ESPRIT Project 21033

HPF+

Optimizing HPF for Advanced Applications

Deliverable D1.2c

Final Evaluation Report

Status:	Final Report
Version:	1.1 (26/04/98)
Confidentiality:	Public
Responsible:	NEC
Authoring Partners:	AVL, ECMWF, ESI, NEC
Project Period:	Year 2 + Extension

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

Collaboration Partner:
C&C Research Laboratories, NEC Europe Ltd. (NEC) - Germany/UK

Abstract

This report is the HPF+ project deliverable, D1.2c, covering the final evaluation of the HPF+ language extensions, as implemented within the HPF+ compiler VFC produced within the project. As part of this evaluation, performance comparisons will be made with hand-code message-passing (MPI) kernels and with HPF-1 kernels in implementations suitable for commercial (and research) compilers. The evaluation of the final language defintion was performed with the final versions of the benchmark kernels as described in the deliverable , D1.1b. Since the development of both kernels and language features has been a cumulative process - in terms of kernel complexity and language extension availability - this final evaluation report will restrict its attention to results obtained with the final set of kernels and the most recent releases of VFC. The evaluation process has been supported by the use of the performance analysis tool MEDEA.

Table of Contents

0 Evaluation Summary

This report covers the final evaluation of the HPF+ extended language using the final set of application kernels as detailed in the deliverable, D1.1b. In Sections 1-4, individual detailed evaluation reports from each of the application developers are included. Where kernels have not been included in that deliverable (or in previous reports) they can be found in sufficient detail in the Appendix. In this Section, we will include a summary of the major conclusions drawn from the evaluations and highlight the major language extensions developed within the project in the light of the application requirements.

0.1 Code Porting Effort

In all cases, it has been clearly demonstrated that the amount of code modification required to reach a parallel HPF code is considerably smaller than that for explicit message-passing. This is a major advantage of HPF as a parallel programming paradigm. However, there are some issues that need to be taken into account:

1. Compiler implementations, with varying levels of completeness and restrictions - currently lead to **non**-portable code.
2. Although HPF_LOCAL does allow a switch back to "local programming", this is less attractive for programmers without a message-passing background.
3. The more advanced features, such as those related to HALO functions, really require the code developer to possess a great deal of parallelisation knowledge. They should thus really only be used by experienced developers.

0.2 Data Distributions

For all applications included, the requirement for GEN_BLOCK distributions with run-time definition is an absolute must in general - though the ESI & NEC kernels were evaluated with test cases where this feature was not a dominant in order to concentrate on algorithmic problems. INDIRECT was necessary for both the FIRE and IFS benchmarks.

The extension of the HPF-2 SHADOW feature to the user-defined HALO functions provided a mechanism for performance optimisation in several kernels. Although the halo functions are included within the data distribution or re-distribution statements, they can also be seen as a mechanism to allow the user to indicate data which needs to be communicated, saving run-time analysis - their impact is thus more to be seen in conjunction with the UPDATE HALO calls and computational phases.

In most cases the requirement for DYNAMIC and RE-DISTRIBUTE was for a "clean" way to allow for the irregular distributions to be performed at run-time and in the initialisation phase. That is, the data redistribution would only be performed once. Such provision is expected to be most common and the inclusion of the RANGE option to specify that, although DYNAMIC the distribution will only be of a single type, is expected to be of great advantage in general. Example of frequent distributions requirements were seen within the applications: real data re-mapping within the ECMWF kernels (see Section 2.2) and halo redefinitions for the generic crash kernels (deliverable D1.1b).

0.3 Independent Loops

The need for efficient parallelisation of INDEPENDENT loops with indirect addressing, conditional statements, and subroutine calls - and in some cases a nesting of such loops - has been the central subject of the language developments and evaluations. It is not at all realistic to expect that complex, irregular codes will be able to be re-written in F90 array statements or simple FOR ALL constructs. Indeed, some investigations highlighted the possible performance losses when code segments were rewritten in this way. The focus has thus been on constructs to allow the user to assist with the minimisation of overheads generated for such use.

The two main mechanisms for this have been the schedule reuse directives (or, in the most general case, the provision of schedule variables) and halo regions.

The schedule reuse options allow the user to declare the invariance (or conditional invariance) of communication schedules to be generated by the compiler. What has been seen for all applications is that such directives can lead to extremely large performance gains - this can be seen easily when using the MEDEA tool and looking for the costs for the "inspector" phase of the independent loops.

The second mechanism places much more control in the hands of the programmer - the definition of HALO regions for distributed array and the specification that these halo regions are to be updated. The halo regions are, for each process and array, the non-local array elements required for particular calculations on a given process. Clearly, the global "HPF view" has been sacrificed in order to provide an optimisation possibility. Combined with schedule reuse (a requirement of the current VFC implementation), it has been seen that significant performance gains can be achieved.

Combined with the possibility to call PUREST subroutines from the INDEPENDENT loops and to allow reduction operations (REDUCTION option), the above developments are absolutely crucial if performant HPF codes for irregular applications are to be developed.

0.4 Performance

The evaluations carried out have set the goal of achieving absolute times approaching those possible with MPI implementations in F77. Any overheads due to the underlying F90 compilers have been highlighted. Thus, the application developers have tried to provide a realistic appraisal of the relative computing times to be expected were they to rely on the HPF or HPF+ languages for the parallel implementations of their codes.

The HPF+ language implementation within VFC has made major steps towards providing performant parallel versions of the project kernels - both in terms of absolute times and scaling behaviour. For the LG kernel from ECMWF, absolute performance has been reached which essentially matches that of the MPI kernel. Otherwise the achieved scaling with processor number is similar to that exhibited b the MPI codes. **However**, the major drawback remains the (in some cases enormous) computational overhead in the generated code. This can be seen throughout the evaluations when comparing the absolute computational times on a single CPU. The current performance hit would prevent the application developers from adopting HPF within their product code versions.

1 AVL

1.1 Introduction

In this section the final version of the AVL HPF+ kernel will be discussed. The complexity of the final kernel, that contains the solution of a scalar transport equation as it is used inside the FIRE program, has increased significantly compared to kernel version 2. In the first part an overview of the HPF+ kernel, the required HPF+ features and the porting effort will be discussed. The following section contains a description of the message passing kernel, followed by the performance evaluation of the HPF+ and the message passing kernels on the Meiko CS2 under the SUN F90 compiler for several different input datasets. Finally some results of a performance analysis produced by MEDEA are presented.

1.2 HPF+ kernel

Within the FIRE program up to 15 different transport equations are solved, one of them being the passive scalar transport equation, that is mainly used for the simulation of mixing processes between different species. Within the HPF+ kernel this transport equation is solved for an invariant, predefined velocity field that has been extracted from FIRE applications and is read by the kernel from an input data set. For the time marching solution of this equation it is necessary to compute convection, diffusion and source terms that give their contributions to the coefficients. Several additional parameters that are necessary to solve the equation are provided by the input data set. The equation solver itself is identical to the conjugate gradient equation solver of kernel version 2. The number of solver iterations performed at each time-step depend on the convergence criterion which is defined by a residual ratio of less then 1e-3. The discretisation of the equation has been implemented by two different differencing schemes that can be used alternatively: the first order upwind differencing scheme and the second order hybrid differencing scheme, which is a blending of upwind and central differencing depending on the local grid Peclet number.

The data structure within the kernel is the same as in the FIRE program, where all dependent variables (e.g.: velocity, pressure ...) are stored at the cell centres and only the fluxes are computed and stored at the cell faces. Neighbouring elements are accessed via the connectivity arrays LCC (for cell centre data) and LCF (for cell face data). For the parallelisation of the HPF+ and the message passing kernel all internal, cell-center based arrays are distributed in blocks of equal size, while arrays over the external or total number of cells are distributed indirectly. In order to minimise the amount of communication, cell face array elements are not distributed according to their numbering. These array elements are distributed indirectly so that the cell faces of each cell are distributed to the same processor as the respective cell center array element.

1.2.1 HPF+ features used

Data Distribution

All globally used arrays are defined within the F90 module globvar. A BLOCK distribution is used for arrays where only internal cell elements are accessed, while an INDIRECT distribution is used for arrays over the total number of cells, the boundary cells and the cell face arrays. The block distribution is specified in module globvar where all indirectly distributed arrays are also declared as dynamic. The indirect distribution is done by 5 different map arrays that are read from different files at runtime. These map arrays are generated by a separate program, that computes the data distribution for any number of processors and writes the distribution to a file. A dynamic generation of the map arrays during runtime is not possible because for the calculation of the map arrays the connectivity arrays LCC and LCF are required, which are distributed arrays.

```
!HPF$ DYNAMIC :: PS
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: PSO

    OPEN(20,STATUS='OLD',FILE='map1',FORM='FORMATTED')
    READ(20,*) map1
    CLOSE(20)

!HPF$ REDISTRIBUTE(INDIRECT(MAP1)) ONTO NODES :: PS
```

The above code fragment shows the data distribution of the passive scalar array PS, which is defined over the total number of cells and distributed indirectly, and the block distribution of array PSO (where the passive scalar values of the previous time step are stored), which is defined over the internal cells. The function NUMBER_OF_PROCESSORS has been used to determine at runtime how many processors are being used.

Independent Loops

The use of different data distributions and indirect data access within nearly all loops of the HPF+ kernel requires an extensive use of the INDEPENDENT directive. Due to the developments and improvements of the VFC compiler loops of high complexibility can be handled with the INDEPENDENT directive, like for example the computation of the convection coefficients, where several indirectly distributed arrays (which are distributed by different map arrays) are accessed indirectly over different connectivity arrays and are updated within the same loop. Within independent loops the following directives are used in the kernel:

The REDUCTION directive had to be used for the computation of global sums in the conjugate gradient equation solvers and for the calculation of the normalised residual.

The ON HOME clause was required for all loops where arrays with different data distribution are accessed.

For all loops where indirect data access is performed the REUSE clause was specified. As the time measurments presented in the following section show the REUSE clause is the most important feature in order to achieve a good performance and scalability of the code.

The RESIDENT directive was required within all loops where indirect data access to boundary cells is performed, which is always local, due to the data distribution.

```
!HPF$ INDEPENDENT, REDUCTION(CNORM1_LOC, OMEGA, RESNEW),
!HPF$* ON HOME(DIREC2(NC)), REUSE,
!HPF$* RESIDENT(ACOEFF, DIREC2, ADXOR1, RESVEC)
         DO 4 NC=NINTCI,NINTCF
             DIREC2(NC)=ACOEFF(NC)*DIREC1(NC)
        X           -ACOEFF(NC+NSFR01)*DIREC1(LCC(NC+NSFI01))
        X           -ACOEFF(NC+NSFR02)*DIREC1(LCC(NC+NSFI02))
        X           -ACOEFF(NC+NSFR03)*DIREC1(LCC(NC+NSFI03))
        X           -ACOEFF(NC+NSFR04)*DIREC1(LCC(NC+NSFI04))
        X           -ACOEFF(NC+NSFR05)*DIREC1(LCC(NC+NSFI05))
        X           -ACOEFF(NC+NSFR06)*DIREC1(LCC(NC+NSFI06))
            CNORM1_LOC=CNORM1_LOC+DIREC2(NC)*DIREC2(NC)
            OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
            RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
            ADXOR1(NC)=DIREC2(NC)
         4 CONTINUE
```

The above loop shows a code segment extracted from the equation solver where arrays of different size and different distribution are accessed indirectly and three sum-reductions are computed within the same loop.

1.2.2 Porting Effort

The final version of the kernel is based on several routines extracted from the FIRE code and on the conjugate gradient equation solver of benchmark version 2. The following modifications of the serial kernel had to be done to port the code to HPF+:

- Since FIRE is coded in F77, additional effort had to done in order to adapt the code to F90, e.g. the transformation of the COMMON block into an F90 module.

- Due to some restrictions of the VFC compiler the syntax of some loops had to be changed (e.g.: in-lining of array indices).

- The data distribution has been modified compared to kernel version 2: five different map arrays are required for the indirect data distribution. For the generation of these map arrays a separate program was implemented.

- New input data-sets have been extracted from the FIRE program and a routine to read these input data-sets has been implemented.

1.2.3 Message Passing version

The message passing version is based on the same serial code as the HPF+ version. The data structure used is equivalent to the HPF+ version except that instead of a module a common block was used for the definition of all globally used arrays. With respect to functionality the MPI and the HPF+ versions are exactly the same. The data distribution under MPI, which is done in a separate subroutine, is implemented so that array elements are distributed in the same way as it is done in the HPF+ version. The input data-set (which is the same as for the HPF+ version) is read by node zero and distributed to all other nodes.

For the indirect data access send and receive lists have to be computed out of the connectivity array LCC. During the computation these communication lists are used for the data exchange among the processors. Since the send and receive lists are computed dynamically during runtime, the same executable can be used for any processor configuration. Communication lists are used only for cell centre based data while cell face arrays are distributed in a way that no communication is necessary.

1.3 Performance

For the performance evaluation of the HPF+ and the MPI kernel two input data sets of different size have been used:

Dataset	No. of internal cells	No. of total cells
PIPE	10240	12672
DUCT	26400	30720

These input data sets have been extracted from FIRE applications and contain the connectivity arrays, the Cartesian velocity components in three dimensions as well as all required geometrical and boundary data.

All performance measurements have been done on the Meiko CS-2 at the University of Vienna. Since there was never a partition for higher number of processors available on the system, the maximum number of processors that has been used for the measurements was 16. All timings are related to the time marching solution of the equation, whereas the time required to read the input data set, for distributing the data and the initialisation has been neglected.

Performance of the HPF+ kernel

For the HPF+ version the SUN WorkShop F90 compiler was used with and without optimisation. The following time measurements have been obtained for the HPF+ Version of the Benchmark (all times in seconds):

Dataset	Compiler	1p	2p	4p	8p	16p
PIPE	f90 Speedup	821.53	400.50 2.05	213.27 3.85	127.39 6.45	97.59 8.42
	f90 -O3 Speedup	318.16	170.32 1.86	104.76 3.02	68.17 4.64	67.05 4.72
	f90 -O3 no reuse Speedup	-	2298.51 -	1169.33 3.93	714.12 6.43	519.91 8.84
DUCT	f90 -O3 Speedup	-	549.74 -	295.78 3.72	184.12 5.97	137.49 8.00

When using the optimisation level -O3 a significant performance increase has been obtained. On the other hand the time required for the communication between the processors becomes more dominant, so that the speedup is decreasing with higher optimisation.

As the timings obtained by the kernel without using the REUSE clause indicate, the schedule reuse concept is one of the most important features for a good performance or the kernel. Figure 1.1 shows the parameter statistics of a 16 processor result obtained by Medea when using the REUSE clause, while in Figure 1.2 the same statistics for a run without the REUSE clause is shown. It is obvious that the number of communication events and the time used for communication is an order of magnitude larger when the REUSE clause is not used.

rohr_hpf_16p_4iter.IndepLoops
# of components: 21

	t_exec	t_comm	n_exec	n_comm
CGCCG.266	2.710010	0.858062	119.000000	261.500000
CGCCG.266_work	0.000266	0.000000	1.000000	0.000000
CGCCG.266_inspector	0.653622	0.038310	1.000000	23.500000
CGCCG.266_gather	1.332943	0.819752	119.000000	238.000000
CGCCG.266_executor	0.723178	0.000000	119.000000	0.000000
CGCCG.178	2.667481	0.833459	119.000000	261.500000
CGCCG.178_work	0.000274	0.000000	1.000000	0.000000
CGCCG.178_inspector	0.655157	0.042702	1.000000	23.500000
CGCCG.178_gather	1.320364	0.790757	119.000000	238.000000
CGCCG.178_executor	0.691687	0.000000	119.000000	0.000000

Fig. 1.1: Medea parameter statistic: 16 Processor result of the HPF+ kernel with REUSE.

rohr_hpf_16p_4iter_noreuse.IndepLoops
# of components: 21

	t_exec	t_comm	n_exec	n_comm
CGCCG.178	101.880730	27.308081	119.000000	3034.500000
CGCCG.178_work	0.037475	0.000000	119.000000	0.000000
CGCCG.178_inspector	98.726486	25.467888	119.000000	2796.500000
CGCCG.178_gather	2.389459	1.840193	119.000000	238.000000
CGCCG.178_executor	0.727309	0.000000	119.000000	0.000000
CGCCG.266	101.855362	27.234257	119.000000	3034.500000
CGCCG.266_work	0.036662	0.000000	119.000000	0.000000
CGCCG.266_inspector	98.721291	25.457869	119.000000	2796.500000
CGCCG.266_gather	2.341098	1.776388	119.000000	238.000000
CGCCG.266_executor	0.756311	0.000000	119.000000	0.000000

Fig.1.2: Medea parameter statistic: 16 Processor result of the HPF+ kernel without REUSE clause.

The data distribution used does not take any geometrical information into account. The distribution depends strongly on the numbering of the cells and as a consequence disconnected subdomains will occur for block-structured grids. The amount of communication could be further by using different mesh partitioning methods. In such a case an indirect distribution of all arrays used in the kernel would be necessary.

The speedups for the larger dataset have been evaluated relative to the 2 processor measurements, since single processor results have not been available because of memory problems.

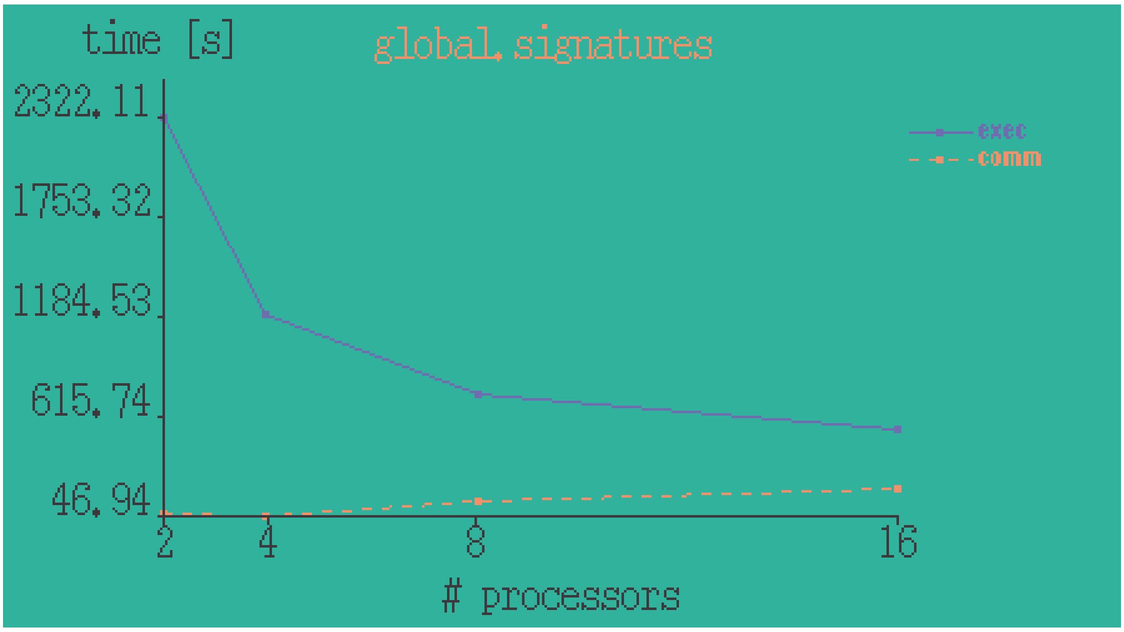


Fig.1.3: Medea signature statistic: Result of the HPF+ kernel.

The signatures obtained by Medea, shown in Figure 1.3, show that for the larger data-set the execution time is still dominant even for the 16 processor case.

Performance of the MPI kernel

For the MPI kernel the same input data-sets have been used as for the HPF+ version. Since the MPI kernel is coded in F77 the SUN F77 and the SUN F90 compiler have been used for the compilation of the code.

Dataset	Compiler	1p	2p	4p	8p	16p
PIPE	F77 Speedup	97.14	50.70 1.92	27.53 3.53	17.99 5.40	14.08 6.90
	F77 -O3 Speedup	47.27	24.22 1.95	13.39 3.53	9.29 5.09	8.76 5.39
	f90 -O3 Speedup	55.58	28.76 1.93	16.17 3.44	10.53 5.28	9.61 5.78
DUCT	f90 -O3 Speedup	209.74	105.65 1.98	53.11 3.95	31.65 6.63	24.22 8.66

The executable obtained by the F77 compiler was in average about 20% faster compared to the executable of the F90 compiler. Speedups obtained for the larger data-set (DUCT) are significantly better then for the smaller data-set, which is due to the fact that in case of the PIPE dataset on 16 processor only 640 cells are computed on each node, so that communication becomes the dominant part.

Time measurements for the MPI kernel have also been done on an IBM SP system:

Dataset	Compiler	1p	2p	4p	8p
DUCT	xlf	10.22	7.04	5.86	9.49
	Speedup		1.45	1.74	1.08

The results show that the problem size is already too small for the high single processor performance of the IBM system, so that nearly no speedup is obtained.

1.4 Discussion

With the current version of the VFC compiler very complex loops with highly irregular data access can be handled. The language features supplied would be sufficient for a port of the full FIRE code to HPF+.

The performance of the MPI kernel is significantly better compared to the HPF+ version. This seems to be due to the fact that schedule reuse is still expensive, although performance has increased significantly compared to previous VFC versions. On the other hand the porting effort of the HPF+ version is much smaller compared to the message passing version, where the quite complex calculation of the communication lists and the local data exchange has to be coded by hand.

Medea is a very helpful tool for the analysis of the communication behaviour of the different kernels. The profiles, metrics and signatures obtained give a very good insight into the communication behaviour of the program. Program parts with high communication costs can be easily identified and appropriate optimisations strategies can be worked out. Because of the automatic instrumentation facility with the VFC compiler the the trace-file generation is very comfortable.

2 ECMWF

2.1 Introduction

In this section of the report, three final kernels from ECMWF (TS, LG and SL) are evaluated. These kernels represent more complex features of the IFS weather forecast model. The IFS is briefly described in the following section. After this, each kernel is described separately, including 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. All kernels have been coded to use fortran 90 modules, rather than common blocks which were used in the version 2 kernels.

Each kernel is 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 that the HPF+ compiler (VFC) is a pre-processor that generates Fortran 90 source code, including calls to a HPF+ runtime library. For the MPI kernels, version MPICH-1.0.13 has been used. The release of VFC used for the evaluation of the HPF+ kernels is version 1.00. Finally, a summary of the main points found during the evaluation and future HPF+ requirements are presented in a 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.

2.1.2 HPF+ Kernel Issues

It is the representation of the data spaces used in the IFS that present a technical difficulty for HPF as they are all irregular, as shown in Figure 2.1. 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.

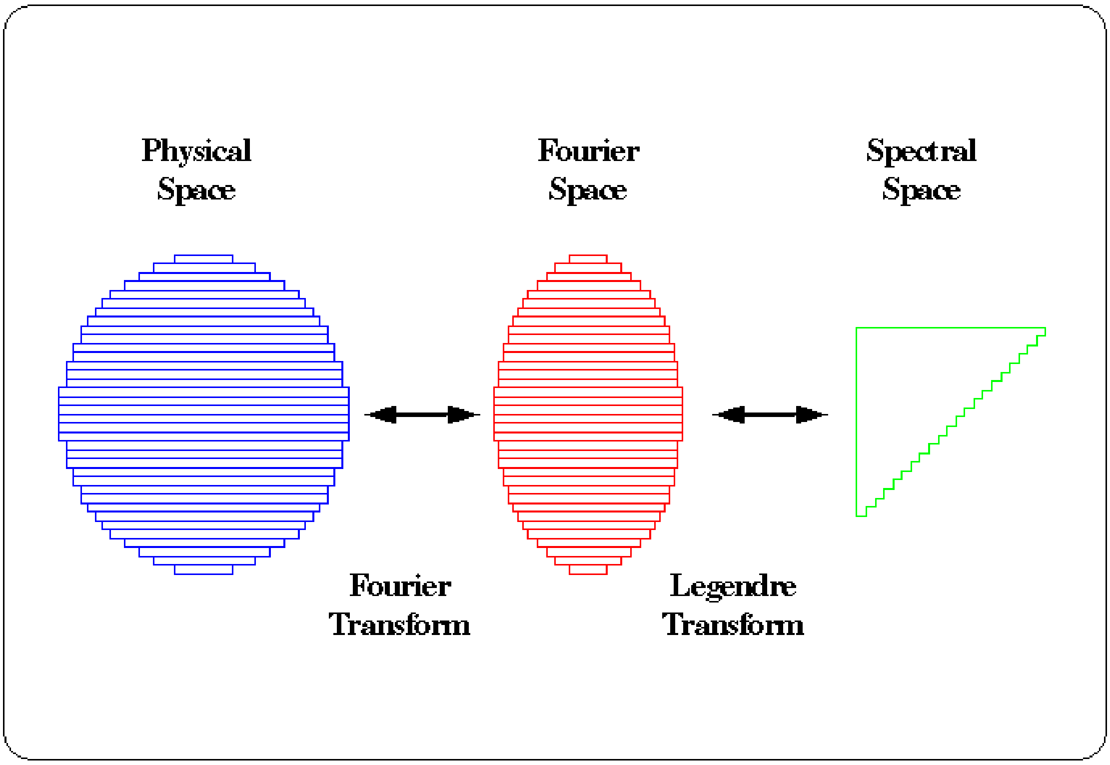


Fig. 2.1. Spectral transform method

Instead GEN_BLOCK and INDIRECT distributions are required as explained in the following sections.

