Report on Evaluation Results with Benchmark Programs Vers.1

Deliverable:	D1.2a
Date:	December 1996
Status:	draft
Confidentiality:	public
Responsible:	ECMWF
Revised:	January/7/1997

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

Abstract

This report constitutes deliverable D1.2a of the Esprit project "HPF+ -Optimizing HPF for Advanced Applications" Project No. 21033. In this report we provide an evaluation of HPF+ and its implementation, based on version 1 of the project benchmarks, supplied by the application partners at project month 6. The benchmark programs are evaluated by a number of criteria, such as, object code performance, language expressivity, coding effort, and maintainability. This is done mainly by comparing HPF+ versions of the benchmark programs with their message passing equivalents, and where possible HPF-1 versions. In addition to the performance evaluations by the respective application partners, the University of Pavia provide a complementary evaluation of these benchmarks by using the MEDEA tool.

Contents

1 AVL

1.1 Introduction

The current version of the AVL kernel is a part of the FIRE conjugate gradient equation solver, which can be applied to structured and unstructured meshes. The method is designed for iterative sparse matrix computations whereby non zero elements only occur in the six bands and in the main diagonal.
The data structure within the FIRE program is based on three different types of arrays: cell center based, cell vertex based and cell face based arrays. The computations done in the benchmark requires cell center based arrays that are dimensioned either over the number of internal cells or over the number of internal and external cells (e.g. DIREC1). The position of the elements is obtained from the indirect addressing array LCC(1:6,0:NNINTC), which is part of the input data set. The LCC array reflects the geometrical connection of the computational cells in physical space, where the second index refers to the cell number and the first index refers to the direction in the mesh.

1.2 HPF+ kernel

All required arrays are dimensioned in a common block, either over the number of internal cells (NNINTC) or over the total number of cells (NNCELL). The values for all external cells are initialised by zero, so that a simple block distribution over the number of internal cells could be used to distribute the data. All required arrays (conectivity array LCC, coefficients and sources) are read in from the input dataset, that has been generated previously with the FIRE program.
In the Benchmark two different communication concepts are tested: sum reduction operations (e.g. the computation of the equation residual) where all processors are involved in the communication simultaneously. Global sums have to be computeted frequently in the equation solver, so their fast and efficient implementation is an important issue for the FIRE kernel.
When the matrix vector product is performed, components of the direction vector DIREC1 can be situated on different processors, so that communication between these processors is necessary, which is the second form of communication in the benchmark. In HPF+ a schedule for the array DIREC1 is computed, which is reused during the calculation of the matrix vector product. For the message passing version this asynchronuous local data exchange must be coded in two seperate routines.
At the end of the benchmark a second sum reduction operation is performed.

1.2.1 HPF+ features used

For the current version of the kernel only the internal cells have non-zero values, therefore a block data distribution of all arrays could be used. For the porting of the entire equation solver where also the external cells have non-zero values, it will be necessary to use an indirect data distribution by using a map array. The array SU has been distributed to the processors, while for all other arrays the distribution has been obtained by an alignment to the array SU. The schedule/reuse concept of HPF+ is essential for the computation of the matrix vector

product, because of the indirect access to elements of array DIREC1 that can be situated on different processors.
All do loops that can be computed locally on each processor without data exchange have been marked with the 'INDEPENDENT' statement.
For the computation of the two global sums the 'REDUCTION' statement was used.

1.2.2 Porting Effort

For the porting of the HPF kernel parts of the FIRE conjugate gradient solver have been extracted from the serial version, a common block that contains all required arrays has been setup and the apropriate HPF+ directives have been inserted into the kernel.

1.2.3 HPF-1 version

Because of its irregular structure the AVL kernel can not be coded in HPF-1. Nevertheless, some experiments with HPF-1 compilers have been performed, but the generated code could not be run.

1.2.4 Message Passing version

In order to have a comparison of the performance of the HPF+ Benchmark a message passing version has been ported under MPI.
For the MPI version it was necessary to introduce two additional routines to obtain correct results when using the indirect addressing of the cells during the calculation. The routine cspgau.f computes the send and reveive lists required for the local data exchange. This routine is called once, after the input of the dataset. The routine cmpxchg.f performs the local data exchange and is called previous to the computation of the direction vector (DIREC1).
With respect to the coding of these routines the porting effort of the MPI version was higher compared to the HPF+ version, where the necessary communication structures are performed invisible to the user by the VFCS.
The advantage of the MPI version is that the number of processors used during the execution need not be known at compile time.

1.3 Performance

The performance of the HPf+ and the MPI kernel have been tested with three datasets of different size. The datasets have been extracted from regular FIRE applications and contain the size of the problem the two-dimensional connectivity array (LCC), the coefficients of the continuity equation and the source term.

Dataset	No. of internal cells
TJUNC	13845
SWIRL	47312
PENT	108000

For the time measurement of the benchmark the routine syswclock.c has been implemented, which determines the wall clock time by calling the function gettimeofday. The time measurements have been done on the Meiko CS2 on dedicated nodes. All measured times are in seconds and refer to the pure computational part of the kernel, but they do not contain the time required to read in all arrays from the input file. For the message passing version also the calls to the routines cspgau.f and cmpxchg.f are included in the measured time.

The following time measurements have been obtained with the HPF+ Version of the Benchmark:

Dataset	1p	2p	4p	8p	16p	32p
TJUNC	-	0.71	0.55	0.47	0.34	0.34
SWIRL	-	1.87	1.55	1.22	0.82	0.73
PENT	-	3.97	3.14	1.95	1.23	0.88

The following time measurements have been obtained with the MPI Version of the Benchmark:

Dataset	1p	2p	4p	8p	16p	32p
TJUNC	0.08	0.18	0.14	0.14	0.19	0.43
SWIRL	0.18	0.49	0.42	0.40	0.49	1.01
PENT	0.61	0.73	0.52	0.45	0.49	0.66

1.4 Discussion

The residuals computed with the HPF and the MPI version are identical for all different datasets and for all number of processors that have been tested. The HPF+ code failed to run on one single processor.

For smaller processor numbers the MPI version is about three times faster compared to the HPF+ version. When running on 32 processor and using the TJUNC dataset the HPF+ version becomes faster then the MPI version.

The performance degradation of the MPI version when comparing the single processor and the multi-processor results arises from the fact that the setup phase, where the send and receive lists are computed and the local data exchange is done, requires a high amount of the total execution time and is not necessary for the single-processor execution. When running on a larger number of processors the time required for the setup becomes the dominant factor during the execution of the benchmark.

The future requirements to HPF+ regarding the AVL Benchmark are the indirect distribution of the data by using a map array and a dynamic definition of general blocks. Since many arrays have the same structure in the benchmark (and also in the FIRE program) it would be important to have a possibility of using one computed schedule for many different arrays.

2 ECMWF

2.1 Introduction

The forecast model of the ECMWF Integrated Forecasted System (IFS) uses a transposition strategy to redistribute data between its major computational phases, which are the grid-point space computations, fourier transforms, legendre transforms and spectral space computations. In the course of a single time step, six transpositions are performed. These transpositions in effect contain the bulk of the inter-processor communication in the IFS and allow the computational phases to run for relatively long periods of time without any interprocess communication. The efficiency of the IFS on distributed memory systems is therefore very much dependent on the successful implementation of the transpositions. The IFS LG kernel represents two of the six transpositions, and is used to redistribute data between the grid-point space computations and fourier transforms.

The data space being redistributed is a quasi-regular (so-called reduced) grid in which the number of grid points per latitude is reduced as the poles are approached. This is represented as a two dimensional array, where the first dimension contains the horizontal domain of the reduced grid, and the second dimension the levels in the atmosphere or surface for some number of variables. The total number of grid points for some different IFS model resolutions is shown in Table 1.

IFS Spectral Resolution	Grid Points
T63	13280
T106	35718
T213	134028

Table 1: Number of Grid Points for different IFS configurations

With the grid-point computations the dependencies are in the vertical dimension (levels in the atmosphere), requiring the reduced grid to be distributed over the available processors. This is done in the IFS using a 2 dimensional (NPROCA, NPROCB) partitioning scheme where the reduced grid is partitioned in the N-S direction by NPROCA and then in the E-W direction by NPROCB, so that each partition has an equal number of grid-points, give or take one point. Two dimensional partitioning is also needed to minimise the amount of halo data required by the semi-lagrangian scheme. It should be evident that to maintain ideal static load balance for the grid-point computations that NPROCA partitioning requires some latitudes to be split between partitions.

In contrast, the data dependencies in the fourier transforms are E-W, which requires that the reduced grid be distributed in complete latitudes by NPROCA. As there are no vertical dependencies in the fourier transforms, this allows model levels to be distributed by NPROCB.

2.2 HPF kernel

The LG HPF kernel consists of two programs, a relatively complex 5K line f90 initialisation program and a short 210 line f77 HPF+ test program. The f90 initialisation program mainly contains the IFS partitioning routines for grid-point space and can be set to produce partitioning information required by HPF+ for different IFS problem sizes (e.g.T63, T213), processor configurations (NPROCA, NPROCB) and numbers of model variables. The number of 3D and 2D model variables have been set to 1 with number of model levels fixed at 19. These settings were used to be able to run the T213 problem serially on the Meiko CS2. In its general form, the HPF+ kernel requires more than one array dimension to be distributed with INDIRECT or GEN_BLOCK as shown below,

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB) = RESHAPE(PROCS)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGPA),INDIRECT(MAPFLDO)) ONTO PROCSAB :: ZGL
      REAL ZGL(NGPTOTG,NGTO)
```

This form of distribute directive is currently not supported by VFCS 1.3, which only allows one array dimension to be distributed if INDIRECT is used. However, the kernel can run in a degraded mode if NPROCB is set to 1, in which case the distribute directive can be specified as,

```
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGPA),*) ONTO PROCS :: ZGL
```

When the kernel is run in this mode, communication is only required on latitudes which are split between processor partitions and the rest of the data is moved locally within each processor.

2.2.1 HPF+ features used

The HPF+ features used in the LG kernel are the INDIRECT and GEN_BLOCK distribution directives and the INDEPENDENT do loop directive. The two-dimensional (NPROCA, NPROCB) partitioning of grid-point space can only be represented by an INDIRECT distribution. Also GEN_BLOCK distribution satisfies the requirement for expressing how complete groups of latitudes are distributed for the IFS reduced grid. The INDEPENDENT directive had to be specified on all loops containing distributed data structures, even though it could be easily determined that there was no data dependency between the iterations of those do loops.

2.2.2 Porting Effort

The HPF+ version was written from scratch so it is difficult to quantify porting effort. Nevertheless, it was still necessary to make some changes to the HPF+ version from its original serial version.

The most important of these was to switch the loop order of the key transpose loops from,

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

to the following where the innermost loop is over fields, and the outermost loop is the distributed dimension of the reduced grid.

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

Given that NGT0 is typically small (100's) and NGPTOTG very large (134,028 at T213), this transformation is both bad for vector and cache based architectures, and hopefully should not be necessary in future production HPF compilers.

Other changes to the serial code involved the initialisation of the map data structures specified in the INDIRECT and GEN_BLOCK distribution directives. While the map required for the INDIRECT distribution could be read in at execution time, this was not the case for the map used in the GEN_BLOCK distribution which had to be specified as an array constant.

It should be noted that large production applications (such as the IFS) are typically directed by run-time input, to determine the problem they need to solve and the environment (e.g. number of processors) to be used. Having to specify a GEN_BLOCK map at compile time is therefore not a practical solution and should only be viewed as a restriction with VFCS today and not a requirement for future HPF compilers.

2.2.3 HPF-1 version

No experiments were performed to use HPF-1 distribution directives as alternatives to INDIRECT and GEN_BLOCK, as it was felt that this was only meaningful in the context of two dimensional partitioning when NPROCB greater than one is used. In addition, the LG kernel would also require additional loops and data structures to simulate those found in the grid-point computations and fourier transforms. For instance, in grid-point computations, the majority of loops can simply run from 1 to NGPTOTG, the total number of grid-points, but in the fourier transforms we would expect to see loops running from 1 to NDGLG, the total number of latitudes.

2.2.4 Message Passing version

The message passing version was built around the transposition routines TRGTOL and TRLTOG used in the operational IFS, which are each over 300 lines in length. These 3D transposition routines and the data structures they use have been optimised for the parallelisation strategy used in the IFS, to avoid the need for global communication. In this strategy, the LG transpositions result in E-W communications that are independent within each of the NPROCA sets (of NPROCB processors), which is important for scalability.

2.3 Performance

The HPF+ and message passing versions of the IFS LG kernel were run for 3 problem sizes and on different numbers of processors, the results of which are summarised in the following tables.

Version	Compiler		1p	2p	4p	8p	16p	32p
MPI	apf90	ms	163.5	73.4	34.3	16.9	9.8	6.0
		s/u	0.78	1.74	3.73	7.56	13.4	21.3
HPF+	f77	ms		170.2	91.6	57.7	55.5	81.7
		s/u		0.77	1.44	2.28	2.37	1.61
SERIAL	apf90	ms	127.8					
SERIAL	f77	ms	131.8					

Table 2: Comparison of run times (in millesecs per time step) and speedups for the versions of the IFS LG kernel (T63 resolution) for different numbers of processors on a Meiko CS2

Version	Compiler		1p	2p	4p	8p	16p	32p
MPI	apf90	ms	501.5	229.2	111.9	48.9	23.8	14.6
		s/u	0.70	1.52	3.12	7.13	14.7	23.8
HPF+	f77	ms		470.2	243.6	136.9	96.4	108.6
		s/u		0.75	1.44	2.57	3.65	3.24
SERIAL	apf90	ms	348.7					
SERIAL	f77	ms	351.9					

Table 3: Comparison of run times (in millesecs per time step) and speedups for the versions of the IFS LG kernel (T106 resolution) for different numbers of processors on a Meiko CS2

The message passing version being written in f90, was compiled using the Apogee apf90 compiler with -O3 optimisation, whereas the HPF+ version was compiled using the default SUN f77 compiler and also with -O3 optimisation. Performance was only measured for the time-stepping loop in terms of millesecs per time-step. The cost per time step was computed as simple average over the total number of time steps, which was 1000 for the T63 resolution problem, and 100 for T106 and T213. Serial performance was determined by using both compilers on the HPF+ source. Speedup for both message passing and HPF+ versions has been calculated relative to the serial time with the same compiler. No single processor times

Version	Compiler		1p	2p	4p	8p	16p	32p
MPI	apf90	ms		1002.4	465.0	225.2	116.7	53.1
		s/u		1.35	2.91	6.01	11.6	25.5
HPF+	f77	ms		1830.1	950.5	562.0	309.3	236.9
		s/u		0.73	1.41	2.39	4.34	5.66
SERIAL	apf90	ms	1352.5					
SERIAL	f77	ms	1340.9					

Table 4: Comparison of run times (in millesecs per time step) and speedups for the versions of the IFS LG kernel (T213 resolution) for different numbers of processors on a Meiko CS2

are available for the HPF+ version as the code generated by VFCS had compilation errors relating to the size of some internal data structures.

The difference in performance between the serial (HPF+ source) and message passing version is explained by the fact that the message passing version's loop structures are significantly different, for instance, a typical gathering loop nest in TRLTOG.F would be,

