JF,IRM)=ZL0(JF,IRL)
            ZM(JF,IIM)=ZL0(JF,IIL)
          ENDDO
        ENDDO
      ENDDO
```

where NDGLG is the number of latitudes, NMENG the wave cut-off per latitude and NGT0 the number of fields to be transposed. This fragment can be coded in HPF+ using GEN_BLOCK for describing the layout of both distributions of fourier data over NPROCA processors, and INDIRECT for describing the distribution of fields over NPROCB processors. Like the LG kernel, the (JGL,JM) loops above have been collapsed into one (J) loop to workaround the VFC constraint that independent loop bounds must be invariant. In addition, the more complex index pointer code has been replaced by pre-computed values of index arrays I0 and I1 in the new J loop. As the distribution of fourier data is invariant over time steps we can ignore any setup cost for these arrays.

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCB,NPROCA)
!HPF$ DISTRIBUTE(INDIRECT(MAPFLD0),GEN_BLOCK(MAPFL)) ONTO PROCSAB :: ZL0
      REAL ZL0(NGT0,NFTOT2G)
!HPF$ DISTRIBUTE(INDIRECT(MAPFLD0),GEN_BLOCK(MAPFM)) ONTO PROCSAB :: ZM
      REAL ZM (NGT0,NFTOT2G)

!HPF$ INDEPENDENT,NEW(JF),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT
          DO JF=1,NGT0
            ZM(JF,I1(J)  )=ZL0(JF,I0(J)  )
            ZM(JF,I1(J)+1)=ZL0(JF,I0(J)+1)
          ENDDO
        ENDDO
```

Two HPF+ variants have been coded, selectable at runtime, one like the above where an INDIRECT distribution is used to distribute fields over NPROCB processors, and another (as in the LG kernel) where 3D and 2D variables are stored in separate arrays and distributed using BLOCK.

2.3.1 HPF+ features used

The LM kernel is characterised by the use of nested INDEPENDENT loops, and BLOCK, GEN_BLOCK and INDIRECT distributions. The transpositions used also involve a large amount of communication, and in each timestep there are some $NPROCB \times NPROCA \times (NPROCA - 1)$ message exchanges for each transposition. Note that NPROCA controls the parallelisation over latitudes and waves, and NPROCB the parallelisation over levels in the atmosphere, where the number of processors being used is $NPROCA \times NPROCB$. In this context we would typically set NPROCA to a large number and NPROCB small, as increasing NPROCB has the effect of reducing the vector performance of FFTs and legendre transforms. This kernel therefore represents a difficult problem for the IFS, especially on distributed memory systems with large numbers of processors.

2.3.2 Porting Effort

Development of the HPF+ version of the LM kernel was based on the serial version of the kernel. As mentioned earlier, the main difficulty was to resolve the conflict of the non-invariant loops used in the transpositions. The only other work that had to be done was to set up the integer arrays used to describe the distributions.

2.3.3 Other HPF compilers

The LM kernel was inspected for a conversion (from serial or HPF+ versions) to the NAS HPF-plus compiler. However, the current release of this compiler did not support GEN_BLOCK or INDEPENDENT loops, and as no practical alternative was available, the port was abandoned. The LM kernel, like the LG kernel, also uses POINTERs in common which are used for allocating data structures. While this is legal Fortran 90 and accepted by other Fortran 90 compilers, the NAS HPFplus compiler flags this as a compilation error.

2.3.4 Message Passing kernel

The message passing kernel has been coded using a similar technique to that used in the IFS. In this method, the distribution of fourier space is managed by index arrays that are optimised for the transpositions. This optimisation results in no gathering or scattering being required in the transposition message passing routines. In the real IFS, code also exists to perform recursive transpositions that reduce the total number of communications required for the transpositions. This recursive transposition method is only useful on large processor configurations (e.g. > 500 processors), and has therefore not been implemented in this kernel.

2.3.5 Performance

Performance is summarised in Table 8 for three problem sizes, T63, T106 and T213, representing a small, medium and large problem, against an increasing number of processors. For each of the problem sizes, performance is shown for the serial kernel, message-passing kernel, and two HPF+ variants. In this table, timing is presented in millesecs (m/s) per time-step and speed-up (s/u) is the performance attained relative to the serial version of the kernel. The performance of the message-passing (MPI) kernel shows some good scaling on up to 32 processors with a speed-up of 16 to 18. It should be noted that this kernel is dominated by the large amount of communication and no computation and as a result is sensitive to issues of machine topology and message passing library implementation.

Problem	Version		1p	4p	16p	32p
T63	SERIAL	ms	408.8			
		s/u	1.00			
	MPI	ms	372.3	99.6	35.9	25.2
		s/u	1.10	4.10	11.39	16.22
	HPF+ variant 1	ms	6,977	2,114	599	370
		s/u	0.06	0.19	0.68	1.10
	HPF+ variant 2	ms	8,139	2,453	764	479
		s/u	0.05	0.17	0.54	0.85
T106	SERIAL	ms	1,142.5			
		s/u	1.00			
	MPI	ms	1,029.6	289.6	90.0	56.4
		s/u	1.11	3.95	12.69	20.26
	HPF+ variant 1	ms	18,817	5,448	1,577	834
		s/u	0.06	0.21	0.72	1.37
	HPF+ variant 2	ms	22,077	6,776	1,995	1,146
		s/u	0.05	0.17	0.57	1.00
T213	SERIAL	ms	4,136.2			
		s/u	1.00			
	MPI	ms	3,878.2	1,110.8	307.9	219.5
		s/u	1.07	3.72	13.43	18.84
	HPF+ variant 1	ms	not enough	not enough	6,170	3,277
		s/u	memory	memory	0.67	1.26
	HPF+ variant 2	ms	not enough	not enough	7,417	4,145
		s/u	memory	memory	0.56	1.00

Table 8: Comparison of run times (in millesecs per time step) and speedups for the versions of the LM kernel for different numbers of processors on a QSW CS2

The HPF+ variants of the kernel, perform very poorly, running over 16 times slower than the serial version even on a single processor. This is with schedule REUSE. Figure 2 shows detailed timing information displayed by Medea when the HPF+ kernel (variant 1) is run on a single processor. It can be seen that this performance is due to a combination of the high cost for executor phases of the key INDEPENDENT loops (those executed 100 times) and for the corresponding gather/scatter phases. The high executor cost is mainly explained by the fact that the generated code for INDEPENDENT loops uses indirection arrays that contain the local array indices for accesses to the distributed arrays. On the other hand, the high cost for the gather/scatter phases is somewhat surprising considering the program is executing on a single processor, which has probably not been considered for optimisation. As with the LG kernel, the variant 2 HPF+ kernel shows no benefit in performance over variant 1, suggesting that it is better to have fewer INDEPENDENT loops albeit with more complex distributions. While the HPF+ performance of this kernel is poor, it should be noted that the large amount of independent computation in the IFS will have the effect of minimising its effect when we consider total application performance.

2.4 SL kernel

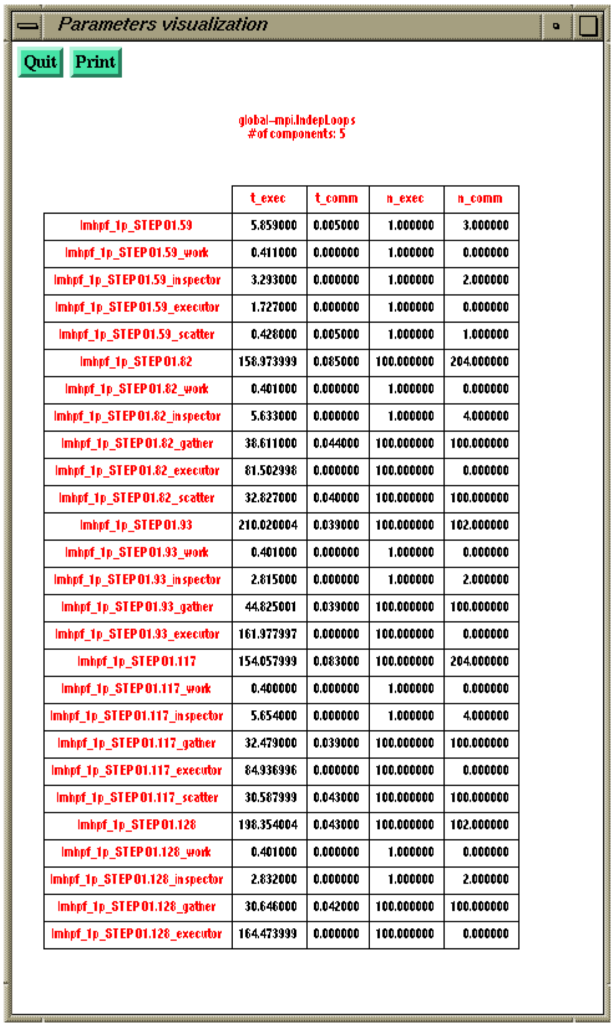
The semi-Lagrangian (SL) calculation in the IFS consists of two parts, the computation of a trajectory from a grid point backwards in time, and the interpolation of various quantities at the departure point and at the mid-point of the trajectory. The problem for distributed memory systems is that both these parts require access to grid-point data held on neighbouring processors. The approach used in the IFS today is to construct a halo of all grid columns from these processors based on the global maximum wind likely to be encountered as shown in Figure 3. A halo mechanism is required, as the computation of the trajectory for each grid-point is an iterative process (only 2 or 3 iterations due to cost), where the use of an inspector mechanism at each iteration would result in a high overhead.

2.4.1 HPF+ features used

The SL kernel computes such a halo, and in the case of the HPF+ version uses a HALO specification as part of INDIRECT distribution directive and and a UPDATE_HALO directive to indicate where the halo should be updated. This HALO specification simply specifies a data structure that describes for each processor, the list of non-local global indices that are needed. In the IFS, the SL halo data is only read and never written to. These language features while originally being proposed for inclusion in the present VFC release, are the subject of further investigation, and as a result have not being implemented. In particular, an alternative syntax is being looked at, whereby, the programmer would specify for each grid-point the set of non-local grid-points that potentially could be accessed. As a consequence of this ongoing development, the HPF+ kernel has been modified to run without the use of halos. It should be noted that the HPF-2 SHADOW feature is only applicable for regular nearest neighbour problems, and cannot be used in the above context where a single array dimension contains an irregular structure that is distributed INDIRECT.

2.4.2 Porting Effort

The HPF+ kernel was developed by starting with the version 1 LG kernel and adding initialisation routines for computing the halo required by each partition in grid-point space. The SL interpolation routines in the IFS are very complex and have many runtime controls that specify, among other things, the interpolation method to be used (e.g. 32 point stencil, 3D interpolation) and the variables to be interpolated. For this reason they are only approxi-



	t_exec	t_comm	n_exec	n_comm
lmhpf_1p_STEP01.59	5.859000	0.005000	1.000000	3.000000
lmhpf_1p_STEP01.59_work	0.411000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.59_inspector	3.293000	0.000000	1.000000	2.000000
lmhpf_1p_STEP01.59_executor	1.727000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.59_scatter	0.428000	0.005000	1.000000	1.000000
lmhpf_1p_STEP01.82	158.973999	0.085000	100.000000	204.000000
lmhpf_1p_STEP01.82_work	0.401000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.82_inspector	5.633000	0.000000	1.000000	4.000000
lmhpf_1p_STEP01.82_gather	38.611000	0.044000	100.000000	100.000000
lmhpf_1p_STEP01.82_executor	81.502998	0.000000	100.000000	0.000000
lmhpf_1p_STEP01.82_scatter	32.827000	0.040000	100.000000	100.000000
lmhpf_1p_STEP01.93	210.020004	0.039000	100.000000	102.000000
lmhpf_1p_STEP01.93_work	0.401000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.93_inspector	2.815000	0.000000	1.000000	2.000000
lmhpf_1p_STEP01.93_gather	44.825001	0.039000	100.000000	100.000000
lmhpf_1p_STEP01.93_executor	161.977997	0.000000	100.000000	0.000000
lmhpf_1p_STEP01.117	154.057999	0.083000	100.000000	204.000000
lmhpf_1p_STEP01.117_work	0.400000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.117_inspector	5.654000	0.000000	1.000000	4.000000
lmhpf_1p_STEP01.117_gather	32.479000	0.039000	100.000000	100.000000
lmhpf_1p_STEP01.117_executor	84.936996	0.000000	100.000000	0.000000
lmhpf_1p_STEP01.117_scatter	30.587999	0.043000	100.000000	100.000000
lmhpf_1p_STEP01.128	198.354004	0.043000	100.000000	102.000000
lmhpf_1p_STEP01.128_work	0.401000	0.000000	1.000000	0.000000
lmhpf_1p_STEP01.128_inspector	2.832000	0.000000	1.000000	2.000000
lmhpf_1p_STEP01.128_gather	30.646000	0.042000	100.000000	100.000000
lmhpf_1p_STEP01.128_executor	164.473999	0.000000	100.000000	0.000000

Figure 2: Independent loop and communication statistics displayed by Medea for the LM HPF+ kernel executing on 1 processor for 100 time steps at T63 resolution.

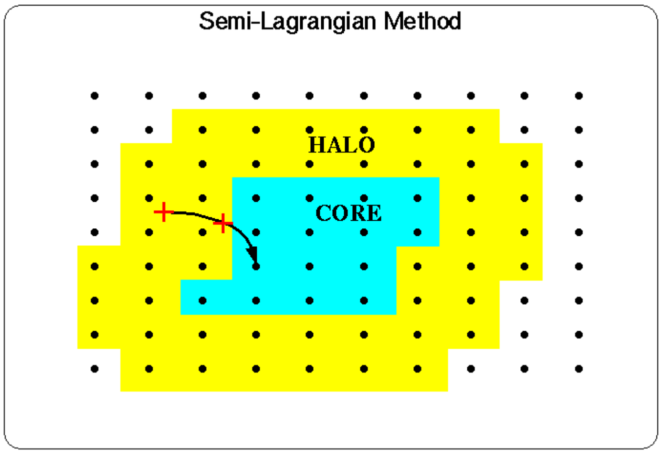


Figure 3: Semi Lagrangian method, showing computation of trajectory, requiring interpolation of grid-point data held on neighbouring processors (halo area)

mated in the HPF+ kernel, by a simple average of neighbouring grid-points, as shown in the HPF+ extract below,

```
!HPF$ INDEPENDENT,NEW(J),REDUCTION(ZSL2)
         DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(I1,I2,I3,I4,IB,IT,Z1,Z2,Z3,Z4,Z5,Z6)
           DO J=1,NGPTOTG
             I1=NSLIND(1,J,JSL)
             I2=NSLIND(2,J,JSL)
             I3=NSLIND(3,J,JSL)
             I4=NSLIND(4,J,JSL)
             IB=MAX(1,JLEV-1)
             IT=MIN(NFLEVG,JLEV+1)
             Z1=ZSL1(I1,JLEV)
             Z2=ZSL1(I2,JLEV)
             Z3=ZSL1(I3,JLEV)
             Z4=ZSL1(I4,JLEV)
             Z5=ZSL1(I2,IT)
             Z6=ZSL1(I3,IB)
             ZSL2(J,JLEV)=ZSL2(J,JLEV)+
     +         (Z1+Z2+Z3+Z4+Z5+Z6)/(6.*ISL)
           ENDDO
         ENDDO
```

However, these interpolation loops were sequentialised by VFC, due to the local integer scalars being both defined in the inner loop and used in the subscript of mapped array ZSL1. This was resolved by applying scalar forward substitution for these scalars, as shown in the recoded version,

```
!HPF$ INDEPENDENT,NEW(J),REDUCTION(ZSL2)
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(Z1,Z2,Z3,Z4,Z5,Z6)
```

```
          DO J=1,NGPTOTG
            Z1=ZSL1(NSLIND(1,J,JSL),JLEV)
            Z2=ZSL1(NSLIND(2,J,JSL),JLEV)
            Z3=ZSL1(NSLIND(3,J,JSL),JLEV)
            Z4=ZSL1(NSLIND(4,J,JSL),JLEV)
            Z5=ZSL1(NSLIND(2,J,JSL),MAX(1,JLEV-1))
            Z6=ZSL1(NSLIND(3,J,JSL),MIN(NFLEVG,JLEV+1))
            ZSL2(J,JLEV)=ZSL2(J,JLEV)+
     +        (Z1+Z2+Z3+Z4+Z5+Z6)/(6.*ISL)
          ENDDO
        ENDDO
```

This transformation will be performed automatically by VFC in a future release of the compiler.

2.4.3 Other HPF compilers

As halo support was not provided in the current VFC release, there seemed little benefit in porting the HPF+ kernel to run with another HPF compiler. Further, the restrictions of NAS HPFplus compiler described in section 2.3.3 precluded a simple conversion.

2.4.4 Message Passing kernel

A message passing kernel was developed to use the same SL halo strategy as used in the IFS today. This involved determining in initialisation those non-local grid columns that could be reached from a local processors partition during a time step period using a global maximum wind speed. The message receive lists were then compiled by examining which processor owned each of the non-local grid columns identified by global (lat,lon) coordinates. To compile the corresponding send lists a global communication was performed. Thereafter, in each timestep, inter-processor communication only involved nearest neighbour processors to obtain the halo.

2.4.5 Performance

The performance of the SL kernel is summarised in Table 9. As with the other ECMWF kernels, runs were performed on 1, 4, 16 and 32 processors. The message passing version showed scaling of 11 to 23 on 32 processors for different problem sizes. It should be noted that the halo data volume does not reduce in proportion to a corresponding reduction in processor grid-points. This non-scaling issue and high communication to compute ratio mainly explain the observed performance for the message passing kernel.

The HPF+ kernel performs very slowly, over 60 times slower than the serial version at T63 resolution. Figure 4 shows detailed timing information displayed by Medea when the HPF+ kernel is run on a single processor. From this timing information we can see that the reason for the poor performance is the inspector overhead for the interpolation loop nest (starting at stepo line 100). This performance problem is expected to be resolved in this case by the implementation of user specified halos in VFC, which obviate the need to call an inspector.

Problem	Version		1p	4p	16p	32p
T63	SERIAL	ms	1,029.0			
		s/u	1.00			
	MPI	ms	1,110.6	304.3	105.3	86.7
		s/u	0.93	3.38	9.77	11.87
	HPF+	ms	71,008	21,459	8,906	
		s/u	0.014	0.048	0.116	
T106	SERIAL	ms	4,153.4			
		s/u	1.00			
	MPI	ms	4,325.7	1,161.7	374.2	269.1
		s/u	0.96	3.58	11.10	15.43
	HPF+	ms	not enough	92,326	31,239	
		s/u	memory	0.045	0.133	
T213	SERIAL	ms	17,532.8			
		s/u	1.00			
	MPI	ms	not enough	4,467.3	1,281.6	754.6
		s/u	memory	3.92	13.68	23.2
	HPF+	ms	not enough	not enough	108,446	
		s/u	memory	memory	0.162	

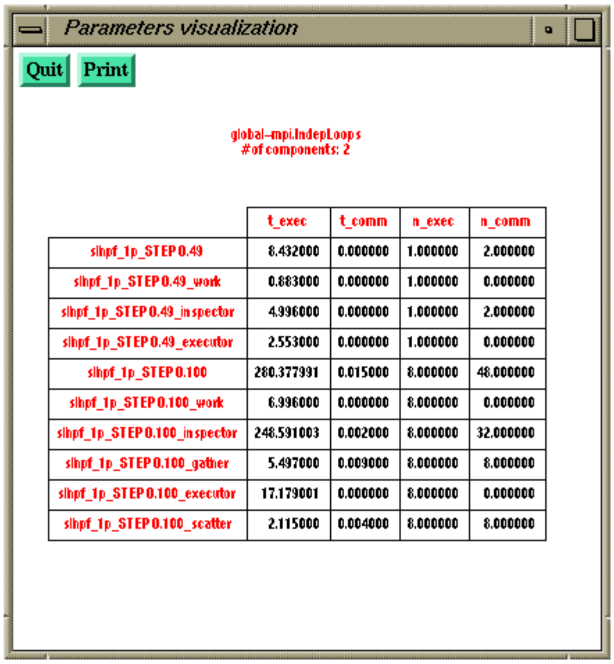
Table 9: Comparison of run times (in millesecs per time step) and speedups for the versions of the SL kernel for different numbers of processors on a QSW CS2

2.5 Discussion

The new Fortran 90 based VFC HPF+ compiler is a significant improvement over the previous VFCS (release 1) HPF+ compiler. In terms of usability, it provides good diagnostics, an easy to use command line interface for instrumentation and the separate compilation of subroutines.