As the computational routines of the IFS require no communication, they can take advantage of the efficient HPF_LOCAL extrinsic mechanism. Further, the complexity and number of these routines preclude an evaluation with an experimental HPF compiler such as VFC and for this reason are not part of this evaluation. Rather, the kernels explore the issues relating to the representation of the IFS data spaces and the transformation of these spaces in a forecast model timestep.

Finally, the kernels have all been coded using modules with allocatable data structures. This dynamic structure is exactly the same as that used in the production IFS and allows the same executable to run all forecast resolutions. For the HPF+ kernels this involves computing map information for GEN_BLOCK and INDIRECT distributions.

2.1.3 Grid point space

Grid point space (also known as physical 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 2.1.

IFS Spectral Resolution	Grid Points
T63	13280
T106	35718
T213	134028

Table 2.1: 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.

2.1.4 Fourier space

Fourier space is similar to grid point space in its structure, with the number of fourier coefficients for each latitude reducing towards the poles. At model resolution T213, each field has 213 (complex) 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. A field is a 2 dimensional model variable at the surface or at one of 31 levels in the atmosphere.
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 on all latitudes 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.

2.1.5 Spectral space

Spectral space is a 3 dimensional space in which 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 using GEN_BLOCK where for practical and communication efficiency reasons the zonal wave numbers (m) would be distributed as in the legendre transforms.

2.2 TS kernel

The Time-Step (TS) kernel is by far the most complex of the three ECMWF kernels. It uses fourier and legendre transforms to transform data between grid-point space and spectral space. In a single time-step data is transformed from grid-point space to spectral space and back to grid-point space.

Two HPF+ variants of the TS kernel have been coded, with the fourier and legendre transforms being performed by using INDEPENDENT loops (variant 1) or by using the extrinsic HPF_LOCAL mechanism (variant 2).

Unfortunately, VFC 1.00 was unable to parallelise the critical INDEPENDENT loops of variant 1 due to a compiler restriction that bounds of inner loops must be invariant, that is, they cannot be dependent on outer loops. No practical workaround was found to bypass this restriction in this kernel.

On the other hand, the second variant using the HPF_LOCAL extrinsic mechanism encountered no such compiler restrictions and is described further in the next section.

2.2.1 HPF+ features used

Fourier and spectral spaces were distributed using GEN_BLOCK distribution. This allowed the fourier and legendre transforms to execute efficiently using the HPF_LOCAL extrinsic mechanism and without any interprocessor communication. While grid-point space would ideally be distributed with an INDIRECT distribution for a general two dimensional partitioning strategy, a one dimensional approach was used to simplify the complexity of these transforms. For this reason grid-point space could be distributed with a simple BLOCK distribution.
The distribution of the legendre polynomials (computed during initialisation) represented a difficulty for HPF, as they both needed to be distributed (because of their size) and to have different distributions for the initial computation (over latitudes) and in the legendre transforms (over waves). This was achieved by using an initial BLOCK distribution for the polynomials over latitudes (all waves) and then redistributing over waves (all latitudes). This was accomplished by the specification of the DYNAMIC attribute on the initial BLOCK distribution, and then by using a REDISTRIBUTE directive to perform the redistribution. This is much easier than the equivalent message passing routine (sutrle.F), which takes 150 lines to do what is achieved by a single HPF directive!

2.2.2 Porting Effort

The HPF+ kernel was developed by using experience gained from the version 2 LG kernel and adding the infrastructure to support real (rather than simulated) fourier and legendre transforms. Although, the coding effort for the HPF+ version was significantly less than for the message passing version, testing uncovered a few VFC bugs (all quickly resolved by the Vienna compiler team) and the occasional application bug (of my making).

2.2.3 Other HPF compilers

An attempt was made to port the HPF kernel (variant 2) to the pghpf (version 2.2) compiler on the QSW CS-2. It was found that this compiler did not support GEN_BLOCK distributions. Furthermore, as there was no straightforward way to modify the kernel to get it to compile and run correctly, the port was abandoned. This is a serious problem when using HPF compilers. If you have a Fortran 90 code, you know (or are at least 99% certain) that it will compile and run correctly whatever F90 compiler you use. The same cannot be said for HPF. Are users expected to analyse which features HPF compiler suppliers have decided to implement?
For this reason HPF programs today are not portable. If on the other hand, programmers restricted themselves to a common subset of HPF language features, then this would not be sufficient for programs with irregular data structures.

2.2.4 Message Passing kernel

The message passing version of the TS kernel was developed by enhancing the version 2 LG MPI kernel to include fourier and legendre transforms. This was achieved by including routines and code sections from the production IFS and adding a driver routine (stepo.F) which successively called routines to perform real to spectral (reespe.F) and spectral to real transforms (speree.F). Correct execution was confirmed by comparing fields that had been transformed from real to spectral and back again with the original fields, and further by printing out the mean of the spectral fields on each time step.

2.2.5 Performance

Problem	Version		1p	4p	16p
T63	SERIAL	ms	(1p) 10,857		
		s/u	(1p) (1.00)		
	MPI	ms	(1p) 8,797	2,097	584
		s/u	1.27 (1.00)	5.28 (4.20)	18.59(15.06)
	HPF+variant 2	ms	15,259	4,449	1,274
		s/u	0.71(1.00)	2.44 (3.43)	8.82(11.98)
T106	SERIAL	ms	34,953		
		s/u	1.00		
	MPI	ms	29,574	7,399	1,953
		s/u	1.18(1.00)	4.72 (4.00)	17.90(15.14)
	HPF+variant 2	ms	not enough	14,740	3,898
		s/u	memory	2.37 (1.00)	8.97(3.78)
T213	SERIAL	ms	206,797		
		s/u	1.00		
	MPI	ms	not enough	52,051	12,349
		s/u	memory	3.97(1.00)	16.75(4.21)
	HPF+variant 2	ms	not enough	not enough	not enough
		s/u	memory	memory	memory

Table 2.2: Comparison of run times (in millisecs per time step) and speedups for the versions of the TS kernel for different numbers of processors on QSW CS2

Performance is summarised in Table 2.2 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 HPF+ kernel (variant 2). In this table, timing is presented in millisecs (m/s) per time-step, and speed-up (s/u) is the performance attained relative to the serial version of the kernel. Where speedup is shown in brackets, this is relative to the same code version.

Figure 2.2 shows graphically the performance of the message-passing and HPF+ kernels at T63 resolution. As can be seen the MPI kernel scales very well, achieving ideal speedups relative the the serial version. Further, the single processor version of the MPI kernel is 27% faster than the original serial version. This is due to the legendre transforms (direct and inverse) running about twice as fast. This, in turn, is thought to be due to different compiler optimisation of the respective source files. Besides this there is evidence of some super-linear speedup, which is most probably a caching effect on the QSW CS-2. In comparison, the HPF+ kernel scales moderately well, achieving a speedup of 8 to 9 on 16 processors. For LTOM and MTOL (see source files) transpositions, these INDEPENDENT loops run 4 times slower, while the GTOL and LTOG transpositions run 7 and 11 times slower respectively.

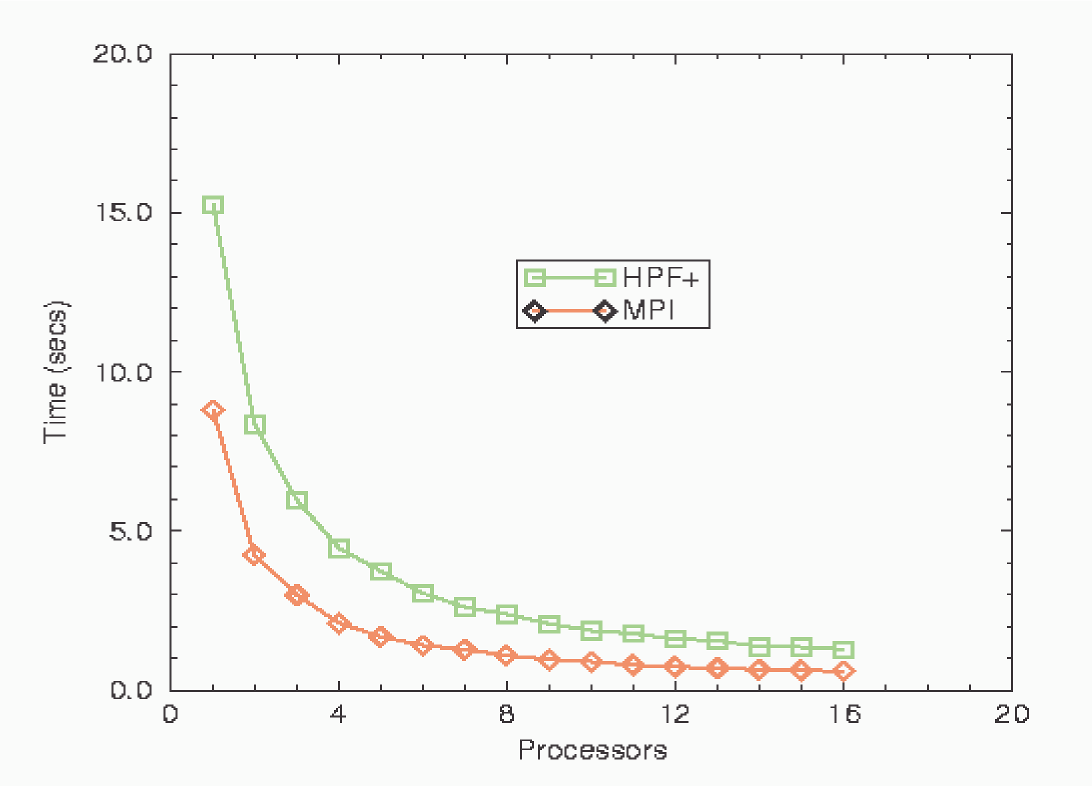


Figure 2.2: TS kernel performance

This is no surprise, as HPF overheads will always be greater than message-passing overheads, especially for simple loops where communication is required. However, if we were to include the large amount of code in the grid-point, fourier and spectral space computations (as used in the IFS forecast model) in a HPF_LOCAL context then the message-passing and HPF+ performances would get much closer.

2.3 LG kernel

The LG kernel is provided to address the IFS program issues of 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 memory required by the particular FFT package used at ECMWF.
- E-W data dependencies of Fast Fourier Transforms (SIMFFT).
- Vertical dependencies of physics calculations (SIMPHYS).
- Blocking of grid-point space to conserve memory usage in physics calculations and exploit locality of reference.

This kernel differs from the version 2 LG kernel in that it uses modules rather than common blocks for containing its data structures. The serial version of the GTOL transposition (kernel fragment) is coded as:

```
REAL ZGA(NPROMA,NGT0,NBLOCKS)
REAL ZGL(NPROMAG,NGT0)
REAL ZTMP(NGPTOTG)

DO JFLD=1,NGT0
    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
```