```
       DO JFLD=1,KLATFLDS
         DO JGL=1,NDGLL
           DO JL=NSTA(IOFF+JGL,ISENDSET)+NSTABUF(JGL),
     &           NSTA(IOFF+JGL,ISENDSET)+NSTABUF(JGL)+
     &           NONL(IOFF+JGL,ISENDSET)-1
             IPOS=IPOS+1
             ZCOMBUF(IPOS)=PGLAT(JL,JFLD)
           ENDDO
         ENDDO
       ENDDO
```

where, NSTA and NONL data structures provide the start and length of each latitude segment owned by a processor. This should be compared with the simple loop structure of the serial/HPF+ version shown in the Appendix.

As described earlier, the LG kernel was run in a degraded mode (NPROCB set to 1) where only one array dimension was distributed, due to constraints with the current VFCS implementation. As a result of this, communication only takes place on latitudes that are split between partitions, at most one latitude to the north and south of each partition, data on all other latitudes being transposed locally. The performance of the kernel should therefore be dominated by the local transpose and not communication. This is confirmed by the fact the message passing version scales very well even at the smallest resolution (T63) on 32 processors. The local transpose being a memory operation is subject to caching issues, which can be seen by occasional instances of super-linear speedup in the message passing version.

2.4 Discussion

It is evident from the tabulated performance that the message passing version of the LG kernel performs significantly better than the HPF+ version. This is not surprising given the overhead of INDIRECT and GEN_BLOCK distributions today compared with an optimised message-passing code. As the communication patterns used are identical on each time step, the LG kernel should clearly benefit from the support of schedule re-use in future releases of HPF+.

As stated before, the performance of the IFS is very much dependent on the performance of the transpositions, that they should be both efficient and scalable. The LG kernel is representative of two of the six transpositions used in the IFS, but remains to be evaluated in its normal mode of use with 2 dimensional partitioning, requiring general support for INDIRECT distributions.

Future requirements of HPF+ are the need to remove the constraints on the use of INDIRECT distributions, and to allow GEN_BLOCK maps to be specified at run time.

3 ESI

3.1 Introduction

This report discusses the initial HPF port of different kernels of the PAM-CRASH code within the Esprit Project HPF+. The aim is to have all features needed for a complete HPF benchmark with the better expressiveness and optimized in terms of performance and memory requirements. Some kernels of growing difficulty in term of HPF requirements were written to be adapted and to follow the evolution of the tools used [11][12] in the project.

A brief overview of PAM-CRASH's modules are given, followed by the HPF+ features needed for a more advanced benchmark. The report also describes an initial port using a mix of standard available HPF directives and gives a comparison with the corresponding f77 kernels and the hand-coded message-passing kernels. In a subsequent section, the performance of different F90 compilers on these kernels is described.

3.1.1 Overview of the Code

PAM-CRASH is a Finite-Element Program specialized for solving the highly non-linear dynamic problems arising in crashworthiness simulation. It uses an explicit time-marching scheme with unstructured meshes comprising mechanical elements which model the behavior of the structure under consideration: beams, shells solids and trusses. Quantities to be calculated by the program are either defined on the elements or at the nodal points defining the elements.

A central-difference scheme is employed for the explicit time-marching at nodal points. Around 80which are performed element-wise (all force calculations are performed on elements, then there is a need for an interaction between nodal point and element-wise calculations, which is as follows : Given velocities and displacements at nodal points the forces on elements defined by those nodal points may be calculated. These forces are then distributed to the nodal points in order to calculate accelerations from the equations of dynamic equilibrium. The central difference scheme now allows the calculation of displacements at the new time-level.).

3.1.2 Outline of the Algorithm

For the understanding of the HPF+ port, we outline the skeleton of the algorithm, which is shown in Figure 1.

3.2 HPF kernel

3.2.1 HPF+ features used

In the HPF kernel, BLOCK and ALIGN directives were used to distribute nodes and elements. Ideally, elements should be distributed in such a way that all nodes that belong to local elements reside as much as possible on the same processor. This is necessary to avoid excessive inter-processor communication during the local computation of forces. Indirect distribution would be more appropriate for this, however, it was not used due to restrictions in the initial kernel. Further, the INDEPENDENT do loop directive was used with the REDUCTION

```
(1)      Read in problem sizes
(2)      Initialization of variables
(3)      Compute Force
(4.1)    Gather values of nodal quantities
(5.2)    Compute Internal forces of elements
(6.3)    Update of force of nodal quantities
(7)      Compute Contact forces
(8)      Compute Minimum Time-step: Global min over shells
(9)      Compute Acceleration
(10)     Apply Boundary conditions
(11)     Update velocity
(12)     Update displacement
(13)     output
```

Figure 1: Basic Shell/Time-Marching Scheme

clause to gather forces computed by elements to nodes. In addition, the ON HOME clause was added to the INDEPENDENT directive to optimize the analysis of different loops by compilers. Finally, the HPF+ kernel includes the PUREST directive to allow the use of a local subroutine in an independent loop.

3.2.2 Porting Effort

The storage management is performed by allocating a single one-dimensional work array. Sections of this array are reserved for the storage of individual arrays by the initialization of run-time defined pointers into this array. Obviously, each subroutine uses different sections of the same array.

Data structures

Another problem is the extensive use of common statements for storage of run-time pointers or real arrays mixed with global and/or local data. Moreover, in all subroutines, we have assumed-sized arrays.

Below, we illustrate how the one-dimensional work array is used in the code.

```
PROGRAM PCRASH
COMMON / / DATA(LENGTH)
COMMON /DATA1/ NUMNP,N4NODE,NDOF
INTEGER IPTR1,IPTR2,IPTR3,IPTR4,NUMNP,N4NODE

CALL PAM_INPUT(NUMNP,N4NODE,NDOF)
...
CALL PAM_INIT(NUMNP,N4NODE,IPTR1,IPTR2,IPTR3,IPTR4)
...
CALL PAM_SOLV(DATA(IPTR1),DATA(IPTR2),DATA(IPTR3),DATA(IPTR4))
...
END


SUBROUTINE PAM_SOLV(X,V,A,IY)
COMMON /DATA1/ NUMNP,N4NODE,NDOF
DIMENSION X(3,*),V(NDOF,*), A(NDOF,*), IY(6,*)
.. .
END
```

As discussed above, the main work was spent in:
Extracting the salient parts of the solver
The code was completely reorganized
Write specific code for the initialization of data structure.

This is especially true for all data structures. This includes:

* Cleaning data structures
- remove the one-dimensional working array
- do not use complex data structures
- use (automatic) arrays to compute nodal forces
- avoid assumed-sized arrays
- avoid common blocks with different shapes and containing a mix of distributed and local variables

* All I/O is performed in the main program
- Avoid implicit do-loops

* Only one subroutine to compute internal forces
- Static arrays (parameter statement was used to dimension arrays)
- Simple indirect addressing in the main loop

* Remove all optimizations relating to vector groups (first benchmark).
- Simple indirect addressing in the main loop.
- Local arrays used to store indirections were removed.

We give below, an overview of main variables manipulated by the program for one specific example:

```
Data structures used for NODES :

      NUMNP : Number total of nodes
      NDOF : Degree of freedom (always 6 in our case)

      A(NDOF,NUMNP) : Acceleration or Forces (AC or FINT in subroutines)
      V(NDOF,NUMNP) : Nodal velocities
      X(3,NUMNP)    : Nodal coordinates
```

```
Data structures used for ELEMENTS :

    N4NODE : Number of elements

    IY(1,N4NODE)         : Id of material of element
    IY(2 :5,N4NODE)      : Connectivities of elements
    STRS(NSTR,N4NODE)    : List of physical values computed per element
```

3.2.3 HPF+ version

Initial Distribution

In the first version of the HPF kernel, all data structures use the BLOCK distribute directive. This is the first distribute directive provided by the HPF-1 specifications. Nevertheless, the data-set is organized to have the list of nodes reordered to reduce communications. Thus, to be compliant with the initial definition of the HPF language, the interesting INDIRECT directive of the VCFS was not used. Besides, this kernel do not have some others subroutines that use the distribute directives.

```
Distribute directives used:

    !HPF$ PROCESSORS :: PROC(NPROC)
    C    elements
    !HPF$ DISTRIBUTE(BLOCK) :: INDEX
    !HPF$ ALIGN WITH INDEX  :: ROEL,DT2AR
    !HPF$ ALIGN (*,I) WITH INDEX(I) :: IY,STRS,FORCE1,FORCE2,FORCE3,FORCE4
    C     nodes
    !HPF$ DISTRIBUTE(BLOCK) :: XC
    !HPF$ ALIGN WITH XC :: YC,ZC
    !HPF$ ALIGN (*,I) WITH XC(I) :: XMSI,V,A,XMS,BLAMB,X1,X,VGLOB
```

Unstructured communication

The different routines for computing internal forces remain certainly one of the most critical points in the initial HPF benchmark. As mentioned above, the major part of the computational cost of the algorithms used is due to element-wise force calculations. The indirect addressing which occurs within the solver is described in the technical annex [10]. In practice, we duplicate all distributed arrays, except the STRS array in automatic arrays. Then, we must replicate these arrays on each processor. In this portion of code, we need the Halo specification (irregular overlaps) for nodes. And, it is crucial to reuse the communication schedule computed the first time or in an external loop (cf. inspector/executor strategy[9]). For the first kernels, we have to remove all optimizations used for vectorization and completely rewrite this part to be able to use the new PUREST procedure provided by Vienna. This implies to have only local variables in the subroutine.

```
Example in HPF-1 version:

    !HPF$ INDEPENDENT, ON HOME(IY(2,I)), NEW (IL,XL,VL,RL, ... )
       DO I = 1, N4NODE                 ! Number of elements
    C
         DT2AUX = DT2AR(I)
    C
         XL(1) = X(1,IY(2,I))           !
         XL(2)  = X(2,IY(2,I))          ! Gahter connectivies of nodes
         XL(3)  = X(3,IY(2,I))          !
    C
         CALL FORCEI(I, XL, FORCE ...)  ! Compute internal forces
    C
         DT2AR(I) = DT2AUX              ! Minimum time-step by element
         FORCE1(1,I) =-FORCE(1)         !
         FORCE1(2,I) =-FORCE(2)         ! Update arrays of forces
         FORCE1(3,I) =-FORCE(3)
    C
       END DO
```

The INDEPENDENT directive contains:
- On Home clause with limited restriction
- New statement (private variables: scalars and arrays with different shapes)
Nevertheless, we outline that the next kernel, The INDEPENDENT directive must also contains:
- Reductions (not limited to one reduction by independent loop)
- Handle Local subroutines with distributed arrays (cf: HPF LOCAL)
IN HPF-2, the proposed RESIDENT[6] directive could be another alternative.

Local subroutine

The PUREST directive defined by the HPF+ specifications is used only in the subroutine called in independent loop. This subroutine computes internal forces of one element. Actually, we outline that all distributed variables used to compute forces must be replicated in local variable to use subroutines independent loops.

```
      SUBROUTINE FORCEI(I, XL, FORCE ...)
!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)....
```

Reduction

The distribution of the internal forces of each element over the attached nodes requires a reduction operation in an independent loop. With the storage of internal forces in four distributed automatic arrays indexed by the connectivities, we can use different strategies available in F90/HPF+. Below, we show two different syntax with 4-nodes shells. The second solution without the array syntax reduction was used for the initial kernel:

```
!HPF$ INDEPENDENT
      DO I=1,NDOF
        A(I,:) = SUM_SCATTER(FORCE1(I,:),A(I,:),IY(2,:))
        A(I,:) = SUM_SCATTER(FORCE2(I,:),A(I,:),IY(3,:))
        A(I,:) = SUM_SCATTER(FORCE3(I,:),A(I,:),IY(4,:))
        A(I,:) = SUM_SCATTER(FORCE4(I,:),A(I,:),IY(5,:))
      END DO


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

Global minimum

To compute the stable time-step, we need to find the minimum time-step over all elements. Thus, a special global reduction is required. In fact, in the kernel, we have to modify this calculation to be compliant with different compilers. We have allocated an array of elements to store the time-step and then we apply a min-reduction. Nevertheless, some others different strategies will be considered.

```
!HPF$ INDEPENDENT,REDUCTION(DT2)
      DO N=1,N4NODE
        DT2 = MIN(DT2,DT2AR(N))
      END DO
```

3.2.4 Message Passing version

The hand-written message-passing version is based on the second revision of the HPF-1 kernel used for the time-measurements on the CS-2 machine provided by the VCPC. The first implementation was written using the PVM library. It uses the host/node paradigm based on the full parallel version of the PAM-CRASH code.
The characteristics are:

Host code:
- Very simple input code
- It was constructed to read the same input data as the others kernels
- Construction of mailing-lists for the exchange of information between nodes
- Distribution of all information to the nodes

Node code:
- Initialization by receiving all necessary information from host
- Same kernel as the HPF-1 and sequential implementation
- One global broadcast to compute the minimal time-step
- Exchange Forces and update velocities of the boundaries nodes for the neighbors processors.
- Outputs are redirected in local files to validated results

A second implementation based on the MPI library was written to generate trace-files for the MEDEA tool and to compare with the HPF kernel. The same input file is used by all different kernels.

3.2.5 Fortran 90 with derived data types

Introduction

In parallel with the HPF kernel, a kernel using the F90 syntax was written. In fact, this kernel was totally rewritten to use the derived data types for Elements and Nodes based on notes provided by NA Software during previous meetings. The main objectives are to be able to use the distribute directive for derived data types and pointers to obtain a HPF code more easier to write and maintain. The need for portability between different platforms and HPF compilers is also very important for such code. The kernel is subdivided in two parts: Initialization to read data input and the core to compute internal forces. Two subroutines were developed to measure impact on performance: The first uses the F90 syntax with derived data types and the second subroutine is the same subroutine used in the HPF kernel.