As for HPF+ language features, the most significant for the ECMWF kernels is the implementation of communication schedule reuse by supporting a conditional REUSE clause for INDEPENDENT loops. The benefit of this feature can be clearly seen in the analysis of performance of the LG kernel as summarised in table 6. However, schedule reuse comes with a cost of increased memory usage which is a function of the communication and distributions used in the associated INDEPENDENT loops. This cost is expected to be instrumentated in the next release of VFC.

The performance of INDEPENDENT loops, although improved with schedule reuse, are still expensive. This is mainly due to the fact that the generated code for such loops use indirection arrays that contain the local array indices for accesses to distributed arrays. It should be noted that recoding regular loops to array syntax avoids such overheads, however, in the case of the IFS such loops are few and far between.



	t_exec	t_comm	n_exec	n_comm
slhpf_1p_STEP0.49	8.432000	0.000000	1.000000	2.000000
slhpf_1p_STEP0.49_work	0.883000	0.000000	1.000000	0.000000
slhpf_1p_STEP0.49_inspector	4.996000	0.000000	1.000000	2.000000
slhpf_1p_STEP0.49_executor	2.553000	0.000000	1.000000	0.000000
slhpf_1p_STEP0.100	280.377991	0.015000	8.000000	48.000000
slhpf_1p_STEP0.100_work	6.996000	0.000000	8.000000	0.000000
slhpf_1p_STEP0.100_inspector	248.591003	0.002000	8.000000	32.000000
slhpf_1p_STEP0.100_gather	5.497000	0.009000	8.000000	8.000000
slhpf_1p_STEP0.100_executor	17.179001	0.000000	8.000000	0.000000
slhpf_1p_STEP0.100_scatter	2.115000	0.004000	8.000000	8.000000

Figure 4: Independent loop and communication statistics displayed by Medea for the SL HPF+ kernel executing on 1 processor for 4 time steps at T63 resolution.

The use of HPF_LOCAL provides a useful mechanism for handling code sections where no communication is required which cannot be simply expressed by array syntax. This is used in the LG kernel variants 1 and 2. HPF_LOCAL also provides a means by which the blocking of grid-point space can be achieved in a HPF program, which is used in the IFS to conserve memory usage in the physics routines.

VFC also supports nested INDEPENDENT loops, which is required by all the ECMWF kernels. This presently requires an INDEPENDENT directive to be positioned in front of each DO statement in the nest and care in specification of NEW variables, and other clauses. A more user friendly syntax for nested INDEPENDENT loops is shown in 2.2.2, where a single directive is used. This syntax (or similar) is proposed for the next VFC HPF+ compiler release.

The present kernels have used some workarounds to avoid implementation restrictions, specifically for INDEPENDENT loops. However, to be able to handle more complex kernels, such restrictions will need to be relaxed, in particular,

- access to distributed arrays within the body of conditional statements
- use of non-invariant bounds for independent loops (i.e. bounds of inner loops dependent on outer loops)
- subscript expressions of distributed arrays referencing more than one loop variable, e.g. A(IND(i)+j) is required where i and j are loop control variables

To be accepted more widely in industry, HPF compilers require to conform to standards, where HPF 2.0 represents the current standard. However, relegating the more sophisticated and useful features to the approved extension sections of this standard, does not encourage use of HPF. Note that, INDIRECT and GEN_BLOCK distributions are approved extensions, as are extrinsics HPF_LOCAL and HPF_SERIAL. Should organisations need to review each HPF compiler to see which features they do or do not support? Equally, should they restrict HPF usage to language features that are supported by the majority of manufacturers and compiler vendors, for fear of being locked into one? In the past such questions were answered by a compilers compliance to a standard. This is not the case with HPF today.

Finally, it is evident from the HPF+ project that the HPF language requires further development, with features such as schedule reuse, and support of halos. The HPF+ project has made significant progress with schedule reuse, and hopefully halos will be addressed in the near future.

3 ESI

3.1 Introduction

This section of the report discusses the HPF port of the benchmarks Rel.2 of the PAM-CRASH code within the Esprit Project HPF+. The different versions of the kernel is a part of the PAM-CRASH solver, which can be applied to unstructured meshes. The aim is to have the maximum of features needed for a complete HPF benchmark with greater expressiveness and optimized in terms of performance and memory requirements. Some kernels of different levels of difficulty in terms of HPF requirements were written to be adapted and to follow the evolution of the tools used in this project. The HPF+ benchmarks were written from the previous release of the commercial PAM-CRASH code. So, the major difficulty was to extract the most important part of the solver to have a kernel with a realistic behavior which underline all data-dependencies and load-balancing that contain the bulk of the inter-processor communication, and more particularly the dynamic behavior in the contact routines.

A brief overview of PAM-CRASH's modules using the contact-impact algorithms used in the kernels is given, followed by a description of the different HPF+ kernels used in the second year of the project. This is followed by an outline of the modifications of the kernels to allow different strategies of parallelization due to the pseudo-global search used in the contact algorithm. The report also contains a more detailed comparison between the HPF+ and F90 kernels using the new directives developed during the second year. Moreover, a message-passing version of kernels with the self-impact algorithm was written to compare with different F90/HPF compilers available on the market. In the last part, performance comparison between compilers with a first release of kernels using derived data types is shown to study some more advanced HPF+ features for expressiveness.

3.1.1 Overview of the Code

PAM-CRASH is an explicit time-marching Finite Element program used for the numerical simulation of the highly non-linear, dynamic phenomena arising in short- duration contact-impact problems. It uses a central difference explicit time-marching scheme with unstructured meshes comprised of mechanical elements. The major computational costs of the algorithm comes in the calculation of the forces at the nodal points. These force calculations can be broken down into:

- stress-strain calculations (Benchmark Version 1)
- contact-impact calculations (Benchmark Version 2)

Both parts display different levels of data locality, when considering parallel implementations.

3.2 HPF+ kernel

The first HPF+ version of the benchmark is a subset of the solver. It is sufficient to compute the stress-strain calculations. The stress-strain calculations are performed over the elements. The calculation on each element requires as input the latest coordinates and velocities from only those nodal points defining the element. Once calculated, the force on the element is distributed as individual forces at the nodal points. These calculations produce the largest contribution to the overall computational cost (between 60% and 80%, depending on the particular model) and exhibit a high degree of data locality. Thus, the second release of the HPF+ kernel based on this subset compute the stress-strain and is updated to take in account the new HPF+ directives developed during the second period of this project. Moreover, the second version of the kernel has been totally rewritten to use the expressiveness of F90, in particular, use of array syntax. In addition, this new syntax should assist the compiler analysis.

But, in contrast to the stress-strain calculation, the contact-impact algorithm uses more complicated data structures. The high level of indirection arises as a consequence of the need to loop over subsets of nodes and elements coupled via connectivity and inverse-connectivity lists. Furthermore, these nodal lists are not mutually exclusive. The peculiarity of the self-impact algorithm selected is that slave nodes belong to the master segments and the contact search is made by a bucketing algorithm. The data-dependencies could be resumed in two parts : we need to gather the coordinates of the slaves nodes and master segments defined by the user at run-time. Then, we construct the list of nodes that determine which master node is closest to the slave node. Access to this list is made by arrays of pointers. A loop over the touched buckets is made for each bucket and all slave nodes are checked if they are in the bounding box of the segment concerned. Then, for each contact pair, we apply a function that computes the new forces to add to the existing forces of the touched nodes of the impacted segment and the slave node. This same list of contact pairs is reused for NACC cycles.

The algorithm is divided in three parts :

1. Pseudo-Global Search (not at every cycle)
   (performing a proximity and penetration search : Bucketing algorithm)

2. Close search (remove nodes of the impacted segments)

3. Contact force application
   (penalty method : apply forces :
   -at the impacting node
   -at the 3 or 4 nodal points of the impacted segment.)

The sequential kernel is taken from the PAM-CRASH solver written in F77. However, because of the highly optimized and vectorized constructs of the standard version, the most important loops of the contact search algorithm as well as the penalty method were rewritten in a simplified form so as to comply with available HPF compilers. The sequential kernel uses dynamically defined lists and pointers to bucket constructs and lists of contact pairs. Actually, the HPF compiler used does not handle correctly all features needed to have a correct version of this algorithm. Thus, in collaboration with the compiler partners, a version without the bucket search and fewer features than the first version was written to be compliant with some compiler restrictions. We hope in the near future to handle all main features used in this kernel.

Some separate versions of this kernel of increasing levels of complexity w.r.t HPF language features including the contact routines with different options were produced and some were adapted to available F90 and HPF compilers.

3.2.1 HPF+ features used

Distribute directives

The HPF+ features used in the HPF kernel are the BLOCK directive to distribute nodes and elements. Contrarily to the first version, all mapped arrays are dynamically allocated. We have to distribute the elements in such a way that all nodes that belong to local elements reside as much as possible on the same processor. Otherwise, the local computation of forces will cause excessive inter-processor communication, thus more appropriate for indirect distribution. The ALIGN directive was not used as this feature is not yet implemented in the VFC. Another variant using the GEN_BLOCK distribute directive for elements was written, but not yet evaluated to study problems of different materials which introduce computational imbalance between processors.

We outline that the we need support for irregular shadow zones especially for the contact algorithm, based on a first specification of the HALO directive. A kernel using this directive has started, but because of different requirements and to adopt a more general form combined with other HPF+ directives, the syntax is not yet well defined. Therefore, no more analysis has been done until now.

Independent loops

Most loops benefit by recoding to array syntax and some HPF directives were removed thanks to an improvement of compiler analysis. Nevertheless, there are still some constraints which prevent optimized generated code when loops use array sections.

The independent loops use the REDUCTION clause to gather forces computed by elements to nodes and the ON HOME clause was added to optimize the analysis phase by compilers. The HPF+ kernels also contain the PUREST directive to allow use of local subroutines in two independent loops. The first local subroutine computes internal forces for one element and the second routine apply forces to one contact pair. Actually, we outline that all distributed variables used to compute forces must be replicated in local variables to be used in PUREST subroutines. Nevertheless, one kernel variant using the PURE attribute was written to test with commercial compilers.

```
!HPF$ INDEPENDENT, ON HOME(IY(:,N)), REDUCTION(A), REUSE
      DO N=1,N4NODE
        DO I=1,NDOF
          A(I,IY(2,N)) = A(I,IY(2,N)) + FORCE1(I,N)
          A(I,IY(3,N)) = A(I,IY(3,N)) + FORCE2(I,N)
          A(I,IY(4,N)) = A(I,IY(4,N)) + FORCE3(I,N)
          A(I,IY(5,N)) = A(I,IY(5,N)) + FORCE4(I,N)
        END DO
      END DO
```

In the independent loop, the gather operations and min-reductions to compute internal forces imply unstructured communications in spite of the INDIRECT directive. Thus, the REUSE directive recognized as a key factor was used to avoid re-executing the inspector phase which describes the same communication patterns at each time-step loop. Also, the COND clause of the INDEPENDENT directive is used in the contact routines to take advantage of some algorithm features.

Dummy argument mapping

The first approach was to use the INHERIT directive in all contact routines. Besides,these routines were rewritten to use the assumed-shape array syntax and reshaping was avoiding by clean constructs. Moreover, whole arrays were passed every time with always the same distribution. However, the use of INHERIT for dummy arguments may be the reason that some optimizations cannot be applied, especially, if the distribution of contact arrays is always the same. Thus, a different version of contact routines is evaluated with the VFC compiler, where the distribution of dummy arguements is specified explicitly. Note that sometimes leaving out an interface block may cause incorrect data mappings between routines with some compilers. This is important to outline, because such constructs do not always have the same default distributions or semantics in different compilers. Obviously, the maturity of actual HPF compilers is not the same for this point and we always need to find what is possible and what compilers do on a call to a subroutine.

3.2.2 Porting Effort

As discussed in the previous report (D1.2a), initially storage management was performed by allocating a single one-dimensional work array. Sections of this array were reserved for the storage of individual arrays by the initialization of run-time defined pointers into this array. Then, the main work was spent in:

- Cleaning data structures :
  - remove the one-dimensional working array
  - no complex data structure used (no pointer arrays were used)
  - use of auxiliary arrays to compute nodal forces
  - assumed-size arrays were translated to assumed-shape arrays in all subroutines
  - all mapped arrays were dynamically allocated
  - common statements were removed
  - interfaces were written for all subroutines
  - the most important parts used in the self-impact algorithm were selected
- Rewrite totally the initialization phase, and find some workarounds because of the I/O restrictions with SC4.0 F90 and VFC.

- Complete reorganization of the code, especially for the contact routines with new data structures more adapted to the HPF syntax. All optimized F77 constructs used were removed because they were not HPF compliant.
- Contact routines were rewritten with F90 array syntax and some different constructs were used to store contact pairs.
- Some independent loops were too complicated for available HPF compilers, due to the high level of indirection, and use of non-invariant bounds and conditional statements. Consequently, some features were temporary removed, namely, (c.f. bucket sort, dynamic lists) and more simpler loops were written.

Example of loop not handled in parallel by any evaluated compiler:

```
!HPF$ INDEPENDENT, ON HOME(IRECT(:,II)),REDUCTION(A), NEW(...), REUSE
      DO 100 II=1,NRT
        DO 101 ICONT=1,LSEMO(II)
          XSNX= X(:,NSV(IMNOD(ICONT,II)))
          RDIST    = DIST1(:,ICONT,II)
          XMASI    = XMS(1,IRECT(1:4,II))
          CALL SELF(XNSNX, ,XMASI ,RDIST , ....)
          A(1:3,NSV(IMNOD(ICONT,II)))=A(1:3,NSV(IMNOD(ICONT,II)))+FORCES
          A(1:3,IRECT(1:4,II)) = A(1:3,IRECT(1:4,II)) + FORCEM
        ENDDO
      ENDDO
```

3.2.3 Other HPF Compilers

Before discussing and comparing with other HPF compilers, the parallelization strategy used to develop the HPF+ kernels has been strongly influenced by available features of the VFC during the first year of this project. Thus, the goal was not to rewrite kernels for other compilers, but to adapt them with a minimum of alterations and when it was possible to run the data-sets. The first recommendations on how to modify kernels for the NA and PGI compilers was provided by the VCPC. However, at the time of writing, performance results are only available for the latest release of the PGI compiler on a CRAY T3E machine. It seems that one of the major problems with commercial compilers is the indirect addressing in independent loops. The solution found for the NA compiler and ADAPTOR was to use auxiliary arrays to help the compiler analysis. Besides, the PUREST directive have to be changed in PURE subroutines and thus avoid temporary variables. Moreover, to handle the reduction with indirect addressing in independent loop, the HPF intrinsic SUM_SCATTER function was used. Some minor restrictions with ADAPTOR with this function prevent to use the auxiliary array forces. We have to collapse them in one big array. The latest revision of the PGI compiler was able to handle all HPF directives used in the HPF+ kernels except the REUSE. Some problems with mapped array sections in independent loops were found and the SUM_SCATTER function was used.

3.2.4 Message Passing version

The message-passing kernel is based on the second revision of the HPF+ kernel used for the time-measurements on the QSW CS-2 machine. This implementation was written with the MPI library and used the host/node paradigm based on the parallel version of the PAM-CRASH code. Moreover, some loops and I/O were rewritten from the previous version to compare with the HPF+ kernel using the reuse schedule directives. Besides, a new complete MPI kernel with the contact feature was developed from the standard parallel code with new parallel optimized constructs specifically written for the benchmarks. It uses the same solver routine in the first part. Nevertheless, the contact routines were written to have the best performance with F77 compilers. Thus, a lot of loops used are not identical to the HPF+ kernels.

This second implementation based on one Pseudo-SPMD model and using the MPI library was used to generate trace-files for the MEDEA tool.

The characteristics are:

- Very simple input code (ASCII input file)
- It was construct to read the same input data that HPF+ kernels
- Construction of mailing-list for exchange information between nodes
- Distribution of all information to the nodes

Node code:

- Initialization was performed by receiving all necessary information from host
- Mailing-lists used for the contact routines are locally generated
- Same kernel as HPF+ Rel.1 and sequential implementation (solver part)
- Optimized F77 constructs for the contact routines were written (not HPF compliant)
- Routines using MPI non-blocking calls were written to parallelize the contact algorithm
- One global broadcast to compute the minimal time-step
- Exchange forces and update velocities of the boundaries nodes for neighbor processors.

3.3 Performance

Four specific input files with a more realistic behavior of different size were generated for the different kernels. All input files are based on the BOX-BEAM data-set and were translated from the standard code. The two first input files were used to for the kernels without contact. The performance measurements of the F90/HPF+ kernels with the different data-sets were made principally on the QSW CS-2 at the University of Vienna.

The characteristics of data-sets without and with contacts are shown in tables 10 and 11 respectively.

	Small	Large
Number of nodes	715	32760
Number of elements	640	32400
Number of CYCLE	600	1000 (or 250)

Table 10: Characteristics of data-sets without contacts

	Small	Medium
Number of nodes	715	6560
Number of elements	640	6400
Number of segments	640	6400 (or less)
Number of Slave nodes	715	6560(or less)
Number of CYCLE	600	1440
NB Max of buckets :xyz	6-20-7	13-12-42
Number of contact found	[1-300]	[8-170] (start at 850)
Number of contact pairs	[1-200]	[8-170]

Table 11: Characteristics of data-sets with contacts

The small data-sets were essentially used to validate the correctness of different kernels on different platforms with available compilers and to generate trace-files for MEDEA with the HPF+ kernels.

3.3.1 F90 kernels with the contact algorithm

The performance of the initial F77 and F90 kernels with contact and the small data-set is shown in table 12.

The F90 sequential code has exactly the same data structure as the HPF+ kernels. It uses the same syntax loops. Nevertheless, F90 array syntax is used for all loops. The difference between the F77 kernel and the F90 kernels are due to the constraints about the HPF actual syntax. The F90/HPF+ kernels do not use optimized constructs with pointers and dynamic lists. In table 13, only F90 compilers were used to evaluate different versions of kernels with

Compiler	DEC F77	DEC F90	B-HP-F90
Machine	DEC 4100	DEC 4100	HP 9000/889
user time(timex-s)	8,4	12,3	25,3
options used	-O5 -fast	-O5 -fast	-O2

Table 12: Initial F77 and F90 kernels with contact (small data-set)

Compiler	apf90 V4 .0	NA V1.3.57	SC4.0 F90 V1.1	DEC F90
Machine	QSW CS-2	QSW CS-2	QSW CS-2	DEC 4100
compile time (s)	18,84	64,0	51,27	2,65
user time(timex)	1531	1912	I/O core	335
options used	-O5 -fast	-fast	-O3 -fast	-O5 -fast

Table 13: Performance of F90 compilers with medium data-set (contact kernels v0)

NPROCS	without allocatable arrays	with allocatable arrays
1	851.3	797.2
2	474.1	437.9
4	283.8	255.4
8	237.5	193.0
12	189.6	186.7
16	259.7	184.4
32	210.1	core dump

Table 14: Performance of HPF+ kernel (small dataset) on QSW CS-2

the medium data-set. However, because of actual constraints about array syntax with the VFC compiler, a few modifications in parallel and independent loops had to be made to make performance comparisons.

3.3.2 HPF+ kernel with schedule reuse

All HPF+ kernels with the REUSE directive were successfully translated in F90 with the VFC compiler. Besides, a first release of the contact kernel without the bucket sort was generated. But, at the time of writing, no performance results were obtained because of problems with the SUN-F90 V1.2 compiler on the QSW CS-2 machine and no support for conditionals in independent loops with the VFC.

The performance of the HPF+ kernel (small dataset) is shown in table 14. In table 15, the performance of the HPF+ kernel (large dataset, 250 cycles) using allocatable arrays is tabulated against the equivalent MPI kernel (F77).

Moreover, trace-files were successfully generated for the small data-set (using the VFC -mcils option) on up to 30 processors. However, the number of cycles had to be reduced to 10 due the size of the generated trace files (see figure 5).