In the above kernel fragment the blocking of ZGA (by NPROMA) is required to conserve space in physics routines where some automatic arrays are dimensioned (NPROMA, NFLEV, NFLEV) - where NPROMA is the block size (or vector length) and NFLEV 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 can therefore be coded as:

```
!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
        REAL ZGA(NGPTOTG,NGT0)

!HPF$   INDEPENDENT,NEW(JFLD),ONHOME(ZGL(INDL(J),:)),REUSE(LREUSE)
        DO J=1,NGPTOTG
           DO JFLD=1,NGT0
                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.

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,*)
!HPF$&            ONTO PROCSAB :: ZGL3
        REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
!HPF$   DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK)
!HPF$&            ONTO PROCSAB :: ZGL2
        REAL ZGL2(NPROMAG,NG2D0)
!HPF$   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,JVAR),ONHOME(ZGL3(INDL(J),:,:)),
!HPF$& REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JLEV=1,NFLEVG
            DO JVAR=1,NG3D0
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$   INDEPENDENT,NEW(JVAR),ONHOME(ZGL2(INDL(J),:)),
!HPF$& REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JVAR=1,NG2D0
            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.

2.3.1 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 transpositions 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 transpositions 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 a PUREST subroutine called within a 3 deep nested INDEPENDENT loop
- GTOL and LTOG transpositions coded as 3 and 2 deep nested INDEPENDENT loops
- use of REUSE on INDEPENDENT loops

2.3.2 Porting Effort

The LG kernel was relatively straightforward to code in HPF+ and certainly easier than the equivalent message passing version. HPF_LOCAL provided 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.

The only problem area encountered in the HPF+ version was for the SIMFFT loop in variant 3. Here VFC complained that arrays ZGL3 and ZGL2 could not be both used and defined. The workaround was to copy these arrays to temporaries ZGL3X and ZGL2X ahead of the independent loop, and uses these to avoid the conflict. Another workaround was used for the DO J=1,NLOENG(JGL) loop which VFC objected to as JGL is the control variable for the outer loop. The workaround here was to loop over the maximum latitude length (NDLON) and use an IF statement inside the loop to restrict the range of the iterations. Variant 3 finally compiled without any restrictions or warning messages, however, it failed in execution even on a single processor with a VFC runtime library diagnostic "PARS_GEN_BLOCK_OWNER : owner of 1820 not found". This diagnostic was issued while executing in the SIMFFT loop nest for 3D variables.

2.3.3 Other HPF compilers

The LG kernel was coded for the NASL Release 2.1 compiler by N.A. Software Ltd., (thanks to Gabriel Howard and Mike Delves) and reported separately in a HPF+ working paper dated 5th March 1998. Significant progress has been made in the 2.1 release over the earlier 2.0 release.

2.3.4 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.

2.3.5 Performance

Problem	Version		1p	4p	16p
T63	SERIAL	ms	15,888		
		s/u	1.00		
	MPI	ms	15,649	3,911	1,006
		s/u	1.02 (1.00)	4.06 (4.00)	15.79 (15.56)
	HPF+variant 1	ms	16,933	4,282	1,120
		s/u	0.94 (1.00)	3.71 (3.95)	14.19 (15.12)
	HPF+variant 2	ms	20,843	4,885	1,344
		s/u	0.76 (1.00)	3.25 (4.27)	11.82 (15.51)
T106	SERIAL	ms	42,645		
		s/u	1.00		
	MPI	ms	42,554	10,608	2,673
		s/u	1.00 (1.00)	4.02 (4.01)	15.95 (15.92)
	HPF+variant 1	ms	45,881	11,515	2,926
		s/u	0.93 (1.00)	3.70 (3.98)	14.57 (15.68)
	HPF+variant 2	ms	not enough	13,132	3,340
		s/u	memory	3.25 (1.00)	12.77 (3.93)
T213	SERIAL	ms	164,642		
		s/u	1.00		
	MPI	ms	not enough	40,951	10,327
		s/u	memory	4.02 (1.00)	15.94 (3.97)
	HPF+variant 1	Ms	not enough	44,470	11,233
		s/u	memory	3.70 (1.00)	14.66 (3.96)
	HPF+variant 2	Ms	not enough	not enough	13,357
		s/u	memory	memory	12.33 (1.00)

Table 2.3: Comparison of run times (in millisecs per time step) and speedups for the versions of the LG kernel for different numbers of processors on a QSW CS2

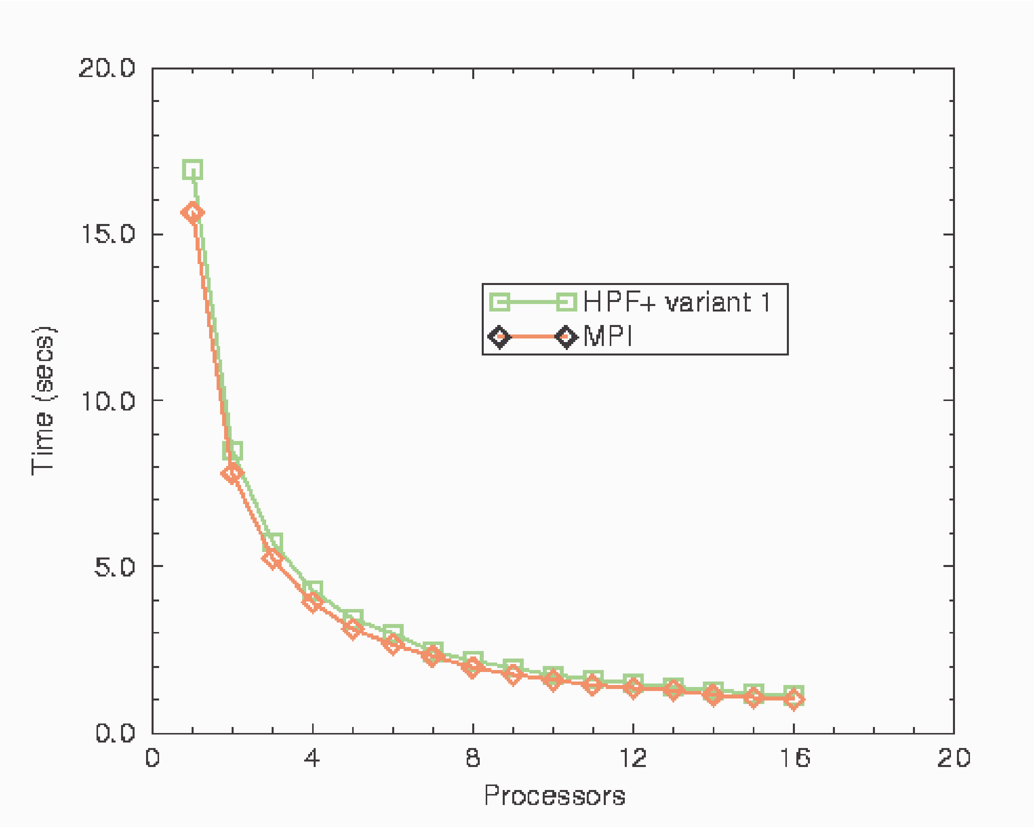


Figure 2.3: LG kernel performance

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. As before, performance is summarised in Table 2.3 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 millisecs (m/s) per time-step and speed-up (s/u) is the performance attained relative to the serial version of the kernel. Where speedup is shown in brackets, this is relative to the same code version.

Figure 2.3 shows graphically the performance of the message-passing and HPF+ (variant 1) kernels at T63 resolution. The performance of the message-passing (MPI) kernel shows near perfect scaling on up to 16 processors with a speed-up of over 15. The HPF+ kernel variants also perform well against the MPI version, with variant 1 showing a speedup of over 14 and variant 2 a speedup of about 12 on 16 processors. This represents a 11 to 17% improvement (on 16 processors) for variant 1 when compared to the equivalent version 2 kernel.

As the variant 2 kernel ran slower than variant 1, it is fairly safe to conclude that the approach of using a simpler BLOCK distribution for fields and having separate arrays for 3D and 2D variables was not a good idea. It is clearly better to store 3D and 2D variables in the same data structure and distribute the fields dimension using an INDIRECT distribution.

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 message-passing 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 2.4. A halo mechanism is required, as the computation of the trajectory for each grid-point is an iterative process where only 2 or 3 iterations are performed due to the cost.
For the final (version 3) evaluation, the SL kernel now takes advantage of the HALO capability provided by the VFC 1.00 compiler.

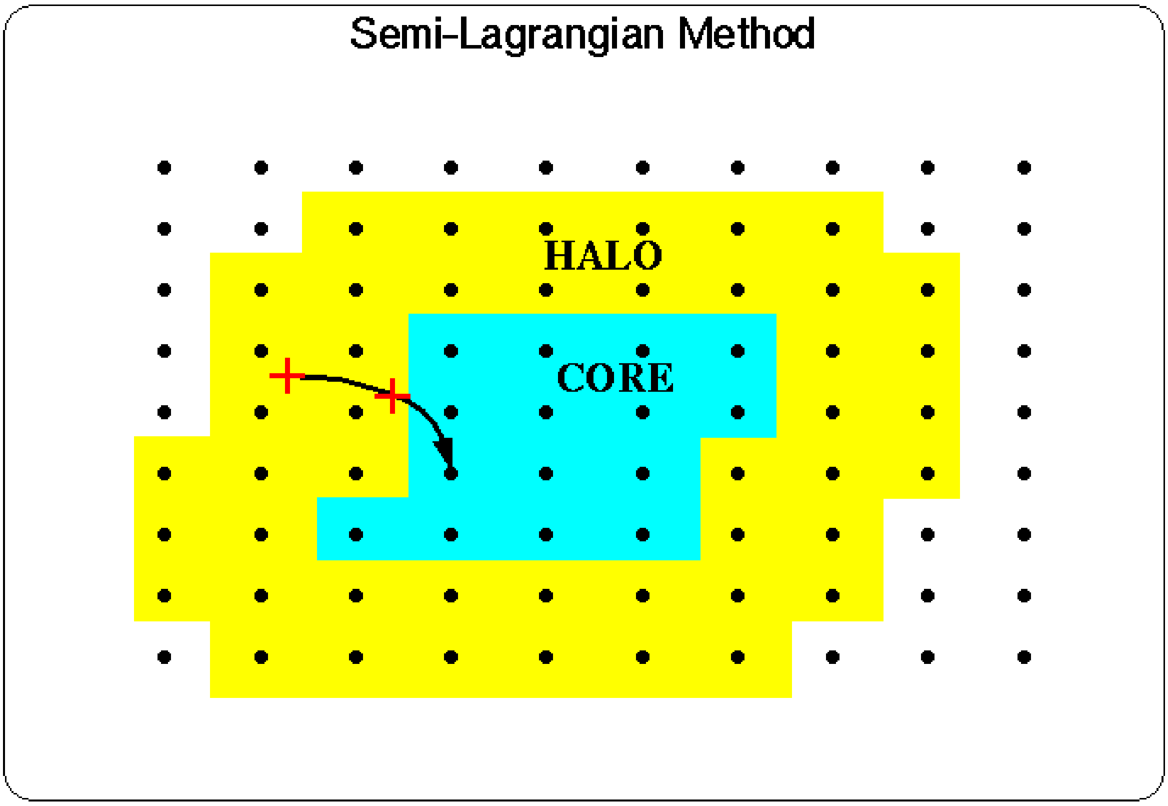


Figure 2.4: Semi Lagrangian method, showing computation of trajectory, requiring interpolation of grid-point data held on neighbouring processors (halo area)

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. Also a UPDATE_HALO directive is used to indicate when 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.

The above HALO features were not available during the version 2 evaluation of the SL kernel, and therefore enables a direct comparison of the performance benefits of these features. It should be noted that the HPF-2 SHADOW capability 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 approximated in the HPF+ kernel by a simple average of neighbouring grid-points, as shown in the HPF+ extract below:

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

2.4.3 Other HPF compilers

The HPF+ kernel was modified to run with the pghpf compiler (release 2.2). As this release did not support INDIRECT distributions (it is basically a HPF 1.1 compiler), the distribution for grid-point space was changed to BLOCK. Other changes were made to remove the halo module and associated references and allocations of data structures. This was necessary as the compiler complained that the use of POINTER in a derived type was not supported. The pghpf compiler also required HPF distribute directives to follow the declaration of the array being distributed, which was unnecessary with VFC.

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 computing 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

Problem	Version		1p	4p	16p
T63	SERIAL	ms s/u	1,036.8 1.00		
	MPI	ms s/u	1,103.8 0.94 (1.00)	276.4 3.75 (3.99)	104.0 9.97 (10.61)
	HPF+	ms s/u	14,806 0.07 (1.00)	3,857 0.27 (3.84)	1,255 0.83 (11.80)
	PGHPF	ms s/u	40,260 0.03 (1.00)	12,057 0.09 (3.34)	4,163 0.25 (9.67)
T106	SERIAL	ms s/u	3,944.9 (1.0)		
	MPI	ms s/u	4,297.1 0.92 (1.00)	1,138.2 3.47 (3.78)	341.3 11.56 (12.59)
	HPF+	ms s/u	not enough memory	16,232 0.24 (1.00)	5,160 0.76 (3.15)
T213	SERIAL	ms s/u	16,410.8 1.00		
	MPI	ms s/u	not enough memory	4,308.2 3.81 (1.00)	1,187.6 13.82 (3.63)
	HPF+	ms s/u	not enough memory	not enough memory	18,354 0.89 (1.00)

Table 2.4:Comparison of run times (in millisecs per time step) and speedup for the versions of the SL kernel for different numbers of processors on a QSW CS2

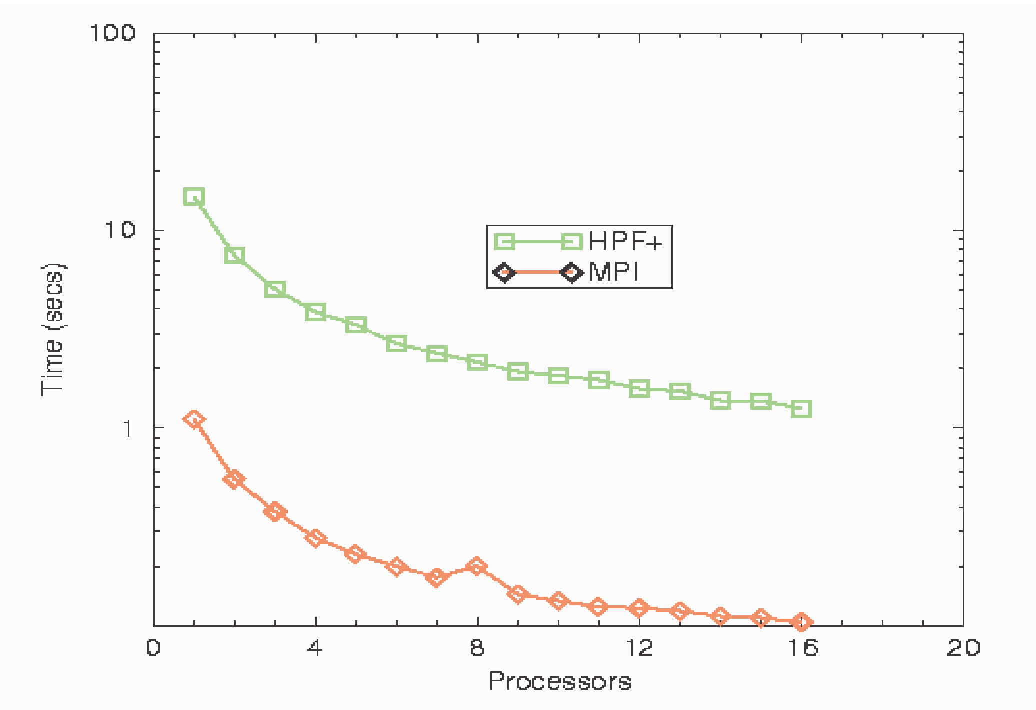


Figure 2.5.: SL kernel performance

The performance of the SL kernel is summarised in Table 2.4. As with the other ECMWF kernels, runs were performed on 1, 4 and 16 processors.

Figure 2.5 shows graphically the performance of the message-passing and HPF+ kernels at T63 resolution. The message passing version showed scaling of 10 to 14 on 16 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.

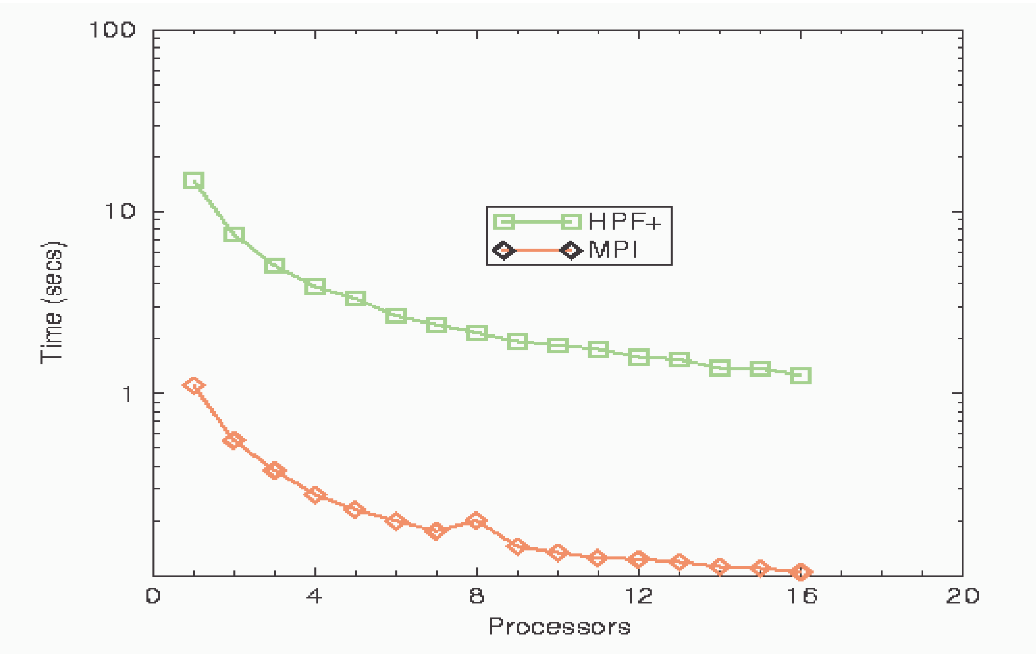
The HPF+ version 3 kernel has improved by a factor of 6 to 7 times over the version 2 kernel. This is attributed to the use of the HALO capability in HPF+. In comparison the pghpf (v2.2) kernel ran about 3 times slower than the HPF+ version.

While the VFC improvement represents significant progress, the MPI version is still many times faster. This is not of great concern as the real SL interpolation routines used in the IFS have a computational intensity at least an order of magnitude greater than the much simplified SL kernel.

2.5 Discussion

The VFC compiler has successfully demonstrated that the HPF language can be extended to provide the functionality and performance required for the IFS forecast model. In particular, the HPF+ language extensions for communication schedule reuse and halo support have been shown to be both easy to use and provide acceptable performance. Schedule reuse has been used in the TS and LG kernel, and has typically resulted in performance improvements of 5 to 6 times for the associated INDEPENDENT loops. Whereas the HALO capability has been used successfully in the SL kernel.

It is also clear that the HPF-2 GEN_BLOCK & INDIRECT distributions and HPF_LOCAL extrinsic mechanism are equally important for the IFS. These features are classed as approved extensions in the HPF-2 standard, which means that it is left up to the compiler developer to decide whether to incorporate them. This presents a problem for application developers, as they have to decide whether to tailor their programs for each HPF compiler, or use a common reduced subset of the language.



Hopefully these issues will be resolved as HPF compiler developers implement more of the present HPF-2 standard, and the HPF language is further enhanced by features such as those implemented in HPF+.

An important focus of the HPF+ project has been the support of nested INDEPENDENT loops in the context of multi-dimensional distributions of arrays. This has been important for all the IFS kernels, and much progress has been made to remove compiler restrictions as these kernels have evolved. Some restrictions still exist, however they can often be programmed around. This is probably the most annoying feature of programming in HPF, and much time can be spent trying to get INDEPENDENT loops to parallelise successfully. This should be compared with programming for message passing where the paradigm imposes little restrictions on Fortran 90 (communication of types being an obvious one). The counter arguement of course is that coding message passing library calls and being responsible for data partitions are equally tiresome.

It should be noted that VFC is implemented as a pre-processor that accepts HPF/HPF+ programs and generates Fortran 90 source code to be compiled by a Fortran 90 compiler. Included in the VFC generated code are calls to a VFC runtime library which are resolved at link time. This approach should be contrasted with message-passing programs which are simply compiled by a Fortran 90 compiler and linked with your favourite message-passing library, such as MPI. The VFC approach clearly has some disadvantages. The generated code is typically very difficult to understand due to its size and sometimes results in extended compilation times. An example of this can be seen with subroutine stepo3.f90 in the LG kernel, where serial compilation (f90 -O3) takes 23 secs on the QSW CS-2 for the 300 source lines. VFC compiles the same source in under 2 seconds, but generates 1400 source lines which takes 880 secs to compile (f90 -O3), a factor of 38 times longer. Fortunately very few routines are like this, but we don't need many of them for compilation time to be a serious problem.

Another problem occurs when we want to debug HPF+ programs. It is not possible today to debug the original HPF+ source. That leaves the voluminous VFC generated source which is unreadable by mere mortals. What do we do to resolve this? We use good old-fashioned print statements in the original source. This works in the global context of HPF programs, but requires writing to separate units in the HPF_LOCAL context.

Parallel I/O has not been addressed in the HPF+ project and equally in HPF standardisation efforts. This is a serious disadvantage for HPF, as the MPI-2 standard now supports parallel I/O which has been incorporated successfully into the production IFS. A possible solution for HPF may be to use this same interface within a HPF_LOCAL context.

Issues such as these would need to be resolved before ECMWF could consider an IFS port to HPF, or writing new applications in HPF. Nevertheless, the HPF+ project has made a big step towards such a port being practical.

3 ESI

3.1 Introduction

This section of the report discusses the HPF port of the Rel.3 benchmarks of the PAMCRASH code within the Esprit Project HPF+. The different versions of the kernel are a part of the PAM-CRASH solver, which can be applied to unstructured meshes. The aim is to have the most representative features needed for a complete HPF benchmark with a greater expressiveness, 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 V96 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 the load-balance 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 modules used in the kernels is given, followed by a description of different HPF+ kernels. The report also gives a more detailed comparison between the HPF+ kernels using new directives implemented recently in the HPF+ project. Moreover, the message-passing version of these kernels with the self-impact algorithm was improved to obtain a better scalability on multiple processors. Nevertheless, at the time of writing, no comparison could be done with the HPF+ version of this kernel. In the last part, performance comparisons between compilers and kernels on different platforms is shown to study the advanced HPF+ features included in the last revision of the kernels.

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 part of the computational cost of the algorithm comes from the calculation of the internal forces at the nodal points. These force calculations can be broken down into two parts:

- stress-strain calculations (First set)
- contact-impact calculations (Second set)

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

3.2 HPF+ kernel

The final release of HPF+ benchmarks was broken in different sets. The first set 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. The detailed description of the HPF+ kernel is extensively explained in the previous reports. The last release of the HPF+ kernel based on this subset computes the stress-strain and is updated to take in account the new HPF+ directives developed during the last months of this project. Moreover, The kernels were totally rewritten to use the expressiveness of the F90 (e. g: Array syntax). This new syntax should assist the compiler analysis to generate a more efficient code.

In contrast to the stress-strain calculation, the contact-impact algorithm used in the second set of benchmarks has a more complicated data structure. The high level of indirection arises as a consequence of the need to loop over subsets of nodes and elements coupled via the connectivity and inverse-connectivity lists. Moreover, these nodal lists are not mutually exclusive. Besides, the peculiarity of the self-impact algorithm selected is that slaves node belong to the master segments. The data-dependencies could be resumed in two parts: we need to gather coordinates of slave nodes and master segments defined by user at run-time. Then, we construct the list of nodes that determines which master node is closest to the slave node. In the sequential code, the access to this list is made by arrays of pointers. 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 and MPI kernel use dynamically defined lists and pointers to construct buckets 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 more features than the second version was written to be compliant with some compiler restrictions. Nevertheless, some big improvements were done in the VFC to handle all complex parallel loops found in the contact kernels.

In the meantime, some separate versions of this kernel of increasing levels of complexity w.r.t HPF language features including or not the contact routines with different options were produced. Different programming strategies combined with the new HPF+ directives were analyzed to match with the consecutive releases of the VFC.

3.2.1 HPF+ features used

The HPF+ features used in the HPF kernels are the GEN_BLOCK/BLOCK directives to distribute nodes and elements. Contrarily to the first version, all mapped arrays are dynamically allocated. Some versions with the DYNAMIC directive were also evaluated. 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. The ALIGN directive is now used and correctly handled by the VFC. One variant using the GEN_BLOCK distribute and a weight function for elements was writing to study the possibility to include different materials which could introduce load-imbalance during the computation. Another variant was written with the INDIRECT directive with a MAP array computed at run-time and successfully executed on the CS-2. Nevertheless, the current set of benchmarks does not have the adequate features to really use this interesting feature.

But, the most important feature used is the HALO directive. The HALO data structure was written to return an integer array of the non-local nodes per processor. The use of this Halo computed at run-time during the initialization will alleviate the storage allocation for non-local data and will reduce the preprocessing overhead generated by the inspector during the time-stepping phase. It is important to underline that thanks to the ALIGN directive, all perfectly aligned arrays are able to use the same HALO. Other directives used are described in the D1.1b report.

Independent loops

Most loops benefit by the use of array syntax. In addition, some HPF directives were removed as a results of improvements in compiler analysis. Nevertheless, there are still some constraints that prevent optimized generated code when some parallel loops use array sections. (e. g , the FORALL is missing for mapped arrays).

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. The HPF+ kernels also contains the PUREST directive to allow the 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 carry out tests with commercial compilers.

The REUSE clause is recognised as a key capability of HPF+ as it eliminates the need for an inspector phase on subsequent time-steps. In addition, the RESIDENT/ON HOME clauses were added to some INDEPENDENT loops to help the compiler to generate a more efficient code. Further, as a result of a significant improvement of the VFC capabilities, nearly all independent loops used in the contact routines were handled (e g: Mapped arrays referenced in the bounds of loops and within the body of conditional statements, subscript expressions of distributed arrays that reference at most one loop variable: x(1, nsv(imnod (i,j))).

Nevertheless, at the time of writing, the HPF+ implementation of the bucket search was not possible because of too complex statements in independent loops and the heavy use of array pointers. By example, the bounds-expressions of loops that are nested within INDEPENDENT loop reference outer loop variables that are distributed arrays. As a consequence, the strategy for collection of references in the inspector fails. Consequently, the bucket sort included in the final contact kernel have been put commented-out for performance measurements. Another strategy using non-mapped arrays to compute the bounds was written, but the compiler failed because of restrictions about non-mapped arrays in independent loops.

A simpler form using non-mapped arrays to compute the bounds was written, but not yet carry out. Nevertheless, all other constraints discussing in the previous reports due to the inspector/executor strategy were removed.

Dummy arguments

INHERIT/RANGE directives have been used in all subroutines and interfaces were generated. All routines used in the contact kernel were rewritten to use the assumed-shape array syntax and reshaping was avoided by clean constructs.

3.2.2 Porting Effort

As outlined in previous reports, the initial storage management strategy was implemented by using a 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. Thus, the main work was spent in:

* Cleaning data structure:
    - 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-sized arrays were translated to assumed-shape arrays in all subroutines
    - All mapped arrays are dynamically allocated
    - Common statements were removed
    - Interfaces with HPF directives were generated for all subroutines
    - Complete reorganization of the code, especially for the contact subroutines.

Contact routines were rewritten with F90 array syntax and some different constructs were used to store contact pairs. Some independent loops seem too complicated with actual HPF compilers, because of the high level of indirection, the non-invariant bounds, the goto statements,… . This implies sometimes to rethink the algorithm to have a HPF compliant code. Consequently, some features were temporary removed (CF: bucket sort, dynamic lists).

On the other hand, the interest from the code developer is that the data distribution and the indirect data access can be easily handled in HPF+ by using distribute directives and the INDEPENDENT statement before the parallel loops. The resulting code is very clean and easier to maintain. The parallelization effort of the message passing is much more important. A lot of additional code has been developed. The programmer has to write the dynamic send/receive lists coupled with a renumbering phase to avoid a large amount of communications.

3.3 Other HPF Compilers

Before comparing HPF+ with other HPF compilers it should be noted that the parallelisation strategy used in the HPF+ kernels has been strongly influenced by available features of the VFC during the second year of this project. Besides, the new features are not available in the current commercial compilers, and as no practical alternative existed without significant rewriting, the port was abandoned. Nevertheless, some new performance results on a slightly modified version the HPF+ kernels were made with the latest release of the PGI compiler on the T3E machine.

3.4 Message Passing kernel

The hand-written message-passing kernel is based on the second revision of the HPF+ kernel used for the time-measurements on the CS-2 machine. This implementation was written with the MPI library and use the host/node paradigm based on the parallel version of the PAM-CRASH code.

Further, a more scalable algorithm was included in the MPI contact-kernel to improve the overall performance. However, parallel contact-impact algorithms are not the topic of this project and no more investigation was done. But, we have to underline the effort to write such kernel in message-passing. The construction of dynamic mailing lists and the management of such data-structure in parallel on distributed platforms lead to a very complex coding and it is difficult to maintain. Besides, the parallel algorithm is based on the bucket sort, which is itself based on a nodal distribution. Therefore, a the time of writing, it was not possible to remove this feature to make some absolute performance comparisons with the HPF+ contact kernel.

3.5 Performance

3.5.1 Data-sets

The four specific input files generated in the second part were reused to make comparisons with previous kernels and to see the new improvements. 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 benchmark with the different data-sets were made principally on the CS-2.

Characteristics of data-sets without contacts:

	Small	Large
Number of nodes	715	32760
Number of elements	640	32400
Number of CYCLE	600	250

Data-sets with 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]

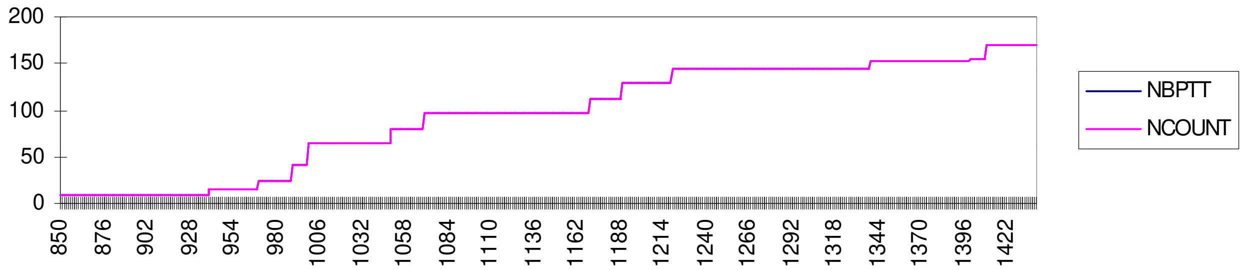


Figure 3.1: Number of contact pairs during the execution for the medium data-set

3.5.2 HPF+/F90 kernels with the contact algorithm

The last revision of HPF+ kernels with the self-contact and the medium data-set compiled without VFC:

Compiler	DEC F95	SUN F90	XLF90
Machine	DEC 4100	CS-2	IBM
User time(s)	1366.7	4404.0	812.0
Options used	-O5 –fast	-O3	-O3
Number of CP	170	188	168

On all platforms, the contact starts at 850 with 8 contact pairs.

3.5.3 HPF+ kernel V3.0

All HPF+ kernels without the contact routines with the different distribute directives and new HPF+ features were successfully translated in F90 with the latest release of VFC. Besides, a first release of the contact kernel without the bucket sort was also generated thanks to a significant improvement of VFC capabilities and a rewrite of some independent loops.
The first subset of kernels is derived from the HPF+ kernels developed during the second year of this project. It is divided in four versions:

Only the most representative kernels are pointed out:
K2: Kernel used for the D1.2b report (static arrays + reuse+ variants for PGI)
K3: Improved Kernel to test some intermediate versions of the VFC.
K4: Kernels V3.0 using the DYNAMIC/GEN_BLOCK/HALO directives
K4.1: GEN_BLOCK with a weight function to evaluate the load-imbalance.
K4.2: GEN_BLOCK + INDIRECT directive for array of nodes.
K4.4: Kernel using the HALO and some others advanced features found in HPF+. The K4.41 variant uses the ALIGN directive for perfectly aligned arrays. In this case, all the aligned arrays are able to use the same HALO.

The second subset of kernels is derived from the HPF+ kernels developed during the second year of this project that contains the contact routines. All the K kernels were successfully compiled with the VFC and successfully executed on the CS-2 up to 16 processors. Unfortunately, the port of these kernels to the CENJU-3 was abandoned because of run-time errors and segmentation violation.

C1: sequential F77 kernel only used to make comparison (D1.2b) between different compilers. Beside, the MPI kernel is based on this kernel. In another part, the HPF+ variants of this kernel were recoded in F90 to be HPF and VFC compliant. The data-structure has changed. The dynamic coupled lists were replaced with a 2-dim array. The number of contact nodes per impacted segment was limited to a constant and some minor features were removed. All performance measurements were done until 16 processors (maximum number of processor per partition on the CS-2).

Small data-set on the CS-2 (600 cycles):

MEIKO CS-2	VFC 0.9	VFC1.0	VFC1.0	VFC1.1
NPROCS	K2 (D1.2b)	K4.4	K4.41(ALIGN)	K4.4
1	797,2	114,4	114,7	(1)
2	437,9	57,8	58,1	1872,9
4	255,4	34,0	34,4	696,5
8	193,0	32,2	31,0	410,1
16	184,4	38,9	39,6	210,6

Large data-set on the CS-2 (250 cycles):

NPROCS	K2 (D1.2b)	MPI kernel	K3	K4.4	K4.41
1 (sun f90)	1300,0	F77 -O5	VFC 1.0	VFC 1.0	VFC 1.0
1 (apf90 V4)	402,0				
1 (vfc+f90)	(1)	480,8	(1)	(1)	(1)
2	2810,0	260,2	1509.6	2075.3	2206.6
4	1135,7	131,3	1161.5	736.8	746.8
8	672,2	118,2	462.7	416.6	465.1
16	370,3	28,4	241.9	222.0	289.6
20	293,5	25,4			
30	221,4	16,5			
32	210,3				

(1) : Not Enough memory.

We notice some big improvements with the last release of VFC thanks to a more efficient generated code and the new HPF+ directives. Nevertheless, because of internal problems with the used platform, it is not possible to confirm the accuracy of performance measurements. Therefore, we cannot explain why the combination of aligned arrays and the HALO is more slower.

The following kernels were essentially developed to evaluate the cost of the different distribute directives. In the case where the previous strategies were not sufficient to handle a more complete benchmark (.e g: load-balancing problems). For a production code, the combination of GEN_BLOCK+DYNAMIC+ALIGN+INHERIT+REDISTRIBUTE have to be used to because of possible load-imbalance during the computation.Unfortunately, the time-measurements were made with a previous version of VFC on a very unreliable machine.

Large data-set on the CS-2 with VFC 0.93:

NPROCS	K3.1	K4.2	K4.3	K4.1
		GEN_B+IND	DYNA+ALIGN	INHERIT+weight func
1	MEM	Dview*	MEM	MEM
2	1466	3389	1466	4602
4		2381		1604
8	866	1518	1773	
12		1380	Dview*	
16	687		Dview*	1638

*dview: not enough space in buffer

3.5.4 HPF+ kernels with MEDEA

MEDEA V.2.2 was installed and evaluated on a Silicon Graphics machine. Moreover, trace-files were successful generated for the small data-set.(-mcils and –miclsb options were used) up to 16 processors with the V3.0 kernel. This kernel includes the REUSE clause and the HALO directive. The number of cycles has been reduced to 6 because of time constraints and the size of trace files.

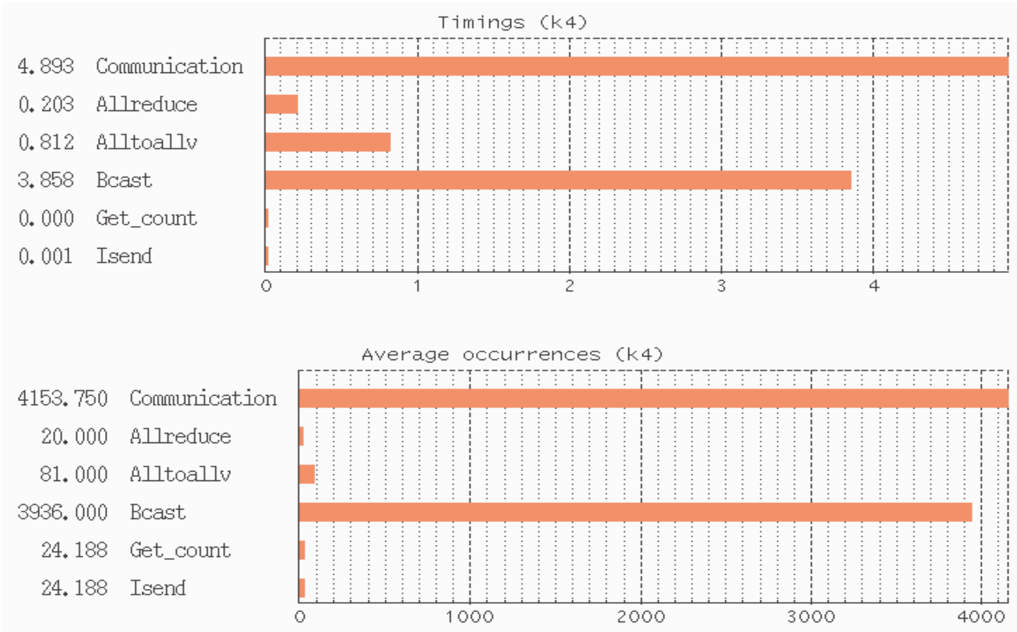
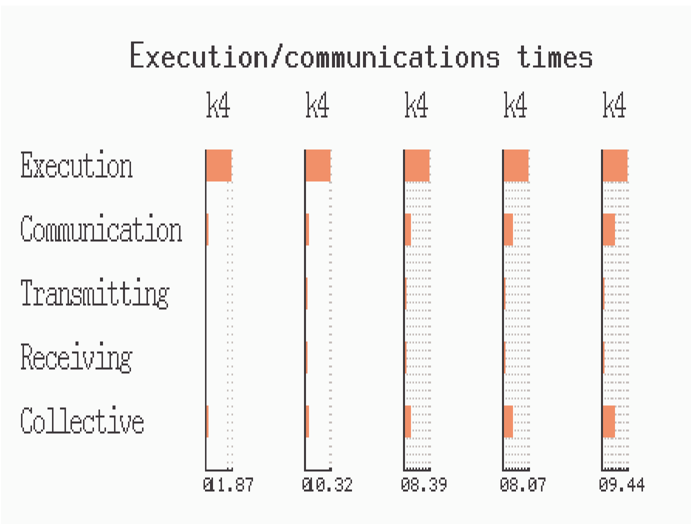
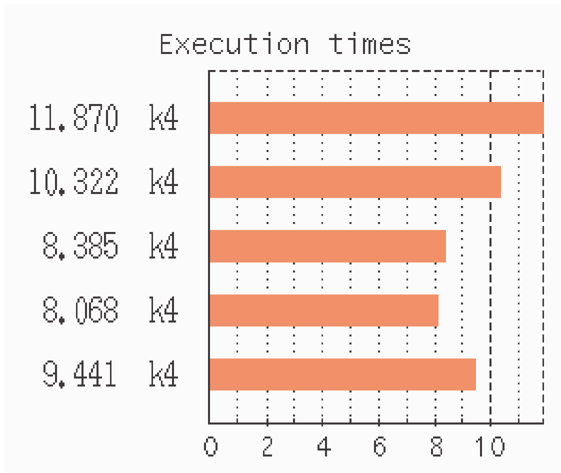


Figure 3.2: Detailed communication times from 1 to 16 processors (small data-set)





Figures 3.3-3.4: Execution times from 1 to 16 processors (small data-set)

global.IndepLoops
\# of components: 16

	t_exec	t_comm	n_exec	n_comm
k4_PC_K4.705	1.721004	0.079733	6.000000	62.000000
k4_PC_K4.705_work	0.006164	0.000000	1.000000	0.000000
k4_PC_K4.705_inspector	1.200772	0.048714	1.000000	14.000000
k4_PC_K4.705_gather	0.029943	0.006646	6.000000	18.000000
k4_PC_K4.705_executor	0.419705	0.000000	6.000000	0.000000
k4_PC_K4.705_scatter	0.064420	0.024374	6.000000	30.000000

Figure 3.5: Total execution time in the Independent loops

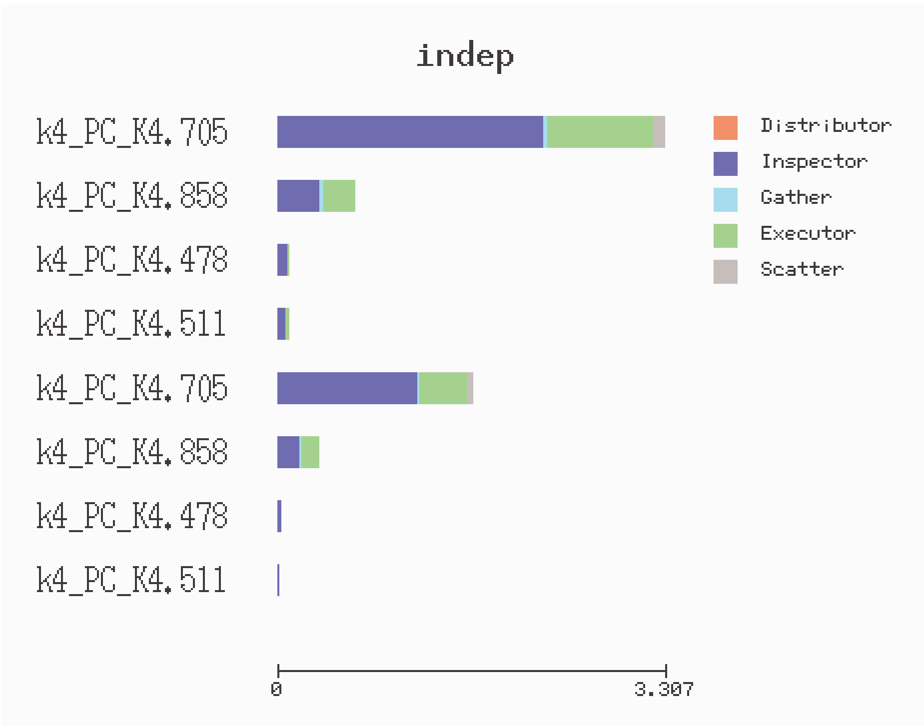


Figure 3.6: Execution time in the Independent loops

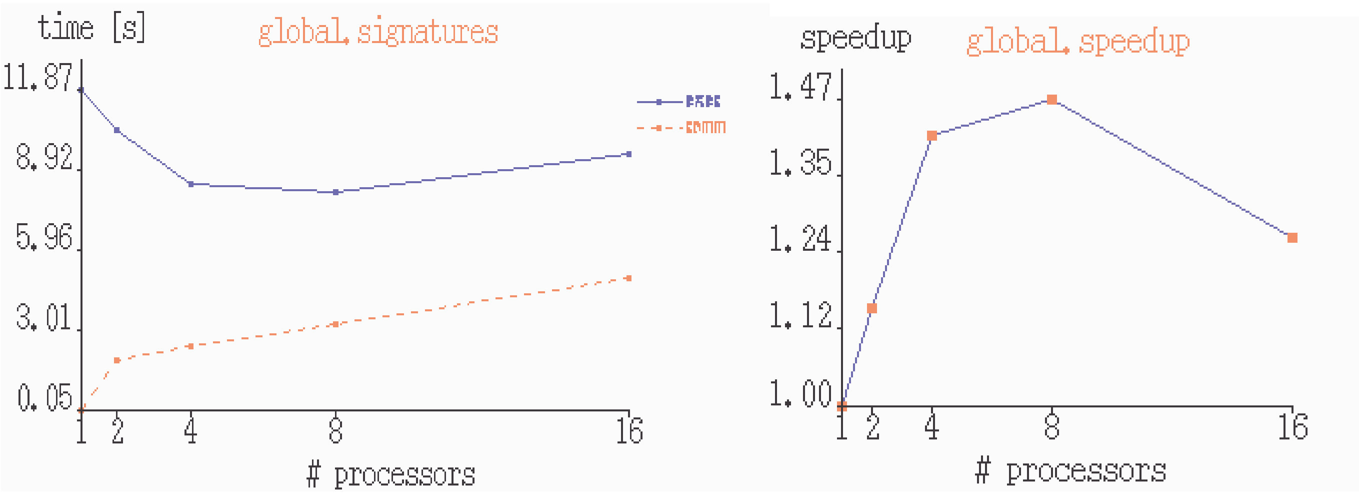


Figure 3.7: Signature & speed-up from 1 to 16 processors

Nproc	K4.4 (-mcils)	K4.4 (without the SIS)
1	4.33	3.79
2	2.33	1.92
4	1.22	1.08
8	0.83	0.86
16	1.21	1.08

Figure 3.8: Total Time with VFC 1.0 + -mcils + SIS 0.8 (small data-set 6 cycles)

3.5.5 PGI compiler on CRAY T3E

The PGI version of the third release of the HPF+ kernel uses the same F90 routines that the HPF+ kernel developed in collaboration with the University of Vienna. 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, The analysis of the generated code has indicated that all loops are parallelized. But, in spite of a good scalability until 128 processors (compared to 1 processor), the CPU over-head is always very high (native CRAY compiler: 172.02 s with CF90, -03). Thus, the PGI code is always 7 more slowly on 1 processor. Tests were done to be sure that no schedule reuse was used. Separate compilation and indexes arrays with the INOUT attribute were used in the K2.4 kernel. Independent loops use the NEW and the REDUCTION clause. Besides, the prescriptive mapping was used in the subroutines.

Compiler	PGI R2.3			
NPROCS	Elapse time(s)		-Msmp(shmem)	MEIKO CS-2
	K2.0(D1.2b)	K2.2(October 97)	K2.4	K2.0 (PGI R2.2)
1	2227,18	2213,05	**1207,60**	7391
2	1315,66	1296,77	**647,10**	7616
4	666,30	655,39	**328,39**	7447
8	340,39	336,56	**168,58**	7484
16	177,25	176,52	**88,97**	
32	94,49	93,77	**48,37**	
64	57,94	54,47	**29,54**	
128	37,26	37,29	**27,09**	

In the previous report, timing data shows that the gather/scather operations with both compilers are a major drawback in performance. It was no possible to remove the SUM_SCATTER with the PGI compiler. But, some tests were done with the DEC compiler with and without this intrinsic library:

Compiler	F95 V5.0	DEC ALPHA (EV5 300 Mhz)	XLF90 IBM
1 proc	SUM_SCATTER	F77 loop	F77 loop
Times(s)	860.549	384.92	834.84

3.5.6 Performance of MPI kernels

The time-measurements of the message-passing version were performed with the MPI implementation on the CS-2, the CENJU-3 and the 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 elements, 250 cycles). The time-measurements are completed by the partition name on the CS-2, because of different behaviors due to the hardware. The number of cycles was reduced to 250 cycles on the CENJU-3 and the T3E machine.

MPI kernel compiled in FORTRAN 77 without the contact algorithm:

	CS-2	250 cycles	
NPROCS	time(s)	Partition	Speed-Up
1	409.2	P16b	1
2	204.1	P16b	2
4	106.2	P16b	4
8	49.2	P16b	7,5
16	22.8	P32a	16,7

3.5.7 HPF+ kernels with contact-impact routines

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-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 strongly 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. Characteristics of these kernels are assumed shape arrays, static arrays and array syntax (a more detailed description can be found in the deliverable D1.1b). All loops were rewritten to be compliant with the HPF syntax and some new arrays have to be added. Nevertheless, at the time of writing, the HALO directive was not included in the contact kernel. Below, we show some preliminary results on the CS-2 with the BLOCK distribute directive and without the bucket sort. The number of cycles have reduced because of time constraints. Therefore, tests were made until 850 cycles. The aim was to know if the generated code provide good results. Until now, the HPF+ contact kernel finds the contact at 850 with 8 contact pairs. This is very promising, because some directives could be more judiciously added to increase the code performance thanks to the help of the compiler developers.

Contact kernel on the CS-2 machine (time measurements up to 850 cycles):

CS-2	VFC1.0	VFC1.1	VFC1.1
NPROCS	Time(s)	Time(s)	Speed-Up
F90 Seq	2607		1
1		17375.1	2.08
2		8322.1	3.94
4	4343	4401.1	7.65
8		2269.0	13.9
16		1043.4	16.65
32		802.6	21.6

The complete run of the contact kernel with 16 processors on the CS-2 gives 3391 seconds. (1440 cycles, 210 contact pairs). The number of contact pairs found is the same that for the MPI kernel. Unfortunately, up to 10 processors, the amount of memory used by the generated code to apply the contact forces was too important to have a complete run. The CPU overhead generated by the HPF compiler is still inacceptable. The super-linear speed-up seems to show that this is due to some cache effects and surely to memory swap. But, at the time of writing, it was not possible to evaluate exactly how much memory is used. But, in spite of a good scalability, it is not possible to compare the absolute times with the message-passing kernels (the bucket sort was not activated). Nevertheless, it could be possible to increase the absolute performance and reduce the memory requirements of the contact kernel whether some restrictions are removed:

- Constraints about outer loops of independent loops (in this case, it is very easy to handle the bucket sort)

- Multiple halos for the same array in combination with a directive to select the update of halos
- Add/modify the schedule variables computed during the inspector phase to be able to reuse them for different loops.

3.5.8 MPI kernels with contact-impact routines

The MPI kernel R2.0 have the same loops in the self-impact routine that the F77 C1 kernel. Therefore, the native optimized constructs were used. Performance results were measured with the gettimeofday() function on different partitions. Some fluctuations until 30-50 % were observed with different partitions. Besides, in MPI the bucket sort was rewritten to obtain a better scalability on parallel platforms. Moreover, it is written in F77 contrarily to the HPF kernels. At the time of writing, no comparisons could be done with the HPF+ kernels. The bucket sort is the major drawback. In fact, the MPI kernel is based on the distribution of slave nodes and the parallel bucket sort. The HPF kernel is more based on the distribution of the slave segments than the distribution of slave nodes. Therefore, there is any sense to remove this because this involves of changing completely the parallel strategy. Nevertheless, we can also underline the load-imbalance due to different number of contact pairs per processor which reduces the scalability of this code. The redistribution of data is a complex task to do for a code developer using the explicit message-passing. Thus, this gives a true advantage to the HPF code thanks to the redistribute directives that could make this work automatically.

PROCS	Native F77 time(s) CS-2	Time(s) CS-2	Speed -Up	Time(s) CENJU-3	Speed -Up	NBPTT : processor Id (number of contact pair)
	R1.0 (D1.2b)	R2.1	R2.1	R2.1		R2.0
1	547,81	468,9	1	492,6	1	168
2	296,11	274,5?	1,7	243,1	2,0	P1(168)
4	171	132,04	3,5	131,3	3,7	P4(168)
8	117,8	73,8	6,3	69,5	7,0	P8(168)
12	102,2	60,0	7,8	56,0	8,8	P5(18), P6(4), P11(8),P12(160) = 190
16	100,3	53,2(15p)	8,8	48,2	10,2	P7(16),P8(15),P9(2),P15(27) ,P16(141) = 201

(BOX with :6560 nodes and 6400 elements, 1440 cycles)

3.6 Discussion

In this report, we have described the evolution of the PAM-CRASH kernel within the HPF+ project. After investigating some problematic parts in the self-impact routines, the contact kernels were restructured for better analysis by the VFC. In the second step, we have adapted some different subroutines to use the HPF+ directives. The third version of the kernels now written in F90 and including the last HPF+ features has been carried out with the VFC compiler on the CS-2 machine thanks to the close collaboration with the compiler developers. The new VFC HPF+ compiler is a significant improvement over the previous releases. In terms of usability, it provides good diagnostics, an easy to use command line interface and handle the separate compilation of subroutines. Besides, significant improvements to handle more complex independent loops were made.

As for HPF+ language features, the most significant for the ESI kernels are the implementation of communication schedule reuse, generalized block distribution and the HALO for the specification of irregular overlaps. The benefit of these features was seen in the analysis of performance of different kernels. Nevertheless, the cost of independent loops is still expensive and it is always due to the fact that the generated code use indirection arrays. It is not always possible to avoid the array-syntax in the independent loops. Thus, a more powerful mechanism to handle regular loops could improve considerably the performance of the VFC.

Moreover, the complete hand-coded version with a parallelized version of the self-impact algorithm was written in MPI, improved and evaluated on different parallel platforms. Unfortunately, at the time of writing, it was not possible to compare the HPF+ contact kernel with the message-passing version. Trace files were successfully generated with the HPF+ kernels V3.0 and experiments were made with MEDEA.

Some performance measures were made to evaluate the impact of the HPF+ directives developed during this project. Results show that the overheads generated by the inspector phase are well amortized over a complete simulation. The HALO which generalizes the shadow zone concept is very powerful and can handle any kind of data-dependencies. Nevertheless, intrinsic functions to help the application developer should be available to find more easily the underlying data-distribution policy. The contact-impact kernels need many more improvements from the HPF compilers to handle more efficiently all the independent loops. This project has shown that it was possible. Finally, it seems evident that the HPF language requires further development with features such as schedule reuse and halo. Thus, to be accepted more widely in industry, HPF compilers must not only to conform to the full standards but also to be improved in term of efficiency and memory requirements.

4 NEC

4.1 Introduction

The generic crash kernels developed within the project have served to permit an accelerated introduction to the problems to be found in crashworthiness codes like PAM-CRASH and have supported the evaluation of language features. Since the generic kernels were designed to refelect the general algorithmic approaches within PAM-CRASH, while incorporating simplifications in data structures, it is not surprising that their HPF requirements are very close to those of the PAM-CRASH benchmark codes from ESI.

The various codes included in the generic kernel sets developed within the project have been described in some detail in the deliverables D1.1a, D1.1b and also (in terms of modifications to fit to available compiler options) in the previous evaluation deliverable, D1.2b. Rather than repeating previously provided information, reference is made to these complimentary reports. In the sections below, only indications of necessary code modifications for particular compilers will be given.

The emphasis of the final evaluation included below is on the use of user-defined halo functions and the achievement of HPF+-performance for the most demanding contact-impact kernels. The use of halo-functions was highlighted in the previous evaluation as a possible way to approach the high performance of explicit message-passing code. Some of the results from the previous evaluation deliverable for the K2-set are repeated again for ease of reference.

4.2 HPF+ Kernel

The kernels used for the final evaluation are taken from the K2- & CK1-Kernel sets. The descriptions of the kernels and implementation options for HPF+ features can be found in the deliverables D1.1a, D1.1b and D1.2b.

Although the CK1-kernels include all routines of the K2 kernels, the evaluation has been performed with both sets. Maintaining use of the K2-set allows a higher number of iterations to be performed which more clearly demonstrates the REUSE facility, since only static communication schedules are exhibited by these kernels. Tests with the contact kernels were performed with a reduced number of iterations (only 30) because of the non-physical simulation performed (see D1.1b). The focus of interest is in the ability to efficiently handle algorithms with dynamically changing computational and communication requirements.

4.2.1 HPF+ features used

Communication schedule reuse using the REUSE directive for INDEPENDENT loops is the central feature affecting the performance of all the kernels included in the evaluation. For the implementation of the HALO facility within VFC, which allows the user to specify a halo in combination with DISTRIBUTION - or in our cases REDISTRIBUTION - directives, REUSE remains necessary. This is despite the fact that a previously specified UPDATE_HALO command allows loops to be specified as RESIDENT, which would seem to imply that communication is in fact unnecessary.

4.2.2 Porting Effort

Although the porting effort in terms of code modification is extremely low in comparison with message-passing implementations, this fact alone would tend to give a somewhat distorted picture. The main problem lies in the lack of portability between HPF compilers. This is reflected within the HPF+ developments by the need to interact strongly with the VFC team in order to take account of restrictions in the implementations of various releases. When coming to commercial compilers where this interaction may not be possible, the user is faced with legitimate code which may not be accepted by a particular compiler or may lead to terrible performance. Code analysis then becomes extremely difficult - a way out is offered by the MEDEA tool, but this would of course require the instrumentation possibilty with the other compilers for full effect.

Otherwise, the move to halo functions will be seen to provide performance advantages, but does require almost a message-passing background to enable efficient exploitation and could easily lead to erroneous programs.

However, to return to the comparison with the hand-coded MPI kernels: the construction of local lists of contact segments and data structures to enable minimised volume & number of dynamic communications requires a great deal of effort and rather complicated coding. The HPF+ kernels retain a far greater clarity and much less code intrusion due to parallelisation.

4.2.3 Other HPF Compilers

The choice of alternative HPF compilers chosen within the final evaluation was very much related to the machine availability. The reduction of accessible processors on the QSW CS2, meant that the NEC Cenju-3 was more suitable for scalability testing. On that platform, the research compiler from GMD is publically available. In this final report, we restrict our attention to the strictly directive options offered by the Adaptor tool - a TRACE option similar in effect to REUSE. For the K2HPF_a kernel, this was the ADP1 option of deliverable D1.2b. The version using hand-code run-time system calls has not been included - the previous investigation aimed to point ahead to the benefits of halo use and such an option is included now automatically within VFC. Since Adaptor is also available on the IBM SP2 at GMD, this provides a mechanism for an indirect comparison of achieved performance with the PGI compiler, whose release (2.2.1) on the GMD's SP2 is more recent than that on the CS2.

Modifications to the kernels for the PGI and Adaptor compilers were discussed in deliverables D1.1b and D1.2b. The main feature in the CK1HPF_a implementations which differs strongly from that in the HPF+ implementation was a data redistribution at the subroutine boundary, with the coordinate array required for searching being replicated within the contact routine.

4.2.4 Message-passing Kernel

Hand-coded MPI versions have been produced for all kernel sets and their major features have been described in the previous deliverables. The kernel to be used in the results below, CK1MPI_a, uses the results of the search cube phase to construct dynamic mailing lists. The non-blocking communications are then performed only between processors which have to exchange data. This will minimise latency costs, as no unnecessary communication costs are involved. An optimisation of contact force communication volume was not implemented.

4.3 Performance

4.3.1 K2-Kernels

The major feature to be tested in the final evaluation with the K2-set is the use of HALO distributions and corresponding UPDATE. The test case taken was, as in the previous evaluation reports, the 4 X 40 X 160 rectangular box (resembling the box-beam test case from ESI). 1000 iterations were performed. Tables 4.2-4.4 show the achieved performance with the K2-set on the following three parallel platforms: QSW CS2 at VCPC, NEC Cenju-3 at GMD, IBM SP2 at GMD. All K2HPF+ kernels were compiled with the VFC compiler - the VFC version numbers are given in brackets, togeher with compiler flags when results for various options are presented). Compiler options used were as in Table 4.1. Other abbreviations used in the tables are:

ADP1 - GMD research compiler Adaptor using trace options (similar to REUSE) as in Deliverable D1.2b

PGI - HPF-1 compiler from Portland Group

Platform	Compiler	Compile flags	Version/Release
QSW CS2	F77	-O3	4.2
	VFC	-s -b # (& -h)(&-d)	0.91, 0.92, 1.0
	MPIF77	-O3	mpich
IBM SP2	GMDHPF	-O3 –qstrict	5.0
	PGI	-O3 –qstrict	2.2.1
NEC Cenju-3	GMDHPF	-Kmips2 –O	5.0
	MPIF77	-Kmips2 –O	mpich
	VFC (F90 back-end)	-Chopt –qstatic –Kmips2	0.92, 1.1

Table 4.1: Compiler options

Number of Procs		1	2	4	8	16	32	64
K2SEQ (F77)		497						
K2F90 (F90)		1005						
K2HF+_a	(0.91)	4351	2094	1050	563	298	187	174
K2HF+_a	(0.92)	1938	969	461	243	129	86	
K2HF+_a NO REUSE	(0.92)			9790	5995	2928	1709	
K2HPF+_a	(1.00)	1988	943	454	223	135		
K2HPF+_b	(1.00)	NEM	911	444	209	128		
K2HPF+_b NO RESIDENT	(1.00)	NEM	913	449	211	129		
K2MPI_a		497	246	126	68	38	24	22

Table 4.2: K2 results on QSW CS2 (NEM = Not enough memory)

No. Processors	1	2	4	8	16	32	64	128
K2HPF+_a (0.92)	1243	586	286	142	76	50	NEM	NEM
K2HPF_a (ADP1)	1264	588	267	137	65	39	25	22
K2HPF+_a (1.1)	1128	560	253	128	67	44.7	NEM	NEM
K2HPF+_b (1.1, -h -d)	NEM	911	445	207	104	63.4	62.5	61.5
K2HPF+_b (1.1, -h –b2)	1243	610	298	151	73.6	45.9	41.2	NEM
K2HPF+_b (1.1, -h)	1084	532	255	121	67.8	42.8	40.9	NEM
K2HPF+_b (1.1 ,-h) NO RESIDENT	1074	534	256	124	69.4	45.6	NEM	NEM
K2MPI_a	340	162	82	42	22	12	6,9	4,5

Table 4.3: K2 results on NEC Cenju-3 (NEM = Not enough memory)

No. Processors	1	2	4	8	16
K2HPF_a (ADP1)	665,8	344,0	182,7	97,9	57,8
K2HPF_a (PGI)	3120	1769	922	524	378
K2MPI_a	319,7	161,6	80,6	43,5	23,5

Table 4.4: K2 results on IBM SP2

Just to recall the nomenclature used in D1.1b: the kernel K2HPF+_a uses REUSE, the K2HPF+_b kernel uses REUSE, HALO and RESIDENT (and does require DYNAMIC & REDISTRIBUTE, though this is performed only once). Comparing the HPF and HPF+ results from Tables 4.2 and 4.3, it can be seen that the inclusion of REUSE and HALO options has lead to a major increase in performance and that there is an advantage in including the RESIDENT directive. In all cases, REUSE is absolutely necessary - HALO brings an additional gain, but not of the same magnitude. A larger gain would have been possible if multiple loops could have made use of the updated halo region - this was not the case with the K2-kernels. The relative performance with respect to commercial HPF-1 compilers can be inferred from the relative performances of the VFC, Adaptor and PGI compilers on the NEC Cenju-3 and IBM SP2: the availability of schedule reuse results in an improvement of around a factor of five.

However, several major worries remain:

- As in the previous evaluation, the single processor performance of the HPF code remains the dominant source of performance loss. Even in the best cases, a performance loss of approaching a factor four in comparison with the MPI kernel is exhibited. Such a performance loss is inacceptable for application developers. Analysing the results with MEDEA (as illustrated in Figure 4.1) shows that the executor phase - that is, the generated computational code - is highly inefficient. The major loss in performance is neither due to the inspector phase nor, for parallel execution, communication overheads.
- Through the use of temporary arrays included by the application developer or introduced by the compiler (in particular in conjunction with halo regions) the memory requirements are greatly increased - which precludes execution of larger problems on some platforms.

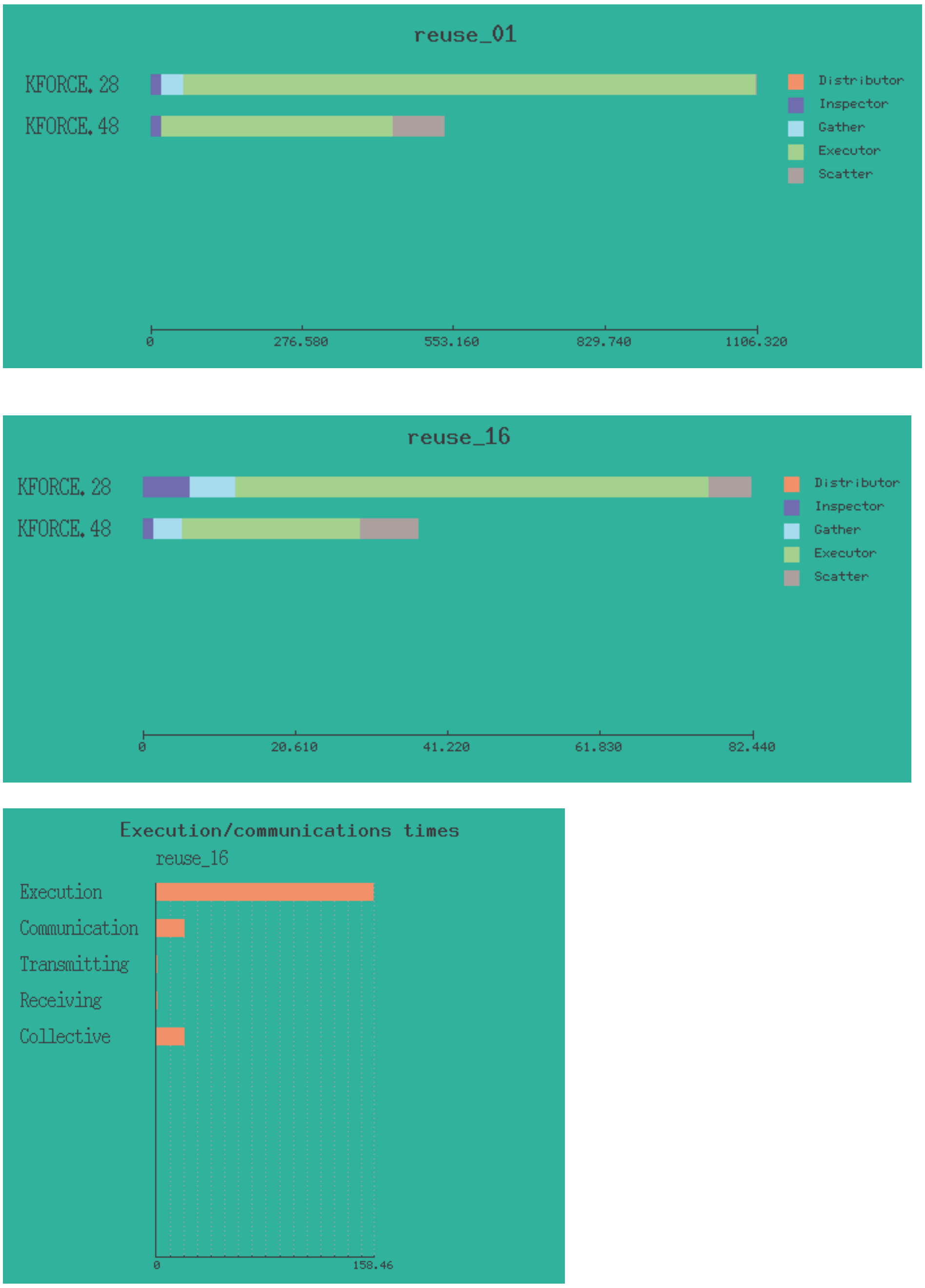


Figure 4.1: MEDEA Profiles of K2HPF+_a with 1 and 16 processors on the QSW CS2
reuse_01 - 1 processor, reuse_16 - 16 processors

4.3.2 CK1-Kernels

The computation within the "search box" phase (see D1.1b) is a non-sorted linear search, whose computational time dominates totally the remainder of the calculation. Thus, the timings for the search box calculations (which are performed once every 10 iterations) are subtracted from the results presented (referring to the kernels included in deliverable D1.1b - this means that all calculations within the RECALC IF block are omitted). The timings include the time-marching phase (routines KFORCE and UPDATE - as used within the K2-kernels) plus contact box penetration checking and correction calculations in CONTK1.

As discussed in deliverable D1.1b, the deformations produced by the modelled contact calculations are non-physical and a simulation with too many iterations would not be representative of the computational and communications requirements of actual contact-impact algorithms. Thus, only 30 iterations are performed. Two model sizes have been chosen for the investigations carried out. The results for the two box models, 4 X 40 X **80,** 4 X 40 X **160**, are given in Tables 4.5-4.9. Compiler options are as in Table 4.1. The HPF-1 results (kernel CK1HPF_a) have been produced on the NEC Cenju-3 and IBM SP2 with the GMD Adaptor and PGI compilers. The Adaptor version did not employ a communication schedule reuse -as included in the ADP1 kernel for the K2 set above. Current problems with the run-time system mean that full comparison with HPF+ kernel performance has not been possible.

No. Processors	1	2	4	8	16
CK1HPF+_a (1.00), -d NO REUSE	2630	131 1	690	361	127
CK1HPF+_a (1.00, -d)		541	270	146	72
CK1MPI_a	23.1	12.8	6.9	3.8	3.6

Table 4.5 CK1 results on the Meiko CS2, Problem size 4 X 40 X **80**

No. Processors	1	2	4	8	16	32
CK1SEQ	7.21					
CK1HPF_a (Adaptor)	160	52	34	26	24	25.5
CK1MPI_a	7.45	4.05	2.20	1.27	0.86	0.69

Table 4.6 CK1 results on the NEC Cenju-3, Problem size 4 X 40 X **80**

No. Processors	1	2	4	8	16	32
CK1SEQ	5.7					
CK1HPF_a (Adaptor)	94	57	39	28	20	19
CK1HPF_a (PGI)	101	58	37	27	30	33
CK1MPI_a	5.9	3.14	1.87	1.05	0.71	0.51

Table 4.7 CK1 results on the IBM SP2, Problem size 4 X 40 X **80**

No. Processors	1	2	4	8	16	32
CK1SEQ	15.08					
CK1HPF_a (Adaptor)	432	246	135	79	65	51
CK1MPI_a	15.13	8.23	4.51	2.46	1.51	0.92

Table 4.8 CK1 results on the NEC Cenju-3, Problem size 4 X 40 X **160**

No. Processors	1	2	4	8	16	32
CK1SEQ	12.29					
CK1HPF_a (Adaptor)	189	113	72	43	33	30
CK1HPF_a (PGI)	202	117	66	47	48	76
CK1MPI_a	12.5	6.58	3.71	2.15	1.21	0.83

Table 4.9 CK1 results on the IBM SP2, Problem size 4 X 40 X **160**

Although current problems with the VFC compiler mean that results are not available for the halo function based kernel (CK1HPF+_b) and only results on the QSW CS2 with the schedule reuse kernel (CK1HPF+_a) for the smaller test case, some conclusions may already be drawn. Schedule reuse is again demonstrated as a crucial component although the number of iterations over which the inspector phase for the contact box penetration checking is only 10. However, the relative performance with respect to the MPI code is extremely disappointing - whether or not an improvement over HPF-1 compilers has been achieved with the HPF+ extensions remains to be investigated. In any case, an order of magnitude performance degradation on a single CPU would prevent HPF exploitation for industrial codes.

The limited number of time-steps performed is not an explanation of the performance loss: as can been in the Medea analysis of independent loops shown in Figure 4.2. The dominant execution times are all within the contact-impact routine CONTK1, whise costs can only be amortised over 10 outer iterations. Since the search box times have been extracted from all results, attention may be focussed on the remaining CONTK1 loops. The following provides a link between the loop indentification used in the tracefiles (and shown in Figure 4.2) and the kernel source provided in deliverable D1.1b:

CONTK1 **.75** RECALC Phase (Not fully visible in Figure 4.2 due to Zoom)

. **.147** Contact Force Calculation -
1st loop, Fig. 10, D1.1b

.212 Contact Force Correction, Impacting Node -
2nd Loop, Fig. 10, D1.1b

.230 Contact Force Correction, Impaced Segment -
3rd Loop, Fig. 10, D1.1b

All of the latter three independent loops show a total dominance of generated overheads in comparison with the actual execution phase. For both of the reduction loops (212, 230) where schedule reuse is not at all possible, the inspector phase plays the major role. For the contact force calculation (loop 147) schedule reuse is effective in reducing the inspector costs. However, given that temporary arrays were introduced and aligned with the array of contact segments used within the ON_HOME directive, the occurrance of high scatter costs seems strange and will be investigated further. Nevertheless, the dominant costs were for the inspector within the reduction loop 230.

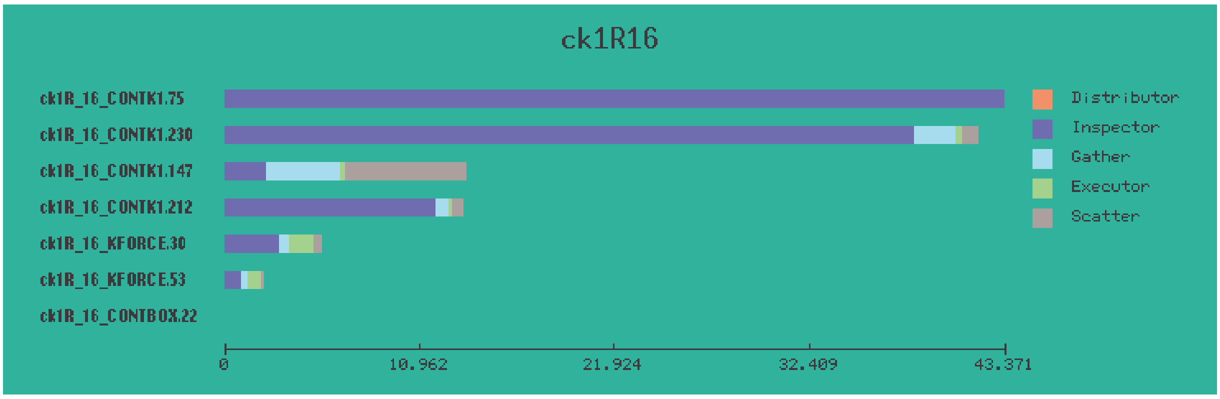


Figure 4.2: MEDEA Profiles of CK1HPF+_a with 16 processors on the QSW CS2

4.4 Discussion

The improvement of the VFC compilation system within this project showed that a large potential for optimization is still available in the HPF-language. The interaction of both new language concepts and their efficient realization within compiler and/or run time system could lead to competitive alternatives in the near future. Even though the current overhead in terms of elapsed times would not justify the use of HPF in performance critical applications today, the achieved performance with certain language constructs are very promising. In particular, the re-use of communication schedules for INDEPENDENT loops showed extremely good speed-ups. Furthermore, the possibility of defining so-called HALO regions is another feature that could dramatically influence the run time behaviour of a program under certain conditions. Both extensions of the HPF standard have proven their effectiveness within the generic crash-kernels although the advantages of the latter were not as high as had been hoped.
On the other hand, the HALO concept assumes a detailed knowledge of the underlying data-distribution policy of the compiler in order to calculate all non-local data elements of each processor by the programmer. Thus, a language feature like this is only suited for advanced programmers. The ideal solution would be to leave all this program analysis to the compiler. Unfortunately, the performance of commercial HPF-compilers (PGI) so far has shown that this goal is hard to achieve in the general case. High performance will only be achieved by an optimal interaction between user and compiler (-builders) as demonstrated within this project.

The final performance achieved with the contact-impact kernels is most disappointing. However, in the lifetime of the project it has been discovered that, in contrast to expectations and experiences with HPF compiler technology at project commencement, such complex algorithms may indeed be expressed in the language. The project developments raise expectations for generation of performant code in the future.

5 Acknowledgements

The authors would like to thank the other members of the HPF+ project for the many fruitful discussions on HPF and to VCPC for keeping the QSW CS-2 alive during the project evaluation periods. Special thanks are due to Siegfried Benkner, Viera Sipkova and other members of the VFC team in Vienna for making VFC a useable HPF compiler, and being responsive to the requirements of the application partners.

6 Appendix - Kernel Extracts

6.1 AVL HPF+ Kernel