The main characteristics of this benchmark are:

- Initial F90 benchmark was taken from the initial HPF+ benchmark. (free format syntax was used)
- Totally rewritten with Derived Data Types (cf : NA HPF+ Working note, to be compliant with Fortran 95 syntax)
- Dynamic allocation and pointer arrays.
- Assumed shape arrays from Derived Data Types
  TYPE(T_NODE), DIMENSION(:),INTENT(IN) :: NODES
  TYPE(T_SHELL), DIMENSION(:),INTENT(IN) :: SHELLS
- Obsolete features: COMMON, DATA or BLOCK DATA were rewritten
  Use of modules to avoid storage association.
- Automatic arrays to compute nodal forces.
- Intrinsics functions : ex: MINLOC and MINVAL
- Explicit interfaces for procedures with the extrinsic directive (to be done).
- All variables are declared (IMPLICIT NONE)

Data structures

```
!   Nodes
TYPE :: T_NODE
  TYPE(T_DOFVAL)  :: A        ! Acceleration
  TYPE(T_DOFVAL)  :: V        ! Velocities
  TYPE(T_POSVAL)  :: X        ! Coordinates
END TYPE T_NODE
!     Shells elements
TYPE :: T_SHELL
  INTEGER         :: ID       ! Id
  REAL, POINTER   :: STRS(:)  ! physical values
END TYPE T_SHELL
!
```

Examples taken from the F90 kernel

Main loop:

```
!
!    Transform moments from global to nodal axes
!
DO N=1,NUMNP
  AX = NODES(N)%A%VALUE(4)
  AY = NODES(N)%A%VALUE(5)
  AZ = NODES(N)%A%VALUE(6)
! A(4,N) = BLAMB(1,N)*AX + BLAMB(2,N)*AY + BLAMB(3,N)*AZ
  NODES(N)%A%VALUE(4) = NODES(N)%BLAMB%COEF(1)*AX +&
                        NODES(N)%BLAMB%COEF(2)*AY + &
                        NODES(N)%BLAMB%COEF(3)*AZ
! A(5,N) = BLAMB(4,N)*AX + BLAMB(5,N)*AY + BLAMB(6,N)*AZ
  NODES(N)%A%VALUE(5) = NODES(N)%BLAMB%COEF(4)*AX + &
                        NODES(N)%BLAMB%COEF(5)*AY + &
                        NODES(N)%BLAMB%COEF(6)*AZ
! A(6,N) = BLAMB(7,N)*AX + BLAMB(8,N)*AY + BLAMB(9,N)*AZ
  NODES(N)%A%VALUE(6) = NODES(N)%BLAMB%COEF(7)*AX + &
                        NODES(N)%BLAMB%COEF(8)*AY + &
                        NODES(N)%BLAMB%COEF(9)*AZ
END DO
```

Min-reduction in Fortran 90:

```
DO I=1,NDOF
  NODES(SHELLS(:)%IY%CONNECT(2))%A%VALUE(I) =
     NODES(SHELLS(:)%IY%CONNECT(2))%A%VALUE(I) + FORCE1(I,:)
  NODES(SHELLS(:)%IY%CONNECT(3))%A%VALUE(I) =
     NODES(SHELLS(:)%IY%CONNECT(3))%A%VALUE(I) + FORCE2(I,:)
  NODES(SHELLS(:)%IY%CONNECT(4))%A%VALUE(I) =
     NODES(SHELLS(:)%IY%CONNECT(4))%A%VALUE(I) + FORCE3(I,:)
  NODES(SHELLS(:)%IY%CONNECT(5))%A%VALUE(I) =
     NODES(SHELLS(:)%IY%CONNECT(5))%A%VALUE(I) + FORCE4(I,:)
END DO
```

3.3 Performance

Two specific input files of different size were generated for the different kernels. The first input file was used to validate the Vienna Fortran Compiler System. All results have been validated with the two data-sets. Comparisons with the standard version of the PAM-CRASH code have given the same results (Minimal time-step and nodal coordinates). The performance measurements were made by the University of Vienna on the Meiko CS-2.

Characteristics of input files:
Small: Number of nodes : 483. Number of shell elements: 504.
Large: Number of nodes : 35571. Number of shell elements: 35000.

3.3.1 F77 kernel

The F77 sequential code has exactly the same data structure as the HPF kernel. It uses the same loops. The first version is the initial version that uses replicated arrays to gather information about elements and some other details. The second version is the simplified version by the VCPC team to be able to use their HPF compiler. The time- measurements of different kernels with other partner compilers were not yet available during the writing of this report.

Initial fortran 77 kernel:

Machine	Small dataset	Compiler
SGI R4400 150 MHz 0S5.3	21.66 s	F77 compiler (-O3)
HP-UX 10.20 PA8000	19.17 s (B-F90 compiler)	B-HPf90 (-O2)
Ultra sparc SunOS 5.3	6.47 s	F77 compiler (-O5)

Second kernel with modifications:

Machine	Small dataset	Large dataset
SGI R4400 150 MHz 5.3	17.30 s	828 s
HP-UX 10.20 PA8000	2.91 s (B-HPf90)	618 s (B-HPf90)
Ultra sparc SunOS 5.3	5.90 s	400 s
Power Challenge R8000/75	8.11 s	491 s

3.3.2 HPF+ kernel

(provided by the VCPC)

Machine MEIKO CS-2	Large dataset first run (s)	Large dataset second run (s)
Sequential (F77 -03)	545.873	549.341
2 procs	601.447	601.582
4 procs	324.180	329.918
8 procs	169.121	170.571
16 procs	110.312	111.412
32 procs	79.165	83.636

3.3.3 Hand-coded Message passing kernel

The time-measurements of the message-passing version were performed with the PVM implementation on a 4 processor Power Challenge. To evaluate the results, it must be observed that the PVM implementation used was not optimized for this machine (the Shared-Memory feature was not used (TCP/IP protocol) and compilation was made on 32 bits machine). Moreover, the simulation with more processes than processors degrade performances because of the SGI-scheduler.

Machine Power Challenge R8000 (PVM: arch SGI5)	Large dataset (s)
4 procs	345.0
8 procs	164.7
16 procs	87.63
32 procs	37.04

3.3.4 F90 kernels

The first evaluation is based on the initial kernel written with derived data types, but the subroutine that computes internal forces is based on the HPF-1 kernel, thus this part of code is in pure Fortran 77. The second version is taken from the final HPF benchmark totally rewritten with derived data types. Actually, it seems that the use of derived data types is very compiler implementation dependent.

Mix of Derived Data types and Fortran 77:

Machine	Small dataset	Compiler
CRAY EL-II	108.025 s	Cray F90 V2.0 (default)
HP-UX 10.20 PA8000	44.45 s	B-HPf90 (-O2)

Computation of forces with derived data types:

Machine	Small dataset	Compiler
CRAY EL-II	221.31 s	Cray F90 V2.0 (default)
HP-UX 10.20 PA8000	66 s	B-HPf90 (-O2)

3.4 Discussion

In this report, we have described the parallelization strategy of a PAM-CRASH kernel to be performed within the HPF+ project. The first step, consisted in localizing most problematic parts of the code and restructuring it to allow better analysis by the HPF compilers. Nevertheless, we had to remove some very optimized constructs to stay in compliance with the data-parallel HPF constructs. In the second step, we have adapted some different subroutines to use HPF directives. A first version of the kernels in F77 has been carried out in the VF compiler system and a more advanced benchmark with some new HPF+ was constructed in close cooperation with the University of Vienna. Moreover, a complete hand-coded version of the kernel was written with different message-passing libraries to be able to use the MEDEA tool. Furthermore, a Fortran 90 version of the HPF kernel with derived data types was also written to take in account some future HPF directives. This version could be the basis for further development with the parallelisation of derived data types, pointers, resident clauses and other new advanced features.

Some performance measurements were made on different platforms to evaluate impact of HPF directives on codes and to show the urgent needed for new directives to improve the expressiveness, performance and memory requirements of unstructured HPF code. Nevertheless, first results given by the HPF-1 kernel seem very encouraging.

4 CCRLE

4.1 Introduction

The generic kernels to be produced by CCRLE for the HPF+ project are to be designed so that they represent specific, important features within the PAM-CRASH code from ESI but in a much simplified form. This assists the language and compiler developers to a more rapid understanding of the key issues in the PAM-CRASH code, before dealing with the full benchmark kernels from ESI.

This report covers work performed in the development of several versions of the first such generic kernel (denoted below by K1_ followed by various acronyms for sequential and parallel variants). The kernel represents the most basic calculation procedure within PAM-CRASH: that of performing stress-strain calculations on 4-node shell elements. This is illustrated in Figure 2, and can be described as: for every element of the unstructured mesh (defined simply as a list of 4 nodal points), the calculation procedure takes the latest values of co-ordinates and velocities at the nodal points, calculates "forces" on the element; adds the elemental forces (with some weighting factor) back to the forces stored at nodes. Note that step (iv) in Figure 2 is not included in the K1_ kernels.

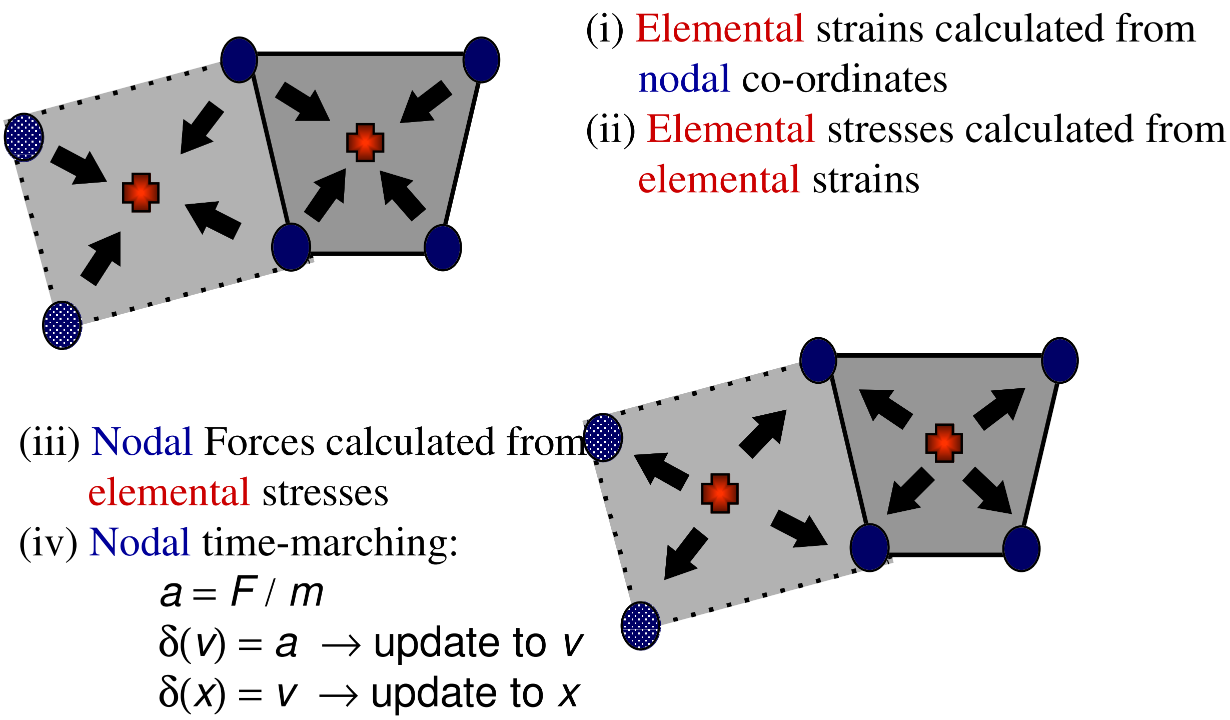


Figure 2: Schematic view of the K1_ generic kernels

4.2 HPF kernel

The "origin" of all the kernels discussed is the F77 kernel K1SEQ given in the Appendix, Section A1. The routines omitted from that section are the routines box, mforce and kout. The K1_ kernels make use of a run-time generated mesh (rather relying on input from file - since this provided a simple mechanism to avoid problems / restrictions in VFCS) corresponding to a box-beam. This is computed within the routine box. The ordering of nodes and elements in the generated mesh is such that a "partitioning" based on the storage-order corresponds to horizontal slicing of the mesh. Thus, each process in the parallel implementations will have at most two neighbouring processes and, for the numbers of processes and mesh size considered below, the interface lengths will be approximately constant with increasing process sizes. Later kernels could be modified to exploit different partitions or other geometries. The modelled force calculations are performed in routine mforce. This routine is in fact a modified version of the force2 routine from the initial ESI benchmark kernel, with all non-available physical constants & arrays initialised to zero or with the corresponding code sections removed. The actual calculation to be performed was not relevant for the generic kernel - its purpose was to represent some computation to be performed on local data. The final missing routine is a simple code section to output a section of the generated "forces" to allow correct execution to be checked.

In all K1_ kernels, the only relevant routine in terms of execution time is the routine kforce. The HPF variants discussed, will essentially differ only in their treatment of this routine. In these versions, the main loop over elements was first split, via the introduction of auxiliary arrays to separate out the reduction operation caused by the summing of forces. Standard block distributions of the main arrays was used throughout.

4.2.1 HPF+ features used

The kernel which has been used to test the HPF+ developments within VFCS, and for which results will be presented below, will be referenced as K1HPF+_b1_vfcs, and the corresponding version of routine kforce is given in the Appendix, section A2. The only HPF+ features employed by this kernel are: the REDUCTION directive in subroutine kforce and the PUREST directive within the subroutine mforce (not included in Section A2 - the subroutine is otherwise unaltered). However, there is within this kernel the potential to exploit many of the other features, and these options will be discussed below. For a general, unstructured mesh, the communication resulting from the kforce subroutine could be minimised by making use of distributions corresponding to optimised mesh partitioning tools. These could be incorporated within HPF+ kernels by the use of either GEN_BLOCK or INDIRECT distributions. However, for this to make sense in the context of PAM-CRASH kernels, the distributions need to be defined at run-time- which for a clean implementation implies the availability of DYNAMIC, otherwise subroutine boundary re-distribution. The impact of this feature will be investigated with future VFCS releases. However, the main gain from HPF+ extensions is expected to be the "schedule re-use" facility. The next level up from the K1_ kernel is to embed it within the outer time-marching iteration (as exemplified in the ESI benchmark code). Thus, the routine kforce would be executed many thousands of times (for industrial models, 100 000 time-steps is not unusual). The overhead within the handling of the INDEPENDENT loops (the inspector phase) could be extracted to a higher level by a schedule re-use construct.