3.3.3 PGI compiler on CRAY T3E

The performance of the HPF+ kernel using the PGI compiler release R2.3 is shown in table 16. The PGI version of the second release of HPF+ kernel use the same F90 routines as the

K3B.IndepLoops
#of components: 20

	t_exec	t_comm	n_exec	n_comm
k3_4_PCRASH.219	2.615000	0.347000	21.000000	323.000000
k3_4_PCRASH.219_work	0.047000	0.000000	1.000000	0.000000
k3_4_PCRASH.219_inspector	1.033000	0.115000	1.000000	50.000000
k3_4_PCRASH.219_gather	0.219000	0.104000	21.000000	147.000000
k3_4_PCRASH.219_executor	1.096000	0.000000	21.000000	0.000000
k3_4_PCRASH.219_scatter	0.220000	0.128000	21.000000	126.000000
k3_8_PCRASH.219	1.886000	0.639000	21.000000	327.500000
k3_8_PCRASH.219_work	0.047000	0.000000	1.000000	0.000000
k3_8_PCRASH.219_inspector	0.605000	0.106000	1.000000	54.500000
k3_8_PCRASH.219_gather	0.316000	0.226000	21.000000	147.000000
k3_8_PCRASH.219_executor	0.556000	0.000000	21.000000	0.000000
k3_8_PCRASH.219_scatter	0.362000	0.306000	21.000000	126.000000
k3_32_PCRASH.219	4.544000	4.095000	21.000000	336.125000
k3_32_PCRASH.219_work	0.045000	0.000000	1.000000	0.000000
k3_32_PCRASH.219_inspector	0.274000	0.107000	1.000000	63.125000
k3_32_PCRASH.219_gather	2.242000	2.169000	21.000000	147.000000
k3_32_PCRASH.219_executor	0.136000	0.000000	21.000000	0.000000
k3_32_PCRASH.219_scatter	1.846000	1.819000	21.000000	126.000000
k3_40_PCRASH.219	4.560000	4.148000	21.000000	338.600006
k3_40_PCRASH.219_work	0.046000	0.000000	1.000000	0.000000
k3_40_PCRASH.219_inspector	0.298000	0.153000	1.000000	65.599998
k3_40_PCRASH.219_gather	2.700000	2.624000	21.000000	147.000000
k3_40_PCRASH.219_executor	0.115000	0.000000	21.000000	0.000000
k3_40_PCRASH.219_scatter	1.401000	1.371000	21.000000	126.000000

Figure 5: Main Independent loop and communication statistics displayed by Medea for the HPF+ kernel executing from 4 to 40 processors for 10 time-steps with the small data-set

NPROCS	HPF+ kernel	MPI kernel (F77 -O5)
1 (sequential)	1300,0 (SUN F90 V1.2)	
1 (sequential)	402,0 (apf90 V4.0)	
1 (vfc+f90)	Not enough memory	480,85
2	2810,0	260,26
4	1135,7	131,39
8	672,2	118,28
12	439,6	55,88
16	370,3	28,44
20	293,56	25,4
30	221,44	16,5
32	210,33	29,9 (partition p64)

Table 15: Performance of HPF+ kernel (large dataset) on QSW CS-2, compared against equivalent MPI kernel

HPF+ kernel developed in collaboration with the University of Vienna. Therefore, the kernel is not optimized for this compiler. Nevertheless, because of restrictions, different constructs and directives were used in some loops (ex : sum_scatter). The time-step calculation was reduced to 250 cycles because of quota limits on this machine during the time-measurements. Besides, analysis of the generated code, has indicated that all loops are parallelized and no important constraints were found. But, in spite of a very good scalabililty upto 128 processors (but not as well as the MPI kernel), the high over-head generated to compute the schedule of different loops at every cycle dramatically reduce the code performance. It is important to underline the performance of the same kernel directly compiled with the native CRAY compiler : 172.02 s (CF90, -03). Then, the PGI code is 12,9 times slower serially. In comparison, if we extrapolate the VFC time to 1 processor, we find it is only 4,3 times slower.

The main source of the inefficiencies reported by the PGI could be the same source that the VFC compiler. In other terms, all depends of the runtime parallelization strategy like inspector/executor with all its associated huge overheads (in memory as wells as in preprocessing time) used for such codes. Besides, the use of large indirection arrays are not well recommended because of the cache, especially if they are used in regular loops with regular array assignment.

However, timing data shows that the GATHER/SCATTER operations with both compilers are a major drawback in performance. Therefore, more attention will be needed for the next kernels especially with the high level of indirection arising in the contact routines.

3.3.4 Performance of MPI kernels

The time-measurements of the message-passing version were performed with the MPI implementation on the QSW CS-2 and CRAY T3E (ELV5 300 Mhz) to compare with the HPF+ kernel using the schedule reuse. The large BOX data-set was used (32760 nodes and 32400

NPROCS	Elapse time(s)	Speed-Up
1	2227,18	1,00
2	1315,66	1,69
4	666,30	3,34
8	340,39	6,54
16	177,25	12,57
20	144,46	15,42
32	94,49	23,57
40	79,26	28,10
50	66,94	33,27
60	57,94	38,44
80	51,09	43,59
100	46,38	48,02
128	37,26	59,77
150	43,34	51,39
180	39,94	55,76
191	40,22	55,37
200	40,16	55,46
230	39,82	55,93
250	38,83	57,36
270	64,45	34,56

Table 16: Performance of HPF+ kernel on T3E using PGI compiler (large data-set, 250 cycles)

NPROCS	Elapse time(s)	partition	Speed-Up
1	1831,99	p16b	1,00
2	916,45	p16b	2,00
4	451,50	p16b	4,06
6	333,40	p16b	5,49
8	244,20	p16b	7,50
12	159,60	p16b	11,48
16	120,70	p32a	15,18
	109,2	p32b	16,78
20	97,80	p32a	18,73
	88,3	p32b	20,75
31	85,20	p32a	21,50
	62,8	p32b	29,17
40	51,00	p64	35,92
50	43,80	p64	41,83

Table 17: Performance of MPI kernel (large dataset, 1000 cycles) on QSW CS-2 using SCO 3.01 F77 (-O5 -fast)

elements, 1000 cycles). Because of a different behavior with some partitions, the partition name is indicated.

3.3.5 MPI kernel on QSW CS-2 machine

The performance of the MPI kernel on the QSW CS-2 is tabulated in table 17, and shown graphically in figure 6.

3.3.6 MPI kernel on CRAY T3E machine

The performance of the MPI kernel on the CRAY T3E is shown in table 18. The large BOX data-set was used, but contrary to the HPF kernel tested on this machine, the number of cycles was 1000 and no optimization options were used. The kernel was written in F77 and compiled with the native F90 CRAY compiler.

3.3.7 Speed-Up comparison on T3E

A comparison of HPF(F90-PGI V2.3) and MPI (F77) performance on the CRAY T3E is shown in figure 7.

3.3.8 HPF+ kernels with contact-impact routines (sequential performance)

The benchmarks used for time-measurements are the first release of the contact algorithms. Main features are included to have the most realistic behavior underlining the dynamic data-

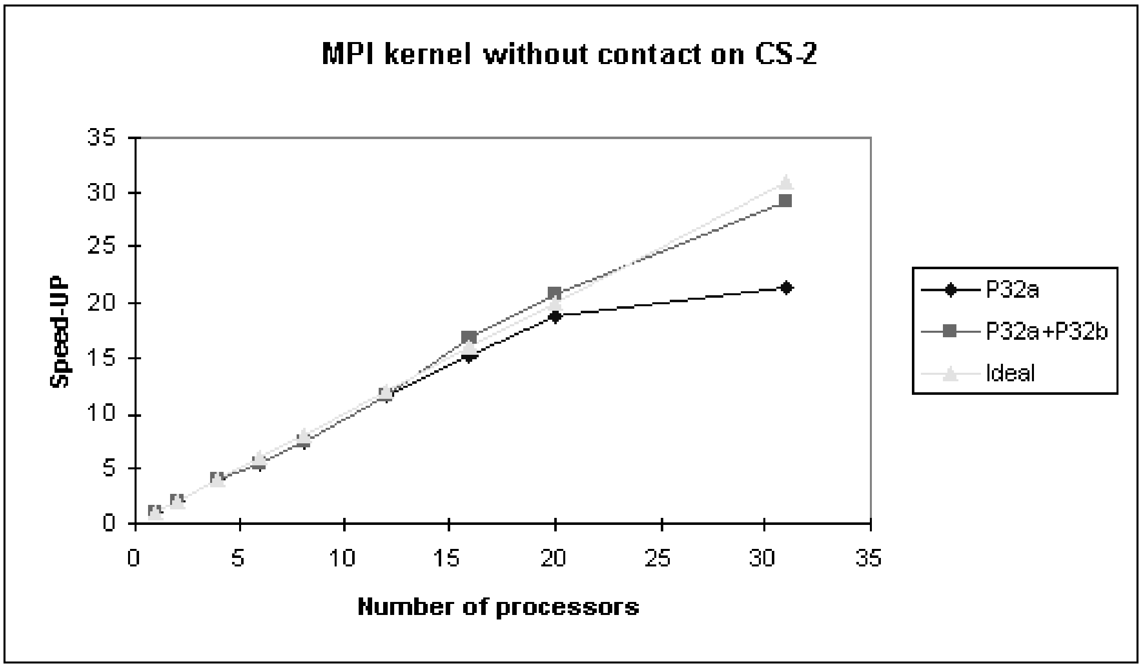


Figure 6: MPI kernel performance on QSW CS-2 (large dataset, 1000 cycles)

NPROCS	Elapse time(s)	Speed-Up
1	573,36	1,00
2	294,16	1,95
4	147,4	3,89
8	77,4	5,49
12	53,2	10,82
16	42,9	13,37
20	35,0	16,38
30	23,9	24,00
40	21,9	26,18
50	16,7	34,33
60	14,8	38,74
64	12,5	45,87
90	11,3	50,74
128	7,0	81,91

Table 18: Performance of MPI kernel (large dataset, 1000 cycles) on CRAY T3E using F90

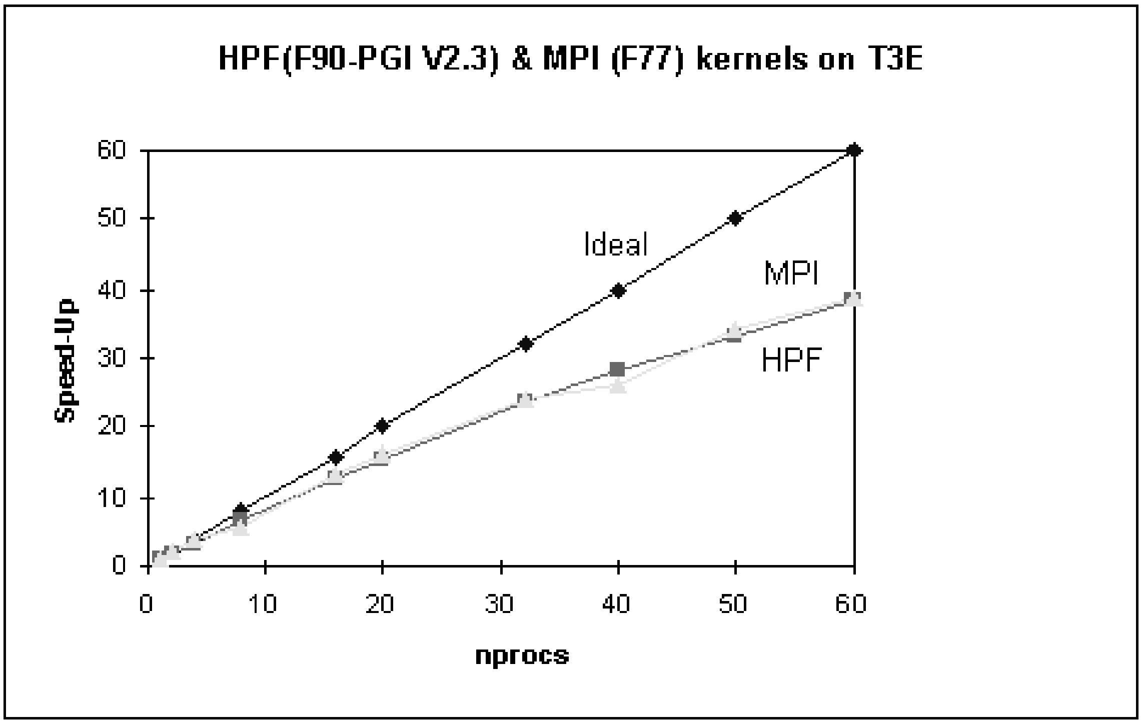


Figure 7: HPF and MPI kernel performance on CRAY T3E (large dataset, 1000 cycles)

dependencies in different loops and the load-balancing problems arising during computation. The data-set used is of medium size. Thus, the scalability will be limited (BOX example with : 6560 nodes and 6400 elements, 1440 cycles) to a few processors. Besides, all surfaces are declared in contact, this is the worse case. The characteristics of these kernels are : assumed shape arrays, static arrays and array syntax. All loops were rewritten to be compliant with the HPF syntax and some new arrays have to be added. Nevertheless, because of constraints of the actual HPF+ compiler and problems with the SUN-F90 compiler (internal compiler error), no time-measurements were made with multiple processors.

Therefore, only single processor performance is shown in table 19. In this table, * denotes

compiler	apf90 V4.0	NA V1.3.56	SC4 F90	DEC F90	DEC F90*	DEC F77*
Machine	QSW CS-2	QSW CS-2	QSW CS-2	DEC 4100	DEC 4100	DEC 4100
compile time	34,67	2,31	**	8,87	6,3	6,3
time(timex)	578	1916	**	141,7	102,1	100,0
options used	-O5 -fast	-fast	-O3	-O5 -fast	-O5 -fast	-O5 -fast
Nb of Ct pair	168			170	170	170

Table 19: Single processor performance of HPF+ contact-impact kernel

PROCS	Elapse time(s)	Speed-Up	NBPTT : processor Id (number of contact pair)
1	597,4	1,00	168
2	214,4	2,77	P1(168)
4	171,6	3,48	P4(168)
8	97,2	6,12	P8(168)
12	69,0	8,61	P5(18), P6(4), P11(8),P12(160) = 190
16	60,34	9,85	P7(16),P8(15),P9(2),P15(27) ,P16(141) = 201
20	58,00	10,25	P19(44),P20(124) = 168
31	*	*	P15(12),P16(20),P17(32),P18(31),P19(31)=126

Table 20: Multi-processor performance of MPI contact-impact kernel

the respective kernel is written with F77 optimized constructs from the production code and also that loops used in the self-impact routines are not HPF compliant. In the same table ** denotes an excessive compilation time.

3.3.9 MPI kernels with contact-impact routines

The MPI kernel have the same loops in self-impact routine that the F77 kernel. Therefore, the native optimized constructs were used. Performance results were measured with the gettimeofday() function on different partitions, and are shown in table 20. Also, a large variation in performance was observed when running on different processor partitions (up to 30-50 %).

3.3.10 F90 kernels using the derived data type syntax

In table 21, the performance of the kernel using derived data types constructs is shown for the large data-set (1000 cycles). The second evaluation is based on the HPF+ kernel written with array syntax. But the subroutine that compute internal forces is always based on the HPF kernel, thus this part of code is in pure Fortran 77. Actually, it seems that the use of derived data types is too compiler dependent and more optimized constructs have to be coded. The same data-set was used for all F90 benchmarks. We have to notice that the bad performance on the QSW CS-2 could be due to the lack of memory as a result of the size of the data-set being used.

3.4 Discussion

In this report, we have described the evolution of the PAM-CRASH kernel to be performed within the HPF+ project. After localising some problematic parts in the self- impact routines, the contact kernels were restructured for better analysis by HPF compilers. Nevertheless, we had to remove some very optimized constructions to stay in compliance with the data-parallel HPF constructs. In the second step, we have adapted some different subroutines to use HPF directives. The second version of the kernels now written in F90 has been tested with different

Compiler	apf90 V4 .0	NA V1.3.56	DEC F90
Machine	QSW CS-2	QSW CS-2	DEC 4100
compile time	147	53,62	6,5
user time(timex)	5624	****	560
options used	-O5 -fast	-fast	-O5 -fast

Table 21: Single processor performance of F90 (derived data type) kernel on different systems

F90/HPF compilers and more particularly with the VFC thanks to the close collaboration with the compiler developers.

Moreover, a complete hand-coded version with a parallelized version of the self-impact algorithm was written in MPI and evaluated on different parallel platforms (SGI, DEC, T3E, CS-2). Some trace files were generated and preliminary experiments were made with MEDEA on the kernels. Besides, more extensive experiments of the F90 version of the benchmark with derived data types were also performed on different platforms.

Some performance measurements were made to evaluate the impact of the new HPF+ directives developed during this project. First results show that the overheads generated by the inspector phase are amortized over a complete simulation. Nevertheless, the contact-impact routines in different kernels quickly need new improvements within independent loops and compiler constraints have to be relaxed to handle the more efficiently this new challenge. Besides, the dynamic data-dependencies imply to define user and run-time defined general halo regions and integrated with others optimization which have been addressed for the future HPF+ developments.

4 NEC - CCRLE

4.1 Introduction

This report provides an intermediate "status report" on the suitability of the HPF+ language constructs for the project applications, judged using the intermediate implementation of this language within the VFC compiler (release 0.91). The basis for the evaluation in this section is the K2 Kernel set as described in deliverable D1.1a (which itself references kernel descriptions in D1.2a). The HPF+ features required by the K2-set, described below, correspond to the restrictions included in release 0.91. As discussed in D1.1a, the K2-set are generic crash kernels which represent the explicit time-marching scheme built around the element-node interactions for 4-node elements.

4.2 HPF+ kernel

Several variants of kernels were produced in order to meet compilation and performance restrictions of the various compilers tested. All variants took the K2HPF+_a kernel described in D1.1a with the further modification that the routine UPDATE was transformed to use purely F90 array syntax (this is also true for the K2F90 kernel used for sequential comparisons in the results section). This can be seen in the extracts from the HPF+ kernel included in the Appendix. Further modifications will be described in the sub-sections below.

4.2.1 HPF+ features used

The final version of the HPF+ kernel used for this evaluation is referred to below as K2HPF+_a_vfc1. The crucial routines are given in the Appendix. In comparison with the 'target' versions discussed in the deliverable D1.1a, several modifications were necessary in order to conform with restrictions within release 0.91 of the VFC compiler.

- Since alignment is not supported and INHERIT was reported to be problematic in some cases, all subroutines use explicit distribution statements for distributed arrays.
- Array sections of distributed arrays are not yet supported within INDEPENDENT loops, thus F90 statements within the INDEPENDENT loop in routine KFORCE were expanded to F77 do loops.
- The use of purely array syntax in routine UPDATE avoided both restrictions in *use-and-definition* of distributed arrays in INDEPENDENT loops and avoided the non-optimised handling of INDEPENDENT loops with regular array accesses.

The most crucial HPF+ features employed are all within routine KFORCE and linked to the INDEPENDENT loops: the schedule re-use directive, REUSE; and the REDUCTION directive.

4.2.2 Porting Effort

Since no **major** re-coding took place for the production of the various HPF kernels included in the evaluation, the question of porting effort in comparison with MPI has not changed significantly from the previous evaluation. The upgrade from the K1-set to the K2-set revolves around the inclusion of the velocity field and the UPDATE routine. The effort to perform this within the MPI kernel is significantly higher: since the field needs to be mailed to node processes in local form, not just simply included in relevant subroutines. However, it must be stated at this point, that there is currently **no such thing as portable HPF code !** The various variants included below, for both research and commercial compilers, illustrates this - at least for kernels like those included in this project. This does greatly increase the porting effort.

4.2.3 Other HPF compilers

HPF versions were produced for three other HPF compilers, two commercial and one research.

In line with the advice of NA Software, included in the project working paper "A study of the HPF+ V1.1b Kernels", modifications leading to the kernel K2HPF_a_nas1 were:

- Replication of the indirection array IX.
- Specification of the routine MFORCE as EXTRINSIC(HPF_SERIAL).
- Avoidance of the need for NEW variables by the use of forward substituted array sections in the routine argument list (in fact, this had to be modified to pass 4 arguments instead of the construct

  ```
  x(1:3,ix(1:4,iel))
  ```

  due to a restriction of the release of the compiler).

However, there appears to be no way currently to avoid the need for a REDUCTION directive, so that (since the compiler includes only a sequential implementation of the HPF LIBRARY routine SUM_SCATTER) the second loop in routine KFORCE remains problematic; i.e though compiled, it is likely to be sequentialised. Indeed, execution with more than one processor was not possible on the QSW CS-2 (multiple processor execution leading to a segmentation error).