```
C=======================================================================
      PROGRAM CALC_PS
C=======================================================================
C
C     SOLUTION OF PASSIVE SCALAR TRANSPORT EQUATION
C     H P F + VERSION
C
C=======================================================================
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C-----
C----------------------------------------------------------------------
C     READ MAP ARRAYS FROM FILE AND REDISTRIBUTE ARRAYS
C----------------------------------------------------------------------
C
C-----CONECTIVITY ARRAYS
      OPEN(20,STATUS='OLD',FILE='map1',FORM='FORMATTED')
      READ(20,*) map1
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP1)) ONTO NODES :: LCC, LCF
C
C-----COEFFICIENTS
      OPEN(20,STATUS='OLD',FILE='map2',FORM='FORMATTED')
      READ(20,*) MAP2
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP2)) ONTO NODES :: ACOEFF
C
C-----CELL FACE DATA
      OPEN(20,STATUS='OLD',FILE='map3',FORM='FORMATTED')
      READ(20,*) MAP3
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP3)) ONTO NODES :: XJAF11, XJAF21, XJAF31
!HPF$ REDISTRIBUTE(INDIRECT(MAP3)) ONTO NODES :: XJAF12, XJAF22, XJAF32
!HPF$ REDISTRIBUTE(INDIRECT(MAP3)) ONTO NODES :: XJAF13, XJAF23, XJAF33
C
C-----CELL CENTER DATA
      OPEN(20,STATUS='OLD',FILE='map4',FORM='FORMATTED')
      READ(20,*) MAP4
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP4)) ONTO NODES :: PS, U1, U2, U3
!HPF$ REDISTRIBUTE(INDIRECT(MAP4)) ONTO NODES :: DIREC1
!HPF$ REDISTRIBUTE(INDIRECT(MAP4)) ONTO NODES :: VOL
C
C-----BOUNDARY CELLS
      OPEN(20,STATUS='OLD',FILE='map5',FORM='FORMATTED')
      READ(20,*) MAP5
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP5)) ONTO NODES :: LBOUND
      write(6,*) 'reading map arrays finished'
C
C--------------------------------------------------------------------
C     ZERO ALL ARRAYS
C--------------------------------------------------------------------
C
      CALL CZERO
C
C--------------------------------------------------------------------
C     READ INPUT DATA-SET
```