4.2.2 Porting Effort

From the size and complexity of the HPF kernels, produced as stated from the sequential kernel K1SEQ, it is quite apparent that the coding effort to create these versions was minimal - which is the major attraction to that programming paradigm, should this prove to be true for large codes. That said, the idiosyncrasies of various HPF (or HPF+) compilers, in the sense of varying levels of supported features, restrictions or "bugs" which result in sequentialisation of what are in fact parallel loops, means that portability is not really achieved. However, the main comparison should be with hand-coded message-passing. Since the message-passing version to be discussed later, K1MPIa was also constructed from K1SEQ, a direct comparison can be made. The porting effort involved in the construction of the MPI version is significantly greater. The domain partitioning approach employed requires a localisation of node numbering and element references which must be done by the applications programmer. While the computational levels of the code remain unchanged - the routine kforce, and thus all underlying routines implied by this generic modelling, is exactly that in K1SEQ - the data management levels acquire an extra level of complexity. As a very rough indicator, the lines of code in the kernels K1SEQ, K1HPF+_b1_vfcs, K1MPI_a are (comment cards included) : 411, 531, 1504, respectively. A large portion of the extra code arises from the construction of the partitioned node-element lists and inter-process mailing lists.

4.2.3 HPF-1 version

Two basis variants of the kernel were produced using only HPF-1 constructs, the only difference between them being the reduction loop in routine kforce. The first variant employs the HPF Library function SUM_SCATTER as a replacement for this loop. A version of this kernel which has been "tuned" to best exploit the commercial compiler from the Portland Group is given in section A3 of the Appendix, and will be referred to as K1HPF1_a_PGI. The second variant is identical to the kernel K1HPF+_b1_vfcs, except that the directives REDUCTION and PUREST are omitted. A version of this variant was successfully compiled and executed using the NAS Mapper - producing correct results. However, the compiler version on the Meiko CS2 in Vienna does not make use of the INDEPENDENT, NEW directives. As a consequence, the main loop is serialised. Performance measurements have therefore been omitted below.

4.2.4 Message Passing version

As stated above, an MPI-based message-passing version of the generic shell kernel (K1MPI_a) was constructed starting from the sequential kernel. For the sake of brevity, only the core computational routine (local) and the two inter-process communications routines (xa2b, fb2a) have been included in section A4 of the Appendix. Although the initialisation phase of the program provides all processes with the correct interface co-ordinates, the corresponding call to the co-ordinate update routine xa2b has nevertheless been included in routine local. Were a time-stepping procedure to be built around the call to the routine local, this update would then be necessary. The inclusion of the communication thus gives an indication of the performance in the more general case.

4.3 Performance

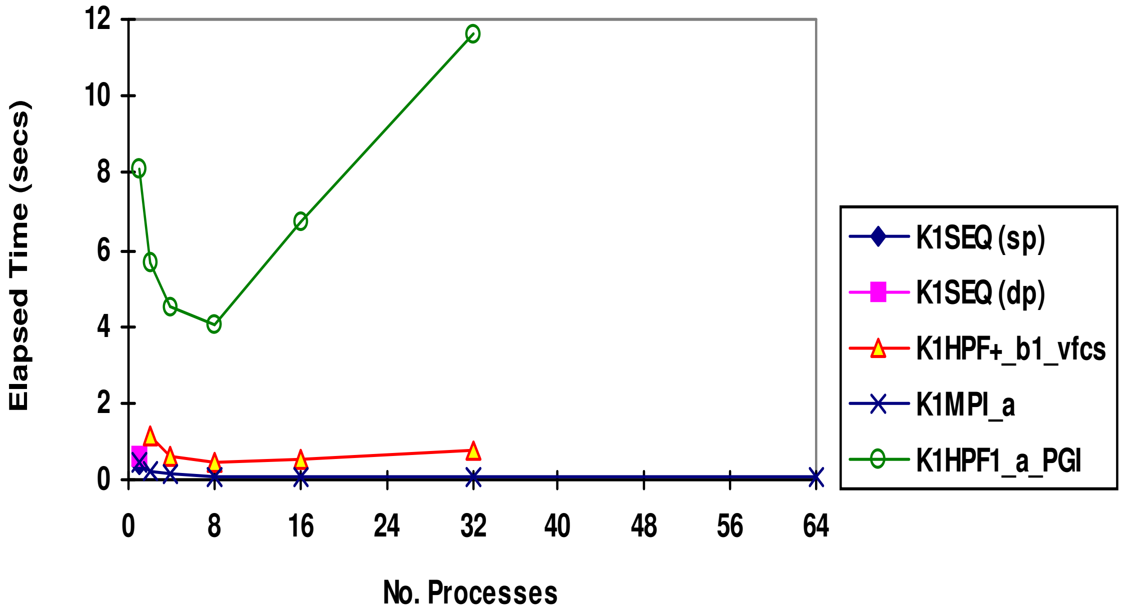


Figure 3: Elapsed Times for all Kernels

At the time of writing, performance results are available for the box model with element dimensions 40 40 160 (x- y- z- directions) giving a total of 25600 elements. Additional testing has been performed on an IBM SP2 and an NEC Cenju-3 at GMD, but attention will be restricted here to the project platform: the Meiko CS2 at VCPC. Figures 3 & 4 show the relative performance of the kernels on up to 64 processors of the Meiko CS2. The timings given are for the routine kforce for the sequential & HPF kernels and the routine local for K1MPI_a. All time are wall clock times in seconds. The timings were created using: ETIME with K1HPF+_b1_vfcs, system_clock with K1HPF1_a_PGI and MPI_WTIME with both K1SEQ and K1MPI_a. In all cases, compile option -O3 was used. Comparison times are given for K1SEQ in both single & double precision, since the K1HPF+_b1_vfcs kernel was run in double precision to ensure identical results in the sample output. It should be noted that some differences between results for a small number of nodes (e.g. 10 nodes when comparing results for 1 and 64 processes) were detected in the message-passing version, but these were

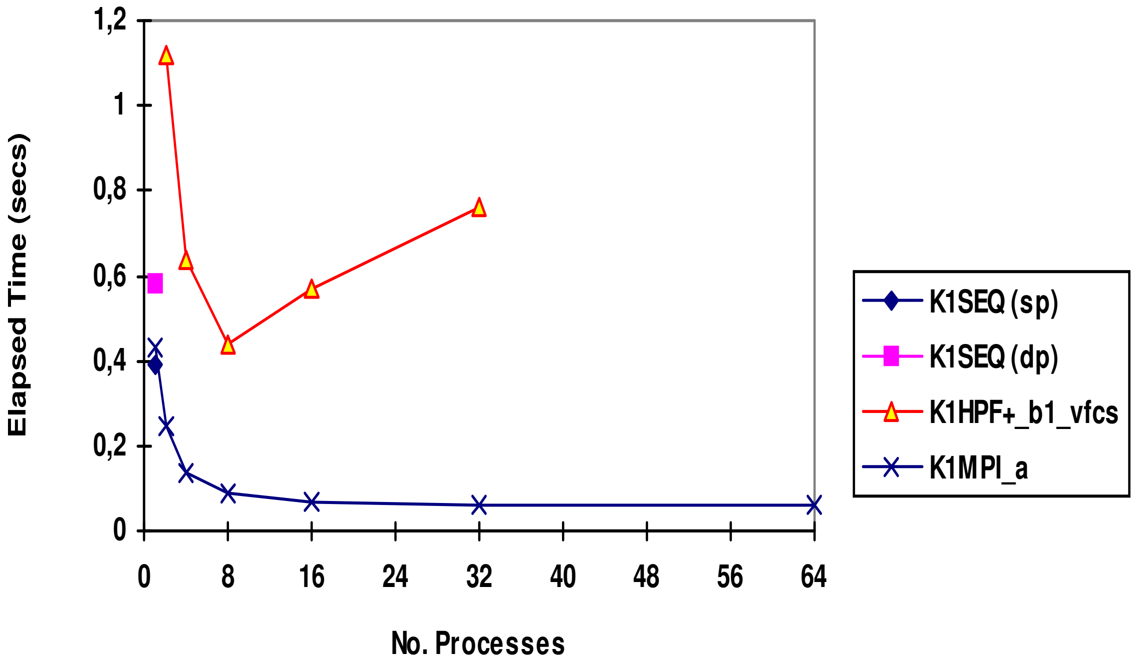


Figure 4: Elapsed times excluding HPF-1 Kernel

in the 3rd decimal place in force components several order of magnitude smaller than the largest components and the largest x-values: the re-ordering within the distributed sum-scatter operation is enough to create such differences and is deemed to be within single precision accuracy. A double precision version could be created to check on this, but would require minor changes in the F77 work space management employed. For the K1MPI_a kernel, the measurements at higher processor numbers exhibited a significant amount of noise, in the sense of relatively widely varying results for repeated runs. However, an inspection of the detailed timings indicated that the times for the two communication routines xa2b and fb2a dominates. An average of the (max. elapsed wall clock) measurements of the repeated runs (typically 3 to 4 per process number) was taken for the results in Figures 3 & 4.

4.4 Discussion

Although further testing based on the K1_ kernels - also with larger model sizes and/or varying mesh partitions - is to be performed in order investigate in more detail the areas of overhead (here, improved interaction between VFCS & MEDEA could prove most useful), the results obtained to date already indicate two features:

1. Major advances have been made with the VFCS facilities for handling INDEPENDENT loops including indirect addressing, subroutine calls and reductions.
2. Improvement are still required in order to approach message-passing performance.

The main expectation is that the static nature of the communication within the kernel can be exploited when the computation is embedded within the time-marching algorithm by using schedule re-use. The overheads seen could then be amortised over the complete simulation.

As discussed in the introductory remarks, this kernel represents the most basic computational component found within PAM-CRASH. The inclusion of contact-impact algorithms, which involve global or pseudo-global searching, will pose a much greater challenge for the future HPF+ developments. Here, user- and run-time defined general shadow/halo regions may provide the answer - but this is yet to be investigated.

Acknowledgements

The authors would like to thank Sigi Benkner & Viera Sipkova for running the PGI & VFCS kernels on the Meiko CS2. Additional thanks go to them and also to Guy Robinson for support and most useful discussions.

5 University of Pavia

5.1 Introduction

Performance evaluation of the kernel benchmarks is based on measurements collected at run–time during their execution on the project platform. Such evaluation includes HPF+ versions of the codes as well as explicitly message passing codes implemented using MPI.

All the executions are traced by means of MPI profiling libraries which record into tracefiles the occurrences of the various types of communication events used by the codes (i.e., inserted by the compiler into HPF+ codes or by the application developers into explicit message passing codes). Basically, each record of the tracefile contains a time stamp and the identifiers of the processor that generated the event and of the event itself.

A post–mortem analysis of these tracefiles has been carried out by means of the MPI filtering module developed and integrated within Medea. Various types of metrics (e.g., profiles, signatures) and parameters (e.g., execution and communication times, message size) are computed and presented by means of the visualization facilities of Medea. The importance of these analyses is related to their ability to extract from the large amount of data stored into tracefiles (of the order of a few Mbytes), information which eases the understanding and the interpretation of the achieved performance. Insights into the behavior of individual processors are obtained by computing basic statistics and by applying clustering techniques. Similarities with respect to parameters, reflecting specific activities, are pointed out by analyzing the groups produced by the clustering algorithms.

All these results allow the application developers to evaluate the efficiency of their codes. These results are also very helpful for compiler developers in that they provide hints about possible optimizations.

In what follows, the performance of each kernel benchmark is evaluated by analyzing the tracefiles generated by an extensive set of executions on the project platform. Note that we have discovered that most of the runs are characterized by a sort of "noise" due to fact that communications as well as I/O activities have been measured.

5.2 AVL kernel

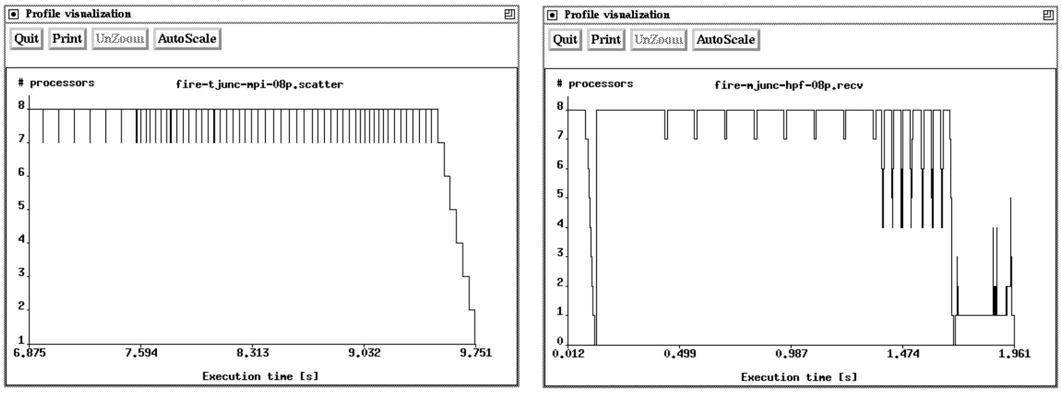
The performance of an HPF+ version and of an explicit message passing version of the kernel, both executed with three datasets, corresponding to a small (1.5 Mbytes), medium (4.5 Mbytes) and large (9.8 Mbytes) size of the input data, has been evaluated.
Table 5 presents the execution times of the runs of the MPI version of the kernel varying the problem size and the number of allocated processors. These times include the times spent in the input phase, in the pure computation part of the kernel, as well as in the output phase.

A more detailed analysis of the performance of the kernel focuses on the phases it consists of. The execution of the kernel starts by reading and distributing the input data among the processors. The MPI version distributes the data by means of `MPI_SCATTER` primitives. The HPF+ version adopts a different implementation based on `MPI_SEND` and `MPI_RECV` primitives, inserted by the VFCS. Although the two implementations are different, the communications required by data distributions exhibit a similar behavior (see Fig. 5). Figure 5(a) shows the profile of the `MPI_SCATTER`, that is, the number of processors performing scatter operations,

Number of processors	t_exec [s]		
	Small	Medium	Large
2	9.36	15.53	29.54
4	7.09	15.61	31.62
8	9.95	18.32	44.58
16	8.74	19.86	38.29
32	11.07	20.29	43.96
64	27.51	22.44	48.09

Table 5: Execution times of the MPI version of the kernel varying the problem size and the number of allocated processors.

as a function of the execution time. The profile of `MPI_RECV` is shown in Figure 5(b). Both profiles refer to executions with 8 `nodes` for a problem with a small dataset. HPF+ version requires 9 processors since an extra processor acts as a host.



(a) (b)

Figure 5: Profiles of the `MPI_SCATTER` and of the `MPI_RECV` communications in the MPI version (a) and in the HPF+ version (b) of the kernel.