Essentially the same kernel as K2HPF_a_nas1 was used for the PGI compiler. Differences were that: the indirection array IX was maintained as a distributed array; the second loop in KFORCE was re-written as six SUM_SCATTER function calls; routine MFORCE was defined as PURE. The avoidance of NEW variables was achieved as described above - here, the replacement of the complex single argument with four simpler array sections was necessary to avoid a sequentialisation of the loop, rather than a compiler restriction.

In collaboration with Thomas Brandes at GMD, investigations with development versions of the HPF compiler system ADAPTOR were performed. The purpose of this was to demonstrate that ideas similar to those being discussed in the HPF+ project may lead to high performance. As will be seen in the brief description of the kernel variants below, the use of the constructs is not yet at an automatic stage, nor suitable for a general problem. However, the performance results achieved, speak for themselves. The first variant, K2HPF_a_ADP1, uses a technique similar to schedule REUSE by specifying that all accesses via the array IX should be "traced". In addition, and at the cost of additional storage, it pre-stores all necessary co-ordinate values. The second kernel, K2HPF_a_ADP2, uses a large overlap, or SHADOW, region, of fixed size, into which non-local variables are copied andor read. The management of these overlap regions is achieved via direct calls to the run-time system, dalib, of ADAPTOR. As stated earlier, this is not a suggested approach, rather it supports the idea for the use of halo regions as is being discussed in the HPF+ project. The crucial features of these two kernels are illustrated in the extracts included in the appendix.

4.2.4 Message Passing kernel

As discussed in the previous deliverables, D1.1a, D1.2a, the K2MPI_a kernel was produced separately and performs exactly the same computation as the various HPF kernels. Although it exploits non-blocking communication, this does not necessarily make it an "optimised" code - it exploits the communication pattern generated by the construction of neighbourhood mailing lists, which is in any case a necessary initialisation. This initialisation process has not at all been optimised and thus produces, at higher processor numbers, a noticeable contribution to the total execution time. For this reason, two sets of performance figures are given for the MPI results. Since representative crash simulations would perform an order of magnitude more iterations with significantly larger computational models, we believe the performance figures without the initialisation times represent more closely the expected scaling of an MPI implementation.

4.3 Performance

As in the previous evaluation, a simple box model with mesh of size $4 \times 40 \times 160$ has been used. Given the focus on the performance behaviour of REUSE and REDUCTION, an investigation of the behaviour when employing dynamically defined, indirect or general block distributions has been postponed to a later date.

In order to specify performance goals for the HPF compilers (and, of course, underlying language constructs) several sequential measurements were made: with the F77 kernel, K2SEQ; the F90 kernel, K2F90 (which is equivalent to the base HPF codes with the exception that the splitting of the computational loop in routine KFORCE can be avoided); the HPF and HPF+ kernels. All HPF kernels executed on the QSW CS-2 were checked with both available F90 compilers - major variations between different HPF variants were not evident. Figure 8 summarises all compiler options used on the two machines used for the evaluation: the QSW CS-2 at VCPC and the NEC Cenju-3 located at GMD.

Where feasible, several executions of a given run were performed: timings on the QSW are highly unreliable (being both time and partition dependent) in addition to problems exhibited by the system_clock function which sometimes delivered negative timings. Timings given later are average values (with the exclusion of clearly outlying results). The HPF results reported

Platform	Compiler	Compile flags	Version/Release
QSW CS2	F77	-O3	4.2
	F90	-freeform -O3	1.2
	APF90	-O3	4.0
	VFC	-f	0.91
	HPF	-free -O	2.01
	PGHPF	-Mmpi -Mfreeform -O3	2.1
	MPIF77	-O3	mpich
NEC Cenju-3	GMDHPF	-Kmips2 -O	development
	MPIF77	-Kmips2 -O	mpich

Figure 8: Compiler Options

on the CS-2 were from ETIME; on the Cenju-3 with system_clock. MPI results are with MPI_WTIME.

The performance measurements for the variety of kernels, machines and compilers is given in Figure 9. The notation used for the VFC results is as follows: R denotes the use of schedule re-use (the logical variable set to .true.), N denotes execution without schedule re-use (the logical variable set to .false.).

Graphical representations of the HPF versus MPI scaling behaviour are given in Figures 10 and 11, where logarithmic scales have been used on both axes.

4.4 Discussion

The single processor performance of all the HPF compilers on the CS-2 is highly unsatisfactory. The inclusion of the numerous sequential performance results in Figure 9 was to highlight the fact that although a portion could be due to the underlying SUN F90 compiler - where appropriate - there is nevertheless a large (to huge) overhead generated. (Additional tests with 5 variants of the UPDATE routine, from purely F77 as in K2SEQ through to purely F90 array constructs as in the Appendix, showed that the SUN F90 compiler produced a slow-down of a factor of two on just the pure F90 array syntax version. The Apogee compiler on the CS-2 was able to either maintain or improve on performance with array constructs, as indeed did the SGI compiler on an in-house Indy). A further performance loss with VFC can

			Number of Processors							
Platform	Kernel	Compiler	1	2	4	8	16	32	64	128
QSW CS2										
	K2SEQ	F77	497							
	K2SEQ	F90	712							
	K2SEQ	APF90	414							
	K2F90	F90	1005							
	K2F90	APF90	520							
	K2HF+_a_vfc1	F90	1107							
	K2HF+_a_vfc1	APF90	673							
	K2HF+_a_vfc1 (R)	VFC	4351	2094	1050	563	298	187	174	
	K2HF+_a_vfc1 (N)	VFC				11397	8537	6919		
	K2HPF_a_nas1	HPF	16000	X						
	K2HPF_a_pgi1 *	PGHPF	32450	47650	71100					
	K2MPI_a (total)	MPIF77	497	246	126	68	38	24	22	
	K2MPI_a (no init)	MPIF77	497	243	122	64	34	19	14	
NEC Cenju-3										
	K2HPF_a_ADP1	GMDHPF	1264	588	267	137	65	39	25	22
	K2HPF_a_ADP2	GMDHPF	584	287	148	77	42	25	17	15
	K2MPI_a (total)	MPIF77	340	162	82	42	23	13	8,6	7,7
	K2MPI_a (no init)	MPIF77	340	162	82	42	22	12	6,9	4,5

X - runtime error
* - Extrapolated from a run with 20 iterations

Figure 9: Elapsed times (secs) for the evaluated kernels

be attributed to the VFC-internal use of POINTERS, which could be alleviated by a switch to ALLOCATABLE and AUTOMATIC arrays. The expected reduction in performance penalty w.r.t. the statically allocated "original" code would be (this has been identified by tests performed by the VFC developers) a factor of 1.1 instead of the current 1.6. Put together, the overhead resulting from the underlying SUN F90 compiler and the use of Pointers amounts to a penalty factor of approx. 3.2.

Further investigations will look into the non-scaling and erroneous execution behaviour with the NAS and PGI compilers (the latter produced warnings of excessive communication being generated, later versions of both compilers may offer alternative solutions). **However,** the same codes could be compiled and run with both F90 compilers (SUN & Apogee) delivering execution times similar to those in Figure 9 for the K2HPF+_a_vfc1 kernel.

Setting aside the current single processor performance of VFC, the scalability exhibited (as illustrated in Figure 9) is nevertheless good, showing that the REUSE facility is effective. This is underlined by the elapsed times exhibited when the schedule reuse is switched off (see Fig. 9). A detailed investigation of the inspector-executor costs for the INDEPENDENT loops in routine KFORCE is provided by MEDEA. In Figure 12 these costs are presented for the K2HPF+_a_vfc1 kernel with (bottom table) and without (top table) the REUSE directive for execution on 16 processors (note that the run without schedule reuse was shortened to 100 iterations). While gather, executor and scatter costs remain, these are an order of magnitude

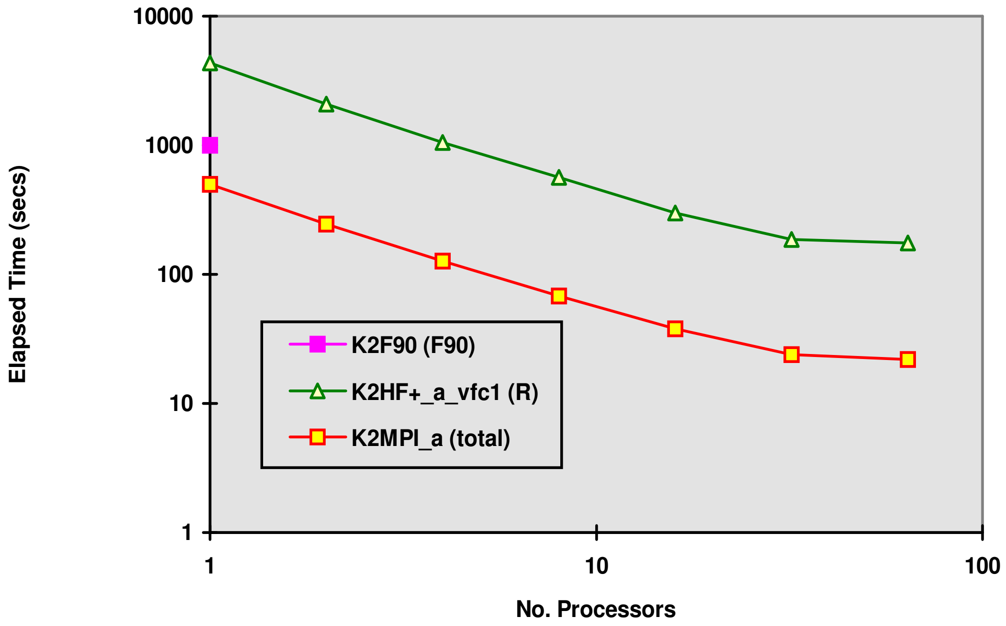


Figure 10: HPF+ - MPI comparison on the QSW CS-2

less significant than the inspector and work distributor phases which may be avoided in the REUSE case.

Further information provided by Medea related to the communication taking place. Figure 13 shows that this only begins to become significant for processor numbers of 32 (and obviously above). Using the profile options of Medea, it was clearly to be seen that all-to-all communications were persistent and dominant. Since the structure under consideration actually requires each process to communicate with only two neighbours, this shows that further optimisation within VFC could focus on the use of communication constructs based on the information provided by the generated schedules, rather than the use of the MPI collective operations.

Although resorting to direct calls to the runtime library, the second ADAPTOR kernel, K2HPF_a_ADP2, demonstrates just how effective the use of a generalised overlap, or "halo", region could be. This suggests that the "halo" concept should be followed-up, and a combination with schedule re-use could perhaps deliver the sort of performance demonstrated by the MPI kernels.

4.5 Acknowledgements

The authors would like to thank their colleagues in the HPF+ project for support and many fruitful discussions. In addition, it is their pleasure to acknowledge the support of Dr. Thomas

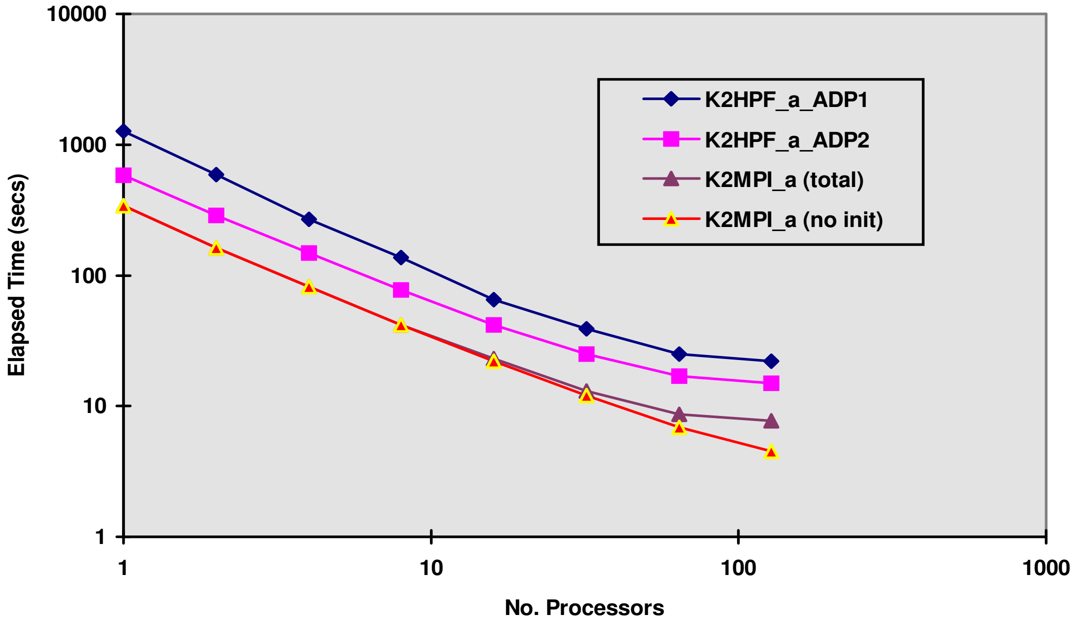


Figure 11: HPF-Adaptor - MPI comparison on the NEC Cenju-3

Brandes, GMD for the development of the Adaptor-based kernels.

nec-noreuse_16p.IndepLoops
# of components: 2

	t_exec	t_comm	n_exec	n_comm
nec-noreuse_16p_KFORCE.30	405.730011	45.782001	100.000000	1550.000000
nec-noreuse_16p_KFORCE.30_work	187.274994	0.000000	100.000000	0.000000
nec-noreuse_16p_KFORCE.30_inspector	203.839005	44.077999	100.000000	1350.000000
nec-noreuse_16p_KFORCE.30_gather	1.028000	0.504000	100.000000	100.000000
nec-noreuse_16p_KFORCE.30_executor	10.735000	0.000000	100.000000	0.000000
nec-noreuse_16p_KFORCE.30_scatter	2.853000	1.200000	100.000000	100.000000
nec-noreuse_16p_KFORCE.53	423.813995	44.000999	100.000000	1550.000000
nec-noreuse_16p_KFORCE.53_work	187.223007	0.000000	100.000000	0.000000
nec-noreuse_16p_KFORCE.53_inspector	225.473007	42.615002	100.000000	1350.000000
nec-noreuse_16p_KFORCE.53_gather	2.971000	0.507000	100.000000	100.000000
nec-noreuse_16p_KFORCE.53_executor	6.756000	0.000000	100.000000	0.000000
nec-noreuse_16p_KFORCE.53_scatter	1.391000	0.879000	100.000000	100.000000

nec-reuse_16p.IndepLoops
# of components: 2

	t_exec	t_comm	n_exec	n_comm
nec-reuse_16p_KFORCE.30	157.311996	24.732000	1000.000000	2013.500000
nec-reuse_16p_KFORCE.30_work	1.954000	0.000000	1.000000	0.000000
nec-reuse_16p_KFORCE.30_inspector	6.320000	4.588000	1.000000	13.500000
nec-reuse_16p_KFORCE.30_gather	9.150000	4.361000	1000.000000	1000.000000
nec-reuse_16p_KFORCE.30_executor	108.074997	0.000000	1000.000000	0.000000
nec-reuse_16p_KFORCE.30_scatter	31.813000	15.782000	1000.000000	1000.000000
nec-reuse_16p_KFORCE.53	113.874001	15.579000	1000.000000	2013.500000
nec-reuse_16p_KFORCE.53_work	1.949000	0.000000	1.000000	0.000000
nec-reuse_16p_KFORCE.53_inspector	3.529000	1.657000	1.000000	13.500000
nec-reuse_16p_KFORCE.53_gather	27.884001	5.378000	1000.000000	1000.000000
nec-reuse_16p_KFORCE.53_executor	66.933998	0.000000	1000.000000	0.000000
nec-reuse_16p_KFORCE.53_scatter	13.578000	8.543000	1000.000000	1000.000000

Figure 12: Medea Parameter displays - 16 Processors, with & without schedule reuse

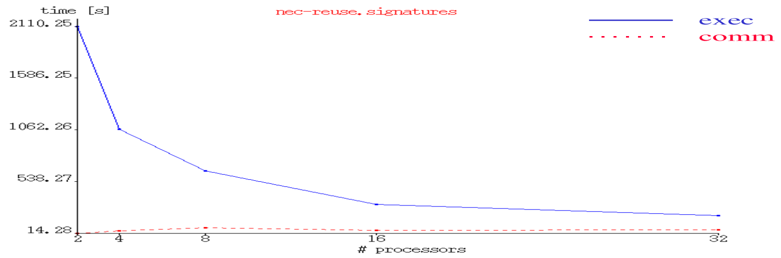


Figure 13: Medea Signature for K2HPF+_a_vfc1, 2 -32 procs

5 University of Pavia

5.1 Performance Evaluation Approach

The objective of this section is to define a framework for the analysis of the performance of benchmark programs vers. 2, as provided by Medea. As we will describe, such an analysis can be focused on different levels of details.

Benchmark programs are typically characterized by performance figures, such as, execution times and achieved speedup. These figures show how the program scales, as a function of the number of allocated processors. More details about the behavior of the program are obtained by looking at the communication activities. Communication times and patterns are derived on a per protocol and per processor basis. Loss of synchronization among processors and differences in their communication behavior are discovered by these analyses.

Although all these results provide a detailed view of the performance of the benchmark programs, they fail to analyze specific components of the code (e.g., loops, subroutines). Hence, a sort of profiling, similar to what is done for sequential codes, has been adopted in order to capture the "cost" of the various components of the code. Medea has been enhanced by introducing such new features.

Each of the components currently analyzed, that is, loops, `INDEPENDENT` loops, subroutines, is characterized by the following parameters:

- execution time
- communication time
- number of occurrences
- number of communications.

All these parameters represent the averages taken over all the processors.

Another important performance issue adopted by the profiling of the code, is the analysis of the overhead introduced by the compiler, as a consequence of the parallelization/optimization strategies adopted. In the case of `INDEPENDENT` loops, the overhead is due to the inspector/executor strategy. The Instrumentation System of VFC recognizes the five phases of the inspector/executor strategy, that is:

- work distributor
- inspector
- gather
- executor
- scatter.

The profiling provided by Medea breaks down, for each of these phases, the parameters characterizing each `INDEPENDENT` loop, such that the "weight" of the compiler overhead is easily recognized. Moreover, the performance benefits attained in using the schedule `REUSE` directive can be studied.

A further enhancement of Medea consists of the link to the HPF+ source code. This enhancement was made possible since the Instrumentation System of VFC identifies each component and its corresponding phases by the name of the unit they belong to and by their line numbers.

6 Summary

The second release of kernels from AVL, ECMWF, ESI and NEC/CCRLE have been evaluated in this report.

Application partners report that the new Fortran 90 based VFC (release 2) compiler provides a significant improvement in functionality and useability compared with the previous VFCS (release 1) compiler. In addition, the implementation of communication schedule reuse results in a significant improvement in performance for HPF+ programs where the data distributions used are static or changing infrequently.

The performance of independent loops in the kernels have been analysed, and the combination of the instrumention facilities of VFC and visualisation facilities of MEDEA have been used to give greater insight into the overhead of such loops. It is observed that independent loops have a high overhead, even with the benefit of schedule reuse. This is mainly due to the use of indirection arrays for the local array indices of distributed array sections and also the use of the POINTER attribute for these indirection arrays. In the latter case, the use of allocatable arrays in the next release of VFC is expected to reduce this overhead.

Fortran 90 array syntax notation has been used in some kernels to replace do loops containing accesses to distributed arrays. It is recognised that this syntax should be used where possible, as the generated code for such statements do not require the inspector or work distributor phases of independent loops as implemented by VFC. Furthermore, as the generated code does not involve the overhead of indirection arrays it can be very efficient. However, array syntax notation cannot be used to replace more complex do loops, for example, containing subroutine calls or accesses to non-regular data structures. For these, independent loops are required. It should be noted that VFC will be enhanced in the future to also recognise more regular do loops, and where possible bypass the generation of calls to an inspector and work distributor.

There is agreement among application partners that their HPF kernels are easier to write than the corresponding message-passing kernels. However, it is not evident today that HPF programs are more portable. This is mainly due to the present state of HPF compilers and the different subsets of the HPF 2.0 standard that are implemented. In particular, the portability of HPF programs is reduced by the use of HPF extensions which are not directives, such as PURE or EXTRINSIC. Such extensions are not recognised by Fortran 90 compilers, and need to be removed before compilation for serial execution. It is expected that HPF program portability will improve with time, as compiler suppliers provide more complete subsets of the HPF language.

Finally, some future requirements for HPF+ have been identified in this report. These focus on relaxing some of the present constraints of independent loops and the provision of a halo facility for irregular data distributions.