```
C--------------------------------------------------------------------
C
      CALL CXREAD
      CALL CINDEX_SHIFT
C
C--------------------------------------------------------------------
C-----USER_FUNCTION INITIALISATION
C--------------------------------------------------------------------
C
      CALL CPS_INIT
C
C----------------------------------------------------------------------
C     SETUP CONVECTION AND DIFFUSION COEFFICIENTS
C----------------------------------------------------------------------
C
      IF (ISDIFF.EQ.0) THEN
        CALL COEFF_UPWI
      ELSE IF (ISDIFF.EQ.1) THEN
        CALL COEFF_HYBR
      END IF
C
C-----ZERO BOUNDARY COEFFICIENTS
C
      CALL COEFFZ
C
C----------------------------------------------------------------------
C     INITIALISE SOURCES AND ASSEMBLE POLE COEFFICIENT
C----------------------------------------------------------------------
C
      CALL CSTORE
      CALL CSORPS
      CALL CASSEM
C
C--------------------------------------------------------------------
C                       START OF TIME LOOP
C--------------------------------------------------------------------
C
      ITIME=1
    1 CONTINUE
C
C----------------------------------------------------------------------
C     SETUP SOURCES
C----------------------------------------------------------------------
C
      CALL CSTORE
      CALL CSORPS
C
C-----ADD CONTRIBUTIONS FROM INLET/OUTLETS AND WALLS INTO SP AND SU
C
      CALL CMOD_SPS
C
C--------------------------------------------------------------------
C     SOLVE PASSIVE SCALAR EQUATION
C--------------------------------------------------------------------
C
C
C-----NORMALISED RESIDUAL
C
      CALL CRESID
C
C----------------------------------------------------------------------
C     LINEAR EQUATION SOLVER
C----------------------------------------------------------------------
C
      CALL CGCCG
C
C----------------------------------------------------------------------
```

```
C-----UPDATE PASSIVE SCALAR AT BOUNDARIES
C--------------------------------------------------------------------
C
      CALL CUP_PS
C
C--------------------------------------------------------------------
C     END OF TIME LOOP
C--------------------------------------------------------------------
C
      ITIME=ITIME+1
      IF (ITIME.LE.NTIME) GOTO 1
C
C-----
      STOP
      END


C========================================================================
      SUBROUTINE CASSEM
C========================================================================
C
C     ASSEMBLE POLE COEFFICIENT FOR ALL TRANSPORT EQUATIONS
C
C========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C--------------------------------------------------------------------
!HPF$ INDEPENDENT, ON HOME(SP4(NC)), RESIDENT, REUSE
      DO 1 NC=NINTCI,NINTCF
         ACOEFF(NC)=ACOEFF(NC+NSFR01)
     X             +ACOEFF(NC+NSFR02)
     X             +ACOEFF(NC+NSFR03)
     X             +ACOEFF(NC+NSFR04)
     X             +ACOEFF(NC+NSFR05)
     X             +ACOEFF(NC+NSFR06)
     X             -SP4(NC)
    1 CONTINUE
C
      RETURN
      END


C========================================================================
      SUBROUTINE CGCCG
C========================================================================
C
C     CONJUGATE GRADIENT EQUATION SOLVER
C
C========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
      REAL RATII,RATIO
C--------------------------------------------------------------------
C
C-----DELIMITER
C
      RATIO=1.0
      CM15=1.E-15
      CM30=1.E-30
      OMEGA=0.0
C
C--------------------------------------------------------------------
C     INITIAL RESIDUAL
```

```
C-----------------------------------------------------------------------
C
      RESREF=0.0
!HPF$ INDEPENDENT, REDUCTION(RESREF), ON HOME(RESVEC(NC)), REUSE
      DO 11 NC=NINTCI,NINTCF
        RESREF=RESREF+RESVEC(NC)*RESVEC(NC)
   11 CONTINUE
C
      RESREF=SQRT(RESREF)
C
C-----RETURN IF NOTHING IS HAPPENING
C
      IF(RESREF.LT.CM15) GOTO 9999
C
C-----------------------------------------------------------------------
C     DIAGONAL SCALING
C-----------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(CGUP(NC)), REUSE
      DO 49 NC=NINTCI,NINTCF
        CGUP(NC)=1.0/ACOEFF(NC)
   49 CONTINUE
C
!HPF$ INDEPENDENT, ON HOME(DIREC1(NC)), REUSE
      DO 50 NC=NEXTCI,NEXTCF
        DIREC1(NC)=0.0
   50 CONTINUE
C
C-----------------------------------------------------------------------
C     INITIALISE DIRECTION VECTOR
C-----------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(DXOR1(NC)), REUSE
      DO 10 NC=NINTCI,NINTCF
        DIREC1(NC)=RESVEC(NC)*CGUP(NC)
        DXOR1(NC)=DIREC1(NC)
   10 CONTINUE
C
C-----------------------------------------------------------------------
C     START OF THE ORTHOMIN ITERATION LOOP
C-----------------------------------------------------------------------
C
C-----RESET ITERATION COUNTER
      ITERCG=1
  100 CONTINUE
C
C-----------------------------------------------------------------------
C     MATRIX-VECTOR PRODUCT
C-----------------------------------------------------------------------
C
      OMEGA=0.0
      RESNEW=0.0
      CNORM1_LOC=0.0
!HPF$ INDEPENDENT, REDUCTION(CNORM1_LOC, OMEGA, RESNEW),
!HPF$* ON HOME(DIREC2(NC)), REUSE,
!HPF$* RESIDENT(ACOEFF, DIREC2, ADXOR1, RESVEC)
      DO 4 NC=NINTCI,NINTCF
        DIREC2(NC)=ACOEFF(NC)*DIREC1(NC)
     X          -ACOEFF(NC+NSFR01)*DIREC1(LCC(NC+NSFI01))
     X          -ACOEFF(NC+NSFR02)*DIREC1(LCC(NC+NSFI02))
     X          -ACOEFF(NC+NSFR03)*DIREC1(LCC(NC+NSFI03))
     X          -ACOEFF(NC+NSFR04)*DIREC1(LCC(NC+NSFI04))
     X          -ACOEFF(NC+NSFR05)*DIREC1(LCC(NC+NSFI05))
     X          -ACOEFF(NC+NSFR06)*DIREC1(LCC(NC+NSFI06))
        CNORM1_LOC=CNORM1_LOC+DIREC2(NC)*DIREC2(NC)
        OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
        RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
```

```
        ADXOR1(NC)=DIREC2(NC)
    4 CONTINUE
C
C-----------------------------------------------------------------------
C     RESIDUAL SUM AND NORMALISATION
C-----------------------------------------------------------------------
C
      RESNEW=SQRT(RESNEW)
      IF(CNORM1_LOC.GT.CM30) THEN
        OMEGA=OMEGA/CNORM1_LOC
      ELSE
        OMEGA=1.0
      ENDIF
C
C-----------------------------------------------------------------------
C     CONVERGENCE CHECK
C-----------------------------------------------------------------------
C
      RATII=RATIO
      RATIO=RESNEW/RESREF
C
C-----STEADY MODE
C
      IF(RATIO.LE.1E-5) GOTO 9999
C
C-----CONVERGENCE RATE CONDITION
C
      IF(ITERCG.GT.3) THEN
        REDUC=ABS((RATII-RATIO)/RATII)
        IF(REDUC.LT.0.001) GOTO 9997
      ENDIF
C
C-----CHECK FOR MAXIMUM NO. OF ITERATIONS
C
      IF(ITERCG.GE.500) GOTO 9998
C
C-----------------------------------------------------------------------
C     1st SOLUTION UPDATE
C-----------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(PS(NC)), REUSE
      DO 440 NC=NINTCI,NINTCF
        PS(NC)=PS(NC)+OMEGA*DIREC1(NC)
        RESVEC(NC)=RESVEC(NC)-OMEGA*DIREC2(NC)
        DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
  440 CONTINUE
C
C-----INCREMENT ITERATION COUNTER
C
      ITERCG=ITERCG+1
C
C-----------------------------------------------------------------------
C     DIREC2 IS ORTHOGONALIZED TO THE PREVIOUSLY COMPUTED VECTOR (WHICH
C     IS STORED ON ADXOR1)
C-----------------------------------------------------------------------
C
      OC1=0.0
!HPF$ INDEPENDENT, REDUCTION(OC1), ON HOME(DIREC2(NC)), REUSE
      DO 5 NC=NINTCI,NINTCF
        DIREC2(NC)=ACOEFF(NC)*DIREC1(NC)
     X          -ACOEFF(NC+NSFR01)*DIREC1(LCC(NC+NSFI01))
     X          -ACOEFF(NC+NSFR02)*DIREC1(LCC(NC+NSFI02))
     X          -ACOEFF(NC+NSFR03)*DIREC1(LCC(NC+NSFI03))
     X          -ACOEFF(NC+NSFR04)*DIREC1(LCC(NC+NSFI04))
     X          -ACOEFF(NC+NSFR05)*DIREC1(LCC(NC+NSFI05))
     X          -ACOEFF(NC+NSFR06)*DIREC1(LCC(NC+NSFI06))
        OC1=OC1+ADXOR1(NC)*DIREC2(NC)
```