Note that, due to the different implementation of the input phase, the times related to the executions of two versions are not comparable. For example, in the MPI version the data distribution starts after the whole data set is read (that is, after 6.875 seconds, in our example), while in the HPF+ version reads and broadcasts of the input data are intermixed. In what follows, examples related to the analysis of runs with the small dataset are shown.
Critical operations performed by the kernel are the sum reductions. Figure 6 shows their profiles for an execution with 16 processors, for both the MPI and the HPF+ versions.
Table 6 summarizes the total execution times and the average times spent by each processor in write, reduce and communication operations in the HPF+ version of the kernel, varying the number of allocated processors. As can be seen, the percentage of time taken by the reduction

operations is always less than 3.8% of the total execution time. The MPI version executed with the medium dataset and 32 processors requires less than 0.1% of the execution time for reduction operations, while the percentage of the time spent for I/O operations reaches 50%. Clustering technique has also been applied in order to characterize and to highlight similar behaviors of the processors. As an example, the analysis of a run of the MPI version of the kernel with 64 processors and medium dataset size is presented.

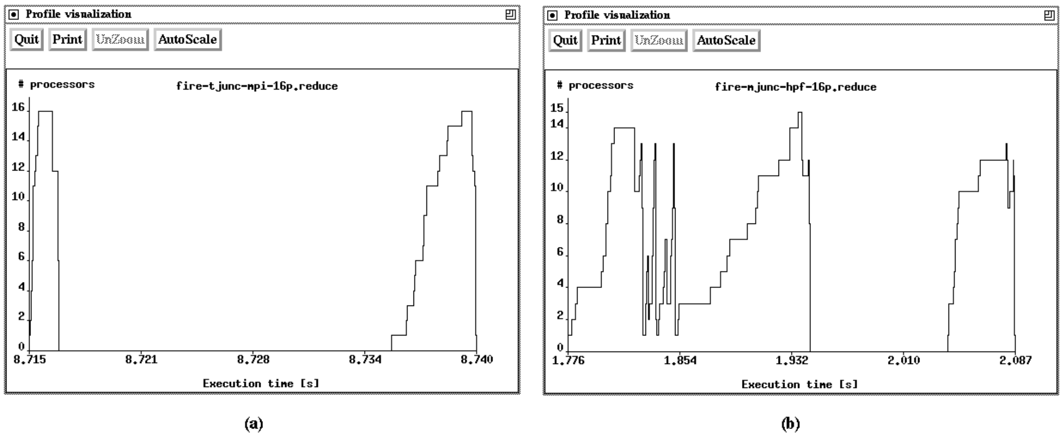


Figure 6: Profiles of the sum reduction operations in the MPI version (a) and in the HPF+ version (b) of the kernel with 16 processors.

Number of processors	t_exec [s]	t_comm [s]	t_reduce [s]	t_write [s]
3	2.33	1.70	0.08476	0.007115
5	2.06	1.67	0.07650	0.004533
9	1.96	1.70	0.05881	0.023622
17	2.09	1.91	0.077878	0.006841
33	2.49	2.35	0.077967	0.006718

Table 6: Execution, communication, sum reduction and write times of the HPF+ version of the kernel.

Figure 7 shows the basic statistics of some of the parameters derived from the tracefiles for the description of the behavior of the individual processors.
In Fig. 8, the statistics produced by clustering analysis are shown. As can be seen, the first group of processors is characterized by large time spent in barrier operations and by small receive times. Processors belonging to the second group are very fast during the barrier phase, but are only 3. Larger send times, with respect to the first two groups, characterize groups 3 and 4: differences between them are in the barrier and wait times.

Statistics

Quit Print

Basic statistics for fire-mpi-64p
# of components: 64

	mean	std-dev	min	max
t_comm	21.991739	1.434196	11.719050	22.310230
t_snd	0.065210	0.025868	0.011179	0.125398
t_rcv	0.914957	0.295131	0.003471	1.453154
t_wait	0.381191	0.245496	0.021891	0.692264
t_scat	9.839605	0.185254	9.529024	10.148650
t_barr	0.446603	0.503854	0.000910	1.410334
t_reduce	0.010088	0.002354	0.002808	0.014769
n_comms	160.250000	55.783600	44.000000	199.000000

Figure 7: Basic statistics of a few parameters describing the behavior of the 64 processors executing the kernel.

Statistics

Quit Print

Cluster 4 summaries for fire-mpi-64p
# of components: 64

	Cluster 1.4 # components : 17		Cluster 2.4 # components : 3		Cluster 3.4 # components : 25		Cluster 4.4 # components : 19	
	mean	std-dev	mean	std-dev	mean	std-dev	mean	std-dev
t_snd	0.032501	0.011661	0.026943	0.019252	0.078728	0.012390	0.082731	0.011953
t_rcv	0.611484	0.157084	1.445685	0.012563	0.822560	0.140282	1.224259	0.099511
t_wait	0.197003	0.199601	0.054414	0.002575	0.634594	0.045249	0.264163	0.138978
t_barr	1.206798	0.240889	0.005784	0.001736	0.297751	0.185054	0.031888	0.028429

Figure 8: Statistics of the parameters in a partition of the processors into four groups.

5.3 ESI and ECMWF HPF+ kernels

The evaluation of the HPF+ versions of these two kernels is presented together because they exhibit a very similar behavior. Three distinct phases can be recognized corresponding to specific activities performed by the kernels. In the input phase the `host` reads the data from a file and broadcasts this information to the `nodes`. The computation phase represents the core part of each kernel, where computation and communication activities are intermixed on the `nodes` while the `host` is performing a blocking receive operation. In the output phase, the `nodes` send the computed results to the `host`, which prints them.

A single size of the HPF+ version of the ESI kernel has been evaluated, varying the number of allocated processors. Although the size of the input file is relatively small (about 0.5 Mbytes), the number of collected events is very high. The tracefiles for the executions with 3, 5, and 9 processors contain 602726, 742934, and 1058762 events, respectively.

Table 7 summarizes the execution times of the kernel and the times spent by each processor in the input phase, varying the number of processors.

Number of processors	t_exec [s]	t_input [s]
3	16.482	11.824
5	23.502	19.359
9	23.333	19.374

Table 7: Execution times and times spent during the input phase of the kernel, varying the number of allocated processors.

The kernel is characterized by a high communication activity, which mainly takes place for the data distribution, that is, during the input phase. Communication times are 11.83, 19.63, and 20.94 seconds for the three analyzed executions. The average numbers of communicating processors are equal to 2.15, 4.17, and 8.07, respectively.
During the input phase, the number of `MPI_BCAST` events ranges from 35% to 60% of the total number of communication events. For the analyzed executions, the average times for send/receive operations are equal to 0.17, 0.57, and 0.30 seconds only, while the times spent in broadcast activities are 10.25, 17.43, and 18.36 seconds, respectively.
As an example, Figure 9 shows a zoom of a portion of the communication profile of the execution of the kernel with 9 processors; although the time range is relatively small, the number of events is very high, as stated before.

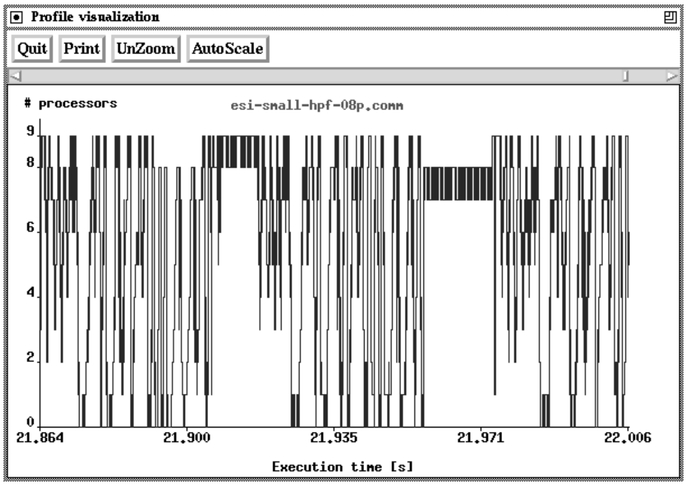


Figure 9: Zoom of a portion of communication profile of the kernel.

A single size of the HPF+ version of the ECMWF kernel (namely, T63, corresponding to an input dataset of 199849 bytes) has been considered and three runs of the kernel (with 5, 9, and 17 processors) have been analyzed. Table 8 summarizes the execution times for each of these runs, together with the contributions of each of the three phases identified in the kernel.

			Phases		
Number of processors	t_exec [s]	t_comm [s]	input [s]	computation [s]	output [s]
5	21.336	13.886	10.958	9.666	0.711
9	21.611	16.974	14.597	6.214	0.800
17	26.321	23.376	19.694	5.905	0.721

Table 8: Timings of the kernel.

As can be seen, a large amount of time is spent in the input phase, namely, 51.36%, 67.54%, and 74.82% of the total execution time for these three runs. The VFCS implements the data distribution by means of broadcast operations which take place after each read operation performed by the `host`.
In order to better characterize the behavior of this kernel, a characterization of the performance, based on a comparative analysis between the code generated by the VFCS and the communication events traced at run–time, has been carried out. For example, in the execution with 17 processors, the code generated by the VFCS contains three explicit types of calls to the MPI primitives, namely, `MPI_BCAST`, `MPI_SEND`, and `MPI_RECV`, plus a call to the `VF_BARRIER` routine which implies an `MPI_BARRIER`. Note that, in order to execute the computation phase,

a number of communication primitives, hidden in the PARTI library, are involved. These communications (MPI_ALLTOALL, MPI_ALLREDUCE, MPI_WAITALL, MPI_GET_COUNT, MPI_ISEND, MPI_PROBE) have been identified in the tracefile. A detailed analysis of the tracefile obtained at run-time is presented in Table 9.

Primitives	# of calls	Primitives	# of calls
MPI_BCAST	452999	MPI_GET_COUNT	3045
MPI_ALLTOALL	9760	MPI_ISEND	3045
MPI_ALLREDUCE	9720	MPI_PROBE	3045
MPI_WAITALL	9696	MPI_SEND	2469
MPI_RECV	5514	MPI_BARRIER	48

Table 9: Characterization of the message passing primitives for a run with 17 processors.

The number of broadcast operations accounts for about 90% of the total number of communications issued by the kernel, since the data distribution is based on this kind of operations. As a consequence, 79% of the communication time is spent in the broadcast of the input data, while the exchange of messages needed while performing the core of the computations is negligible. The communication primitives inserted by the PARTI library account for the 7% of the number of communications, corresponding to about 20% of the total communication time.

5.4 NEC/CCRL HPF+ kernel

A single size of the HPF+ version of the kernel has been executed, corresponding to $40 \times 40 \times 160$ elements, with a number of processors ranging from 5 up to 17. Table 10 summarizes the execution time of each run together with the average communication time spent by each processor.

Number of processors	t_exec [s]	t_comm [s]
5	30.562	23.272
9	27.988	24.369
17	29.355	27.240

Table 10: Execution and communication times for the kernel.

All the executions are characterized by a large average number of processors which simultaneously communicate, that is 3.81, 7.84, and 15.77, respectively. The situation can be slightly improved if the code uses the INSPECTOR/EXECUTOR approach provided within the VFCS. For example, the communication time in the case of an execution with 17 processors decreases to 25.8284 seconds when the INSPECTOR/EXECUTOR is used. This

corresponds to a gain in time of about 5.18%.
Figure 10 shows the distribution of the size of the messages exchanged among the processors in a run with 17 processors. Three main groups of messages can be recognized: the first group contains 96 messages characterized by about 3 bytes, the second has 60 messages of about 62 Kbytes each and the third has 60 messages whose size is about 520 Kbytes.

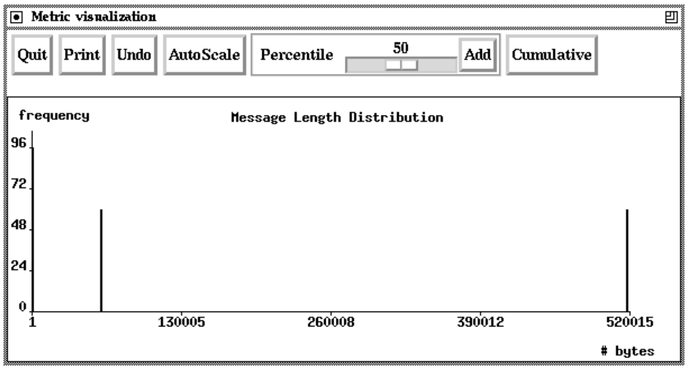


Figure 10: Distribution of the message sizes for an execution with 17 processors.

The large value of the average number of communicating processors reveals evident if we look at the communication profiles (see Fig. 11). The figure shows that a large amount of the execution time is characterized by a steady condition where all the 16 nodes are communicating. From the analysis of the source code, it has been possible to understand that during this phase the `host` is doing physical I/O for saving the results computed by each processor. Correspondingly, each `node` is performing a blocking receive before issuing another send.

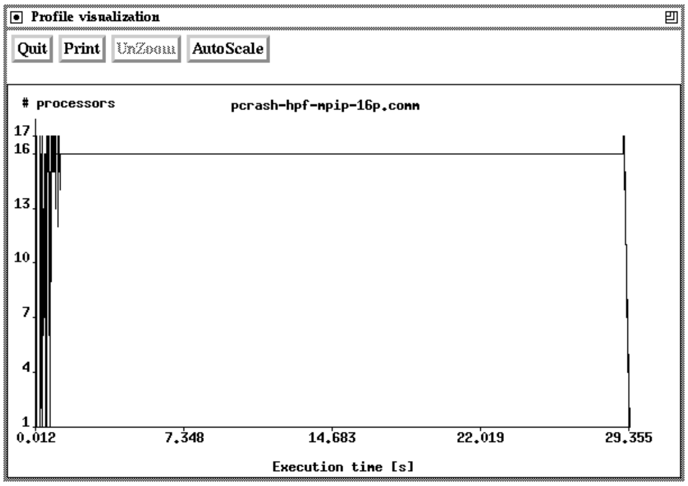


Figure 11: Communication profile of the kernel executed with 17 processors.

The kernel has then been further instrumented in order to better understand its overall

behavior. In the host program three main phases has been recognized: a startup phase (where preliminary information on the execution to be performed are exchanged), and two segments where the results computed by the nodes are written down to files. Each node program contains three main phases as well: two of them correspond to the execution of the routines BOX and KFORCE (where both the domain and the corresponding forces are evaluated) while the third one corresponds to the communications performed by the individual nodes to send their results back to the host.
Table 11 summarizes the percentages of times spent in each of these phases in the case of executions with 9 and 17 processors. The percentages for the host and the node codes have been computed with respect to their execution times. Note that, these phases correspond to the main tasks recognized in the host and node codes: minor routines and computations related to variable initializations, for example, have not been taken into account.