7 Appendix A - Kernel extracts

7.1 AVL HPF+ kernel

file: firebench.hpf

```
C=======================================================================
      PROGRAM FIREBENCH
C=======================================================================
C
C     ORTHOMIN - DIAGONAL SCALING , HPF Version
C
C=======================================================================
C-----
C-----COMMON
      INCLUDE 'comm.inc'
C-----
C-----------------------------------------------------------------------
C-----HPF DEFINITIONS
C-----------------------------------------------------------------------
C-----
!HPF$ PROCESSORS :: NODES(NUMBER_OF_PROCESSORS())
C-----
!HPF$ DYNAMIC :: DIREC1, CGUP
C-----
!HPF$ DISTRIBUTE(BLOCK) :: DIREC2
!HPF$ DISTRIBUTE(BLOCK) :: BP, BE, BW, BN, BS, BH, BL, SU
!HPF$ DISTRIBUTE(BLOCK) :: VAR, RESVEC
!HPF$ DISTRIBUTE(BLOCK) :: DXOR1, DXOR2
!HPF$ DISTRIBUTE(BLOCK) :: ADXOR1, ADXOR2
!HPF$ DISTRIBUTE(*,BLOCK) :: LCC
!HPF$ DISTRIBUTE(*,BLOCK) :: DIREC1V
C-----
      CM0=0.0
C-----
C-----------------------------------------------------------------------
C-----READ MAP ARRAY FROM FILE
C-----------------------------------------------------------------------
C-----
      OPEN(20,STATUS='OLD',FILE='map',FORM='FORMATTED')
      READ(20,*) MAP
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP)) :: DIREC1, CGUP
C-----
C-----------------------------------------------------------------------
C-----READ INPUT DATA FROM IFILE
C-----------------------------------------------------------------------
C-----
      OPEN(20,FILE='../data/ifile.udat',STATUS='OLD',FORM='UNFORMATTED')
C-----
      READ(20) NINTCI,NINTCF,NEXTCI,NEXTCF
C
      READ(20) LCC
C
      READ(20) BS
```

```
      READ(20) BE
      READ(20) BN
      READ(20) BW
      READ(20) BL
      READ(20) BH
      READ(20) BP
      READ(20) SU
C-----
      CLOSE(20)
      WRITE(6,*) 'read finished'
C-----
C-----------------------------------------------------------------------
C-----INITIALISE VARIABLES
C-----------------------------------------------------------------------
C-----
      CM0=0.0
      CM1=1.0
C-----
C----- CALCULATE REFERENCE RESIDUAL VALUE
C-----
      NOMAX=3
      RESREF=CM0
      RESREF=CM0
C-----
!HPF$ INDEPENDENT, REDUCTION(RESREF), ON HOME(RESVEC(NC)), REUSE
      DO 10 NC=NINTCI,NINTCF
        RESVEC(NC)=SU(NC)
        RESREF=RESREF+RESVEC(NC)*RESVEC(NC)
   10 CONTINUE
      RESREF=SQRT(RESREF)
C-----
      IF(RESREF.LT.1.0D-15) THEN
        WRITE(6,'('' I/O - Error: residual sum less than 1.e-15 -''
     X        /'' value: '',G12.6)') RESREF
        STOP
      ENDIF
C-----
      DO 12 NC=1,10
        OC(NC)=0.0
        CNORM(NC)=1.0
   12 CONTINUE
C-----
!HPF$ INDEPENDENT, ON HOME(VAR(NC)), REUSE
      DO 14 NC=NINTCI,NINTCF
        VAR(NC)=0.0
   14 CONTINUE
C-----
!HPF$ INDEPENDENT, ON HOME(CGUP(NC)), REUSE
      DO 16 NC=NINTCI,NINTCF
        CGUP(NC)=1.0/BP(NC)
   16 CONTINUE
C-----
!HPF$ INDEPENDENT, ON HOME(CGUP(NC)), REUSE
      DO 18 NC=NEXTCI,NEXTCF
```

```
        CGUP(NC)=CM0
        DIREC1(NC)=CM0
   18 CONTINUE
C-----
C-------------------------------------------------------
C
C     the following program section should be timed
C
C-------------------------------------------------------
C-----
!HPF$ INDEPENDENT, ON HOME(DIREC1(NC)), REUSE
      DO 20 NC=NINTCI,NINTCF
        DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
   20 CONTINUE
      WRITE(6,*) 'init finished'
C-----
      ITER=1
      NOR=1
      NOR1=NOR-1
  100 CONTINUE
C-----
C-----------------------------------------------------------------------
C-----INDIRECT ADRESSING
C-----------------------------------------------------------------------
C-----
!HPF$ INDEPENDENT, ON HOME(DIREC2(NC)), REUSE
      DO 70 NC=NINTCI,NINTCF
        DIREC2(NC)=BP(NC)*DIREC1(NC)
     X            -BS(NC)*DIREC1(LCC(1,NC))
     X            -BW(NC)*DIREC1(LCC(4,NC))
     X            -BL(NC)*DIREC1(LCC(5,NC))
     X            -BN(NC)*DIREC1(LCC(3,NC))
     X            -BE(NC)*DIREC1(LCC(2,NC))
     X            -BH(NC)*DIREC1(LCC(6,NC))
   70 CONTINUE
C-----
      IF(NOR1.EQ.0) THEN
        OMEGA=CM0
        RESNEW=CM0
        CNORM(NOR)=CM0
!HPF$ INDEPENDENT, REDUCTION(CNORM(NOR), OMEGA, RESNEW),
!HPF$* ON HOME(DIREC2(NC)), REUSE
        DO 419 NC=NINTCI,NINTCF
          CNORM(NOR)=CNORM(NOR)+DIREC2(NC)*DIREC2(NC)
          OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
          RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
  419   CONTINUE
      ENDIF
C-----
      IF(NOR1.EQ.1) THEN
        OC1=CM0
!HPF$ INDEPENDENT, REDUCTION(OC1), ON HOME(DIREC2(NC)), REUSE
        DO 411 NC=NINTCI,NINTCF
          OC1=OC1+ADXOR1(NC)*DIREC2(NC)
```

```
  411   CONTINUE
C-----
        OC1=OC1/CNORM(1)
C-----
        OMEGA=0
        RESNEW=0.0
        CNORM(NOR)=0
!HPF$ INDEPENDENT, REDUCTION(OMEGA, RESNEW, CNORM(NOR)), REUSE
        DO 420 NC=NINTCI,NINTCF
          DIREC2(NC)=DIREC2(NC)-OC1*ADXOR1(NC)
          DIREC1(NC)=DIREC1(NC)-OC1*DXOR1(NC)
          OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
          RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
          CNORM(NOR)=CNORM(NOR)+DIREC2(NC)*DIREC2(NC)
  420   CONTINUE
      ENDIF
C-----
      IF(NOR1.EQ.2) THEN
        OC1=CM0
        OC2=CM0
!HPF$ INDEPENDENT, REDUCTION(OC1, OC2), ON HOME(DIREC2(NC)),
!HPF$* ON HOME(DIREC2(NC)), REUSE
        DO 412 NC=NINTCI,NINTCF
          OC1=OC1+ADXOR1(NC)*DIREC2(NC)
          OC2=OC2+ADXOR2(NC)*DIREC2(NC)
  412   CONTINUE
C-----
        OC1=OC1/CNORM(1)
        OC2=OC2/CNORM(2)
C-----
        OMEGA=CM0
        RESNEW=CM0
        CNORM(NOR)=CM0
C-----
!HPF$ INDEPENDENT, REDUCTION(CNORM(NOR), OMEGA, RESNEW),
!HPF$* ON HOME(DIREC2(NC)), REUSE
        DO 422 NC=NINTCI,NINTCF
          DIREC2(NC)=DIREC2(NC)-OC1*ADXOR1(NC)-OC2*ADXOR2(NC)
          DIREC1(NC)=DIREC1(NC)-OC1*DXOR1(NC)- OC2*DXOR2(NC)
          CNORM(NOR)=CNORM(NOR)+DIREC2(NC)*DIREC2(NC)
          OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
          RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
  422   CONTINUE
      ENDIF
C-----
      OMEGA=OMEGA/CNORM(NOR)
      RESNEW=SQRT(RESNEW)
C-----
      IF(NOR.EQ.NOMAX) THEN
        NOR=1
      ELSE
        IF(NOR.EQ.1) THEN
!HPF$ INDEPENDENT, ON HOME(DXOR1(NC)), REUSE
          DO 490 NC=NINTCI,NINTCF
```

```
            DXOR1(NC)=DIREC1(NC)
            ADXOR1(NC)=DIREC2(NC)
  490     CONTINUE
        ENDIF
        IF(NOR.EQ.2) THEN
!HPF$ INDEPENDENT, ON HOME(DXOR2(NC)), REUSE
          DO 491 NC=NINTCI,NINTCF
            DXOR2(NC)=DIREC1(NC)
            ADXOR2(NC)=DIREC2(NC)
  491     CONTINUE
        ENDIF
        NOR=NOR+1
      ENDIF
C-----
      NOR1=NOR-1
C-----
      RATIO=RESNEW/RESREF
      print*,iter,ratio
      IF(RATIO.LE.1.0D-10) GOTO 9999
C-----
!HPF$ INDEPENDENT, ON HOME(VAR(NC)), REUSE
      DO 440 NC=NINTCI,NINTCF
        VAR(NC)=VAR(NC)+OMEGA*DIREC1(NC)
        RESVEC(NC)=RESVEC(NC)-OMEGA*DIREC2(NC)
        DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
  440 CONTINUE
C-----
      ITER=ITER+1
C-----
      IF(ITER.LT.21) GOTO 100
C      IF(ITER.LT.10000) GOTO 100
C-----
 9999 CONTINUE
C-----
      STOP
      END
```

file: comm.inc

```
-----------------------------------------------------------------------
C-----comm.inc
C-----------------------------------------------------------------------
C----- TJUNC
      PARAMETER (NNINTC= 13845,
     X           NNCELL= 19061)
C----- SWIRL
C      PARAMETER (NNINTC= 47312,
C     X           NNCELL= 60394)
C----- PENT
C      PARAMETER (NNINTC= 108000,
C     X           NNCELL= 124928)
C----- COJACK
C      PARAMETER (NNINTC= 318044,
```

```
C     X           NNCELL= 359899)
C-----
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
C-----
      DOUBLE PRECISION SU(NNINTC)
      INTEGER LCC(6,NNINTC)
C-----
      DOUBLE PRECISION VAR(NNINTC)
      DOUBLE PRECISION BP(NNINTC)
      DOUBLE PRECISION BE(NNINTC)
      DOUBLE PRECISION BW(NNINTC)
      DOUBLE PRECISION BN(NNINTC)
      DOUBLE PRECISION BS(NNINTC)
      DOUBLE PRECISION BH(NNINTC)
      DOUBLE PRECISION BL(NNINTC)
C-----
      INTEGER NINTCI,NINTCF,NEXTCI,NEXTCF
C-----
C-----ORTHOMIN
C-----
      DOUBLE PRECISION CGUP(NNCELL)
      DOUBLE PRECISION DIREC1(NNCELL)
C-----
      DOUBLE PRECISION DIREC2(NNINTC)
      DOUBLE PRECISION DIREC1V(6,NNINTC)
      DOUBLE PRECISION RESVEC(NNINTC)
      DOUBLE PRECISION ADXOR1(NNINTC)
      DOUBLE PRECISION ADXOR2(NNINTC)
      DOUBLE PRECISION DXOR1(NNINTC)
      DOUBLE PRECISION DXOR2(NNINTC)
      DOUBLE PRECISION OC(10),CNORM(10)
C-----DATA DISTRIBUTION
C-----
      INTEGER MAP(NNCELL)
C-----
C-----GENERAL
      INTEGER ITER
      DOUBLE PRECISION RATIO
C-----
      COMMON/C0001/ SU
      COMMON/C0002/ LCC
C-----
      COMMON/C0004/ VAR
      COMMON/C0005/ BP
      COMMON/C0006/ BE
      COMMON/C0007/ BW
      COMMON/C0008/ BN
      COMMON/C0009/ BS
      COMMON/C0010/ BH
      COMMON/C0011/ BL
      COMMON/C0012/ NINTCI,NINTCF,NEXTCI,NEXTCF
C-----ORTHOMIN
      COMMON/C0013/ CGUP
      COMMON/C0014/ DIREC1
```

```
C-----
      COMMON/C0015/ DIREC2
      COMMON/C0016/ DIREC1V
      COMMON/C0017/ RESVEC
      COMMON/C0018/ ADXOR1
      COMMON/C0019/ ADXOR2
      COMMON/C0020/ DXOR1
      COMMON/C0021/ DXOR2
      COMMON/C0022/ OC
      COMMON/C0023/ CNORM
C-----DATA DISTRIBUTION
      COMMON/C0025/ MAP
      COMMON/C0026/ ITER
      COMMON/C0028/ RATIO
```