```
    5 CONTINUE
C
      OC1=OC1/CNORM1_LOC
C
      OMEGA=0.0
      RESNEW=0.0
      CNORM2_LOC=0.0
!HPF$ INDEPENDENT, REDUCTION(OMEGA, RESNEW, CNORM2_LOC),
!HPF$* ON HOME(DIREC2(NC)), REUSE
      DO 520 NC=NINTCI,NINTCF
        DIREC2(NC)=DIREC2(NC)-OC1*ADXOR1(NC)
        DIREC1(NC)=DIREC1(NC)-OC1*DXOR1(NC)
        DXOR2(NC)=DIREC1(NC)
        ADXOR2(NC)=DIREC2(NC)
        OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
        RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
        CNORM2_LOC=CNORM2_LOC+DIREC2(NC)*DIREC2(NC)
  520 CONTINUE
C
C---------------------------------------------------------------------
C     RESIDUAL SUM AND NORMALISATION
C---------------------------------------------------------------------
C
      RESNEW=SQRT(RESNEW)
      IF(CNORM2_LOC.GT.CM30) THEN
        OMEGA=OMEGA/CNORM2_LOC
      ELSE
        OMEGA=1.0
      ENDIF
C
C---------------------------------------------------------------------
C     CONVERGENCE CHECK
C---------------------------------------------------------------------
C
      RATII=RATIO
      RATIO=RESNEW/RESREF
C
C-----STEADY MODE
C
      IF(RATIO.LE.1E-5) GOTO 9999
C
C-----CONVERGENCE RATE CONDITION
C
      IF(ITERCG.GT.3) THEN
        REDUC=ABS((RATII-RATIO)/RATII)
        IF(REDUC.LT.0.001) GOTO 9997
      ENDIF
C
C-----CHECK FOR MAXIMUM NO. OF ITERATIONS
C
      IF(ITERCG.GE.500) GOTO 9998
C
C---------------------------------------------------------------------
C     2nd SOLUTION UPDATE
C---------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(PS(NC)), REUSE
      DO 540 NC=NINTCI,NINTCF
        PS(NC)=PS(NC)+OMEGA*DIREC1(NC)
        RESVEC(NC)=RESVEC(NC)-OMEGA*DIREC2(NC)
        DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
  540 CONTINUE
C
C-----INCREMENT ITERATION COUNTER
C
      ITERCG=ITERCG+1
C
```

```
C---------------------------------------------------------------------
C     DIREC2 IS ORTHOGONALIZED TO THE PREVIOUSLY COMPUTED VECTORS (WHICH
C     ARE STORED ON ADXOR1 AND ADXOR2 RESP.)
C---------------------------------------------------------------------
C
      OC1=0.0
      OC2=0.0
!HPF$ INDEPENDENT, REDUCTION(OC1, OC2), ON HOME(DIREC2(NC)), REUSE
      DO 6 NC=NINTCI,NINTCF
        DIREC2(NC)=ACOEFF(NC)*DIREC1(NC)
     X            -ACOEFF(NC+NSFR01)*DIREC1(LCC(NC+NSFI01))
     X            -ACOEFF(NC+NSFR02)*DIREC1(LCC(NC+NSFI02))
     X            -ACOEFF(NC+NSFR03)*DIREC1(LCC(NC+NSFI03))
     X            -ACOEFF(NC+NSFR04)*DIREC1(LCC(NC+NSFI04))
     X            -ACOEFF(NC+NSFR05)*DIREC1(LCC(NC+NSFI05))
     X            -ACOEFF(NC+NSFR06)*DIREC1(LCC(NC+NSFI06))
        OC1=OC1+ADXOR1(NC)*DIREC2(NC)
        OC2=OC2+ADXOR2(NC)*DIREC2(NC)
    6 CONTINUE
C
      OC1=OC1/CNORM1_LOC
      OC2=OC2/CNORM2_LOC
C
      OMEGA=0
      RESNEW=0.0
      CNORM3_LOC=0.0
!HPF$ INDEPENDENT, REDUCTION(OMEGA, RESNEW, CNORM3_LOC),
!HPF$* ON HOME(DIREC2(NC)), REUSE
      DO 620 NC=NINTCI,NINTCF
        DIREC2(NC)=DIREC2(NC)-OC1*ADXOR1(NC)-OC2*ADXOR2(NC)
        DIREC1(NC)=DIREC1(NC)-OC1*DXOR1(NC)- OC2*DXOR2(NC)
        CNORM3_LOC=CNORM3_LOC+DIREC2(NC)*DIREC2(NC)
        OMEGA=OMEGA+RESVEC(NC)*DIREC2(NC)
        RESNEW=RESNEW+RESVEC(NC)*RESVEC(NC)
  620 CONTINUE
C
C---------------------------------------------------------------------
C     RESIDUAL SUM AND NORMALISATION
C---------------------------------------------------------------------
C
      RESNEW=SQRT(RESNEW)
      IF(CNORM3_LOC.GT.CM30) THEN
        OMEGA=OMEGA/CNORM3_LOC
      ELSE
        OMEGA=1.0
      ENDIF
C
C---------------------------------------------------------------------
C     CONVERGENCE CHECK
C---------------------------------------------------------------------
C
      RATII=RATIO
      RATIO=RESNEW/RESREF
C
      IF(RATIO.LE.1E-5) GOTO 9999
C
C-----CONVERGENCE RATE CONDITION
C
      IF(ITERCG.GT.3) THEN
        REDUC=ABS((RATII-RATIO)/RATII)
        IF(REDUC.LT.0.001) GOTO 9997
      ENDIF
C
C-----CHECK FOR MAXIMUM NO. OF ITERATIONS
C
      IF(ITERCG.GE.500) GOTO 9998
C
```

```
C----------------------------------------------------------------------
C     3rd SOLUTION UPDATE
C----------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(PS(NC)), REUSE
      DO 640 NC=NINTCI,NINTCF
        PS(NC)=PS(NC)+OMEGA*DIREC1(NC)
        RESVEC(NC)=RESVEC(NC)-OMEGA*DIREC2(NC)
        DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
        DXOR1(NC)=DIREC1(NC)
  640 CONTINUE
C
C-----INCREMENT ITERATION COUNTER
C
      ITERCG=ITERCG+1
C
C----------------------------------------------------------------------
C     GO AND GIVE IT ANOTHER TRY
C----------------------------------------------------------------------
C
      GOTO 100
C
C----------------------------------------------------------------------
C     CONVERGENCE FAILURE
C----------------------------------------------------------------------
C
C-----CONVERGENCE RATE CONDITION
C
 9997 CONTINUE
      IF(RATIO.LT.0.9) GOTO 9999
      WRITE(6,*) 'Residual reduction too slow'
      WRITE(6,*) 'latest residual = ',RATIO
      WRITE(6,*) 'at iteration = ',ITERCG
      GOTO 9999
C
C-----CHECK FOR MAXIMUM NO. OF ITERATIONS
C
 9998 CONTINUE
      WRITE(6,*) 'Residual ratio after ',ITERCG,' iterations bigger'
      WRITE(6,*) 'than 1.e-5  [value = ',RATIO,' ]'
C
 9999 CONTINUE
C
      IF (ITIME.EQ.1) WRITE(6,*) ' itime  itercg  ratio '
      WRITE(6,'(2i6,f16.12)') itime,itercg,ratio
C
      RETURN
      END


C======================================================================
      SUBROUTINE CGET_SOURC (NCBOU)
C======================================================================
C
C     CONTRIBUTIONS TO SOURCE FROM INLET/OUTLET BOUNDARIES
C
C======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
      DIMENSION NCBOU(7)
C
C----------------------------------------------------------------------
C
C-----DELIMITER
C
```

```
      CM0=0.0
C
      DO 1 IB=1,6
       IA=NCBOU(IB)
       IE=NCBOU(IB+1)-1
       IS=(IB-1)*NINTCF
C-----MASS FLOW DIRECTION
       IF(IB.EQ.1.OR.IB.EQ.4.OR.IB.EQ.5) THEN
        XMFD=1.
       ELSE
        XMFD=-1.
       ENDIF
       IF ((IE-IA+1).GT.0) THEN
!HPF$ INDEPENDENT, ON HOME(LBOUND(N)), REDUCTION(SP4, SU4)
       DO 11 N=IA,IE
c         NC=LBOUND(N)
C-----LOCATION CELL-CENTER TO CELL-CENTER
c         KC=LCC(LBOUND(N)+IS)
C-----MASS FLOW RATE
C-----INTERPOLATED CARTESIAN CELL-FACE VELOCITIES
C
        IF (IB.EQ.1) THEN
         WF=WF5(LBOUND(N))
         XJ1=XJAF12(LCF(LBOUND(N)+IS))
         XJ2=XJAF22(LCF(LBOUND(N)+IS))
         XJ3=XJAF32(LCF(LBOUND(N)+IS))
        ELSE IF (IB.EQ.2) THEN
         WF=WF1(LBOUND(N))
         XJ1=XJAF11(LCF(LBOUND(N)+IS))
         XJ2=XJAF21(LCF(LBOUND(N)+IS))
         XJ3=XJAF31(LCF(LBOUND(N)+IS))
        ELSE IF (IB.EQ.3) THEN
         WF=WF2(LBOUND(N))
         XJ1=XJAF12(LCF(LBOUND(N)+IS))
         XJ2=XJAF22(LCF(LBOUND(N)+IS))
         XJ3=XJAF32(LCF(LBOUND(N)+IS))
        ELSE IF (IB.EQ.4) THEN
         WF=WF4(LBOUND(N))
         XJ1=XJAF11(LCF(LBOUND(N)+IS))
         XJ2=XJAF21(LCF(LBOUND(N)+IS))
         XJ3=XJAF31(LCF(LBOUND(N)+IS))
        ELSE IF (IB.EQ.5) THEN
         WF=WF6(LBOUND(N))
         XJ1=XJAF13(LCF(LBOUND(N)+IS))
         XJ2=XJAF23(LCF(LBOUND(N)+IS))
         XJ3=XJAF33(LCF(LBOUND(N)+IS))
        ELSE IF (IB.EQ.6) THEN
         WF=WF3(LBOUND(N))
         XJ1=XJAF13(LCF(LBOUND(N)+IS))
         XJ2=XJAF23(LCF(LBOUND(N)+IS))
         XJ3=XJAF33(LCF(LBOUND(N)+IS))
        END IF
C-----
        U1F=U1(LBOUND(N))+WF*(U1(LCC(LBOUND(N)+IS))-U1(LBOUND(N)))
        U2F=U2(LBOUND(N))+WF*(U2(LCC(LBOUND(N)+IS))-U2(LBOUND(N)))
        U3F=U3(LBOUND(N))+WF*(U3(LCC(LBOUND(N)+IS))-U3(LBOUND(N)))
C-----
        FLOWC=U1F*XJ1+U2F*XJ2+U3F*XJ3
C-----
        FLOW=MAX(XMFD*FLOWC,CM0)
C-----ADD INTO SP
        SP4(LBOUND(N))=SP4(LBOUND(N))-FLOW
C-----ADD INTO SU
        SU4(LBOUND(N))=SU4(LBOUND(N))+FLOW*PS(LCC(LBOUND(N)+IS))
   11   CONTINUE
       END IF
    1 CONTINUE
```

```
C-----
      RETURN
      END


C=========================================================================
      SUBROUTINE CINDEX_SHIFT
C=========================================================================
C
C     DEFINE INDEX SHIFTS FOR ADDRESSING AND COEFFICIENT ARRAYS
C
C=========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
C     SHIFTS FOR INTEGER ARRAYS LCC AND LCF
C
      NSFI01=0
      NSFI02=NINTCF
      NSFI03=2*NINTCF
      NSFI04=3*NINTCF
      NSFI05=4*NINTCF
      NSFI06=5*NINTCF
C
C     SHIFTS FOR REAL ARRAY ACOEFF
C
      NSFR01=NINTCF
      NSFR02=2*NINTCF
      NSFR03=3*NINTCF
      NSFR04=4*NINTCF
      NSFR05=5*NINTCF
      NSFR06=6*NINTCF
C
      RETURN
      END


C=========================================================================
      SUBROUTINE CMIR_VAR (NCBOU)
C=========================================================================
C
C     MIRROR VARIABLES AT BOUNDARIES
C
C=========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----LOCAL VARIABLES
      DIMENSION NCBOU(7)
C-----------------------------------------------------------------------
C
      DO 1 IB=1,6
        IA=NCBOU(IB)
        IE=NCBOU(IB+1)-1
        IS=(IB-1)*NINTCF
        IF ((IE-IA+1).GT. 0) THEN
!HPF$ INDEPENDENT, ON HOME(DIREC1(N))
          DO 2 N=IA,IE
            DIREC1(N) = PS(LBOUND(N))
    2     CONTINUE
!HPF$ INDEPENDENT, ON HOME(DIREC1(N))
          DO 3 N=IA,IE
            PS(LCC(LBOUND(N)+IS)) = DIREC1(N)
```

```
    3     CONTINUE
          END IF
    1 CONTINUE
C
      RETURN
      END



C===========================================================================
      SUBROUTINE CMOD_SPS
C===========================================================================
C
C     CONTRIBUTIONS TO SOURCE FROM INLET/OUTLET BOUNDARIES
C     AND IMPERMEABLE WALLS
C
C===========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C     INLET - VELOCITY
C-----------------------------------------------------------------------
C
      IF(NFINLV.GT.0) CALL CGET_SOURC (NCINLV)
C
C-----------------------------------------------------------------------
C     OUTLET - GLOBAL CONTINUITY
C-----------------------------------------------------------------------
C
      IF(NFOUTL.GT.0) CALL CGET_SOURC (NCOUTL)
C
C-----
      RETURN
      END


C===========================================================================
      SUBROUTINE COEFF_UPWI
C===========================================================================
C
C     CONVECTION AND DIFFUSION COEFFICIENTS FOR UPWIND SCHEME
C
C===========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
C-----DELIMITER
C
      CM0=0.0
C
C-----ASSEMBLE CONVECTIVE PART OF COEFFICIENTS
C
C-----
!HPF$ INDEPENDENT, ON HOME(WF1(NC)), REUSE
      DO 1 NC=NINTCI,NINTCF
C
C-----INTERPOLATED CARTESIAN CELL-FACE VELOCITIES
C
C-----U-COMPONENT
C
        U1FE=U1(NC)+WF1(NC)*(U1(LCC(NC+NSFI02))-U1(NC))
        U1FN=U1(NC)+WF2(NC)*(U1(LCC(NC+NSFI03))-U1(NC))
```

```
      U1FH=U1(NC)+WF3(NC)*(U1(LCC(NC+NSFI06))-U1(NC))
      U1FW=U1(NC)+WF4(NC)*(U1(LCC(NC+NSFI04))-U1(NC))
      U1FS=U1(NC)+WF5(NC)*(U1(LCC(NC+NSFI01))-U1(NC))
      U1FL=U1(NC)+WF6(NC)*(U1(LCC(NC+NSFI05))-U1(NC))
C
C-----V-COMPONENT
C
      U2FE=U2(NC)+WF1(NC)*(U2(LCC(NC+NSFI02))-U2(NC))
      U2FN=U2(NC)+WF2(NC)*(U2(LCC(NC+NSFI03))-U2(NC))
      U2FH=U2(NC)+WF3(NC)*(U2(LCC(NC+NSFI06))-U2(NC))
      U2FW=U2(NC)+WF4(NC)*(U2(LCC(NC+NSFI04))-U2(NC))
      U2FS=U2(NC)+WF5(NC)*(U2(LCC(NC+NSFI01))-U2(NC))
      U2FL=U2(NC)+WF6(NC)*(U2(LCC(NC+NSFI05))-U2(NC))
C
C-----W-COMPONENT
C
      U3FE=U3(NC)+WF1(NC)*(U3(LCC(NC+NSFI02))-U3(NC))
      U3FN=U3(NC)+WF2(NC)*(U3(LCC(NC+NSFI03))-U3(NC))
      U3FH=U3(NC)+WF3(NC)*(U3(LCC(NC+NSFI06))-U3(NC))
      U3FW=U3(NC)+WF4(NC)*(U3(LCC(NC+NSFI04))-U3(NC))
      U3FS=U3(NC)+WF5(NC)*(U3(LCC(NC+NSFI01))-U3(NC))
      U3FL=U3(NC)+WF6(NC)*(U3(LCC(NC+NSFI05))-U3(NC))
C
C-----MASS FLOW
C
      FLOWCE=(U1FE*XJAF11(LCF(NC+NSFI02))
     X        +U2FE*XJAF21(LCF(NC+NSFI02))
     X        +U3FE*XJAF31(LCF(NC+NSFI02)))
C
      FLOWCN=(U1FN*XJAF12(LCF(NC+NSFI03))
     X        +U2FN*XJAF22(LCF(NC+NSFI03))
     X        +U3FN*XJAF32(LCF(NC+NSFI03)))
C
      FLOWCH=(U1FH*XJAF13(LCF(NC+NSFI06))
     X        +U2FH*XJAF23(LCF(NC+NSFI06))
     X        +U3FH*XJAF33(LCF(NC+NSFI06)))
C
      FLOWCW=(U1FW*XJAF11(LCF(NC+NSFI04))
     X        +U2FW*XJAF21(LCF(NC+NSFI04))
     X        +U3FW*XJAF31(LCF(NC+NSFI04)))
C
      FLOWCS=(U1FS*XJAF12(LCF(NC+NSFI01))
     X        +U2FS*XJAF22(LCF(NC+NSFI01))
     X        +U3FS*XJAF32(LCF(NC+NSFI01)))
C
      FLOWCL=(U1FL*XJAF13(LCF(NC+NSFI05))
     X        +U2FL*XJAF23(LCF(NC+NSFI05))
     X        +U3FL*XJAF33(LCF(NC+NSFI05)))
C
C-----CONVECTIVE PART OF COEFFICIENTS
C
      ACVE=MAX(-FLOWCE,CM0)
      ACVW=MAX( FLOWCW,CM0)
      ACVN=MAX(-FLOWCN,CM0)
      ACVS=MAX( FLOWCS,CM0)
      ACVH=MAX(-FLOWCH,CM0)
      ACVL=MAX( FLOWCL,CM0)
C
C-----TRANSFORMATION QUANTITIES
C
C-----E/W CELL-FACES
      G11E=XJAF11(LCF(NC+NSFI02))*XJAF11(LCF(NC+NSFI02))
     X     +XJAF21(LCF(NC+NSFI02))*XJAF21(LCF(NC+NSFI02))
     X     +XJAF31(LCF(NC+NSFI02))*XJAF31(LCF(NC+NSFI02))
      G11W=XJAF11(LCF(NC+NSFI04))*XJAF11(LCF(NC+NSFI04))
     X     +XJAF21(LCF(NC+NSFI04))*XJAF21(LCF(NC+NSFI04))
     X     +XJAF31(LCF(NC+NSFI04))*XJAF31(LCF(NC+NSFI04))
```

```
C-----N/S CELL-FACES
       G22N=XJAF12(LCF(NC+NSFI03))*XJAF12(LCF(NC+NSFI03))
     X      +XJAF22(LCF(NC+NSFI03))*XJAF22(LCF(NC+NSFI03))
     X      +XJAF32(LCF(NC+NSFI03))*XJAF32(LCF(NC+NSFI03))
       G22S=XJAF12(LCF(NC+NSFI01))*XJAF12(LCF(NC+NSFI01))
     X      +XJAF22(LCF(NC+NSFI01))*XJAF22(LCF(NC+NSFI01))
     X      +XJAF32(LCF(NC+NSFI01))*XJAF32(LCF(NC+NSFI01))
C-----H/L CELL-FACES
       G33H=XJAF13(LCF(NC+NSFI06))*XJAF13(LCF(NC+NSFI06))
     X      +XJAF23(LCF(NC+NSFI06))*XJAF23(LCF(NC+NSFI06))
     X      +XJAF33(LCF(NC+NSFI06))*XJAF33(LCF(NC+NSFI06))
       G33L=XJAF13(LCF(NC+NSFI05))*XJAF13(LCF(NC+NSFI05))
     X      +XJAF23(LCF(NC+NSFI05))*XJAF23(LCF(NC+NSFI05))
     X      +XJAF33(LCF(NC+NSFI05))*XJAF33(LCF(NC+NSFI05))
C-----INTERPOLATED VOLUMES
       VOLE = 0.5 * (VOL(LCC(NC+NSFI02))+VOL(NC))
       VOLW = 0.5 * (VOL(LCC(NC+NSFI04))+VOL(NC))
       VOLN = 0.5 * (VOL(LCC(NC+NSFI03))+VOL(NC))
       VOLS = 0.5 * (VOL(LCC(NC+NSFI01))+VOL(NC))
       VOLH = 0.5 * (VOL(LCC(NC+NSFI06))+VOL(NC))
       VOLL = 0.5 * (VOL(LCC(NC+NSFI05))+VOL(NC))
C
C-----CONVECTION + DIFFUSION COEFFICIENT
C
       ACOEFF(NC+NSFR01)=ACVS+VISC/VOLS*G22S
       ACOEFF(NC+NSFR02)=ACVE+VISC/VOLE*G11E
       ACOEFF(NC+NSFR03)=ACVN+VISC/VOLN*G22N
       ACOEFF(NC+NSFR04)=ACVW+VISC/VOLW*G11W
       ACOEFF(NC+NSFR05)=ACVL+VISC/VOLL*G33L
       ACOEFF(NC+NSFR06)=ACVH+VISC/VOLH*G33H
C
  1   CONTINUE
C
      RETURN
      END


C=======================================================================
      SUBROUTINE COEFFZ
C=======================================================================
C
C     SET COEFFICIENTS ADJACENT TO BOUNDARIES TO ZERO
C
C=======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
C-----SOUTH
C
!HPF$ INDEPENDENT
      DO 1 N=NBCESI,NBCESF
       ACOEFF(LBS(N)+NSFR01)=0.0
  1   CONTINUE
C
C-----EAST
C
!HPF$ INDEPENDENT
      DO 2 N=NBCEEI,NBCEEF
       ACOEFF(LBE(N)+NSFR02)=0.0
  2   CONTINUE
C
C-----NORTH
C
!HPF$ INDEPENDENT
```

```
      DO 3 N=NBCENI,NBCENF
        ACOEFF(LBN(N)+NSFR03)=0.0
  3   CONTINUE
C
C-----WEST
C
!HPF$ INDEPENDENT
      DO 4 N=NBCEWI,NBCEWF
        ACOEFF(LBW(N)+NSFR04)=0.0
  4   CONTINUE
C
C-----LOW
C
!HPF$ INDEPENDENT
      DO 5 N=NBCELI,NBCELF
        ACOEFF(LBL(N)+NSFR05)=0.0
  5   CONTINUE
C
C-----HIGH
C
!HPF$ INDEPENDENT
      DO 6 N=NBCEHI,NBCEHF
        ACOEFF(LBH(N)+NSFR06)=0.0
  6   CONTINUE
C
      RETURN
      END


C=====================================================
      SUBROUTINE CPS_INIT
C=====================================================
C
C     ROUTINE TO INITIALISE THE PASSIVE SCALAR ARRAY AT
C     THE BOUNDARIES OR AT THE INTERNAL COMPUTATIONAL DOMAIN.
C
C--------------------------------------------------
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C--------------------------------------------------
C-----
C-----INITIALISE SCALAR AT INLET - VELOCITY BOUNDARIES
C-----
      IF(NFINLV.GT.0) THEN
        DO IB=1,6
          IA=NCINLV(IB)
          IE=NCINLV(IB+1)-1
          IS=(IB-1)*NINTCF
!HPF$ INDEPENDENT, ON HOME(LBOUND(N)), RESIDENT(PS)
          DO N=IA,IE
            PS(LBOUND(N))=1.0
            PS(LCC(LBOUND(N)+IS))=1.0
          END DO
        END DO
      END IF
C-----
      RETURN
      END


C======================================================================
      SUBROUTINE CRESID
C======================================================================
C
C     NORMALISED RESIDUALS
```

```
C
C    NOTE: THE RSM EXTENSION CONTAINS AN ADDITIONAL CONVERGENCE JACK
C          OF THE NEAR WALL REGION.
C
C======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C----------------------------------------------------------------------
C
      RESX=0.0
      RESN=0.0
!HPF$ INDEPENDENT, REDUCTION(RESX, RESN), ON HOME(RESVEC(NC)), REUSE
      DO 1 NC=NINTCI,NINTCF
C-----CALCULATE RESIDUAL
       RESVEC(NC)=SU4(NC)-ACOEFF(NC)*PS(NC)
     X          +ACOEFF(NC+NSFR01)*PS(LCC(NC+NSFI01))
     X          +ACOEFF(NC+NSFR02)*PS(LCC(NC+NSFI02))
     X          +ACOEFF(NC+NSFR03)*PS(LCC(NC+NSFI03))
     X          +ACOEFF(NC+NSFR04)*PS(LCC(NC+NSFI04))
     X          +ACOEFF(NC+NSFR05)*PS(LCC(NC+NSFI05))
     X          +ACOEFF(NC+NSFR06)*PS(LCC(NC+NSFI06))
C-----ABSOLUTE SUM OF RESIDUALS
       RESX=RESX+ABS(RESVEC(NC))
C-----NORMALISATION QUANTITY
       RESN=RESN+ABS(ACOEFF(NC)*PS(NC))
    1 CONTINUE
C
C----------------------------------------------------------------------
C    NORMALISATION
C----------------------------------------------------------------------
C
      IF(RESN.GT.1.E-10) THEN
       RSUM=RESX/RESN
      ELSE
       RSUM=RESX
      ENDIF
C
C----------------------------------------------------------------------
C    CHECK FOR NaN
C----------------------------------------------------------------------
C
      IF (RSUM.NE.RSUM) THEN
       WRITE(6,*) 'NaN : Invalid floating point results in'//
     X       'passive scalar equation - Execution terminated.'
      END IF
C
      RETURN
      END