9 processors		17 processors	
Phase	% time	Phase	% time
host: startup	1.63%	host: startup	1.86%
host: write x	26.57%	host: write x	27.89%
host: write f	71.79%	host: write f	70.24%
node: BOX	0.28%	node: BOX	0.19%
node: KFORCE	2.02%	node: KFORCE	1.39%
node: results	95.82%	node: results	95.61%

Table 11: Profiling for the kernel.

5.5 ECMWF MPI kernel

Three different problem sizes (namely, T63, T106, and T213) have been analyzed with a number of processors ranging from 4 up to 64. Table 12 shows the execution time of each run.

Number of processors	t_exec [s]		
	T63	T106	T213
4	6.411	19.225	90.461
8	3.806	9.981	41.378
16	2.166	11.231	30.401
32	5.719	6.367	16.257
64	14.496	16.926	13.022

Table 12: Execution times of the kernel varying the problem size and the number of allocated processors.

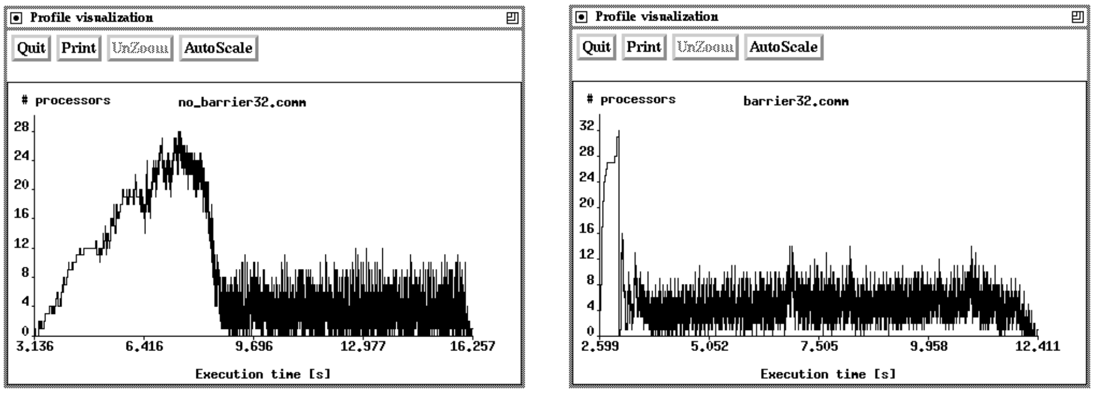
Communication times have been analyzed as well. They are characterized by a high variability. For example, with the T213 problem size, the total communication time accounts from 3.78% of the execution time (in a run with 4 processors) to 20.27% (in a run with 64 processors). More insights into the behavior of the kernel, with respect to the communication times, are summarized in Table 13.

Communication types	T63		T106		T213	
	mean	std dev	mean	std dev	mean	std dev
t_comm	10.966	1.323	11.974	1.755	4.806	1.911
t_send	0.134	0.141	0.177	0.120	0.236	0.117
t_recv	10.832	1.343	11.797	1.755	4.570	1.922

Table 13: Communication times for a run with 64 processors, varying the problem size. Times are expressed in seconds.

As a matter of fact, the communication profile of this kernel exhibits the behavior shown in Figure 12(a). As can be seen, this profile is characterized by a first phase, corresponding to the setup of the kernel, where the data distribution among the processors takes place. Then, a more regular communication pattern is identified, where the successive time iterations are performed. The setup phase takes about 8.6 seconds in a run with 32 processors (about 52% of the execution time) and 68% of the time in a run with 64 processors.

When the number of processors increases, the corresponding communication profile becomes more and more irregular. In order to better understand these results, the analysis was focused on the communication behavior of each individual processor (see Table 14).



(a) (b)

Figure 12: Communication profiles for an execution with 32 processors without (a) and with (b) a synchronization barrier.

Number of processors	t_comm [s]			
	mean	max	min	std dev
32	2.4719	3.5201	0.7290	0.7458
64	4.8062	7.2694	0.2844	1.9109

Table 14: Processor statistics for the kernel.

While in the execution with 32 processors only two nodes have a communication time very different from the mean, the run with 64 processors is characterized by even higher fluctuations. Indeed, 15 processors out of 64 are characterized by a communication time less than 2.9 seconds or greater than 6.7 seconds. This analysis reveals that the processors are not well synchronized. Indeed, the kernel did not include an `MPI_BARRIER` call at the beginning and the end of the setup phase. This synchronization reduces the delays between processors, thus improving the overall performance, as shown in Figure 12(b). Indeed, the gain in the communication time is 51.42% and the execution time decreases by 23.65%.
As far as the computation time is concerned, a detailed study of the most time consuming routine of the kernel has been carried out. Figure 14 shows the corresponding code.
In particular, our attention was focused on the first two loops (`Loop1` and `Loop2`), in order to investigate the overhead involved in addressing mechanisms and loop management activities. The times taken by these loops to complete for the run with 32 processors of the kernel are shown in Figure 13. Note that the execution times for `Loop1` are at least twice those measured for `Loop2`. This behavior has been observed also for executions with 16 and 32 processors.

5.6 Discussion

The preliminary evaluation of the Version–1 of the kernel benchmarks has shown their characterizing behavior. MPI versions as well as HPF+ versions of the kernels have been studied by means of Medea.
Further investigations will be dedicated to refine the analysis and to understand the sources of the "noise" discovered in the various runs.

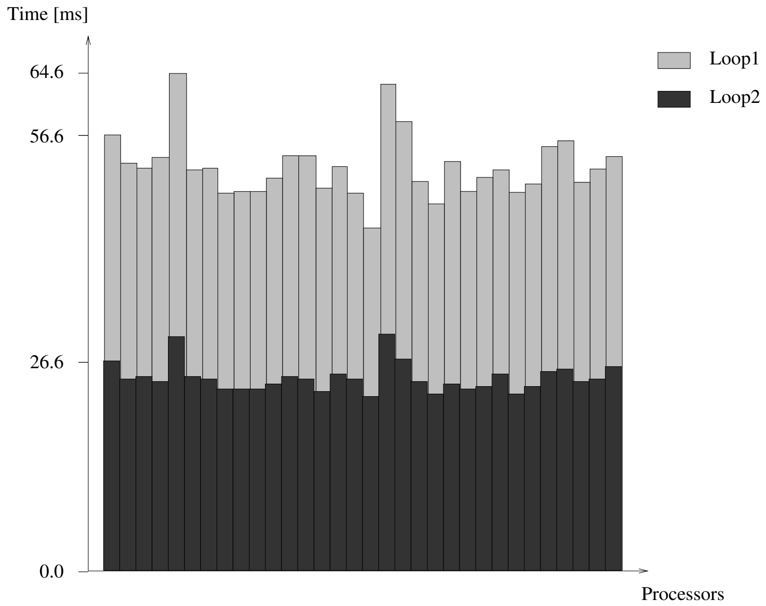


Figure 13: Times taken by `Loop1` and `Loop2` for an execution with 32 processors.

```
c
c     Transpose data used on local processor
c     --------------------------------------

      z0=timef()
      IFLDL=0
                                      Loop1
      DO JVAR=1,KVAR(MYSETB)                              Loop2

        DO JFLD=KFIELD(1,JVAR,MYSETB),KFIELD(2,JVAR,MYSETB)

          ILPOS=IOFF
          IFLDL=IFLDL+1

          DO JGL=1,NDGLL

            DO JL=NSTA(NPTRFLOFF+JGL+IA,MYSETB)+NSTABUF(JGL),
     &            NSTA(NPTRFLOFF+JGL+IA,MYSETB)+NSTABUF(JGL)+
     &            NONL(NPTRFLOFF+JGL+IA,MYSETB)-1
              ILPOS=ILPOS+1
              ZBUF(ILPOS)=PGLAT(JL,IFLDL)
            ENDDO

          ENDDO

          DO JK=1,NGPTOT-JOFF,NPROMA

            IBLOCK=(JK-1)/NPROMA+1
            IKBEG = MAX(IOFF+1,JK)
            IKEND=MIN(NGPTOT-JOFF,JK+NPROMA-1)
            ICOUNT=0
            IF (IOFF+1 .GT. JK) ICOUNT = IOFF + 1 - JK

            DO JK1=IKBEG,IKEND
              ICOUNT=ICOUNT+1
              PGCOL(ICOUNT,JFLD,IBLOCK)=ZBUF(JK1)
            ENDDO

          ENDDO

        ENDDO

      ENDDO
```

Figure 14: Transposition routine on local data.

6 References

References

[1] Lonsdale, G., Clinckemaillie, J., Vlachoutsis, S., Dubois, J., Lecture Notes in Computer Science 796, Springer Verlag, 1994, pp. 55-61.

[2] Lonsdale, G., Elsner, B., Clinckemaillie, J., Vlachoutsis, S., de Bruyne, F., Holzner, M., Lecture Notes in Computer Science 919, Springer Verlag, 1995, pp. 856- 862.

[3] Lonsdale, G., Clinckemaillie, J., Vlachoutsis, S., de Ronde, J.F., Sloot, P.M.A., Floros, N., Reeve, J., "Crashworthiness Simulation migration to distributed memory, MIMD machines", Proc. Supercomputing Applications in the Automotive Industries, 26th ISATA, Aachen, Germany, 13-17 Sept., 1993

[4] Lonsdale, G., Clinckemaillie, J., Vlachoutsis, S., Dubois, J., "Crash- Simulation Migration to HPC Systems", Proceedings, Int. Conf. on Massively Parallel Processing Applications and Development, Delft, 21-23 June, 1994

[5] Brandes, T. ADAPTOR, Programmer's Guide Version 4.0 (April 1996). GMD-SCAI.

[6] L.M Delves, D. Lloyd, and D. Watson. Parallel Extension for HPF+. Draft HPF+ working paper, March 1996.

[7] S. Benkner, Proposed Extensions of HPF 1 for HPF+. Draft Working Paper. VCPC, University of Vienna. April 24, 1996.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.1 TR, Rice University, November 10, 1994.

[9] R. Das, and J. Saltz, A manual for PARTI runtime primitives. University of Maryland, 1995.

[10] HPF+, Optimizing HPF for Advanced Applications (Project No. 21033), Part 2: Scientific and Technical Content.

[11] S. Benkner and al, Vienna Fortran Compilation System. Version 1.2. User's Guide, Institute for Software Technology and Parallel Systems, February 1996.

[12] MEDEA. MEasurements Description, Evaluation and Analysis tool. University of Pavia.

[13] Deliverable 2.1a: HPF+. Initial Language Definition. Sigfried Bencker, Hanz Zima. Institute for software Technology and Parallel Systems. University of Vienna, June 1996.

[14] High Performance Fortran. Version 2.0.. High Performance Fortran Forum. October 19, 1996.

7 Appendix A - Kernels

7.1 AVL HPF+ kernel

```
C=============================================================
      PROGRAM FIREBENCH
C=============================================================
C-----
C-----COMMON
      INCLUDE 'comm.inc'
C-----
C-------------------------------------------------------------
C-----HPF DEFINITIONS
C-------------------------------------------------------------
C-----
!HPF$ PROCESSORS :: PROC(16)
C-----
!HPF$ ALIGN WITH SU :: BP, BE, BW, BN, BS, BH, BL
!HPF$ ALIGN WITH SU :: DIREC2
!HPF$ ALIGN LCC(*,I) WITH SU(I)
!HPF$ ALIGN DIREC1V(*,I) WITH SU(I)
C-----
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC :: SU
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC :: DIREC1
C-----
C-------------------------------------------------------------
C-----READ INPUT DATA FROM IFILE
C-------------------------------------------------------------
C-----
      CALL XREAD
C-----
C-------------------------------------------------------------
C-----INITIALISE VARIABLES
C-------------------------------------------------------------
      RESREF=0.0
      RESNEW=0.0
C-----
      DO 5 NC=NINTCI,NINTCF
        DIREC1(NC)=SU(NC)
    5 CONTINUE
      DO 6 NC=NEXTCI,NEXTCF
        DIREC1(NC)=0.0
    6 CONTINUE
C-------------------------------------------------------------
C-----SUM REDUCTION
C-------------------------------------------------------------
!HPF$ INDEPENDENT, REDUCTION(RESREF)
      DO 10 NC=NINTCI,NINTCF
        RESREF=RESREF+DIREC1(NC)*DIREC1(NC)
   10 CONTINUE
      RESREF=SQRT(RESREF)
C-------------------------------------------------------------
C-----INDIRECT ADRESSING
C-------------------------------------------------------------
!HPF$ INDEPENDENT
      DO 15 NC=NINTCI,NINTCF
```

```
        DIREC1V(1,NC)=DIREC1(LCC(1,NC))
        DIREC1V(2,NC)=DIREC1(LCC(2,NC))
        DIREC1V(3,NC)=DIREC1(LCC(3,NC))
        DIREC1V(4,NC)=DIREC1(LCC(4,NC))
        DIREC1V(5,NC)=DIREC1(LCC(5,NC))
        DIREC1V(6,NC)=DIREC1(LCC(6,NC))
 15   CONTINUE
C-----
      DO 20 NC=NINTCI,NINTCF
         DIREC2(NC)=BP(NC)*DIREC1(NC)
     X          -BS(NC)*DIREC1V(1,NC)
     X          -BW(NC)*DIREC1V(4,NC)
     X          -BL(NC)*DIREC1V(5,NC)
     X          -BN(NC)*DIREC1V(3,NC)
     X          -BE(NC)*DIREC1V(2,NC)
     X          -BH(NC)*DIREC1V(6,NC)
 20   CONTINUE
C---------------------------------------------------------------
C-----SUM REDUCTION
C---------------------------------------------------------------
!HPF$ INDEPENDENT, REDUCTION(RESNEW)
      DO 25 NC=NINTCI,NINTCF
        RESNEW=RESNEW+DIREC2(NC)
   25 CONTINUE
C-----
      STOP
      END
```