7.2 ECMWF HPF+ kernels

7.2.1 LG kernel

```
      SUBROUTINE STEP01          ! LG VARIANT 1

      ...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)

!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),INDIRECT(MAPFLD0))
!HPF$*ONTO PROCSAB :: ZGL
      REAL ZGL(NPROMAG,NGT0)

!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZGA,ZGB
      REAL ZGA(NGPTOTG,NGT0)
      REAL ZGB(NGPTOTG,NGT0)

!HPF$ DISTRIBUTE(BLOCK) ONTO PROCS :: IPROC
      INTEGER IPROC(NPROC)

      INTEGER INDL(NGPTOTG)

      INTERFACE

      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMFFT_LOCAL1(
     + KPROC,PGL,LUPDFFT,
     + NPROC,NPROMAG,NGT0,NPROCA,NPROCB,NDGSAG,NDGENG,
     + NDLON,NFLD0,NPTRLS,NULTPP,NLOENG,NSTABUFL)
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGL(:,:)
!HPF$ INHERIT KPROC,PGL
      LOGICAL LUPDFFT
      INTEGER NFLD0(NPROCB)
      INTEGER NPTRLS(NPROCA)
      INTEGER NULTPP(NPROCA)
      INTEGER NLOENG(NDGSAG:NDGENG)
      INTEGER NSTABUFL(NDGSAG:NDGENG)
      END

      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMPHYS_LOCAL1(
     + KPROC,PGB,LUPDPHYS,
     + NPROC,NGPTOTG,NPROMA,NGT0,NFLEVG,NPROCA,NPROCB,NGPTOT)
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGB(:,:)
!HPF$ INHERIT KPROC,PGB
      LOGICAL LUPDPHYS
      INTEGER NGPTOT(NPROC)
      END

      END INTERFACE

C
```

```
C INITIALISE IPROC ARRAY (USED TO DETERMINE WHOAMI IN HPF_LOCAL CONTEXT)
C
!HPF$ INDEPENDENT
      DO I=1,NPROC
        IPROC(I)=I
      ENDDO

C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT,NEW(J)
      DO JFLD=1,NGTO
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          ZGA(J,JFLD)=FLOAT(JFLD*NGPTOTG+J)
        ENDDO
      ENDDO

C
C INITIALISE INDEX ARRAY TO WORKAROUND VFC CONSTRAINT THAT BOUNDS OF
C NESTED INDEPENDENT LOOPS SHOULD BE INVARIANT
C
      IL=0
      DO JGL=1,NDGLG
        DO JL=1,NLOENG(JGL)
          IL=IL+1
          INDL(IL)=NSTABUFL(JGL)+JL
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C
      DO I=1,NSTOP

C
C       Perform GTOL transposition
C
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JFLD=1,NGTO
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL(INDL(J),JFLD)=ZGA(J,JFLD)
          ENDDO
        ENDDO
C
C       Simulate call to FFT package (E-W dependencies on
C       each latitude)
C
        CALL SIMFFT_LOCAL1(
     +    IPROC,ZGL,LUPDFFT,
     +    NPROC,NPROMAG,NGTO,NPROCA,NPROCB,NDGSAG,NDGENG,
     +    NDLON,NFLDO,NPTRLS,NULTPP,NLOENG,NSTABUFL)
C
```

```
C       Perform LTOG transposition
C
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JFLD=1,NGT0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGB(J,JFLD)=ZGL(INDL(J),JFLD)
          ENDDO
        ENDDO
C
C       Simulate call to physics package (vertical dependencies)
C
        CALL SIMPHYS_LOCAL1(
     +    IPROC,ZGB,LUPDPHYS,
     +    NPROC,NGPTOTG,NPROMA,NGT0,NFLEVG,NPROCA,NPROCB,NGPTOT)
        IF( LUPDFFT.OR.LUPDPHYS )THEN
          ZGA=ZGB
        ENDIF
        RES(I)=ZGB(1,1)

      ENDDO

C
C CHECK CORRECTNESS OF GTOL AND LTOG TESTS
C
      NERROR=0
      IF( .NOT.(LUPDFFT.OR.LUPDPHYS) )THEN
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE),REDUCTION(NERROR)
        DO JFLD=1,NGT0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            IF( ZGA(J,JFLD).NE.ZGB(J,JFLD) ) THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO
      ENDIF

      ...

      RETURN
      END


      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMFFT_LOCAL1(
     + KPROC,PGL,LUPDFFT,
     + NPROC,NPROMAG,NGT0,NPROCA,NPROCB,NDGSAG,NDGENG,
     + NDLON,NFLD0,NPTRLS,NULTPP,NLOENG,NSTABUFL)
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGL(:,:)
!HPF$ INHERIT KPROC,PGL
      LOGICAL LUPDFFT
      INTEGER NFLD0(NPROCB)
      INTEGER NPTRLS(NPROCA)
```

```
INTEGER NULTPP(NPROCA)
INTEGER NLOENG(NDGSAG:NDGENG)
INTEGER NSTABUFL(NDGSAG:NDGENG)

REAL ZL(NDLON+2)

MYPROC=KPROC(1)
MYSETA=MOD(MYPROC-1,NPROCA)+1
MYSETB=(MYPROC-1)/NPROCA+1
IFLDS=NFLD0(MYSETB)
ILAT1=NPTRLS(MYSETA)
ILAT2=ILAT1+NULTPP(MYSETA)-1
DO JFLD=1,IFLDS
  DO JGL=ILAT1,ILAT2
    ILEN=NLOENG(JGL)
    IBEG=NSTABUFL(JGL)+1-NSTABUFL(ILAT1)+1
    IEND=IBEG+ILEN-1
    ZL(1:NLOENG(JGL))=PGL(IBEG:IEND,JFLD)
    CALL SIMFFT(ZL,NDLON,ILEN)
    IF( LUPDFFT )THEN
      PGL(IBEG:IEND,JFLD)=ZL(1:NLOENG(JGL))
    ENDIF
  ENDDO
ENDDO

RETURN
END


SUBROUTINE SIMFFT(PL,KDLON,KLEN)
REAL PL(KDLON+2)
Z=KLEN
K=LOG(Z)
DO I=1,K
  ZMIN=PL(1)
  ZMAX=PL(1)
  DO J=2,KLEN
    IF( PL(J).LT.ZMIN )THEN
      ZMIN=PL(J)
    ENDIF
    IF( PL(J).GT.ZMAX )THEN
      ZMAX=PL(J)
    ENDIF
  ENDDO
  DO J=1,KLEN
    PL(J)=PL(J)-SQRT(PL(J))*SQRT(ZMAX-ZMIN)/100.
  ENDDO
  PL(KLEN+1)=ZMIN
  PL(KLEN+2)=ZMAX
ENDDO
RETURN
END
```

```
      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMPHYS_LOCAL1(
     + KPROC,PGB,LUPDPHYS,
     + NPROC,NGPTOTG,NPROMA,NGT0,NFLEVG,NPROCA,NPROCB,NGPTOT)
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGB(:,:)
!HPF$ INHERIT KPROC,PGB
      LOGICAL LUPDPHYS
      INTEGER NGPTOT(NPROC)

      REAL ZGB(NPROMA,NGT0)
      REAL T(NPROMA,NFLEVG)

      MYPROC=KPROC(1)
      MYSETA=MOD(MYPROC-1,NPROCA)+1
      MYSETB=(MYPROC-1)/NPROCA+1
      IGPTOT=NGPTOT(MYPROC)

      IB=0
      DO J=1,IGPTOT,NPROMA
        IBEG=1
        IEND=MIN(NPROMA,IGPTOT-J+1)
        DO JFLD=1,NGT0
          DO JJ=IBEG,IEND
            ZGB(JJ,JFLD)=PGB(J+JJ-1,JFLD)
          ENDDO
        ENDDO
        IB=IB+1
        T(:,:)=ZGB(:,1:NFLEVG)
        CALL SIMPHYS(T,IEND,NPROMA,NFLEVG)
        IF( LUPDPHYS )THEN
          ZGB(:,1:NFLEVG)=T(:,:)
        ENDIF
        DO JFLD=1,NGT0
          DO JJ=IBEG,IEND
            PGB(J+JJ-1,JFLD)=ZGB(JJ,JFLD)
          ENDDO
        ENDDO
      ENDDO

      RETURN
      END


      SUBROUTINE SIMPHYS(T,KEND,KPROMA,KFLEV)

      REAL T(KPROMA,KFLEV)
      REAL U(KPROMA,KFLEV,KFLEV)

      U(:,:,:)=0.
      DO JK2=1,KFLEV
        DO JK1=1,KFLEV
          IF( JK1.NE.JK2 )THEN
            DO JL=1,KEND
              IF( T(JL,JK1).LT.T(JL,JK2) )THEN
```

```
            U(JL,JK1,JK2)=(T(JL,JK2)-T(JL,JK1))/T(JL,JK2)
          ELSE
            U(JL,JK1,JK2)=(T(JL,JK1)-T(JL,JK2))/T(JL,JK1)
          ENDIF
        ENDDO
      ENDIF
    ENDDO
  ENDDO

  DO JK2=1,KFLEV-1
    DO JK1=1,KFLEV-1
      DO JL=1,KEND
        IF( U(JL,JK2,JK1).GT.U(JL,JK1+1,JK2+1) )THEN
          T(JL,JK1)=T(JL,JK1)+U(JL,JK2,JK1)/10.
        ELSE
          T(JL,JK2)=T(JL,JK2)-U(JL,JK1,JK2)/10.
        ENDIF
      ENDDO
    ENDDO
  ENDDO

  RETURN
  END


  SUBROUTINE STEPO2          ! LG VARIANT 2

  ...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)

!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK,*) ONTO PROCSAB :: ZGL3
      REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK) ONTO PROCSAB :: ZGL2
      REAL ZGL2(NPROMAG,NG2D0)

!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*,*) ONTO PROCS :: ZGA3,ZGB3
      REAL ZGA3(NGPTOTG,NFLEVG,NG3D0)
      REAL ZGB3(NGPTOTG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: ZGA2,ZGB2
      REAL ZGA2(NGPTOTG,NG2D0)
      REAL ZGB2(NGPTOTG,NG2D0)

!HPF$ DISTRIBUTE(BLOCK) ONTO PROCS :: IPROC
      INTEGER IPROC(NPROC)

      INTEGER INDL(NGPTOTG)

      INTERFACE

      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMFFT_LOCAL2(
     + KPROC,PGL3,PGL2,LUPDFFT,NG3D0,NG2D0,NFLEVG,
     + NPROC,NPROMAG,NPROCA,NPROCB,NDGSAG,NDGENG,
     + NDLON,NUMLL,NPSURF,NPTRLS,NULTPP,NLOENG,NSTABUFL)
```

```
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGL3(:,:,:)
      REAL,INTENT(INOUT) :: PGL2(:,:)
!HPF$ INHERIT KPROC,PGL3,PGL2
      LOGICAL LUPDFFT
      INTEGER NUMLL(NPROCB+1)
      INTEGER NPSURF(NPROCB)
      INTEGER NPTRLS(NPROCA)
      INTEGER NULTPP(NPROCA)
      INTEGER NLOENG(NDGSAG:NDGENG)
      INTEGER NSTABUFL(NDGSAG:NDGENG)
      END

      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMPHYS_LOCAL2(
     + KPROC,PGB3,PGB2,LUPDPHYS,NG3D0,NG2D0,NFLEVG,
     + NPROC,NGPTOTG,NPROMA,NGT0,NPROCA,NPROCB,NGPTOT)
      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGB3(:,:,:)
      REAL,INTENT(INOUT) :: PGB2(:,:)
!HPF$ INHERIT KPROC,PGB3,PGB2
      LOGICAL LUPDPHYS
      INTEGER NGPTOT(NPROC)
      END

      END INTERFACE

C
C INITIALISE IPROC ARRAY (USED TO DETERMINE WHOAMI IN HPF_LOCAL CONTEXT)
C
!HPF$ INDEPENDENT
      DO I=1,NPROC
        IPROC(I)=I
      ENDDO

C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT,NEW(JLEV,J)
      DO JVAR=1,NG3D0
!HPF$ INDEPENDENT
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGA3(J,JLEV,JVAR)=FLOAT(((JVAR-1)*NFLEVG+JLEV)*NGPTOTG+J)
          ENDDO
        ENDDO
      ENDDO
!HPF$ INDEPENDENT,NEW(J)
      DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          ZGA2(J,JVAR)=FLOAT((NG3D0*NFLEVG+JVAR)*NGPTOTG+J)
        ENDDO
      ENDDO
```

```
C
C INITIALISE INDEX ARRAY TO WORKAROUND VFC CONSTRAINT THAT BOUNDS OF
C NESTED INDEPENDENT LOOPS SHOULD BE INVARIANT
C
      IL=0
      DO JGL=1,NDGLG
        DO JL=1,NLOENG(JGL)
          IL=IL+1
          INDL(IL)=NSTABUFL(JGL)+JL
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C

      DO I=1,NSTOP

C
C       Perform GTOL transposition
C
!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL2(INDL(J),JVAR)=ZGA2(J,JVAR)
          ENDDO
        ENDDO
C
C       Simulate call to FFT package (E-W dependencies on
C       each latitude)
C
        CALL SIMFFT_LOCAL2(
     +    IPROC,ZGL3,ZGL2,LUPDFFT,NG3D0,NG2D0,NFLEVG,
     +    NPROC,NPROMAG,NPROCA,NPROCB,NDGSAG,NDGENG,
     +    NDLON,NUMLL,NPSURF,NPTRLS,NULTPP,NLOENG,NSTABUFL)
C
C       Perform LTOG transposition
C
!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
```

```
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGB3(J,JLEV,JVAR)=ZGL3(INDL(J),JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGB2(J,JVAR)=ZGL2(INDL(J),JVAR)
          ENDDO
        ENDDO
C
C       Simulate call to physics package (vertical dependencies)
C
        CALL SIMPHYS_LOCAL2(
     +    IPROC,ZGB3,ZGB2,LUPDPHYS,NG3D0,NG2D0,NFLEVG,
     +    NPROC,NGPTOTG,NPROMA,NGT0,NPROCA,NPROCB,NGPTOT)
        IF( LUPDFFT.OR.LUPDPHYS )THEN
          ZGA3=ZGB3
          ZGA2=ZGB2
        ENDIF
        RES(I)=ZGB3(1,1,1)

      ENDDO

C
C CHECK CORRECTNESS OF GTOL AND LTOG TESTS
C
      NERROR=0
      IF( .NOT.(LUPDFFT.OR.LUPDPHYS) )THEN
!HPF$ INDEPENDENT,NEW(JLEV),REDUCTION(NERROR)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              IF( ZGA3(J,JLEV,JVAR).NE.ZGB3(J,JLEV,JVAR) )THEN
                NERROR=NERROR+1
              ENDIF
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REDUCTION(NERROR)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            IF( ZGA2(J,JVAR).NE.ZGB2(J,JVAR) )THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO
      ENDIF
```

```
      ...

      RETURN
      END


      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMFFT_LOCAL2(
     + KPROC,PGL3,PGL2,LUPDFFT,NG3D0,NG2D0,NFLEVG,
     + NPROC,NPROMAG,NPROCA,NPROCB,NDGSAG,NDGENG,
     + NDLON,NUMLL,NPSURF,NPTRLS,NULTPP,NLOENG,NSTABUFL)

      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGL3(:,:,:)
      REAL,INTENT(INOUT) :: PGL2(:,:)
!HPF$ INHERIT KPROC,PGL3,PGL2
      LOGICAL LUPDFFT
      INTEGER NUMLL(NPROCB+1)
      INTEGER NPSURF(NPROCB)
      INTEGER NPTRLS(NPROCA)
      INTEGER NULTPP(NPROCA)
      INTEGER NLOENG(NDGSAG:NDGENG)
      INTEGER NSTABUFL(NDGSAG:NDGENG)

      REAL ZL(NDLON+2)

      MYPROC=KPROC(1)
      MYSETA=MOD(MYPROC-1,NPROCA)+1
      MYSETB=(MYPROC-1)/NPROCA+1
      IFLEV=NUMLL(MYSETB)
      IVAR2=NPSURF(MYSETB)
      ILAT1=NPTRLS(MYSETA)
      ILAT2=ILAT1+NULTPP(MYSETA)-1

      DO JVAR=1,NG3D0
        DO JLEV=1,IFLEV
          DO JGL=ILAT1,ILAT2
            ILEN=NLOENG(JGL)
            IBEG=NSTABUFL(JGL)+1-NSTABUFL(ILAT1)+1
            IEND=IBEG+ILEN-1
            ZL(1:NLOENG(JGL))=PGL3(IBEG:IEND,JLEV,JVAR)
            CALL SIMFFT(ZL,NDLON,ILEN)
            IF( LUPDFFT )THEN
              PGL3(IBEG:IEND,JLEV,JVAR)=ZL(1:NLOENG(JGL))
            ENDIF
          ENDDO
        ENDDO
      ENDDO

      DO JVAR=1,IVAR2
        DO JGL=ILAT1,ILAT2
          ILEN=NLOENG(JGL)
          IBEG=NSTABUFL(JGL)+1-NSTABUFL(ILAT1)+1
          IEND=IBEG+ILEN-1
```

```
          ZL(1:NLOENG(JGL))=PGL2(IBEG:IEND,JVAR)
          CALL SIMFFT(ZL,NDLON,ILEN)
          IF( LUPDFFT )THEN
            PGL2(IBEG:IEND,JVAR)=ZL(1:NLOENG(JGL))
          ENDIF
        ENDDO
      ENDDO

      RETURN
      END


      EXTRINSIC(HPF_LOCAL) SUBROUTINE SIMPHYS_LOCAL2(
     + KPROC,PGB3,PGB2,LUPDPHYS,NG3D0,NG2D0,NFLEVG,
     + NPROC,NGPTOTG,NPROMA,NGT0,NPROCA,NPROCB,NGPTOT)

      INTEGER,INTENT(IN) :: KPROC(:)
      REAL,INTENT(INOUT) :: PGB3(:,:,:)
      REAL,INTENT(INOUT) :: PGB2(:,:)
!HPF$ INHERIT KPROC,PGB3,PGB2
      LOGICAL LUPDPHYS
      INTEGER NGPTOT(NPROC)

      REAL ZGB(NPROMA,NGT0)
      REAL T(NPROMA,NFLEVG)

      MYPROC=KPROC(1)
      MYSETA=MOD(MYPROC-1,NPROCA)+1
      MYSETB=(MYPROC-1)/NPROCA+1
      IGPTOT=NGPTOT(MYPROC)

      IB=0
      DO J=1,IGPTOT,NPROMA
        IBEG=1
        IEND=MIN(NPROMA,IGPTOT-J+1)
        IFLD=0
        DO JVAR=1,NG3D0
          DO JLEV=1,NFLEVG
            IFLD=IFLD+1
            DO JJ=IBEG,IEND
              ZGB(JJ,IFLD)=PGB3(J+JJ-1,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
        DO JVAR=1,NG2D0
          IFLD=IFLD+1
          DO JJ=IBEG,IEND
            ZGB(JJ,IFLD)=PGB2(J+JJ-1,JVAR)
          ENDDO
        ENDDO
        IB=IB+1
        T(:,:)=ZGB(:,1:NFLEVG)
        CALL SIMPHYS(T,IEND,NPROMA,NFLEVG)
        IF( LUPDPHYS )THEN
```

```
    ZGB(:,1:NFLEVG)=T(:,:)
  ENDIF
  IFLD=0
  DO JVAR=1,NG3D0
    DO JLEV=1,NFLEVG
      IFLD=IFLD+1
      DO JJ=IBEG,IEND
        PGB3(J+JJ-1,JLEV,JVAR)=ZGB(JJ,IFLD)
      ENDDO
    ENDDO
  ENDDO
  DO JVAR=1,NG2D0
    IFLD=IFLD+1
    DO JJ=IBEG,IEND
      PGB2(J+JJ-1,JVAR)=ZGB(JJ,IFLD)
    ENDDO
  ENDDO
ENDDO

RETURN
END



SUBROUTINE STEP03          ! LG VARIANT 3

...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)

!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK,*) ONTO PROCSAB :: ZGL3
      REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK) ONTO PROCSAB :: ZGL2
      REAL ZGL2(NPROMAG,NG2D0)

!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*,*) ONTO PROCS :: ZGA3,ZGB3
      REAL ZGA3(NGPTOTG,NFLEVG,NG3D0)
      REAL ZGB3(NGPTOTG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: ZGA2,ZGB2,ZT
      REAL ZGA2(NGPTOTG,NG2D0)
      REAL ZGB2(NGPTOTG,NG2D0)
      REAL ZT  (NGPTOTG,NFLEVG)

      REAL ZL(NDLON+2)

      INTEGER INDL(NGPTOTG)

      INTERFACE

      PURE SUBROUTINE SIMFFT_P(PL,KDLON,KLEN)
      REAL,INTENT(INOUT) :: PL(KDLON+2)
      INTEGER,INTENT(IN) :: KDLON,KLEN
      END SUBROUTINE SIMFFT_P
```

```
      SUBROUTINE SIMPHYS_H(T)
!HPF$ INHERIT T
      REAL,INTENT(INOUT) :: T(:,:)
      END SUBROUTINE SIMPHYS_H

      END INTERFACE


C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT,NEW(JLEV)
      DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGA3(J,JLEV,JVAR)=FLOAT(((JVAR-1)*NFLEVG+JLEV)*NGPTOTG+J)
          ENDDO
        ENDDO
      ENDDO
!HPF$ INDEPENDENT,NEW(J)
      DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          ZGA2(J,JVAR)=FLOAT((NG3D0*NFLEVG+JVAR)*NGPTOTG+J)
        ENDDO
      ENDDO

C
C INITIALISE INDEX ARRAY TO WORKAROUND VFC CONSTRAINT THAT BOUNDS OF
C NESTED INDEPENDENT LOOPS SHOULD BE INVARIANT
C
      IL=0
      DO JGL=1,NDGLG
        DO JL=1,NLOENG(JGL)
          IL=IL+1
          INDL(IL)=NSTABUFL(JGL)+JL
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C

      DO I=1,NSTOP

C
C       Perform GTOL transposition
C
!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
```

```
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL2(INDL(J),JVAR)=ZGA2(J,JVAR)
          ENDDO
        ENDDO
C
C       Simulate call to FFT package (E-W dependencies on
C       each latitude)
C
!HPF+ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF+ INDEPENDENT,NEW(JGL)
          DO JLEV=1,NFLEVG
!HPF+ INDEPENDENT,NEW(ILEN,J,ZL)
            DO JGL=1,NDGLG
              ILEN=NLOENG(JGL)
              DO J=1,NDLON
                IF( J.LE.NLOENG(JGL) )THEN
                  ZL(J)=ZGL3(NSTABUFL(JGL)+J,JLEV,JVAR)
                ENDIF
              ENDDO
              CALL SIMFFT_P(ZL,NDLON,ILEN)
              IF( LUPDFFT )THEN
                DO J=1,NDLON
                  IF( J.LE.NLOENG(JGL) )THEN
                    ZGL3(NSTABUFL(JGL)+J,JLEV,JVAR)=ZL(J)
                  ENDIF
                ENDDO
              ENDIF
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JGL),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT,NEW(ILEN,J,ZL)
          DO JGL=1,NDGLG
            ILEN=NLOENG(JGL)
            DO J=1,NDLON
              IF( J.LE.NLOENG(JGL) )THEN
                ZL(J)=ZGL2(NSTABUFL(JGL)+J,JVAR)
              ENDIF
            ENDDO
            CALL SIMFFT_P(ZL,NDLON,ILEN)
            IF( LUPDFFT )THEN
              DO J=1,NDLON
                IF( J.LE.NLOENG(JGL) )THEN
```

```
                 ZGL2(NSTABUFL(JGL)+J,JVAR)=ZL(J)
               ENDIF
             ENDDO
           ENDIF
         ENDDO
       ENDDO
C
C       Perform LTOG transposition
C
!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGB3(J,JLEV,JVAR)=ZGL3(INDL(J),JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGB2(J,JVAR)=ZGL2(INDL(J),JVAR)
          ENDDO
        ENDDO
C
C       Simulate call to physics package (vertical dependencies)
C
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZT(J,JLEV)=ZGB3(J,JLEV,1)
          ENDDO
        ENDDO
        CALL SIMPHYS_H(ZT)
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGB3(J,JLEV,1)=ZT(J,JLEV)
          ENDDO
        ENDDO
        IF( LUPDFFT.OR.LUPDPHYS )THEN
!HPF$ INDEPENDENT,NEW(JLEV),REUSE(LREUSE)
          DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
            DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
              DO J=1,NGPTOTG
                ZGA3(J,JLEV,JVAR)=ZGB3(J,JLEV,JVAR)
              ENDDO
            ENDDO
```

```
          ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
          DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGA2(J,JVAR)=ZGB2(J,JVAR)
            ENDDO
          ENDDO
        ENDIF
        RES(I)=ZGB3(1,1,1)

      ENDDO

C
C CHECK CORRECTNESS OF GTOL AND LTOG TESTS
C
      NERROR=0
      IF( .NOT.(LUPDFFT.OR.LUPDPHYS) )THEN
!HPF$ INDEPENDENT,NEW(JLEV),REDUCTION(NERROR)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(J)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              IF( ZGA3(J,JLEV,JVAR).NE.ZGB3(J,JLEV,JVAR) )THEN
                NERROR=NERROR+1
              ENDIF
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REDUCTION(NERROR)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            IF( ZGA2(J,JVAR).NE.ZGB2(J,JVAR) )THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO
      ENDIF

      RETURN
      END


      PURE SUBROUTINE SIMFFT_P(PL,KDLON,KLEN)
      REAL,INTENT(INOUT) :: PL(KDLON+2)
      INTEGER,INTENT(IN) :: KDLON,KLEN
      Z=KLEN
      K=LOG(Z)
      DO I=1,K
        ZMIN=PL(1)
        ZMAX=PL(1)
        DO J=2,KLEN
```

```
          IF( PL(J).LT.ZMIN )THEN
            ZMIN=PL(J)
          ENDIF
          IF( PL(J).GT.ZMAX )THEN
            ZMAX=PL(J)
          ENDIF
        ENDDO
        DO J=1,KLEN
          PL(J)=PL(J)-SQRT(PL(J))*SQRT(ZMAX-ZMIN)/100.
        ENDDO
        PL(KLEN+1)=ZMIN
        PL(KLEN+2)=ZMAX
      ENDDO
      RETURN
      END


      SUBROUTINE SIMPHYS_H(T)

      ...

!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ INHERIT T
      REAL,INTENT(INOUT) :: T(:,:)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*,*) ONTO PROCS :: U
      REAL U(NGPTOTG,NFLEVG,NFLEVG)

C
C Note that the HPF REUSE clause is not necessary in the following
C loops as the HPF compiler can easily determine that there is no
C communication required and therefore no inspector/work distribution/
C schedule generation needs to be done.
C

      DO JK2=1,NFLEVG
        DO JK1=1,NFLEVG
!HPF$ INDEPENDENT
          DO JL=1,NGPTOTG
            U(JL,JK1,JK2)=0.
          ENDDO
        ENDDO
      ENDDO

      DO JK2=1,NFLEVG
        DO JK1=1,NFLEVG
          IF( JK1.NE.JK2 )THEN
!HPF$ INDEPENDENT
            DO JL=1,NGPTOTG
              IF( T(JL,JK1).LT.T(JL,JK2) )THEN
                U(JL,JK1,JK2)=(T(JL,JK2)-T(JL,JK1))/T(JL,JK2)
              ELSE
                U(JL,JK1,JK2)=(T(JL,JK1)-T(JL,JK2))/T(JL,JK1)
              ENDIF
            ENDDO
```

```
          ENDIF
        ENDDO
      ENDDO

      DO JK2=1,NFLEVG-1
        DO JK1=1,NFLEVG-1
!HPF$ INDEPENDENT
          DO JL=1,NGPTOTG
            IF( U(JL,JK2,JK1).GT.U(JL,JK1+1,JK2+1) )THEN
              T(JL,JK1)=T(JL,JK1)+U(JL,JK2,JK1)/10.
            ELSE
              T(JL,JK2)=T(JL,JK2)-U(JL,JK1,JK2)/10.
            ENDIF
          ENDDO
        ENDDO
      ENDDO

      RETURN
      END
```