C======================================================================
      SUBROUTINE CSORPS
C======================================================================
C
C    SOURCE OF PASSIVE SCALAR EQUATION
C
C======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C----------------------------------------------------------------------
C
C-----SETUP SP AND SU
C
```

```
!HPF$ INDEPENDENT, ON HOME(SP4(NC)), REUSE
      DO 1 NC=NINTCI,NINTCF
       SP4(NC)=-VOL(NC)/DT
       SU4(NC)=VOL(NC)/DT*PSO(NC)
    1 CONTINUE
C
      RETURN
      END


C=======================================================================
      SUBROUTINE CSTORE
C=======================================================================
C
C     STORE OLD TIME STEP VALUES OF PASSIVE SCALAR
C
C=======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(PSO(NC)), REUSE
      DO 1 NC=NINTCI,NINTCF
       PSO(NC)=PS(NC)
    1 CONTINUE
C
      RETURN
      END


C=======================================================================
      SUBROUTINE CUP_PS
C=======================================================================
C
C     UPDATE PASSIVE SCALAR AT BOUNDARIES
C
C     NOTE: INLET - CONDITIONS OBTAINED FROM BND-FILE
C
C=======================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
C-----------------------------------------------------------------------
C     OUTLET - ADDRESSES
C-----------------------------------------------------------------------
C
      IF(NFOUTL.GT.0) CALL CMIR_VAR (NCOUTL)
C
C-----------------------------------------------------------------------
C     WALL - FIXED
C-----------------------------------------------------------------------
C
      IF(NFFIXW.GT.0) CALL CMIR_VAR (NCFIXW)
C
C-----------------------------------------------------------------------
C     PLANES OF SYMMETRY
C-----------------------------------------------------------------------
C
      IF(NFSYMM.GT.0) CALL CMIR_VAR (NCSYMM)
C
      RETURN
      END
```

```
C========================================================================
      SUBROUTINE CXREAD
C========================================================================
C
C     READ REQUIRED INPUT DATA-SET
C
C========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C-----
C---------------------------------------------------------------------
C-----INQUIRE NUMBER OF TIME-STEPS
C---------------------------------------------------------------------
C-----
      NTIME=0
      WRITE(6,*) 'Number of time-steps [100]:'
      READ(5,*) NTIME
      IF(NTIME.LE.0) NTIME=100
C-----
C---------------------------------------------------------------------
C-----INQUIRE DIFFERENCING SCHEME
C---------------------------------------------------------------------
C-----
 30   CONTINUE
      WRITE(6,*) 'Differencing scheme [0 - UPWI, 1 - HYBR]:'
      READ(5,*) ISDIFF
      IF(ISDIFF.LT.0.OR.ISDIFF.GT.1) ISDIFF=0
C-----
C---------------------------------------------------------------------
C-----INQUIRE TIME STEP
C---------------------------------------------------------------------
C-----
      DT=0.0
      WRITE(6,*) 'Time step [0.001]:'
      READ(5,*) DT
      IF(DT.LE.0.0) DT=0.001
C-----
      VISC=1.0
C-----
C---------------------------------------------------------------------
C-----READ INPUT DATA FROM FILE
C---------------------------------------------------------------------
C-----
C-----
      OPEN(20,FILE='in.dat'
     X       ,STATUS='OLD'
     X       ,FORM='FORMATTED'
     X       ,IOSTAT=IOST,ERR=990)
C-----
      READ(20,*,ERR=991) NINTCI,NINTCF,NEXTCI,NEXTCF
      IF (NINTCF.NE.NNINTC) THEN
        WRITE(6,*) 'Parameter NNINTC in file cpara.f not correct!'
        WRITE(6,*) 'Value: ',NINTCF
        STOP
      END IF
      IF (NEXTCF.NE.NNCELL) THEN
        WRITE(6,*) 'Parameter NNCELL in file cpara.f not correct!'
        WRITE(6,*) 'Value: ',NEXTCF
        STOP
      END IF
C-----
      READ(20,*,ERR=991) NCFACI,NCFACF,NCBMAX
      IF (NCFACF.NE.NNFACE) THEN
        WRITE(6,*) 'Parameter NNFACE in file cpara.f not correct!'
```

```
        WRITE(6,*) 'Value: ',NCFACF
        STOP
      END IF
      IF (NCBMAX.NE.NNBMAX) THEN
        WRITE(6,*) 'Parameter NNBMAX in file cpara.f not correct!'
        WRITE(6,*) 'Value: ',NCBMAX
        STOP
      END IF
C-----
      READ(20,*) LCC
      READ(20,*) LCF
C-----
      READ(20,*) U1
      READ(20,*) U2
      READ(20,*) U3
C-----
      READ(20,*) WF1
      READ(20,*) WF2
      READ(20,*) WF3
      READ(20,*) WF4
      READ(20,*) WF5
      READ(20,*) WF6
      READ(20,*) VOL
C-----
      READ(20,*) XJAF11
      READ(20,*) XJAF12
      READ(20,*) XJAF13
      READ(20,*) XJAF21
      READ(20,*) XJAF22
      READ(20,*) XJAF23
      READ(20,*) XJAF31
      READ(20,*) XJAF32
      READ(20,*) XJAF33
C-----
      READ(20,*) LBS
      READ(20,*) LBN
      READ(20,*) LBL
      READ(20,*) LBH
      READ(20,*) LBW
      READ(20,*) LBE
C-----
      READ(20,*,ERR=992) NBCEWI,NBCEWF,NBCEEI,NBCEEF
     X                  ,NBCESI,NBCESF,NBCENI,NBCENF
     X                  ,NBCELI,NBCELF,NBCEHI,NBCEHF
C-----
      READ(20,*,ERR=992) NFINLV,(NCINLV(N),N=1,7)
      READ(20,*,ERR=992) NFOUTL,(NCOUTL(N),N=1,7)
      READ(20,*,ERR=992) NFFIXW,(NCFIXW(N),N=1,7)
      READ(20,*,ERR=992) NFSYMM,(NCSYMM(N),N=1,7)
C-----
      READ(20,*) LBOUND
      write(6,*) 'Read finished'
C-----
      CLOSE(20,ERR=991,IOSTAT=IOST1)
C-----
      RETURN
C-----
  990 WRITE(6,*) 'Error during opening input file: IOSTAT = ',IOST
      GOTO 1000
  991 WRITE(6,*) 'Error during close input file: IOSTAT = ',IOST1
      GOTO 1000
  992 WRITE(6,*) 'Error during read'
C-----
 1000 STOP
C-----
      END
```

```
C=========================================================================
      SUBROUTINE CZERO
C=========================================================================
C
C     ZERO ALL ARRAYS
C
C=========================================================================
C
      USE GLOBVAR
      INCLUDE 'IMPLICIT.INC'
C
C-----------------------------------------------------------------------
C
      CM0=0.0
C
C-----SET ALL ARRAYS TO ZERO
C
      ACOEFF=0.0
C
      PS=0.0
      PSO=0.0
C
      SP4=0.0
      SU4=0.0
C
      WF1=0.0
      WF2=0.0
      WF3=0.0
      WF4=0.0
      WF5=0.0
      WF6=0.0
      VOL=0.00
C
      XJAF11=0.0
      XJAF22=0.0
      XJAF33=0.0
      XJAF12=0.0
      XJAF13=0.0
      XJAF21=0.0
      XJAF23=0.0
      XJAF31=0.0
      XJAF32=0.0
C
      ISRINI=0
      ISRSPS=0
C
      RETURN
      END


!-----------------------------------------------------------------------
      module globvar
!-----
!-----------------------------------------------------------------------
!-----DIMENSIONS FOR ROHR
!-----------------------------------------------------------------------
!-----
      integer, parameter :: NNINTC = 10240
      integer, parameter :: NNEXTC =  2432
      integer, parameter :: NNCELL = 12672
      integer, parameter :: NNFACE = 33152
      integer, parameter :: NNBMAX =  2400
!-----
!-----------------------------------------------------------------------
!-----DIMENSIONS FOR ROHR2
!-----------------------------------------------------------------------
```

```
!-----
!      integer, parameter :: NNINTC = 26400
!      integer, parameter :: NNEXTC =  4320
!      integer, parameter :: NNCELL = 30720
!      integer, parameter :: NNFACE = 83740
!      integer, parameter :: NNBMAX =  4105
!-----
!-----------------------------------------------------------------------
!-----DIMENSIONS FOR INTAKE
!-----------------------------------------------------------------------
!-----
!      integer, parameter :: NNINTC = 88672
!      integer, parameter :: NNEXTC = 24819
!      integer, parameter :: NNCELL =113491
!      integer, parameter :: NNFACE =277469
!      integer, parameter :: NNBMAX =  4105
!-----
!-----------------------------------------------------------------------
!-----GENERAL
!-----------------------------------------------------------------------
!-----
      integer :: nintci,nintcf,nextci,nextcf
      integer :: ncfaci,ncfacf
      integer :: itime,ntime
      integer :: isdiff
!-----
      double precision :: visc, dt
!-----
!-----------------------------------------------------------------------
!-----INTEGER ARRAYS
!-----------------------------------------------------------------------
!-----
      integer, dimension(6*nnintc) :: lcc
      integer, dimension(6*nnintc) :: lcf
      integer, dimension(nnbmax) :: lbs, lbn, lbl, lbh, lbw, lbe
      integer, dimension(nnextc) :: lbound
      integer, dimension(7) :: ncinlv, ncoutl, ncfixw, ncsymm
      integer               :: nfinlv, nfoutl, nffixw, nfsymm
      integer :: nbcewi,nbcewf,nbceei,nbceef
      integer :: nbcesi,nbcesf,nbceni,nbcenf
      integer :: nbceli,nbcelf,nbcehi,nbcehf
!-----
      integer :: nsfi01,nsfi02,nsfi03,nsfi04,nsfi05,nsfi06
      integer :: nsfr01,nsfr02,nsfr03,nsfr04,nsfr05,nsfr06
!-----
!-----------------------------------------------------------------------
!-----REAL ARRAYS
!-----------------------------------------------------------------------
!-----
      real, dimension(nnintc) :: pso
      real, dimension(nnintc) :: wf1, wf2, wf3, wf4, wf5, wf6
      real, dimension(7*nnintc) :: acoeff
      real, dimension(nncell) :: vol
!-----
!-----FACE DATA
      real, dimension(nnface) :: xjaf11, xjaf21, xjaf31
      real, dimension(nnface) :: xjaf12, xjaf22, xjaf32
      real, dimension(nnface) :: xjaf13, xjaf23, xjaf33
!-----
!-----------------------------------------------------------------------
!-----DOUBLE PRECISION ARRAYS
!-----------------------------------------------------------------------
!-----
      double precision, dimension(nncell) :: ps, u1, u2, u3, direc1
      double precision, dimension(nnintc) :: sp4, su4
      double precision, dimension(nnintc) :: cgup, resvec, direc2
      double precision, dimension(nnintc) :: dxor1, dxor2
```

```
      double precision, dimension(nnintc) :: adxor1, adxor2
!-----
!----------------------------------------------------------------------
!-----DATA DISTRIBUTION
!----------------------------------------------------------------------
!-----
      integer, dimension(6*nnintc) :: map1
      integer, dimension(7*nnintc) :: map2
      integer, dimension(nnface) :: map3
      integer, dimension(nncell) :: map4
      integer, dimension(nnextc) :: map5
!-----
!----------------------------------------------------------------------
!     HPF DEFINITIONS
!----------------------------------------------------------------------
!
!HPF$ PROCESSORS NODES(NUMBER_OF_PROCESSORS())
!
!----------------------------------------------------------------------
!     INDIRECTED DISTRIBUTED ARRAYS
!----------------------------------------------------------------------
!
!-----CONECTIVITY ARRAYS (DIM: 6*NNINTC) --> DISTRIBUTED VIA ARRAY MAP1
!HPF$ DYNAMIC :: LCC, LCF
!
!-----COEFFICIENTS (DIM: 7*NNINTC)        --> DISTRIBUTED VIA ARRAY MAP2
!HPF$ DYNAMIC :: ACOEFF
!
!-----CELL FACE DATA (DIM: NNFACE)        --> DISTRIBUTED VIA ARRAY MAP3
!HPF$ DYNAMIC :: XJAF11, XJAF21, XJAF31
!HPF$ DYNAMIC :: XJAF12, XJAF22, XJAF32
!HPF$ DYNAMIC :: XJAF13, XJAF23, XJAF33
!
!-----CELL CENTER DATA (DIM: NNCELL)      --> DISTRIBUTED VIA ARRAY MAP4
!HPF$ DYNAMIC :: PS, U1, U2, U3
!HPF$ DYNAMIC :: DIREC1, VOL
!
!-----BOUNDARY CELLS (DIM: NNEXTC)        --> DISTRIBUTED VIA ARRAY MAP5
!HPF$ DYNAMIC :: LBOUND
!
!----------------------------------------------------------------------
!     BLOCK DISTRIBUTION FOR ALL ARRAY OVER INTERNAL CELLS (DIM: NNINTC)
!----------------------------------------------------------------------
!
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: PSO
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: WF1, WF2, WF3, WF4, WF5, WF6
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: SU4, SP4
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: CGUP, RESVEC, DIREC2
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: DXOR1, DXOR2
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: ADXOR1, ADXOR2
!-----
      end module globvar
```

6.2 ECMWF HPF+ kernels

6.2.1 TS kernel

(a few of the higher level routines and data structures)

```
      SUBROUTINE STEPO
      ...
!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPSP)) ONTO PROCS :: ZSPEC1,ZSPEC2
      REAL ZSPEC1(NFLSURG,NSPEC2G)
      REAL ZSPEC2(NFLSURG,NSPEC2G)
      ...
!HPF$ DISTRIBUTE(BLOCK,*) ONTO PROCS :: ZREEL1,ZREEL2
      REAL ZREEL1(NGPTOTG,NFLEVG)
      REAL ZREEL2(NGPTOTG,NFLEVG)
      ...
      DO I=1,NSTOP
        ...
C
C       Perform grid point to spectral space transformation
C
        CALL REESPE(NFLSURG,NFLEVG,ZSPEC1,ZREEL1,IPROC)
        ...
C
C       Perform spectral space to grid point space transformation
C
        CALL SPEREE(NFLSURG,NFLEVG,ZSPEC1,ZREEL2,IPROC)

        SPRES(I)=ZSPEC1(1,1)
        ...
      ENDDO
      ...
      END



      SUBROUTINE REESPE(KFLSUR,KFIELD,PSPEC,PREEL,KPROC)
      ...
!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ INHERIT PSPEC,PREEL,KPROC
      REAL,INTENT(OUT) :: PSPEC(:,:)
      REAL,INTENT(IN)  :: PREEL(:,:)
      INTEGER,INTENT(IN) :: KPROC(:)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),*) ONTO PROCS :: ZREEL
      REAL ZREEL(NPROMAG,KFIELD)
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPFL)) ONTO PROCS :: ZBUFL
      REAL ZBUFL(KFIELD,NFTOT2G)
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPFM)) ONTO PROCS :: ZBUFM
      REAL ZBUFM(KFIELD,NFTOT2G)

      INTERFACE
        EXTRINSIC(HPF_LOCAL) SUBROUTINE LTDIR(
     +   PSPEC,PBUFM,PNM,KFIELD,KPROC,
     +   NLED1,NLED3,NLED4,NSMAX,NTMAX,NDGNH,NDGLG,NPROCA,
     +   NDGLU,NPMT,NLTN,NASM0,NPNTGTB1G,NUMPP,MYMS,RW)
!HPF$ INHERIT PSPEC,PBUFM,PNM,KPROC
        REAL,INTENT(IN)  :: PSPEC(:,:)
        REAL,INTENT(OUT) :: PBUFM(:,:)
        REAL,INTENT(IN)  :: PNM(:,:)
        INTEGER,INTENT(IN) :: KPROC(:)
        ...
        END SUBROUTINE LTDIR
```

```
       EXTRINSIC(HPF_LOCAL) SUBROUTINE FTDIR(
     +   PGA,KPROC,KWORK,KFIELDS,NPTRLS,NULTPP,
     +   NPROC,NPROCA,NPROMAG,NDLON,NDGLG,NDGSAG,NDGENG,
     +   NMENG,NLOENG,NCIST,NFAX,NSTABUF,TRIGS)
       REAL,INTENT(INOUT) :: PGA(:,:)
       INTEGER,INTENT(IN) :: KPROC(:)
!HPF$ INHERIT PGA,KPROC
       ...
       END SUBROUTINE FTDIR
       ...
      END INTERFACE
C
C  Transpose grid column data to longitudinal data for the FFT
C
!HPF$ INDEPENDENT,NEW(JFLD),ONHOME(ZREEL(NPNTL(J),:)), REUSE
      DO J=1,NGPTOTG
       DO JFLD=1,KFIELD
        ZREEL(NPNTL(J),JFLD)=PREEL(J,JFLD)
       ENDDO
      ENDDO
C
C     Direct Fourier Transforms.
C
      CALL FTDIR(
     +  ZREEL,KPROC,IWORK,KFIELD,NPTRLS,NULTPP,
     +  NPROC,NPROCA,NPROMAG,NDLON,NDGLG,NDGSAG,NDGENG,
     +  NMENG,NLOENG,NCIST,NFAX,NSTABUF,TRIGS)
C
C     Transfer Fourier data from zreel to zbufl
C
!HPF$ INDEPENDENT, NEW(JF), ONHOME(ZBUFL(:,NPNT0(J))), REUSE
      DO J=1,NFTOTG
       DO JF=1,KFIELD
        ZBUFL(JF,NPNT0(J)  ) =ZREEL(NPNTM(J)  ,JF)
        ZBUFL(JF,NPNT0(J)+1) =ZREEL(NPNTM(J)+1,JF)
       ENDDO
      ENDDO
C
C     Transpose Fourier space from partitioning over latitudes to
C     partitioning over wave numbers
C
!HPF$ INDEPENDENT,NEW(JF),ONHOME(ZBUFM(:,NPNT1(J))), REUSE
      DO J=1,NFTOTG
       DO JF=1,KFIELD
        ZBUFM(JF,NPNT1(J)  )=ZBUFL(JF,NPNT0(J)  )
        ZBUFM(JF,NPNT1(J)+1)=ZBUFL(JF,NPNT0(J)+1)
       ENDDO
      ENDDO
C
C     Legendre Transforms.
C
      CALL LTDIR(
     +  PSPEC,ZBUFM,RPNM,KFIELD,KPROC,
     +  NLED1,NLED3,NLED4,NSMAX,NTMAX,NDGNH,NDGLG,NPROCA,
     +  NDGLU,NPMT,NLTN,NASM0,NPNTGTB1G,NUMPP,MYMS,RW)
      RETURN
      END


      EXTRINSIC(HPF_LOCAL) SUBROUTINE FTDIR(
     + PGA,KPROC,KWORK,KFIELDS,NPTRLS,NULTPP,
     + NPROC,NPROCA,NPROMAG,NDLON,NDGLG,NDGSAG,NDGENG,
     + NMENG,NLOENG,NCIST,NFAX,NSTABUF,TRIGS)
      ...
      REAL,INTENT(INOUT) :: PGA(:,:)
      INTEGER,INTENT(IN) :: KPROC(:)
!HPF$ INHERIT PGA,KPROC
```

```
      ...
      ZWORK=0.
      MYPROC=KPROC(1)
      MYSETA=MOD(MYPROC-1,NPROCA)+1
      ILAT1=NPTRLS(MYSETA)
      ILAT2=ILAT1+NULTPP(MYSETA)-1
      IBEGR=1
      DO J=1,MYPROC-1
        IBEGR=IBEGR+MAPGLA(J)
      ENDDO
      IJUMP=NDLON+2
      DO JLAT=ILAT1,ILAT2
        DO JFLD=1,KFIELDS
          DO J=1,NLOENG(JLAT)
            ZGA(J+IJUMP*(JFLD-1))=PGA(NCIST(JLAT)+J-IBEGR,JFLD)
          ENDDO
        ENDDO
        CALL FFT991(ZGA,ZWORK,TRIGS(:,JLAT),NFAX(:,JLAT),
     &             1,IJUMP,NLOENG(JLAT),KFIELDS,-1)
        DO JFLD=1,KFIELDS
          DO J=1,MIN(NLOENG(JLAT),2*(NMENG(JLAT)+1))
            PGA(NCIST(JLAT)+J-IBEGR,JFLD)=ZGA(J+IJUMP*(JFLD-1))
          ENDDO
        ENDDO
        DO JFLD=1,KFIELDS
          DO J=MIN(NLOENG(JLAT)+1,2*(NMENG(JLAT)+1)+1),NLOENG(JLAT)+2
            PGA(J+NSTABUF(JLAT)-IBEGR+1,JFLD)=0.
          ENDDO
        ENDDO
      ENDDO
      RETURN
      END


      EXTRINSIC(HPF_LOCAL) SUBROUTINE LTDIR(
     +  PSPEC,PBUFM,PNM,KFIELD,KPROC,
     +  NLED1,NLED3,NLED4,NSMAX,NTMAX,NDGNH,NDGLG,NPROCA,
     +  NDGLU,NPMT,NLTN,NASM0,NPNTGTB1G,NUMPP,MYMS,RW)
      ...
!HPF$ INHERIT PSPEC,PBUFM,PNM,KPROC
      REAL,INTENT(OUT)   :: PSPEC(:,:)
      REAL,INTENT(IN)    :: PBUFM(:,:)
      REAL,INTENT(IN)    :: PNM(:,:)
      INTEGER,INTENT(IN) :: KPROC(:)
      ...
      REAL ZLEPO(NLED3,0:NLED1)
      REAL ZSIA(0:NLED1,2*KFIELD)
      REAL ZAIA(0:NLED1,2*KFIELD)
      REAL ZOA(NLED4,2*KFIELD)
      ...
      MYPROC=KPROC(1)
      MYSETA=MOD(MYPROC-1,NPROCA)+1
      IBEGS=1
      IBEGB=1
      IBEGP=1
      DO J=1,MYPROC-1
        IBEGS=IBEGS+MAPSP(J)
        IBEGB=IBEGB+MAPFM(J)
        IBEGP=IBEGP+MAPLP(J)
      ENDDO
      ...
      DO IM=1,NUMPP(MYSETA)
        JM=MYMS(IM,MYSETA)
        ISL=NDGNH-NDGLU(JM)+1
        ISL=MAX(ISL,1)
        DO JN=1,NTMAX-JM+2
          DO JGL=ISL,NDGNH
```

```
        ZLEPO(JN,JGL)=PNM(JGL,NPMT(JM)+JN-IBEGP+1)*RW(JGL)
      ENDDO
    ENDDO
    DO JGL=ISL,NDGNH
      IGLS=NDGLG+1-JGL
      ISTAN=NPNTGTB1G(JM,JGL )*2+1-IBEGB+1
      ISTAS=NPNTGTB1G(JM,IGLS)*2+1-IBEGB+1
      DO JF=1,KFIELD
        IJR=2*(JF-1)+1
        IJI=IJR+1
        ZAIA(JGL,IJR)=PBUFM(JF,ISTAN  )-PBUFM(JF,ISTAS  )
        ZAIA(JGL,IJI)=PBUFM(JF,ISTAN+1)-PBUFM(JF,ISTAS+1)
        ZSIA(JGL,IJR)=PBUFM(JF,ISTAN  )+PBUFM(JF,ISTAS  )
        ZSIA(JGL,IJI)=PBUFM(JF,ISTAN+1)+PBUFM(JF,ISTAS+1)
      ENDDO
    ENDDO
    IFC=2*KFIELD
    CALL LEDIR(JM,IFC,ZAIA,ZSIA,ZOA,ZLEPO,
   +   NSMAX,NTMAX,NDGNH,NLED1,NLED3,NLED4,NDGLU)
    IF(JM.EQ.0) THEN
      DO JN=NLTN(NSMAX),NLTN(JM)
        INM=NASM0(JM)+NLTN(JN)*2-IBEGS+1
        DO JFLD=1,KFIELD
          IR=2*(JFLD-1)+1
          PSPEC(JFLD,INM)  =ZOA(JN,IR)
          PSPEC(JFLD,INM+1) =0.0
        ENDDO
      ENDDO
    ELSE
      DO JN=NLTN(NSMAX),NLTN(JM)
        INM=NASM0(JM)+(NLTN(JN)-JM)*2-IBEGS+1
        DO JFLD=1,KFIELD
          IR=2*(JFLD-1)+1
          II=IR+1
          PSPEC(JFLD,INM)  =ZOA(JN,IR)
          PSPEC(JFLD,INM+1)=ZOA(JN,II)
        ENDDO
      ENDDO
    ENDIF
  ENDDO
  RETURN
  END