7.2 ECMWF HPF+ kernel

```
cat yom77.h

      INTEGER NPROCA
      PARAMETER(NPROCA=    16)
      INTEGER NPROCB
      PARAMETER(NPROCB=     1)
      INTEGER NPROC
      PARAMETER(NPROC=    16)
      INTEGER NGPTOTG
      PARAMETER(NGPTOTG= 13280)
      INTEGER NGT0
      PARAMETER(NGT0=    20)
      INTEGER NGT1
      PARAMETER(NGT1=    20)
      INTEGER NSTOP
      PARAMETER(NSTOP=   100)
      INTEGER MAPGP

      PARAMETER (MAPGPA = (/830,830,830,830,830,
     + 830,830,830,830,830,830,830,830,830,830,830  /))

      INTEGER MAPFLD0
      INTEGER MAPFLD1
      COMMON /MAPS/ MAPGP(NGPTOTG),MAPFLD0(NGT0),MAPFLD1(NGT1)


cat stepo77.F

      SUBROUTINE STEPO

      INCLUDE 'implic.h'
      INCLUDE 'yom77.h'

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

C IF NPROCB GT 1 NEED FOLLOWING
C!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGPA),INDIRECT(MAPFLD0)) ONTO PROCSAB :: ZGL

C IF NPROCB EQ 1 THEN USE FOLLOWING
      REAL ZGA(NGPTOTG,NGT0)
      REAL ZGB(NGPTOTG,NGT0)
      REAL ZGL(NGPTOTG,NGT0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGPA),*) ONTO PROCS :: ZGL
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZGA,ZGB

      REAL ZTIME, TIMERS(NSTOP,2)
      COMMON /TIM/ ZTIME,TIMERS
C
C INITIALISE TIMERS
C
      DO JT=1,2
        DO J=1,NSTOP
```

```
          TIMERS(J,JT)=0.
        ENDDO
      ENDDO
C
C INITIALISE DATA FOR GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT
      DO J=1,NGPTOTG
        DO JFLD=1,NGT0
          ZGA(J,JFLD)=FLOAT(JFLD*1000000+J)
        ENDDO
      ENDDO
C
C TIMESTEPPING LOOP
C
      CALL TIMEF(Z00)
      DO I=1,NSTOP
        CALL TIMEF(Z0)
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          DO JFLD=1,NGT0
            ZGL(J,JFLD)=ZGA(J,JFLD)
          ENDDO
        ENDDO
        CALL TIMEF(Z1)
!HPF$ INDEPENDENT
        DO J=1,NGPTOTG
          DO JFLD=1,NGT0
            ZGB(J,JFLD)=ZGL(J,JFLD)
          ENDDO
        ENDDO
        CALL TIMEF(Z2)
        TIMERS(I,1)=Z1-Z0
        TIMERS(I,2)=Z2-Z1
      ENDDO
      CALL TIMEF(Z99)
      ZTIME=Z99-Z00
C
C CHECK CORRECTNESS OF GTOL AND LTOG TESTS
C
!HPF$ INDEPENDENT
      DO J=1,NGPTOTG
        DO JFLD=1,NGT0
          IF( ZGA(J,JFLD).NE.ZGB(J,JFLD) ) THEN
            CALL ABOR1()
          ENDIF
        ENDDO
      ENDDO

      RETURN
      END
```

7.3 ESI HPF+ kernel

```
      program prcrash
C
      include 'static.h'
C
      COMMON/BK06/NCYCLE
      COMMON/BK25/IELMTC,ITYPTC,ALFMTC
      COMMON/BK02/DT1,DT2
      COMMON/BK03/TIO2,TIOD,PIOT,PIO2,PIOD,DT2OLD,TSSFAC
      CHARACTER*8 ELTYPE(7)
C
      INTEGER INDEX(N4NODE)
      INTEGER IY(6,N4NODE)
      REAL  STRS(NSTR,N4NODE)
      REAL ROEL(N4NODE)
      REAL FORCE1(NDOF,N4NODE)
      REAL FORCE2(NDOF,N4NODE)
      REAL FORCE3(NDOF,N4NODE)
      REAL FORCE4(NDOF,N4NODE)
C
      REAL DT2AR(N4NODE)
      REAL DT2AUX,DT1AUX
      INTEGER ILMAUX,ITYAUX
C
      REAL  XMSI(NDOF,NUMNP)
      REAL     V(NDOF,NUMNP)
      REAL     A(NDOF,NUMNP)
      REAL   XMS(NDOF,NUMNP)
      REAL    BLAMB(9,NUMNP)
      REAL    X1(NDOF,NUMNP)
      REAL     X(3,NUMNP)
      REAL    VGLOB(3,NUMNP)
      REAL    XC (NUMNP),YC(NUMNP),ZC(NUMNP)
      REAL   CM(NPROP,NUMMAT)
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
!HPF$ PROCESSORS :: PROC(NPROC)
C
C    elements
C
!HPF$ DISTRIBUTE(BLOCK) :: INDEX
!HPF$ ALIGN WITH INDEX  :: ROEL,DT2AR
!HPF$ ALIGN (*,I) WITH INDEX(I) :: IY,STRS,FORCE1,FORCE2,FORCE3,FORCE4
C
C     nodes
C
!HPF$ DISTRIBUTE(BLOCK) :: XC
!HPF$ ALIGN WITH XC :: YC,ZC
!HPF$ ALIGN (*,I) WITH XC(I) :: XMSI,V,A,XMS,BLAMB,X1,X,VGLOB
```

```
C
C                     ...........................
C                     ...  read initial values ...
C                     ...........................
C
      open (56,FILE='pcrash.dat',form='formatted')
C
       call readinp( .... )
C
C                     ...........................
C                     ...   Initialization     ...
C                     ...........................
C
       call init( ... )
C
C                     ...........................
C                     ...       Solver         ...
C                     ...........................
C
C
100   CONTINUE
        DT1 = DT2
        DT2 = 1.0E+6
        NECOUN = 0
        IELMTC = 0
        ALFMTC = 1.0
C
        ILMAUX = IELMTC
        ITYAUX = 0
C
C     Zero A
C
       DO N=1,NUMNP
         DO M=1,NDOF
          A(M,N) = 0.0
        END DO
       END DO
C
       DO I=1,N4NODE
         DT2AR(I) = DT2
       END DO
C
C*******************************************************
C                                                      *
C     INTERNAL AND EXTERNAL FORCES                     *
C                                                      *
C*******************************************************
C
!HPF$ INDEPENDENT, ON HOME(IY(2,I)),
!HPF$& NEW (IL,XL,VL,RL,FORCE,STRS1,ROEL1,CM27,YOUNG,ANU,
!HPF$& GM01,GM02,GM03,GM04,GM05,GM06,GM07,GM08,GM09,GM10,GM11,GM12,
!HPF$& DT2AUX,DT1AUX,ILMAUX,ITYAUX)
      DO I = 1, N4NODE
C
```

```
      DT2AUX = DT2AR(I)
      DT1AUX = DT1           ! DT1 is a global value  = DT2 at the previous time-step
C
      IL(1) = IY(2,I)        ! Gather connectivities
      IL(2) = IY(3,I)        ! \
      IL(3) = IY(4,I)        !   quad element
      IL(4) = IY(5,I)        !  /
      IL(5) = IY(1,I)        ! Id  of material
      IL(6) = I              ! Id of element
C
      XL(1)  = X(1,IY(2,I))
      XL(2)  = X(2,IY(2,I))
      XL(3)  = X(3,IY(2,I))
      XL(4)  = X(1,IY(3,I))
      XL(5)  = X(2,IY(3,I))
      XL(6)  = X(3,IY(3,I))
      XL(7)  = X(1,IY(4,I))
      XL(8)  = X(2,IY(4,I))
      XL(9)  = X(3,IY(4,I))
      XL(10) = X(1,IY(5,I))
      XL(11) = X(2,IY(5,I))
      XL(12) = X(3,IY(5,I))
C
      VL(1)  = V(1,IY(2,I))
      VL(2)  = V(2,IY(2,I))
      VL(3)  = V(3,IY(2,I))
      VL(4)  = V(1,IY(3,I))
      VL(5)  = V(2,IY(3,I))
      VL(6)  = V(3,IY(3,I))
      VL(7)  = V(1,IY(4,I))
      VL(8)  = V(2,IY(4,I))
      VL(9)  = V(3,IY(4,I))
      VL(10) = V(1,IY(5,I))
      VL(11) = V(2,IY(5,I))
      VL(12) = V(3,IY(5,I))
C
      RL(1)  = VGLOB(1,IY(2,I))
      RL(2)  = VGLOB(2,IY(2,I))
      RL(3)  = VGLOB(3,IY(2,I))
      RL(4)  = VGLOB(1,IY(3,I))
      RL(5)  = VGLOB(2,IY(3,I))
      RL(6)  = VGLOB(3,IY(3,I))
      RL(7)  = VGLOB(1,IY(4,I))
      RL(8)  = VGLOB(2,IY(4,I))
      RL(9)  = VGLOB(3,IY(4,I))
      RL(10) = VGLOB(1,IY(5,I))
      RL(11) = VGLOB(2,IY(5,I))
      RL(12) = VGLOB(3,IY(5,I))
C
      STRS1(1) = STRS(1,I)
      STRS1(2) = STRS(2,I)
      STRS1(3) = STRS(3,I)
      STRS1(4) = STRS(4,I)
      STRS1(5) = STRS(5,I)
```

```
      STRS1(6) = STRS(6,I)
      STRS1(7) = STRS(7,I)
      STRS1(8) = STRS(8,I)
      STRS1(9) = STRS(9,I)
      STRS1(10)= STRS(10,I)
      STRS1(11)= STRS(11,I)
      STRS1(12)= STRS(12,I)
      STRS1(13)= STRS(13,I)
      STRS1(14)= STRS(14,I)
      STRS1(15)= STRS(15,I)
C
      ROEL1 = ROEL(I)
C
C     Purest subroutine (Computations with only local variables)
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)
C
      DT2AR(I) = DT2AUX
C
      STRS(1,I) = STRS1(1)
      STRS(2,I) = STRS1(2)
      STRS(3,I) = STRS1(3)
      STRS(4,I) = STRS1(4)
      STRS(5,I) = STRS1(5)
      STRS(6,I) = STRS1(6)
      STRS(7,I) = STRS1(7)
      STRS(8,I) = STRS1(8)
      STRS(9,I) = STRS1(9)
      STRS(10,I) = STRS1(10)
      STRS(11,I) = STRS1(11)
      STRS(12,I) = STRS1(12)
      STRS(13,I) = STRS1(13)
      STRS(14,I) = STRS1(14)
      STRS(15,I) = STRS1(15)
C
      FORCE1(1,I) =-FORCE(1)
      FORCE1(2,I) =-FORCE(2)
      FORCE1(3,I) =-FORCE(3)
      FORCE1(4,I) =-GM01
      FORCE1(5,I) =-GM02
      FORCE1(6,I) =-GM03
C
      FORCE2(1,I) =-FORCE(4)
      FORCE2(2,I) =-FORCE(5)
      FORCE2(3,I) =-FORCE(6)
      FORCE2(4,I) =-GM04
      FORCE2(5,I) =-GM05
      FORCE2(6,I) =-GM06
C
      FORCE3(1,I) =-FORCE(7)
      FORCE3(2,I) =-FORCE(8)
```

```
        FORCE3(3,I) =-FORCE(9)
        FORCE3(4,I) =-GM07
        FORCE3(5,I) =-GM08
        FORCE3(6,I) =-GM09
C
        FORCE4(1,I) =-FORCE(10)
        FORCE4(2,I) =-FORCE(11)
        FORCE4(3,I) =-FORCE(12)
        FORCE4(4,I) =-GM10
        FORCE4(5,I) =-GM11
        FORCE4(6,I) =-GM12
C
      END DO
C
!HPF$ INDEPENDENT, ON HOME(IY(2,N)), NEW(N1,N2,N3,N4),REDUCTION(A)
       DO N=1,N4NODE
         DO I=1,NDOF
           A(I,IY(2,N)) = A(I,IY(2,N)) + FORCE1(I,N)
           A(I,IY(3,N)) = A(I,IY(3,N)) + FORCE2(I,N)
           A(I,IY(4,N)) = A(I,IY(4,N)) + FORCE3(I,N)
           A(I,IY(5,N)) = A(I,IY(5,N)) + FORCE4(I,N)
         END DO
       END DO
!HPF$ INDEPENDENT,REDUCTION(DT2)
       DO N=1,N4NODE
         DT2 = MIN(DT2,DT2AR(N))
       END DO
C
C     Transform moments from global to nodal axes
C
      IF (NDOF.EQ.6) THEN
!HPF$ INDEPENDENT
        DO 210 N=1,NUMNP
          A(4,N)=BLAMB(1,N)*A(4,N)+BLAMB(2,N)*A(5,N)+BLAMB(3,N)*A(6,N)
          A(5,N)=BLAMB(4,N)*A(4,N)+BLAMB(5,N)*A(5,N)+BLAMB(6,N)*A(6,N)
          A(6,N)=BLAMB(7,N)*A(4,N)+BLAMB(8,N)*A(5,N)+BLAMB(9,N)*A(6,N)
  210   CONTINUE
      END IF
C
      DT2 = TSSFAC*DT2
      DT2 = MIN(DT2,1.1*DT2OLD)
      DT2OLD = DT2
C
      NCYCLE=NCYCLE+1
C
C*******************************************************
C                                                      *
C     COMPUTE ACCELERATIONS                            *
C                                                      *
C*******************************************************
C
      IF (NDOF.EQ.6) THEN
!HPF$ INDEPENDENT
        DO 350 N=1,NUMNP
```

```
         A(4,N) = A(4,N) + (XMS(5,N)-XMS(6,N))*V(5,N)*V(6,N)
         A(5,N) = A(5,N) + (XMS(6,N)-XMS(4,N))*V(6,N)*V(4,N)
         A(6,N) = A(6,N) + (XMS(4,N)-XMS(5,N))*V(4,N)*V(5,N)
  350   CONTINUE
      END IF
C
!HPF$ INDEPENDENT
      DO N=1,NUMNP
        DO M=1,NDOF
          A(M,N) = A(M,N)*XMSI(M,N)
        END DO
      END DO
C
      DTX=0.5*(DT1+DT2)
C
C     Write output,plot,time history
C
C       Print accelerations, velocities, displacements and stresses
C
       IF (TT.GE.PIOT) THEN
C
         PIOT = PIOT + PIOD
C
         WRITE (6,485) NCYCLE
         WRITE (6,490) TT,DT2,ELTYPE(ITYPTC),IELMTC
C
         WRITE(6,111)
         N = 130
         NOUT = N

         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                     A(1,N),A(2,N),A(3,N),NOUT
         N = 200
         NOUT = N
         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                     A(1,N),A(2,N),A(3,N),NOUT
         N = 476
         NOUT = N
         WRITE(6,222) NOUT,X(1,N),X(2,N),X(3,N),V(1,N),V(2,N),V(3,N),
     .                     A(1,N),A(2,N),A(3,N),NOUT
       END IF
C
C      New time
C
       TT=TT+DT2
C
C*****************************************************
C                                                    *
C     VELOCITY (n+1/2) AND DISPLACEMENTS (n+1)       *
C                                                    *
C*****************************************************
C
       DO N=1,NUMNP
         DO M=1,NDOF
```

```
          V(M,N)  = V(M,N)  + DTX*A(M,N)
          X1(M,N) = X1(M,N) + DT2*V(M,N)
        END DO
      END DO
C
C     Update geometry
C
      DO N=1,NUMNP
        X(1,N) = XC(N) + X1(1,N)
        X(2,N) = YC(N) + X1(2,N)
        X(3,N) = ZC(N) + X1(3,N)
      END DO
C
      IF (TT.GT.TIO2) GOTO 99
C
C*******************
C                   *
      GOTO 100
C                   *
C*******************
C
   99 CONTINUE
C
  485 FORMAT(1H1,17HSOLUTION AT CYCLE,I6/1H ,23(1H*))
  490 FORMAT(' TIME.......................... ',E10.4/,
     .' TIME STEP..................... ',E10.4/,
     .' CONTROLLED BY ',A5,' ELEMENT NB. ',I10)
  111 FORMAT(/19H N O D A L  D A T A//5X,4HN.P.,5X,6HX-COOR,5X,
     .       6HY-COOR,
     .       5X,6HZ-COOR,5X,6H X-VEL,5X,6H Y-VEL,5X,6H Z-VEL,
     .       5X,6HX-ACCL,5X,6HY-ACCL,5X,6HZ-ACCL,5X,4HN.P.)
  222 FORMAT (I8,1X,1P,9E11.3,I8)
C
  505 FORMAT(I10)
  506 FORMAT(E13.4)
C
      END
```