7.2.2 LM kernel

```
      SUBROUTINE STEP01          ! LM VARIANT 1

      ...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCB,NPROCA)

!HPF$ DISTRIBUTE(INDIRECT(MAPFLD0),GEN_BLOCK(MAPFL))
!HPF$*ONTO PROCSAB :: ZL0,ZL1
!HPF$ DISTRIBUTE(INDIRECT(MAPFLD0),GEN_BLOCK(MAPFM))
!HPF$*ONTO PROCSAB :: ZM

      REAL ZL0(NGT0,NFTOT2G)
      REAL ZL1(NGT0,NFTOT2G)
      REAL ZM (NGT0,NFTOT2G)

      INTEGER I0(NFTOTG)
      INTEGER I1(NFTOTG)
      INTEGER IL(NFTOTG)
      INTEGER IM(NFTOTG)

C
C INITIALISE INDEX ARRAYS TO WORKAROUND VFC CONSTRAINT THAT BOUNDS OF
C NESTED INDEPENDENT LOOPS SHOULD BE INVARIANT
C
      J=0
      DO JGL=1,NDGLG
        DO JM=0,NMENG(JGL)
          J=J+1
          I0(J)=NPNTGTB0G(JM,JGL)*2+1
          I1(J)=NPNTGTB1G(JM,JGL)*2+1
          IL(J)=JGL
          IM(J)=JM
        ENDDO
      ENDDO

C
C INITIALISE DATA FOR LTOM AND MTOL TESTS
C
!HPF$ INDEPENDENT,NEW(JF)
      DO J=1,NFTOTG
!HPF$ INDEPENDENT
        DO JF=1,NGT0
          ZL0(JF,I0(J)  )=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+1)
          ZL0(JF,I0(J)+1)=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+2)
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C
      NERROR=0
```

```
      DO I=1,NSTOP

C
C       Perform LTOM transposition
C
!HPF$ INDEPENDENT,NEW(JF),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT
          DO JF=1,NGT0
            ZM(JF,I1(J)  )=ZL0(JF,I0(J)  )
            ZM(JF,I1(J)+1)=ZL0(JF,I0(J)+1)
          ENDDO
        ENDDO

C
C       Check that transposed data (wave order) is correct
C
!HPF$ INDEPENDENT,NEW(JF),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
          DO JF=1,NGT0
            ZGOOD1=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+1)
            ZGOOD2=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+2)
            ZTEMP1=ZM(JF,I1(J)  )
            ZTEMP2=ZM(JF,I1(J)+1)
            IF( ZTEMP1.NE.ZGOOD1 )THEN
              NERROR=NERROR+1
            ENDIF
            IF( ZTEMP2.NE.ZGOOD2 )THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO

C
C       Perform MTOL transposition
C
!HPF$ INDEPENDENT,NEW(JF),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT
          DO JF=1,NGT0
            ZL1(JF,I0(J)  )=ZM(JF,I1(J)  )
            ZL1(JF,I0(J)+1)=ZM(JF,I1(J)+1)
          ENDDO
        ENDDO

C
C       Check that transposed data (latitude order) is correct
C
!HPF$ INDEPENDENT,NEW(JF),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
          DO JF=1,NGT0
```

```
            ZGOOD1=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+1)
            ZGOOD2=FLOAT(((JF*1000+IL(J))*1000+IM(J))*10+2)
            ZTEMP1=ZL1(JF,I0(J)  )
            ZTEMP2=ZL1(JF,I0(J)+1)
            IF( ZTEMP1.NE.ZGOOD1 )THEN
              NERROR=NERROR+1
            ENDIF
            IF( ZTEMP2.NE.ZGOOD2 )THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO

      ENDDO

      ...

      RETURN
      END


      SUBROUTINE STEPO2          ! LM VARIANT 2

      ...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCB,NPROCA)

!HPF$ DISTRIBUTE(BLOCK,*,GEN_BLOCK(MAPFL)) ONTO PROCSAB :: ZL03,ZL13
      REAL ZL03(NFLEVG,NG3D0,NFTOT2G)
      REAL ZL13(NFLEVG,NG3D0,NFTOT2G)
!HPF$ DISTRIBUTE(BLOCK,GEN_BLOCK(MAPFL)) ONTO PROCSAB :: ZL02,ZL12
      REAL ZL02(NG2D0,NFTOT2G)
      REAL ZL12(NG2D0,NFTOT2G)
!HPF$ DISTRIBUTE(BLOCK,*,GEN_BLOCK(MAPFM)) ONTO PROCSAB :: ZM3
      REAL ZM3 (NFLEVG,NG3D0,NFTOT2G)
!HPF$ DISTRIBUTE(BLOCK,GEN_BLOCK(MAPFM)) ONTO PROCSAB :: ZM2
      REAL ZM2 (NG2D0,NFTOT2G)

      INTEGER I0(NFTOTG)
      INTEGER I1(NFTOTG)
      INTEGER IL(NFTOTG)
      INTEGER IM(NFTOTG)

C
C INITIALISE INDEX ARRAYS TO WORKAROUND VFC CONSTRAINT THAT BOUNDS OF
C NESTED INDEPENDENT LOOPS SHOULD BE INVARIANT
C
      J=0
      DO JGL=1,NDGLG
        DO JM=0,NMENG(JGL)
          J=J+1
          I0(J)=NPNTGTB0G(JM,JGL)*2+1
          I1(J)=NPNTGTB1G(JM,JGL)*2+1
          IL(J)=JGL
```

```
          IM(J)=JM
        ENDDO
      ENDDO

C
C INITIALISE DATA FOR LTOM AND MTOL TESTS
C
!HPF$ INDEPENDENT,NEW(JVAR)
      DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(JLEV)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(IF,IVAL)
          DO JLEV=1,NFLEVG
            IF=(JVAR-1)*NFLEVG+JLEV
            IVAL=((IF*1000+IL(J))*1000+IM(J))*10
            ZL03(JLEV,JVAR,I0(J)  )=FLOAT(IVAL+1)
            ZL03(JLEV,JVAR,I0(J)+1)=FLOAT(IVAL+2)
          ENDDO
        ENDDO
      ENDDO
!HPF$ INDEPENDENT,NEW(JVAR)
      DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(IF,IVAL)
        DO JVAR=1,NG2D0
          IF=NG3D0*NFLEVG+JVAR
          IVAL=((IF*1000+IL(J))*1000+IM(J))*10
          ZL02(JVAR,I0(J)  )=FLOAT(IVAL+1)
          ZL02(JVAR,I0(J)+1)=FLOAT(IVAL+2)
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C
      NERROR=0

      DO I=1,NSTOP

C
C       Perform LTOM transposition
C
!HPF$ INDEPENDENT,NEW(JVAR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(JLEV)
          DO JVAR=1,NG3D0
!HPF$ INDEPENDENT
            DO JLEV=1,NFLEVG
              ZM3(JLEV,JVAR,I1(J)  )=ZL03(JLEV,JVAR,I0(J)  )
              ZM3(JLEV,JVAR,I1(J)+1)=ZL03(JLEV,JVAR,I0(J)+1)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),REUSE(LREUSE)
        DO J=1,NFTOTG
```

```
!HPF$ INDEPENDENT
          DO JVAR=1,NG2D0
            ZM2(JVAR,I1(J)  )=ZL02(JVAR,I0(J)  )
            ZM2(JVAR,I1(J)+1)=ZL02(JVAR,I0(J)+1)
          ENDDO
        ENDDO

C
C       Check that transposed data (wave order) is correct
C
!HPF$ INDEPENDENT,NEW(JVAR),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(JLEV)
          DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(IF,IVAL,ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
            DO JLEV=1,NFLEVG
              IF=(JVAR-1)*NFLEVG+JLEV
              IVAL=((IF*1000+IL(J))*1000+IM(J))*10
              ZGOOD1=FLOAT(IVAL+1)
              ZGOOD2=FLOAT(IVAL+2)
              ZTEMP1=ZM3(JLEV,JVAR,I1(J)  )
              ZTEMP2=ZM3(JLEV,JVAR,I1(J)+1)
              IF( ZTEMP1.NE.ZGOOD1 )THEN
                NERROR=NERROR+1
              ENDIF
              IF( ZTEMP2.NE.ZGOOD2 )THEN
                NERROR=NERROR+1
              ENDIF
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(IF,IVAL,ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
          DO JVAR=1,NG2D0
            IF=NG3D0*NFLEVG+JVAR
            IVAL=((IF*1000+IL(J))*1000+IM(J))*10
            ZGOOD1=FLOAT(IVAL+1)
            ZGOOD2=FLOAT(IVAL+2)
            ZTEMP1=ZM2(JVAR,I1(J)  )
            ZTEMP2=ZM2(JVAR,I1(J)+1)
            IF( ZTEMP1.NE.ZGOOD1 )THEN
              NERROR=NERROR+1
            ENDIF
            IF( ZTEMP2.NE.ZGOOD2 )THEN
              NERROR=NERROR+1
            ENDIF
          ENDDO
        ENDDO

C
C       Perform MTOL transposition
C
!HPF$ INDEPENDENT,NEW(JVAR),REUSE(LREUSE)
```

```
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(JLEV)
          DO JVAR=1,NG3D0
!HPF$ INDEPENDENT
            DO JLEV=1,NFLEVG
              ZL13(JLEV,JVAR,I0(J)  )=ZM3(JLEV,JVAR,I1(J))
              ZL13(JLEV,JVAR,I0(J)+1)=ZM3(JLEV,JVAR,I1(J)+1)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT
          DO JVAR=1,NG2D0
            ZL12(JVAR,I0(J)  )=ZM2(JVAR,I1(J))
            ZL12(JVAR,I0(J)+1)=ZM2(JVAR,I1(J)+1)
          ENDDO
        ENDDO

C
C       Check that transposed data (latitude order) is correct
C
!HPF$ INDEPENDENT,NEW(JVAR),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(JLEV)
          DO JVAR=1,NG3D0
!HPF$ INDEPENDENT,NEW(IF,IVAL,ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
            DO JLEV=1,NFLEVG
              IF=(JVAR-1)*NFLEVG+JLEV
              IVAL=((IF*1000+IL(J))*1000+IM(J))*10
              ZGOOD1=FLOAT(IVAL+1)
              ZGOOD2=FLOAT(IVAL+2)
              ZTEMP1=ZL13(JLEV,JVAR,I0(J)  )
              ZTEMP2=ZL13(JLEV,JVAR,I0(J)+1)
              IF( ZTEMP1.NE.ZGOOD1 )THEN
                NERROR=NERROR+1
              ENDIF
              IF( ZTEMP2.NE.ZGOOD2 )THEN
                NERROR=NERROR+1
              ENDIF
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),REDUCTION(NERROR),REUSE(LREUSE)
        DO J=1,NFTOTG
!HPF$ INDEPENDENT,NEW(IF,IVAL,ZGOOD1,ZGOOD2,ZTEMP1,ZTEMP2)
          DO JVAR=1,NG2D0
            IF=NG3D0*NFLEVG+JVAR
            IVAL=((IF*1000+IL(J))*1000+IM(J))*10
            ZGOOD1=FLOAT(IVAL+1)
            ZGOOD2=FLOAT(IVAL+2)
            ZTEMP1=ZL12(JVAR,I0(J)  )
            ZTEMP2=ZL12(JVAR,I0(J)+1)
            IF( ZTEMP1.NE.ZGOOD1 )THEN
```

```
          NERROR=NERROR+1
        ENDIF
        IF( ZTEMP2.NE.ZGOOD2 )THEN
          NERROR=NERROR+1
        ENDIF
      ENDDO
    ENDDO

  ENDDO

  ...

  RETURN
  END
```

7.2.3 SL kernel

```
      SUBROUTINE STEPO

      use nslhalo_module
      ...

!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)

CGM!HPF$ DISTRIBUTE(INDIRECT(MAPGP,HALO=NSLHALO),*) ONTO PROCS :: ZSL1
!HPF$ DISTRIBUTE(INDIRECT(MAPGP),*) ONTO PROCS :: ZSL1
      REAL ZSL1(NGPTOTG,NFLDSLB1)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP),*) ONTO PROCS :: ZSL2
      REAL ZSL2(NGPTOTG,NFLDSLB1)

      REAL RES(NSTOP)

C
C INITIALISE DATA FOR SEMI-LAGRANGIAN COMMS TEST
C
!HPF$ INDEPENDENT,NEW(J)
      DO JFLD=1,NFLDSLB1
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          ZSL1(J,JFLD)=FLOAT(JFLD*NGPTOTG+J)
        ENDDO
      ENDDO

C
C TIMESTEPPING LOOP
C

      DO I=1,NSTOP

C
C       Initialise ZSL2
C
        ZSL2=0.
C
C       Obtain halo data from neigbouring processors
C
CGM!HPF$ UPDATE_HALO(ZSL1)
C
C       Simulate SL interpolation
C
        ISL=NSLWIDE/2
        DO JSL=1,ISL

!HPF$ INDEPENDENT,NEW(J),REDUCTION(ZSL2)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(Z1,Z2,Z3,Z4,Z5,Z6)
            DO J=1,NGPTOTG
              Z1=ZSL1(NSLIND(1,J,JSL),JLEV)
```

```
             Z2=ZSL1(NSLIND(2,J,JSL),JLEV)
             Z3=ZSL1(NSLIND(3,J,JSL),JLEV)
             Z4=ZSL1(NSLIND(4,J,JSL),JLEV)
             Z5=ZSL1(NSLIND(2,J,JSL),MIN(NFLEVG,JLEV+1))
             Z6=ZSL1(NSLIND(3,J,JSL),MAX(1,JLEV-1))
             ZSL2(J,JLEV)=ZSL2(J,JLEV)+
     +         (Z1+Z2+Z3+Z4+Z5+Z6)/(6.*ISL)
           ENDDO
          ENDDO

        ENDDO
C
C       UPDATE ZSL1
C
        ZSL1=ZSL2

        RES(I)=ZSL1(1,1)

      ENDDO

      ...

      RETURN
      END


      MODULE NSLHALO_MODULE
      TYPE HALO
        INTEGER, POINTER    :: INDEX(:)
      END TYPE HALO
      TYPE (HALO),POINTER :: NSLHALO(:)
      END MODULE NSLHALO_MODULE
```