  MODULE YOMLEG
  ...
  REAL,ALLOCATABLE:: RPNM(:,:)
!HPF$ PROCESSORS PROCSL(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,BLOCK) ONTO PROCSL ,DYNAMIC :: RPNM
  ...
  END MODULE YOMLEG


  SUBROUTINE SUINIT
  ...
!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(BLOCK,*) ONTO PROCS :: ZPNM1
  ...
!HPF$ INDEPENDENT,NEW(JGL),ON HOME(RPNM(:,J))
  DO J=1,NSPOLEG
    DO JGL=1,NLEI3
      RPNM(JGL,J)=ZPNM3(JGL,J)
    ENDDO
  ENDDO
  ...
  CALL SUCOPYLE(ZPNM1,RPNM)
  ...
```

```
!HPF$ REDISTRIBUTE(*,GEN_BLOCK(MAPLP)) ONTO PROCS :: RPNM
      ...
      RETURN
      END
```

6.2.2 LG kernel

```
      SUBROUTINE STEPO1          ! 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,ONHOME(IPROC(I))
      DO I=1,NPROC
        IPROC(I)=I
      ENDDO
C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT,NEW(JFLD),ONHOME(ZGA(J,:))
      DO J=1,NGPTOTG
        DO JFLD=1,NGT0
          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(JFLD),ONHOME(ZGL(INDL(J),:)),REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JFLD=1,NGT0
            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,NGT0,NPROCA,NPROCB,NDGSAG,NDGENG,
     +    NDLON,NFLD0,NPTRLS,NULTPP,NLOENG,NSTABUFL)
C
C       Perform LTOG transposition
C
!HPF$ INDEPENDENT,NEW(JFLD),ONHOME(ZGB(J,:)),REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JFLD=1,NGT0
            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
      ...
      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 STEPO3          ! LG VARIANT 3
    ...
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK,*) ONTO PROCSAB :: ZGL3,ZGL3X
    REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
    REAL ZGL3X(NPROMAG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK) ONTO PROCSAB :: ZGL2,ZGL2X
    REAL ZGL2(NPROMAG,NG2D0)
    REAL ZGL2X(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$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: T
        REAL,INTENT(INOUT) :: T(:,:)
        END SUBROUTINE SIMPHYS_H
      END INTERFACE

C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT,NEW(JVAR,JLEV),ONHOME(ZGA3(J,:,:))
      DO J=1,NGPTOTG
        DO JVAR=1,NG3D0
          DO JLEV=1,NFLEVG
            ZGA3(J,JLEV,JVAR)=FLOAT(((JVAR-1)*NFLEVG+JLEV)*NGPTOTG+J)
          ENDDO
        ENDDO
      ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),ONHOME(ZGA2(J,:))
      DO J=1,NGPTOTG
        DO JVAR=1,NG2D0
          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,JVAR),ONHOME(ZGL3(INDL(J),:,:)),REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JLEV=1,NFLEVG
            DO JVAR=1,NG3D0
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),ONHOME(ZGL2(INDL(J),:)),REUSE(LREUSE)
        DO J=1,NGPTOTG
          DO JVAR=1,NG2D0
            ZGL2(INDL(J),JVAR)=ZGA2(J,JVAR)
          ENDDO
        ENDDO
C
C       Simulate call to FFT package (E-W dependencies on
C       each latitude)
C
```

```
      ZGL3X=ZGL3
      ZGL2X=ZGL2
      DO JVAR=1,NG3D0
        DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(J,ZL),REUSE(LREUSE),
!HPF$+ONHOME(ZGL3(NSTABUFL(JGL),JLEV,JVAR))
          DO JGL=1,NDGLG
            DO J=1,NDLON
              IF( J.LE.NLOENG(JGL) )THEN
                ZL(J)=ZGL3X(NSTABUFL(JGL)+J,JLEV,JVAR)
              ENDIF
            ENDDO
            CALL SIMFFT_P(ZL,NDLON,NLOENG(JGL))
            IF( LUPDFFT )THEN
              DO J=1,NDLON
                IF( J.LE.NLOENG(JGL) )THEN
                  ZGL3(NSTABUFL(JGL)+J,JLEV,JVAR)=ZL(J)
                ENDIF
              ENDDO
            ENDIF
          ENDDO
        ENDDO
      ENDDO
      DO JVAR=1,NG2D0
!HPF$ INDEPENDENT,NEW(J,ZL),REUSE(LREUSE),
!HPF$+ONHOME(ZGL2(NSTABUFL(JGL),JVAR))
        DO JGL=1,NDGLG
          DO J=1,NDLON
            IF( J.LE.NLOENG(JGL) )THEN
              ZL(J)=ZGL2X(NSTABUFL(JGL)+J,JVAR)
            ENDIF
          ENDDO
          CALL SIMFFT_P(ZL,NDLON,NLOENG(JGL))
          IF( LUPDFFT )THEN
            DO J=1,NDLON
              IF( J.LE.NLOENG(JGL) )THEN
                ZGL2(NSTABUFL(JGL)+J,JVAR)=ZL(J)
              ENDIF
            ENDDO
          ENDIF
        ENDDO
      ENDDO
C
C       Perform LTOG transposition
C
!HPF$ INDEPENDENT,NEW(JLEV,JVAR),ONHOME(ZGB3(J,:,:)),REUSE(LREUSE)
      DO J=1,NGPTOTG
        DO JLEV=1,NFLEVG
          DO JVAR=1,NG3D0
            ZGB3(J,JLEV,JVAR)=ZGL3(INDL(J),JLEV,JVAR)
          ENDDO
        ENDDO
      ENDDO
!HPF$ INDEPENDENT,NEW(JVAR),ONHOME(ZGB2(J,:)),REUSE(LREUSE)
      DO J=1,NGPTOTG
        DO JVAR=1,NG2D0
          ZGB2(J,JVAR)=ZGL2(INDL(J),JVAR)
        ENDDO
      ENDDO
C
C       Simulate call to physics package (vertical dependencies)
C
      ZT(:,:)=ZGB3(:,:,1)
      CALL SIMPHYS_H(ZT)
      ZGB3(:,:,1)=ZT(:,:)
      IF( LUPDFFT.OR.LUPDPHYS )THEN
        ZGA3=ZGB3
```

```
      ZGA2=ZGB2
    ENDIF
    RES(I)=ZGB3(1,1,1)
  ENDDO
  ...
  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$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: 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
  U(:,:,:)=0.
  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
```

6.2.3 SL kernel

```
      SUBROUTINE STEPO
      ...
      USE YOMHAL
      ...
!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP),*), HALO(NSLHALO(I)%INDEX,*)
!HPF$+ ON PROCS(I) :: ZSL1
      REAL ZSL1(NGPTOTG,NFLDSLB1)
!HPF$ 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(JFLD),ONHOME(ZSL1(J,:))
      DO J=1,NGPTOTG
        DO JFLD=1,NFLDSLB1
          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
!HPF$ UPDATE_HALO :: ZSL1
C
C       Simulate SL interpolation
C
        ISL=NSLWIDE/2
        DO JSL=1,ISL
!HPF$ INDEPENDENT,NEW(JLEV,Z1,Z2,Z3,Z4,Z5,Z6),ON HOME(ZSL2(J,:)),
!HPF$+ REDUCTION(ZSL2)
          DO J=1,NGPTOTG
            DO JLEV=1,NFLEVG
              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 YOMHAL
      TYPE PHALO
        INTEGER, POINTER     :: INDEX(:)
      END TYPE PHALO
      TYPE (PHALO),POINTER :: NSLHALO(:)
      END MODULE YOMHAL


      SUBROUTINE SLCSET
      ...
      USE YOMHAL
      ...
      DO  JA=1,NPROCA
        DO  JB=1,NPROCB
          IPROC=(JB-1)*NPROCA+JA
          ...
          ALLOCATE(NSLHALO(IPROC)%INDEX(IRLOAD))
          DO JVAL=1,IRLOAD
            NSLHALO(IPROC)%INDEX(JVAL)=NGPOSG(nrlstlon(jval),nrlstlat(jval))
          ENDDO
          ...
        ENDDO
      ENDDO
      ...
      RETURN
      END
```

6.3 ESI HPF+ kernels

```
program pc
C
C      03/98-SM :  HPF+ PROJECT
C                  BENCHMARK REL.K4.4
C                  WITHOUT CONTACT
C                  ADAPTED FOR VFC ONLY.
C
      use p_halo
C
      include 'static.h'
C
C < compute the HALO . >
      CALL pcrash(MAPELM,MAPNOD,BHALO)
C
      end program pc
C-----------------------------------------------------------
      subroutine pcrash(MAPELM,MAPNOD,BHALO)
C
C
      use p_halo
C
C dummy arrays
      INTEGER MAPELM(:),MAPNOD(:)
      TYPE (HAL_TYP), POINTER, DIMENSION(:) ::  BHALO
C
C-----------------------------------------------------------
C HPF DISTRIBUTE DIRECTIVES
C ----------------------------------------------------------
C
!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS())
C
C    elements
C
!HPF$ DYNAMIC,DISTRIBUTE(BLOCK) ONTO R:: INDEX
!HPF$ DYNAMIC :: ROEL,DT2AR
!HPF$ DYNAMIC :: IY,STRS
!HPF$ DYNAMIC :: FORCE1,FORCE2,FORCE3,FORCE4
C
!HPF$ ALIGN WITH INDEX          :: ROEL,DT2AR
!HPF$ ALIGN (*,I) WITH INDEX(I) :: IY,STRS,FORCE1,FORCE2,FORCE3,FORCE4
C
!HPF$ RANGE(BLOCK)   :: INDEX,ROEL,DT2AR
!HPF$ RANGE(*,BLOCK) :: IY,STRS
!HPF$ RANGE(*,BLOCK) :: FORCE1,FORCE2,FORCE3,FORCE4
C
C     nodes
C
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R,
!HPF$& HALO(*,BHALO(I)%INDHALO) ON R(I) :: XYZC, X, VGLOB
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R,
!HPF$& HALO(*,BHALO(I)%INDHALO) ON R(I) :: X1,XMSI,V,A,XMS
!HPF$ DISTRIBUTE(*,BLOCK) ONTO R,
!HPF$& HALO(*,BHALO(I)%INDHALO) ON R(I) :: BLAMB
C
< read the data-set >
C
C
```

```
C end of initialization
C
C                       Time loop
C
100   CONTINUE
C
      CALL SYSTEM_CLOCK(ICOUNT0,ICOUNT_RATE,ICOUNT_MAX)
C
        DT1 = DT2
        DT2 = 1.0E+6
C
C     Zero A
C
      A = 0.0
C
C*******************************************************
C                                                      *
C     INTERNAL AND EXTERNAL FORCES                     *
C                                                      *
C*******************************************************
C
C     Internal forces of thin plate elements
C
         CALL SFORCE(INDEX,IY,STRS,ROEL,DT2AR,
     .                  FORCE1,FORCE2,FORCE3,FORCE4,
     .                  XYZC,X,V,VGLOB,A,
     .                  CM,
     .                  ECNTR,DT1,NSTR,NPROP,N4NODE,NDOF,DT2)
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
        CALL CONTACT(X   ,IRECT,NSV  ,LSBUCK,IBUCKT,
     .              XVAL,
     .              DIRN  ,NPENO,
     .              NPENP ,IRTLO,IRTL,IORIEN,IORI,DIST1,A,
     .              XYZMIN,XYZMAX,
     .              LSEM  ,AREA ,NRB,NRE,XMS,
     .              THKSV,NSN,NRT,MAXCT,
     .              NUMBCK,ICOUN,IACCS,NBPTT,NDOF,TISTP,
     .              IWRK1,LSEMO,IBSEG,IMNOD,IMAUX,MXNRT,
     .              INDEX,XYZC,MAX36,NCYCLE)
C
      END IF
C
C

!HPF$ INDEPENDENT, NEW(AX,AY,AZ),ON HOME(A(:,N)),RESIDENT,REUSE
  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
  NCYCLE=NCYCLE+1
C
C     COMPUTE ACCELERATIONS                            *
```

```
C
!HPF$ INDEPENDENT,ON HOME(A(:,N)),RESIDENT,REUSE
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
C
      A = A*XMSI
C
      DTX=0.5*(DT1+DT2)
C
C     Apply displacement boundary conditions
C
C
C     Write output,plot,time history
C
      CALL PAMOUT(X,V,A,NUMNP,NDOF,TT)
C
C     VELOCITY (n+1/2) AND DISPLACEMENTS (n+1)         *
C
      V = V + DTX*A
      X1= X1+ DT2*V
C
C     Update geometry
C
       X = XYZC + X1(1:3,:)
C
  IF (NCYCLE.GT.MAXCYC) GOTO 99
C
      CALL SYSTEM_CLOCK(ICOUNT1,ICOUNT_RATE,ICOUNT_MAX)
      ICOUTT = ICOUTT + ICOUNT1 - ICOUNT0
C
      GOTO 100
C
   99 CONTINUE
C
      WRITE(6,*) ' NPROC=',NUMBER_OF_PROCESSORS()
      WRITE(6,*) 'NB CYCLE=',NCYCLE
      WRITE(6,*) 'TIMEUSED=',TIMEUSED
C
END subroutine pcrash
C
C--------------------------------------------------------------------
      SUBROUTINE SFORCE(INDEX,IY,STRS,ROEL,DT2AR,
     .                  FORCE1,FORCE2,FORCE3,FORCE4,
     .                  XYZC,X,V,VGLOB,A,
     .                  CM,
     .                  ECNTR,DT1,NSTR,NPROP,N4NODE,NDOF,
     .                  DT2)
C
      INTEGER,DIMENSION(:)     ,INTENT(IN)    :: INDEX
      INTEGER,DIMENSION(:,:)   ,INTENT(IN)     :: IY
      REAL   ,DIMENSION(:,:)   ,INTENT(INOUT):: STRS
      REAL   ,DIMENSION(:)     ,INTENT(INOUT):: ROEL,DT2AR
      REAL   ,DIMENSION(:,:),INTENT(INOUT)::FORCE1,FORCE2,FORCE3,FORCE4
      REAL   ,DIMENSION(:,:)   ,INTENT(IN)    :: XYZC
      REAL   ,DIMENSION(:,:)   ,INTENT(INOUT) :: X,V,VGLOB
      REAL   ,DIMENSION(:,:)   ,INTENT(INOUT):: A
      REAL   ,DIMENSION(:,:)   ,INTENT(IN)    :: CM
      REAL   ,INTENT(INOUT) :: ECNTR,DT1,DT2
```

```
      INTEGER NSTR,NPROP,N4NODE,NDOF
C
!HPF$ INHERIT,RANGE(BLOCK)    :: INDEX,ROEL,DT2AR
!HPF$ INHERIT,RANGE(*,BLOCK) :: IY,STRS,XYZC,X,V,VGLOB,A
!HPF$ INHERIT,RANGE(*,BLOCK) :: FORCE1,FORCE2,FORCE3,FORCE4
C     elements
!!HPF$ DISTRIBUTE(BLOCK)            :: INDEX
!!HPF$ RANGE (BLOCK) :: INDEX
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
C     Update the HALO
C
!HPF$ UPDATE_HALO :: X,V,VGLOB
C
!HPF$ INDEPENDENT, ON HOME(IY(:,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$& DT2AUX,DT1AUX,ILMAUX,ITYAUX,ECNT),REDUCTION(DT2),
!HPF$& RESIDENT(X,V,VGLOB),REUSE
DO I = 1, N4NODE
        ECNT   = ECNTR
        DT2AUX = DT2
        IL(7) = NPROP*(IY(1,I)-1)
        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
        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)
        DT2 = MIN(DT2,DT2AUX)
C
C <…>
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(:,N)),NEW(I),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
      RETURN
      END subroutine sforce
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     TT : be careful TT is a new variable in this subroutine
C
      RETURN
      END subroutine forcei

C-----------------------------------------------------------------------
C
      SUBROUTINE CONTACT(X    ,IRECT,NSV ,LSBUCK,IBUCKT,
     .                  XVAL  ,
     .                  DIRN,NPENO,
     .                  NPENP,IRTLO,IRTL,IORIEN,IORI,DIST1,A,
     .                  XYZMIN,XYZMAX,
     .                  LSEM ,AREA ,NRB,NRE,XMS,
     .                  THKSV,NSN,NRT,MAXCT,
     .                  NUMBCK,ICOUN,IACCS,NBPTT,NDOF,TISTP,
     .                  IWRK1,LSEMO,IBSEG,IMNOD,IMAUX,MXNRT,
     .                  INDEX,XYZC,MAX36,NCYCLE)
C
C     (03/97-SM) HPF+ Rel V2.0
C     (07/97-SM) Updated (Version 2.0)
C     (01/98-SM) Updated for VFC restrictions
C     (03/98-SM) Updated for VFC 1.0
!HPF$ INHERIT, RANGE(BLOCK) ::  INDEX
!HPF$ INHERIT, RANGE(*,BLOCK) ::  XYZC
!HPF$ INHERIT, RANGE(*,BLOCK) ::  X,A,XMS
!!HPF$ ALIGN (*,I) WITH XYZC(*,I) :: A,X,XMS
!HPF$ INHERIT, RANGE(BLOCK)    :: LSEMO
!HPF$ INHERIT, RANGE(BLOCK)    :: NPENP
!HPF$ INHERIT, RANGE(*,BLOCK)  :: XYZMIN,XYZMAX
!HPF$ INHERIT, RANGE(*,BLOCK)  :: IRECT,NRB,NRE,IBSEG,IMNOD
!HPF$ INHERIT, RANGE(*,*,BLOCK):: DIRN

!HPF$ INHERIT, RANGE(*,BLOCK)  :: IORIEN,IORI,AREA
!HPF$ INHERIT, RANGE(*,*,BLOCK):: DIST1
!HPF$ INHERIT, RANGE(*,BLOCK) :: X
```

```
C
!HPF$ INHERIT, RANGE(BLOCK)     :: LSEMO
!HPF$ INHERIT, RANGE(*,BLOCK)   :: IRECT,IMNOD
!HPF$ INHERIT, RANGE(*,*,BLOCK):: DIRN
!HPF$ INHERIT, RANGE(*,BLOCK)   :: IORI,AREA
!HPF$ INHERIT, RANGE(*,*,BLOCK):: DIST1
C
C bucket sort
cB    CALL SLBUCK(X,NSV,LSBUCK,IBUCKT,IWRK1,THK,NSN,NUMBCK,NBUCKX,
cB   .            NBUCKY,NBUCKZ,BUBEGX,BUSTPX,BUBEGY,BUSTPY,
cB   .            BUBEGZ,BUSTPZ)
C
!HPF$ INDEPENDENT, ON HOME(X(:,J)),REUSE(LCOND)
      DO J=1,NSN
       XVAL(1,J) = X(1,NSV(J))
       XVAL(2,J) = X(2,NSV(J))
       XVAL(3,J) = X(3,NSV(J))
      END DO
C
C     Loop on slave nodes
C
              CALL  SLA(NBPTT  ,X      ,IRECT  ,NSV  ,NSN    ,XYZC, …)
<...>
C
C
!HPF$ INDEPENDENT,
!HPF$& NEW(XSNX,XL,XNORM1,YNORM1,ZNORM1,XNORM2,YNORM2,ZNORM2,
!HPF$& AREA1,AREA2,RDIST,ICODE,DIRN1,XMAS1,XMASI,
!HPF$& IAUX,FORCES,FORCEM,THK0,TIST0,ICONT),REUSE(LCOND)
      DO II=1,NRT
      DO ICONT=1,MAX36
      IF (ICONT.LE.LSEMO(II)) THEN
        XSNX(1)= X(1,NSV(IMNOD(ICONT,II)))
        XSNX(2)= X(2,NSV(IMNOD(ICONT,II)))
        XSNX(3)= X(3,NSV(IMNOD(ICONT,II)))
C
        XL(1,1) = X(1,IRECT(1,II))
        XL(2,1) = X(2,IRECT(1,II))
        XL(3,1) = X(3,IRECT(1,II))
        XL(1,2) = X(1,IRECT(2,II))
        XL(2,2) = X(2,IRECT(2,II))
C
C       purest procedure : compute forces to apply on the contact pair
C
          CALL SELF(THK0  ,XMAS1   ,XMASI  ,DIRN1 ,… )

        A(1,NSV(IMNOD(ICONT,II))) = A(1,NSV(IMNOD(ICONT,II)))+FORCES(1)
        A(2,NSV(IMNOD(ICONT,II))) = A(2,NSV(IMNOD(ICONT,II)))+FORCES(2)
        A(3,NSV(IMNOD(ICONT,II))) = A(3,NSV(IMNOD(ICONT,II)))+FORCES(3)
<…>
        A(1,NSV(IMNOD(ICONT,II))) = A(1,NSV(IMNOD(ICONT,II)))+FORCES(1)
        A(2,NSV(IMNOD(ICONT,II))) = A(2,NSV(IMNOD(ICONT,II)))+FORCES(2)
        A(3,NSV(IMNOD(ICONT,II))) = A(3,NSV(IMNOD(ICONT,II)))+FORCES(3)
<…>
        END IF
        END DO
       END DO
```

```
C-----------------------------------------------------------------------
C
      SUBROUTINE SLA(NBPTT  ,X      ,IRECT  ,NSV    , …)
C
C      03/97-SM :  HPF+ Project
C                  Search-box routine for VFCS only
C                  Version 2

!HPF$ INHERIT, RANGE(*,BLOCK) :: XYZC
!HPF$ INHERIT, RANGE(*,BLOCK) :: X
!HPF$ INHERIT, RANGE(BLOCK) :: NPENP
!!!HPF$ ALIGN (*,I) WITH XYZC(*,I) :: X
C
!HPF$ INHERIT, RANGE(BLOCK)     :: LSEMO
!HPF$ INHERIT, RANGE(*,BLOCK)   :: XYZMIN,XYZMAX
!HPF$ INHERIT, RANGE(*,BLOCK)   :: IRECT,NRB,NRE,IBSEG,IMNOD
!HPF$ INHERIT, RANGE(*,*,BLOCK) :: DIRN
!HPF$ INHERIT, RANGE(*,BLOCK)   :: IORIEN,IORI,AREA

C     * Determine search box,normals and areas for each segment
C     -------------------------------------------------------
C
!HPF$ INDEPENDENT, ON HOME(IRECT(:,II)),REUSE
      DO II=1,NRT
        XYZMAX(1,II) = MAX(X(1,IRECT(1,II)),X(1,IRECT(2,II)),
     &                 X(1,IRECT(3,II)),X(1,IRECT(4,II))) + THK
        XYZMAX(2,II) = MAX(X(2,IRECT(1,II)),X(2,IRECT(2,II)),
     &                 X(2,IRECT(3,II)),X(2,IRECT(4,II))) + THK
        XYZMAX(3,II) = MAX(X(3,IRECT(1,II)),X(3,IRECT(2,II)),
     &                 X(3,IRECT(3,II)),X(3,IRECT(4,II))) + THK
        XYZMIN(1,II) = MIN(X(1,IRECT(1,II)),X(1,IRECT(2,II)),
     &                 X(1,IRECT(3,II)),X(1,IRECT(4,II))) - THK
        XYZMIN(2,II) = MIN(X(2,IRECT(1,II)),X(2,IRECT(2,II)),
     &                 X(2,IRECT(3,II)),X(2,IRECT(4,II))) - THK
        XYZMIN(3,II) = MIN(X(3,IRECT(1,II)),X(3,IRECT(2,II)),
     &                 X(3,IRECT(3,II)),X(3,IRECT(4,II))) - THK
      END DO
<…>
C     *****************************************************************
C  * loop over segments to determine candidates
*******************************************************************
      NBPTT =0
!HPF$ INDEPENDENT,NEW(NBEG,NEND,J,DISTV1,DISTV2,DISTV,NMNOD),
!HPF$& REDUCTION(NBPTT),REUSE,ON HOME(IRECT(:,II))
      DO II=1,NRT
c       NBEG = IBSEG(1,II)      ! bounds previously calculated
c       NEND = IBSEG(2,II) - 1
        NMNOD = 0
        DO J=1,NSN
          IF (<…>) THEN
            NMNOD = NMNOD+1
            IMNOD(NMNOD,II) = J
          END IF
         END IF
        END DO
        LSEMO(II) = NMNOD
        NBPTT     = NBPTT + NMNOD
      END DO
    RETURN
END subroutine sla
```