7.4 CCRLE HPF+ kernel

A1: Kernel K1SEQ

```
      program shell_kernel1

      parameter (nelx=40, nelz=160, numnp=4*nelx*(nelz+1), numel2=4*nelx*nelz)
      dimension f(6,numnp), x(3,numnp), dimension ix(4,numel2)

      do i=1,numnp
        do j=1,6
          f(j,i) = 0.0
        end do
      end do

      call box(nelx,nelz,x,ix,numnp,numel2)
      call kforce(f,x,ix, numnp,numel2)
      call kout(x,f,numnp)

      stop
      end
c=====================================================================
      subroutine kforce(f,x,ix, numnp,numel2)

c   Shell-element force calculation model
      dimension f(6,numnp), x(3,numnp), ix(4,numel2), xn(3,4),force(6,4)

c   Node - element - Node

      do iel = 1, numel2
        nn1 = ix(1,iel)
        nn2 = ix(2,iel)
        nn3 = ix(3,iel)
        nn4 = ix(4,iel)
        do i=1,3
          xn(i,1) = x(i,nn1)
          xn(i,2) = x(i,nn2)
          xn(i,3) = x(i,nn3)
          xn(i,4) = x(i,nn4)
        end do

        call mforce(force,xn)

        do i=1,6
          f(i,nn1) = f(i,nn1) + force(i,1)
          f(i,nn2) = f(i,nn2) + force(i,2)
          f(i,nn3) = f(i,nn3) + force(i,3)
          f(i,nn4) = f(i,nn4) + force(i,4)
        end do

      end do

      return
      end
```

A2: Kernel K1HPF+_b1_vfcs

```
      subroutine kforce(f,x,ix,force1,force2,force3,force4,numnp,numel2)

c Shell-element force calculation model
      implicit double precision (A-H,O-Z)
      dimension f(6,numnp), x(3,numnp)
      dimension ix(4,numel2)
      dimension xn(3,4),force(6,4)
      dimension force1(6,numel2),force2(6,numel2), force3(6,numel2),force4(6,numel2)

C Distribution in calling program
C
C     !HPF$ DISTRIBUTE (*,BLOCK) :: f,x,ix,force1,force2,force3,force4
!HPF$ INHERIT f,x,ix,force1,force2,force3,force4
!HPF$ INDEPENDENT, NEW(nn1,nn2,nn3,nn4,xn,force)
      do iel = 1, numel2
        nn1 = ix(1,iel)
        nn2 = ix(2,iel)
        nn3 = ix(3,iel)
        nn4 = ix(4,iel)
        do i=1,3
          xn(i,1) = x(i,nn1)
          xn(i,2) = x(i,nn2)
          xn(i,3) = x(i,nn3)
          xn(i,4) = x(i,nn4)
        end do

        call mforce(force,xn)

        do i=1,6
          force1(i,iel) = force(i,1)
          force2(i,iel) = force(i,2)
          force3(i,iel) = force(i,3)
          force4(i,iel) = force(i,4)
        end do
      end do

!HPF$ INDEPENDENT, NEW(nn1,nn2,nn3,nn4), REDUCTION(f)
      do iel = 1, numel2
        nn1 = ix(1,iel)
        nn2 = ix(2,iel)
        nn3 = ix(3,iel)
        nn4 = ix(4,iel)
        do idim=1,6
          f(idim,nn1) = f(idim,nn1) + force1(idim,iel)
          f(idim,nn2) = f(idim,nn2) + force2(idim,iel)
          f(idim,nn3) = f(idim,nn3) + force3(idim,iel)
          f(idim,nn4) = f(idim,nn4) + force4(idim,iel)
        end do
      end do

      return
      end
```

A3: Kernel K1HPF1_a_PGI

```
      subroutine kforce(f,x,ix,force1,force2,force3,force4, numnp,numel2)
c               Shell-element force calculation model use hpf_library
      integer numnp,numel2
      integer,dimension(4,numel2) :: ix
      real,dimension(6,numnp) :: f
      real,dimension(3,numnp) :: x
      real,dimension(6,numel2):: force1,force2,force3,force4
      real,dimension(3,4) :: xn
      real,dimension(6,4) ::force
      integer :: tkf0,tkf1,tkf2,tick

!HPF$ DISTRIBUTE (*,BLOCK) :: f,ix,force1,force2,force3,force4
!HPF$ DISTRIBUTE (*,BLOCK) :: x
!HPF$ DISTRIBUTE (*,*) :: force,xn
      interface
        pure subroutine mforce(force,x)
          real,dimension(6,4),intent(out) ::  force
          real,dimension(3,4),intent(out) ::  x
        end subroutine
      end interface
!HPF$ INDEPENDENT,NEW(nn1,nn2,nn3,nn4,xn,force) ON HOME (ix(1,iel))
      do iel = 1,numel2
        nn1=ix(1,iel)
        nn2=ix(2,iel)
        nn3=ix(3,iel)
        nn4=ix(4,iel)
        do j=1,3
          xn(j,1) = x(j,nn1)
          xn(j,2) = x(j,nn2)
          xn(j,3) = x(j,nn3)
          xn(j,4) = x(j,nn4)
        end do

        call mforce(force,xn)

        do j=1,6
          force1(j,iel) = force(j,1)
          force2(j,iel) = force(j,2)
          force3(j,iel) = force(j,3)
          force4(j,iel) = force(j,4)
        end do
      end do

      do idim=1,6
        f(idim,:) = SUM_SCATTER(force1(idim,:),f(idim,:),ix(1,:))
        f(idim,:) = SUM_SCATTER(force2(idim,:),f(idim,:),ix(2,:))
        f(idim,:) = SUM_SCATTER(force3(idim,:),f(idim,:),ix(3,:))
        f(idim,:) = SUM_SCATTER(force4(idim,:),f(idim,:),ix(4,:))
      end do

      return
      end
```

A4: Kernel K1MPI_a

```
      subroutine local(numnp,nni,nna,nnb,numel2, x, f, ix,
     +                 nnbpa,nnbpb,dima,dimb, maila, mailb, pnta, pntb,
     +                 lnbpa, lnbpb, buffer, dimbuf, tcomms, error)

c       Calculation & communication "driver"
c       Note: Variable numnp is the size of LOCAL data
      implicit none
      include 'mpif.h'
      real x(3,numnp), f(6,numnp), real buffer(dimbuf)
      integer ix(4,numel2)
      integer maila(dima), mailb(dimb), pnta(nnbpa+1), pntb(nnbpb+1),
      integer lnbpa(nnbpa), lnbpb(nnbpb)
      integer numnp, numel2, nni, nna, nnb, nnbpa, nnbpb, dimbuf, dima, dimb, error
      integer i, j, itag, offb1, offb2, offb3, db1, db2, db3
      double precision t1,t2,tlocal,tcomms(2)

      do i=1,numnp
        do j=1,6
          f(j,i) = 0.0
        end do
      end do

c         split buffer (for mpi request & status arrays)
      db1 = nnbpa+nnbpb
      db2 = MPI_STATUS_SIZE*db1
      db3 = dimbuf - db1 - db2
      offb1 = 1
      offb2 = offb1 + db1
      offb3 = offb2 + db2

c         a- to b-node transmission for x  (note: unecessary for 1 cycle)
      t1 = MPI_WTIME()
      itag = 100
      call xa2b(itag,x,maila,mailb,pnta,pntb,lnbpa,lnbpb,
     +          numnp,nnbpa,nnbpb,dima,dimb,
     +          buffer(offb1),buffer(offb2),buffer(offb3),db1,db2,db3,
     +          error)
      t2 = MPI_WTIME()
      tcomms(1)=t2-t1

c         Local computation

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

c         b- to a- transmission-sum for f
      t1 = MPI_WTIME()
      itag = 1000
      call fb2a(itag,f,maila,mailb,pnta,pntb,lnbpa,lnbpb,
     +          numnp,nnbpa,nnbpb,dima,dimb,
     +          buffer(offb1),buffer(offb2),buffer(offb3),db1,db2,db3,
     +          error)
      t2 = MPI_WTIME()
```

```
      tcomms(2)=t2-t1

      return
      end

      subroutine  xa2b(itag,x,maila,mailb,pnta,pntb,lnbpa,lnbpb,
     +                 numnp,nnbpa,nnbpb,dima,dimb,
     +                 req,status,buffer,db1,db2,db3, error)
c
c         update of x values at b-nodes
c           non- blocking receives followed by non-blocking sends

      implicit none
      include 'mpif.h'
      real x(3,numnp),buffer(db3)
      integer maila(dima), mailb(dimb), pnta(nnbpa+1), pntb(nnbpb+1)
      integer lnbpa(nnbpa), lnbpb(nnbpb)
      integer numnp, numel2, nnbpa, nnbpb, db1, db2, db3, error
      integer req(nnbpa+nnbpb), status(MPI_STATUS_SIZE,nnbpa+nnbpb)
      integer mpierr, itag, dima, dim, cnt, ip, nnodes, in, proc, start

      cnt = 1
      do ip = 1, nnbpb
        proc = lnbpb(ip)
        nnodes = pntb(ip+1)-pntb(ip)

        call MPI_IRECV(buffer(cnt),4*nnodes,MPI_REAL,proc,itag,
     +                 MPI_COMM_WORLD,req(ip),mpierr)

        cnt = cnt + 3*nnodes
      end do

      do ip = 1, nnbpa
        proc = lnbpa(ip)
        start = cnt
        nnodes = pnta(ip+1)-pnta(ip)
        do in = pnta(ip), pnta(ip+1)-1
          buffer(cnt)   = x(1,maila(in))
          buffer(cnt+1) = x(2,maila(in))
          buffer(cnt+2) = x(3,maila(in))
          cnt = cnt + 3
        end do

        call MPI_ISEND(buffer(start),3*nnodes,MPI_REAL,proc,itag,
     +                 MPI_COMM_WORLD,req(ip+nnbpb),mpierr)

      end do

      call MPI_WAITALL(nnbpa+nnbpb,req,status,mpierr)

c             Extract data from receipt buffers
      cnt = 1
      do ip = 1, nnbpb
        do in = pntb(ip), pntb(ip+1) - 1
```

```
          x(1,mailb(in)) = buffer(cnt)
          x(2,mailb(in)) = buffer(cnt+1)
          x(3,mailb(in)) = buffer(cnt+2)
          cnt = cnt + 3
        end do
      end do

      return
      end

      subroutine  fb2a(itag,f,maila,mailb,pnta,pntb,lnbpa,lnbpb,
     +                numnp,nnbpa,nnbpb,dima,dimb,
     +                req,status,buffer,db1,db2,db3,
     +                error)
c
c         update of f values at a-nodes (gather and sum)
c           non- blocking receives followed by non-blocking sends

      implicit none
      include 'mpif.h'
      real f(6,numnp),buffer(db3)
      integer maila(dima), mailb(dimb), pnta(nnbpa+1), pntb(nnbpb+1)
      integer lnbpa(nnbpa), lnbpb(nnbpb)
      integer numnp, numel2, nnbpa, nnbpb, db1, db2, db3, error
      integer req(nnbpa+nnbpb), status(MPI_STATUS_SIZE,nnbpa+nnbpb)
      integer mpierr, itag, dima, dimb, integer cnt, ip, nnodes, in, proc, start
      cnt = 1
      do ip = 1, nnbpa
        proc = lnbpa(ip)
        nnodes = pnta(ip+1)-pnta(ip)
    call MPI_IRECV(buffer(cnt),6*nnodes,MPI_REAL,proc,itag,
     +                MPI_COMM_WORLD,req(ip),mpierr)
        cnt = cnt + 6*nnodes
      end do
      do ip = 1, nnbpb
        proc = lnbpb(ip)
        start = cnt
        nnodes = pntb(ip+1)-pntb(ip)
        do in = pntb(ip), pntb(ip+1)-1
          buffer(cnt)   = f(1,mailb(in))
          buffer(cnt+1) = f(2,mailb(in))
          buffer(cnt+2) = f(3,mailb(in))
          buffer(cnt+3) = f(4,mailb(in))
          buffer(cnt+4) = f(5,mailb(in))
          buffer(cnt+5) = f(6,mailb(in))
          cnt = cnt + 6
        end do
        call MPI_ISEND(buffer(start),6*nnodes,MPI_REAL,proc,itag,
     +                MPI_COMM_WORLD,req(ip+nnbpb),mpierr)
      end do
      call MPI_WAITALL(nnbpa+nnbpb,req,status,mpierr)
c             Extract & sum data from receipt buffers
      cnt = 1
      do ip = 1, nnbpa
```

```
  do in = pnta(ip), pnta(ip+1) - 1
    f(1,maila(in)) = f(1,maila(in)) + buffer(cnt)
    f(2,maila(in)) = f(2,maila(in)) + buffer(cnt+1)
    f(3,maila(in)) = f(3,maila(in)) + buffer(cnt+2)
    f(4,maila(in)) = f(4,maila(in)) + buffer(cnt+3)
    f(5,maila(in)) = f(5,maila(in)) + buffer(cnt+4)
    f(6,maila(in)) = f(6,maila(in)) + buffer(cnt+5)
    cnt = cnt + 6
  end do
end do
return
end
```