7.3 ESI HPF+ kernel

```
      program pcrash
C
C       07/96-SM :  HPF+ Project
C                   Benchmark REL.2.0.0
C                   without contact
C                   adapted for VFC 0.91 only.
C                   allocatable arrays
C
C-----------------------------------------------------------------------
C HPF DISTRIBUTE DIRECTIVES
C ----------------------------------------------------------------------
C
!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS())
C
C    elements
C
C
!HPF$ DISTRIBUTE(BLOCK) ONTO R   :: INDEX
!HPF$ DISTRIBUTE(BLOCK) ONTO R   :: ROEL,DT2AR
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R :: IY,STRS,FORCE1,FORCE2,FORCE3,FORCE4
C
C     nodes
C
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R      :: XYZC
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R      :: XMSI,V,A,XMS,BLAMB,X1,X,VGLOB
C
C ----------------------------------------------------------------------
C
      include 'static.h'
C
      CHARACTER*8 ELTYPE(7)
      DATA ELTYPE /'SOLID','SHELL','BEAM','MEMBRANE','USER','AIRBG',
     .            'CNTCT'/
C
Celements
C
      INTEGER,DIMENSION(:)  ,ALLOCATABLE :: INDEX
      INTEGER,DIMENSION(:,:),ALLOCATABLE :: IY
      REAL   ,DIMENSION(:)  ,ALLOCATABLE :: ROEL
      REAL   ,DIMENSION(:,:),ALLOCATABLE :: STRS
      REAL   ,DIMENSION(:,:),ALLOCATABLE :: FORCE1,FORCE2,FORCE3,FORCE4
C
Cnodes
C
      REAL, DIMENSION (:,:), ALLOCATABLE :: XMSI,V,A,XMS,BLAMB,X1,X
      REAL, DIMENSION (:,:), ALLOCATABLE :: VGLOB,XYZC
      REAL, DIMENSION (:,:), ALLOCATABLE :: CM
C
C     local arrays to compute internal forces
C
      DIMENSION IL(8),XL(12),VL(12),CL(12),RL(12),FORCE(12),
     .          STRS1(15)
```

```
C
      REAL DT2AUX,DT1AUX
      INTEGER ILMAUX,ITYAUX
C
C
      ALLOCATE (INDEX(N4NODE))
      ALLOCATE (IY(6,N4NODE))
      ALLOCATE (ROEL(N4NODE))
      ALLOCATE (STRS(NSTR,N4NODE))
      ALLOCATE (FORCE1(NDOF,N4NODE))
      ALLOCATE (FORCE2(NDOF,N4NODE))
      ALLOCATE (FORCE3(NDOF,N4NODE))
      ALLOCATE (FORCE4(NDOF,N4NODE))
      ALLOCATE (DT2AR(N4NODE))
C
      ALLOCATE(XMSI(NDOF,NUMNP),V(NDOF,NUMNP),A(NDOF,NUMNP))
      ALLOCATE(XMS(NDOF,NUMNP))
      ALLOCATE(BLAMB(9,NUMNP),X1(NDOF,NUMNP),X(3,NUMNP))
      ALLOCATE(VGLOB(3,NUMNP),XYZC(3,NUMNP))
      ALLOCATE(CM(NPROP,NUMMAT))
C
C ....................................................................
C .....................  read initial values ...........................
C ....................................................................
C
C files used: fort.30-> fort.36
C
       open (30,FILE='fort.30')
       open (31,FILE='fort.31')
       open (32,FILE='fort.32')
       open (33,FILE='fort.33')
       open (34,FILE='fort.34')
       open (35,FILE='fort.35')
       open (36,FILE='fort.36')
C
       READ(30,*) BLAMB
       READ(32,*) IY
       READ(30,*) XMSI
       READ(30,*) V
       READ(36,*) XMS
       READ(36,*) X1
       READ(36,*) X
       READ(32,*) INDEX
       READ(36,*) CM
       READ(31,*) STRS
       READ(36,*) XYZC
       READ(33,*) VGLOB
       READ(33,*) ROEL
C < ... >
C
      CALL SYSTEM_CLOCK(ICOUNT0,ICOUNT_RATE,ICOUNT_MAX)
C
C < ... >
C end of initialization ..............................
```

```
C
C*******************************************************************************
C                                                                              *
C                             Time loop                                        *
C                                                                              *
C*******************************************************************************
C
100   CONTINUE
C
        DT1 = DT2
        DT2 = 1.0E+6
        NECOUN = 0
        IELMTC = 0
        ALFMTC = 1.0
C
        ILMAUX = IELMTC
        ITYAUX = 0
C
C     Zero A
C
       A = 0.0
       DT2AR = DT2
C
C****************************************************
C                                                   *
C     INTERNAL AND EXTERNAL FORCES                  *
C                                                   *
C****************************************************
C
C     Internal forces of thin plate elements
C
!HPF$ INDEPENDENT, ON HOME(IY(2,I)),
!HPF$& NEW (IL,XL,VL,CL,RL,FORCE,STRS1,ROEL1,CM27,YOUNG,ANU,
!HPF$& GM01,GM02,GM03,GM04,GM05,GM06,GM07,GM08,GM09,GM10,GM11,GM12,
!HPF$& DT2AUX,DT1AUX,ILMAUX,ITYAUX,ECNT, REUSE
      DO I = 1, N4NODE
C
        ECNT   = ECNTR
        DT2AUX = DT2AR(I)
        DT1AUX = DT1            ! DT1 is a global value  = DT2 at the previous time-step
C
        IL(1) = IY(2,I)         ! Gather connectivities
        IL(2) = IY(3,I)         ! \
        IL(3) = IY(4,I)         !   quad element
        IL(4) = IY(5,I)         !  /
        IL(5) = IY(1,I)         ! Id  of material
        IL(6) = I               ! Id of element
        IL(7) = NPROP*(IY(1,I)-1)
        IL(8) = INDEX(I)        ! Pointer on STRS (not used)
C
cccc    XL(1)  = X(1,IY(2,I))  !! or this if problem occurs ??
C
C sigi: performed forward substitution of IL(1) ... IL(4)
C
```

```
      XL(1)  = X(1,IY(2,I))
      XL(2)  = X(2,IY(2,I))
      XL(3)  = X(3,IY(2,I))
      XL(4)  = X(1,IY(3,I))
      XL(5)  = X(2,IY(3,I))
      XL(6)  = X(3,IY(3,I))
      XL(7)  = X(1,IY(4,I))
      XL(8)  = X(2,IY(4,I))
      XL(9)  = X(3,IY(4,I))
      XL(10) = X(1,IY(5,I))
      XL(11) = X(2,IY(5,I))
      XL(12) = X(3,IY(5,I))
C
      VL(1)  = V(1,IY(2,I))
      VL(2)  = V(2,IY(2,I))
      VL(3)  = V(3,IY(2,I))
      VL(4)  = V(1,IY(3,I))
      VL(5)  = V(2,IY(3,I))
      VL(6)  = V(3,IY(3,I))
      VL(7)  = V(1,IY(4,I))
      VL(8)  = V(2,IY(4,I))
      VL(9)  = V(3,IY(4,I))
      VL(10) = V(1,IY(5,I))
      VL(11) = V(2,IY(5,I))
      VL(12) = V(3,IY(5,I))
C
      RL(1)  = VGLOB(1,IY(2,I))
      RL(2)  = VGLOB(2,IY(2,I))
      RL(3)  = VGLOB(3,IY(2,I))
      RL(4)  = VGLOB(1,IY(3,I))
      RL(5)  = VGLOB(2,IY(3,I))
      RL(6)  = VGLOB(3,IY(3,I))
      RL(7)  = VGLOB(1,IY(4,I))
      RL(8)  = VGLOB(2,IY(4,I))
      RL(9)  = VGLOB(3,IY(4,I))
      RL(10) = VGLOB(1,IY(5,I))
      RL(11) = VGLOB(2,IY(5,I))
      RL(12) = VGLOB(3,IY(5,I))
C
      YOUNG   = CM(2,IL(5))
      CM27    = CM(3,IL(5))
      ANU     = CM(4,IL(5))
      CL( 1) = YOUNG
      CL( 3) = ANU/(ANU-1.)
      CL( 5) = CM(14,IL(5))
      CL( 6) = YOUNG*.5/(1.+ANU)
      CL( 7) = YOUNG/(1.-ANU*ANU)
      CL( 8) = ANU*CL( 7)
      CM27   = CM(27,IL(5))
      CL( 9) = CM(29,IL(5))
      CL(10) = CM(30,IL(5))
      CL(11) = CM(31,IL(5))
C
      STRS1(1) = STRS(1,I)
```

```
      STRS1(2) = STRS(2,I)
      STRS1(3) = STRS(3,I)
      STRS1(4) = STRS(4,I)
      STRS1(5) = STRS(5,I)
      STRS1(6) = STRS(6,I)
      STRS1(7) = STRS(7,I)
      STRS1(8) = STRS(8,I)
      STRS1(9) = STRS(9,I)
      STRS1(10)= STRS(10,I)
      STRS1(11)= STRS(11,I)
      STRS1(12)= STRS(12,I)
      STRS1(13)= STRS(13,I)
      STRS1(14)= STRS(14,I)
      STRS1(15)= STRS(15,I)
      ROEL1 = ROEL(I)
C
      CALL FORCEI(I,STRS1,ROEL1,IL,XL,VL,RL,CL,FORCE,CM27,
     .            LSTR,DT1AUX,DT2AUX,ILMAUX,ITYAUX,
     .            GM01,GM02,GM03,GM04,GM05,GM06,
     .            GM07,GM08,GM09,GM10,GM11,GM12,ECNT)
C
      DT2AR(I) = DT2AUX
C
      STRS(1,I) = STRS1(1)
      STRS(2,I) = STRS1(2)
      STRS(3,I) = STRS1(3)
      STRS(4,I) = STRS1(4)
      STRS(5,I) = STRS1(5)
      STRS(6,I) = STRS1(6)
      STRS(7,I) = STRS1(7)
      STRS(8,I) = STRS1(8)
      STRS(9,I) = STRS1(9)
      STRS(10,I) = STRS1(10)
      STRS(11,I) = STRS1(11)
      STRS(12,I) = STRS1(12)
      STRS(13,I) = STRS1(13)
      STRS(14,I) = STRS1(14)
      STRS(15,I) = STRS1(15)
C
      FORCE1(1,I) =-FORCE(1)
      FORCE1(2,I) =-FORCE(2)
      FORCE1(3,I) =-FORCE(3)
      FORCE1(4,I) =-GM01
      FORCE1(5,I) =-GM02
      FORCE1(6,I) =-GM03
C
      FORCE2(1,I) =-FORCE(4)
      FORCE2(2,I) =-FORCE(5)
      FORCE2(3,I) =-FORCE(6)
      FORCE2(4,I) =-GM04
      FORCE2(5,I) =-GM05
      FORCE2(6,I) =-GM06
C
      FORCE3(1,I) =-FORCE(7)
```

```
        FORCE3(2,I) =-FORCE(8)
        FORCE3(3,I) =-FORCE(9)
        FORCE3(4,I) =-GM07
        FORCE3(5,I) =-GM08
        FORCE3(6,I) =-GM09
C
        FORCE4(1,I) =-FORCE(10)
        FORCE4(2,I) =-FORCE(11)
        FORCE4(3,I) =-FORCE(12)
        FORCE4(4,I) =-GM10
        FORCE4(5,I) =-GM11
        FORCE4(6,I) =-GM12
C
      END DO
C
!HPF$ INDEPENDENT, ON HOME(IY(2,N)), NEW(N1,N2,N3,N4),REDUCTION(A)
!HPF$& ,REUSE
       DO N=1,N4NODE
         DO I=1,NDOF
           A(I,IY(2,N)) = A(I,IY(2,N)) + FORCE1(I,N)
           A(I,IY(3,N)) = A(I,IY(3,N)) + FORCE2(I,N)
           A(I,IY(4,N)) = A(I,IY(4,N)) + FORCE3(I,N)
           A(I,IY(5,N)) = A(I,IY(5,N)) + FORCE4(I,N)
         END DO
       END DO
C
!HPF$ INDEPENDENT,REDUCTION(DT2)
       DO N=1,N4NODE
         DT2 = MIN(DT2,DT2AR(N))
       END DO
C (or)
c     DO I=1,NDOF
c       A(I,:) = SUM_SCATTER(FORCE1(I,:),A(I,:),IY(2,:))
c       A(I,:) = SUM_SCATTER(FORCE2(I,:),A(I,:),IY(3,:))
c       A(I,:) = SUM_SCATTER(FORCE3(I,:),A(I,:),IY(4,:))
c       A(I,:) = SUM_SCATTER(FORCE4(I,:),A(I,:),IY(5,:))
c     END DO
C
C     C O N T A C T - I M P A C T  R O U T I N E
C
      IF (NUMSV.NE.0) THEN
        IF (NCYCLE.EQ.0) THEN
          TISTP = DT2
          ICOUN = 0
        END IF
C
      END IF
C
C     Transform moments from global to nodal axes
C
!HPF$ INDEPENDENT, NEW(AX,AY,AZ)
        DO 210 N=1,NUMNP
          AX = A(4,N)
          AY = A(5,N)
```

```
          AZ = A(6,N)
          A(4,N) = BLAMB(1,N)*AX + BLAMB(2,N)*AY + BLAMB(3,N)*AZ
          A(5,N) = BLAMB(4,N)*AX + BLAMB(5,N)*AY + BLAMB(6,N)*AZ
          A(6,N) = BLAMB(7,N)*AX + BLAMB(8,N)*AY + BLAMB(9,N)*AZ
  210   CONTINUE
C
          DT2 = TSSFAC*DT2
          DT2 = MIN(DT2,1.1*DT2OLD)
          DT2OLD = DT2
C
      CONTINUE
      NCYCLE=NCYCLE+1
C
C*****************************************************
C                                                    *
C     COMPUTE ACCELERATIONS                          *
C                                                    *
C*****************************************************
C
      IF (NDOF.EQ.6) THEN
!HPF$ INDEPENDENT
        DO 350 N=1,NUMNP
          A(4,N) = A(4,N) + (XMS(5,N)-XMS(6,N))*V(5,N)*V(6,N)
          A(5,N) = A(5,N) + (XMS(6,N)-XMS(4,N))*V(6,N)*V(4,N)
          A(6,N) = A(6,N) + (XMS(4,N)-XMS(5,N))*V(4,N)*V(5,N)
  350   CONTINUE
      END IF
C
      A = A*XMSI
      DTX=0.5*(DT1+DT2)
C
C       Print accelerations, velocities, displacements and stresses
C
C
       IF (MOD(NCYCLE,20).EQ.0 .OR. NCYCLE.EQ.1) THEN
C
         PIOT = PIOT + PIOD
C
         WRITE (6,485) NCYCLE
         WRITE (6,490) TT,DT2,ELTYPE(ITYPTC),IELMTC
C
         WRITE(6,111)
         N = 200
         NOUT = N
         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                    A(1,N),A(2,N),A(3,N),NOUT
         N = 65
         NOUT = N
         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                    A(1,N),A(2,N),A(3,N),NOUT
         N = 490
         NOUT = N
         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                    A(1,N),A(2,N),A(3,N),NOUT
```

```
      END IF
C
C      New time
C
      TT=TT+DT2
C
C****************************************************
C                                                    *
C     VELOCITY (n+1/2) AND DISPLACEMENTS (n+1)       *
C                                                    *
C****************************************************
C
      V = V + DTX*A
      X1= X1+ DT2*V
      X = XYZC + X1(1:3,:)
C
      IF (NCYCLE.GT.MAXCYC) GOTO 99
C
C*******************
C                  *
      GOTO 100
C                  *
C*******************
C
   99 CONTINUE
C
      CALL SYSTEM_CLOCK(ICOUNT1,ICOUNT_RATE,ICOUNT_MAX)
      TIMEUSED=FLOAT(ICOUNT1-ICOUNT0)/FLOAT(ICOUNT_RATE)
      WRITE(6,*) ' N4NODE=',N4NODE,' NUMNP=',NUMNP
      WRITE(6,*) 'NB CYCLE=',NCYCLE
      WRITE(6,*) 'TIMEUSED=',FLOAT(ICOUNT1-ICOUNT0)/FLOAT(ICOUNT_RATE)
C
  485 FORMAT(1H1,17HSOLUTION AT CYCLE,I6/1H ,23(1H*))
  490 FORMAT(' TIME......................... ',E10.4/,
     .' TIME STEP.................... ',E10.4/,
     .' CONTROLLED BY ',A5,' ELEMENT NB. ',I10)
  111 FORMAT(/19H N O D A L  D A T A//5X,4HN.P.,5X,6HX-COOR,5X,
     .       6HY-COOR,
     .       5X,6HZ-COOR,5X,6H X-VEL,5X,6H Y-VEL,5X,6H Z-VEL,
     .       5X,6HX-ACCL,5X,6HY-ACCL,5X,6HZ-ACCL,5X,4HN.P.)
  222 FORMAT (I8,1X,1P,9E11.3,I8)
C
  505 FORMAT(I10)
  506 FORMAT(E13.4)
C
C-----------------------------------------------------------------------------
      CONTAINS
C
C-----------------------------------------------------------------------------
C
      PURE SUBROUTINE FORCEI(I,STRS,ROEL,IL,XL,VL,RL,CL,FORCE,CM27,
     .                 LSTR,DT1,DT2,IELMTC,ITYPTC,
     .                 GM01,GM02,GM03,GM04,GM05,GM06,
     .                 GM07,GM08,GM09,GM10,GM11,GM12,ECNT)
```

```
C
C      11/96-SM :  HPF+ Project
C                  Initial kernel adapted for VFCS only.
C
!HPF$ PUREST
C dummy arguments
      DIMENSION STRS(15),IL(8),XL(12),VL(12),RL(12),CL(12),
     .          FORCE(12)
C local arrays
      DIMENSION DICOS(9),B(4),ROTL(8),STRAIN(5),SUM(3),FMOM(8)
C
      REAL TN,AREA,DTGREL,TT
C
C < ...>
C
      RETURN
      END subroutine forcei

C
      END
```

7.4 NEC - CCRLE kernels

7.4.1 K2HPF kernel

```
      program K2HPF

      parameter (nelx=40, nelz=160, numnp=4*nelx*(nelz+1), &
       numel2=4*nelx*nelz)
      dimension f(6,numnp), x(3,numnp),  ix(4,numel2)
! Time-stepping
      dimension v(6,numnp)
      real mass(4)
      logical lreuse
      common /bkreuse/lreuse
     INTERFACE
         ::::::: lines omitted :::::
      END INTERFACE

!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (*,BLOCK) :: f, ix, x, v


! Initialisation
      mass(1) = 0.010
      mass(2) = 0.011
      mass(3) = 0.010
      mass(4) = 0.009
      open(unit=55,file='bench.in')
      read(55,*) ittot
      read(55,*) ireuse
      if(ireuse.eq.0) then
        lreuse = .false.
      else
        lreuse = .true.
      end if
!............

      call box(nelx,nelz,x,ix,numnp,numel2)

      call vbox(v,nelx,numnp)


      call wtime(t0)

      do 100 it = 1, ittot

        f = 0.0
        call kforce(f,x,ix,numnp,numel2)
        call update(f,v,x,mass,numnp)

100   continue

      call wtime(t1)
      nprocs = NUMBER_OF_PROCESSORS()
```

```
      open(unit=66,file='bench.out')
      write(66,*) 'Number of Processors = ',nprocs
      write(66,*) 'Model size: numnp = ',numnp,' numel2 = ',numel2
      write(66,*) 'Number of iterations = ',ittot
      write(66,*) 'Elapsed time for computational loop = ',t1-t0
      write(66,*) 'Checked time for computational loop = ',tc-tb
      write(66,*) 'Schedule REUSE variable = ',lreuse
      close(66)

      call kout(x,f,numnp)

      stop
      end

!=======================

      subroutine kforce(f,x,ix,numnp,numel2)

      real, dimension(:,:), intent(in) :: x
      real, dimension(:,:), intent(inout) :: f
      integer, dimension(:,:), intent(in) :: ix
      integer, intent(in) :: numnp,numel2

      real, dimension(6,4,numel2) :: aforce
      real, dimension(3,4) :: xn
      real, dimension(6,4) :: force
      logical lreuse
      common /bkreuse/lreuse

      INTERFACE
         SUBROUTINE MFORCE (force,xn)
! N.B. mforce is !HPF$ PUREST
         real, dimension(:,:), intent(in) :: xn
         real, dimension(:,:), intent(out) :: force
         END SUBROUTINE MFORCE
      END INTERFACE

!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (*,BLOCK) :: f, ix, x
!HPF$ DISTRIBUTE (*,*,BLOCK) :: aforce

!HPF$ INDEPENDENT, NEW(xn,force), ON HOME (ix(1,iel)), REUSE(lreuse)
      do iel = 1, numel2

!       xn(1:3,1:4) = x(1:3,ix(1:4,iel))
        do i=1,3
          xn(i,1) = x(i,ix(1,iel))
          xn(i,2) = x(i,ix(2,iel))
          xn(i,3) = x(i,ix(3,iel))
          xn(i,4) = x(i,ix(4,iel))
        end do

        call mforce(force,xn)
```

```
!       aforce(:,:,iel) = force
        do i=1,6
          aforce(i,1,iel) = force(i,1)
          aforce(i,2,iel) = force(i,2)
          aforce(i,3,iel) = force(i,3)
          aforce(i,4,iel) = force(i,4)
        end do

      end do

!HPF$ INDEPENDENT, ON HOME (ix(1,iel)), REDUCTION(f), REUSE(lreuse)
      do iel = 1, numel2
!       f(:,ix(:,iel)) = f(:,ix(:,iel)) + aforce(:,:,iel)
        do i=1,6
          f(i,ix(1,iel)) = f(i,ix(1,iel)) + aforce(i,1,iel)
          f(i,ix(2,iel)) = f(i,ix(2,iel)) + aforce(i,2,iel)
          f(i,ix(3,iel)) = f(i,ix(3,iel)) + aforce(i,3,iel)
          f(i,ix(4,iel)) = f(i,ix(4,iel)) + aforce(i,4,iel)
        end do
      end do

      return
      end

!=======================

      subroutine update(f,v,x,mass,numnp)

!       Update x & v from Forces: represents calculations of
! accelerations --> velocities --> displacements

      real, dimension(:,:), intent(inout) :: f,v,x
      real, dimension(:), intent(in) :: mass
!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (*,BLOCK) :: f, x, v

      dt1 = 1.0
      dt2 = 1.0

      f(1:3,:) = f(1:3,:) * mass(1)

      f(4,:) = (f(4,:) + (mass(3)-mass(4))*v(5,:)*v(6,:))*mass(2)
      f(5,:) = (f(5,:) + (mass(4)-mass(2))*v(4,:)*v(6,:))*mass(3)
      f(6,:) = (f(6,:) + (mass(2)-mass(3))*v(4,:)*v(5,:))*mass(4)

      v = v + dt1 * f

      x = x + dt2 * v(1:3,:)

      return
      end
```

7.4.2 K2HPF_a_ADP1 kernel

```
      subroutine kforce (f, x, ix, numnp, numel2,.....)

real, dimension(3,numnp), intent(in) :: x
real, dimension(6,numnp), intent(inout) :: f
integer, dimension(4,numel2), intent(in) :: ix
real, dimension (3,4) :: xn
real, dimension (6,4) :: force
real, dimension (3,4,numel2) :: all_xn
real, dimension (6,4,numel2) :: all_force

!HPF$ DISTRIBUTE (*,BLOCK) :: f
!HPF$ ALIGN x(*,I) with f(*,I)
!HPF$ DISTRIBUTE (*,BLOCK) :: ix
!HPF$ ALIGN all_force (*,*,I) WITH ix(*,I)
!HPF$ ALIGN all_xn (*,*,I) WITH ix(*,I)

!ADP$ TRACE ix

      do I = 1, 3
         forall (IEL=1:NUMEL2,K=1:4)
            all_xn (I,K,IEL) = x (I, ix(K, IEL))
         end forall
      end do

!HPF$ INDEPENDENT, NEW(xn,force), ON HOME (ix(1,iel))

      do iel = 1, numel2
        xn = all_xn (:,:,iel)
        call mforce(force,xn)
        all_force (:,:,iel) = force
      end do

      do k = 1, 6
        f = sum_scatter (all_force(k,:,:), f, k, ix)
      end do
```

7.4.3 K2HPF_a_ADP2 kernel

```
  program K2HPF

!HPF$ DISTRIBUTE (*,BLOCK) :: f, ix
!HPF$ ALIGN WITH f :: x,v

!     arrays f and x will get an overlap area for indirect addressing
!     note:  cannot be created dynamically (maybe in next version)
!               both arrays should have same size

!HPF$ SHADOW x(0:0,0:1000)
!HPF$ SHADOW f(0:0,0:1000)
```

```
      integer s_id      ! overlap schedule for ix on x and f

!     ADAPTOR requires interface to know that ix is not changed in KFORCE

!     create schedule for overlap areas of x (and f)

      call Dalib_indirect_overlap2 (s_id, x, ix)

      do 100 it = 1, ittot
        f = 0.0
        call kforce(f,x,ix, numnp,numel2,s_id,ti1,ti2)
        call update(f,v,x,mass,numnp,ti3)
100   continue
:::::::::::::::

      subroutine kforce (f, x, ix, numnp, numel2, s_id, ....)

!HPF$ DISTRIBUTE (*,BLOCK) :: f
!HPF$ ALIGN x(*,I) with f(*,I)
!HPF$ DISTRIBUTE (*,BLOCK) :: ix

!     get the non-local values of x

      call Dalib_indirect_ov_get (s_id, x)

!     update overlap area of forces f with 0.0

      call Dalib_indirect_ov_set (s_id, f, 0.0)

!     Note : RESIDENT directive absolutely necessary
!            user responsible for updating overlap area

!HPF$ INDEPENDENT, NEW(xn,force), ON HOME (ix(1,iel)), RESIDENT

      do iel = 1, numel2

        do J=1,4;  do I=1,3
          xn(I,J) = x(I,ix(J,iel))
        end do;  end do

        call mforce(force,xn)

        do J=1,4;  do I=1,6
          f(I,ix(J,iel)) = f(I,ix(J,iel)) + force(I,J)
        end do;  end do

      end do

!     8 stands for add_reals

      call Dalib_indirect_ov_put (s_id, f, 8)

:::::::::::::::
```