Optimizing HPF for Advanced Applications

Workpackage 1: Applications

Final Report on project benchmarks

Deliverable:	D1.1b
Date:	March 1998
Status:	Draft
Confidentiality:	Confidential
Responsible:	ESI

Revised: 06/03/1998

(Project No 21033)

Abstract

This report corresponds to the deliverable d1.1b of the HPF+ project, describing the final status of the project benchmark kernels using the HPF language and its extensions as proposed by the consortium. This report is the second of the task 1.1 of the project, which focuses on the practical usage of HPF. Its contents reflect some important improvements of the language definition developed during the project if it is to be used efficiently in advanced industrial parallel applications.
The application of this project, the CFD code FIRE of AVL, the IFS code of the ECMWF and the crashworthiness program PAM-CRASH of ESI are relevant in the perspective and cover a very wide and representative range of algorithmic details. Besides, the new range of algorithms described in this report increase the difficulty for HPF compilers to generate some more efficient codes.
The kernels are being analyzed for conversion to the HPF+ paradigm with maintainability and efficiency when it was possible as prime goals. In the sequel, for each application, the most relevant programming structures are analyzed and HPF+ equivalents presented. Problems encountered and appropriate solutions are discussed.
In section 1 the HPF+ kernel from AVL contains the entire conjugate graduate solver for the solution of sparse linear equation systems. Section 2 contains a description of the three final kernels from ECMWF, representing different aspects of the weather forecast application IFS. A new kernel has been developed which performs the data transformations of an IFS time-step, from physical (or grid-point) space to Fourier space and then to spectral space. Moreover, one kernel is provided in an enhanced form to utilize the new halo capability of VFC. Finally, in section 3 and 4 kernels from ESI and NEC/CCRLE are described. These kernels represent challenging problems from the ESI PAMCRASH application, including variants of a simplified version of the contact-impact algorithm where the dynamic behavior and the high level of indirect addressing reflect the requirements to extend the HPF capabilities of compilers. Compilers will have to handle much more complex parallel loops than those studied in the previous report. Besides, the last kernels are also enhanced with the new HALO directive developed by the University of Vienna during the project.

Optimizing HPF for Advanced Applications

Optimizing HPF for Advanced Applications

1. AVL

1.1 Introduction

The final version of the AVL kernel contains the time marching solution of a passive scalar transport equation. This transport equation is part of the FIRE program and is a representative subset of the code that contains all requirements that are necessary for a porting of the entire FIRE program to HPF+. The serial version of this kernel was developed in FORTRAN 90 using the F90 array syntax and modules. This kernel was ported with slight modifications to HPF+, where in particular the indirect data distribution and schedule reuse were essential for a successful porting and for the scalability of the kernel. In addition to these kernels a message passing version under MPI was developed in order to have a comparison of the porting effort of the different parallelisation strategies and to compare the performance of the two kernels.
The following section gives an overview of the kernel and its characteristics,followed by a detailed description of the HPF+ version and the HPF+ features used. Finally an overview of the MPI version will be given.

1.2 HPF+ Kernel Version 3

For the numerical simulation of a passive scalar transport equation convection, diffusion and source terms have to be coded. The steady velocity field is read from an input data set, which was extracted from a FIRE calculation. The initial distribution of the scalar at the internal domain and at the inlet boundaries is setup at the beginning of the calculation. In order to minimise the size of the input data set the cross diffusion term, which requires the indirect access to diagonal neighbour cells, has been neglected. This reduces the size of the connectivity array LCC array by 12 times the number of internal cells. The discretisation of the convection term has been implemented by two different differencing schemes that can be used alternatively: the first order upwind differencing scheme and the second order hybrid differencing scheme, which is a blending of upwind and central differencing depending on the local grid Peclet number. For the solution of the linear equation system the conjugate gradient solver of kernel version 2 was used.
A main target was to maintain absolutely identical calculations for all three kernels (serial, HPF+ and MPI). For the verification of the correctness of the results at each time step the number of iterations and the latest computed residual are written to a file.

1.3 Data Distribution

The HPF+ kernel consists out of 17 subroutines and one module, where the global arrays and variables are defined. The data structure is the same as in the FIRE program: there are arrays containing cell center based data, and arrays containing cell face based data. Cell center based arrays can either be defined over the internal cells, the external cells (for the boundary conditions) or over all cells (internal and external). Internal, cell-center based arrays are block distributed, while arrays over the external or total number of cells are distributed indirectly. All external cells are distributed in a way that adjacent external and internal cells are sent to the same processor: every pair of internal and external cells NC, NE where LCC(I,NC)=NE is distributed to the same processor.
Cell face based data are also distributed indirectly. In order to minimise the communication, cell face array elements are not distributed according to their numbering. The array elements are distribution in a way that all cell faces of one specific cell are distributed to the processor where the cell center element has been distributed to.
For the indirect distributed arrays 5 different map arrays have to be read at runtime. These map arrays are generated by a separate program previous to the calculation, that reads the input data set, computes the correct data distribution for a specific number of processors and writes the distribution to a file. A dynamic generation of the map arrays during runtime is not possible because for the calulation of the map array the LCC connectivity array is necessary, which is a distributed array.
Like for the previous benchmark versions a one dimensional processor array is used. In the following code part the data distribution is shown:

```
!HPF$ PROCESSORS NODES(NUMBER_OF_PROCESSORS())

!HPF$ DYNAMIC :: LCC, LCF
!HPF$ DYNAMIC :: PS, U1, U2, U3

!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: PSO, RDUM
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: VOL
!HPF$ DISTRIBUTE(BLOCK) ONTO NODES :: SU4, SP4

      OPEN(20,STATUS='OLD',FILE='map1',FORM='FORMATTED')
      READ(20,*) map1
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP1)) ONTO NODES :: LCC, LCF

      OPEN(20,STATUS='OLD',FILE='map4',FORM='FORMATTED')
      READ(20,*) MAP4
      CLOSE(20)
!HPF$ REDISTRIBUTE(INDIRECT(MAP4)) ONTO NODES :: PS, U1, U2, U3
```

Processor array, dynamic declaration and block distribution are defined in the module, while the reading of the map array and the redistribution are defined in the main program. This data distribution strategy does not take any geometrical information into account. The distribution depends strongly on the numbering of the cells and as a consequence disconnected subdomains will occur for block-structured grids. The amount of communication can be reduced when using different mesh partitioning methods for the generation of the map arrays.

1.4 Communication

The data distribution remains invariant over the whole computation therefore the reuse concept can be used successfully for all loops with indirect data access.

```
!HPF$ INDEPENDENT, ON HOME(ACOEFF(NC)), REUSE
      DO 1 NC=NINTCI,NINTCF
          ACOEFF(NC)=ACOEFF(NC+NSFR01)
     X              +ACOEFF(NC+NSFR02)
     X              +ACOEFF(NC+NSFR03)
     X              +ACOEFF(NC+NSFR04)
     X              +ACOEFF(NC+NSFR05)
     X              +ACOEFF(NC+NSFR06)
     X              -SP4(NC)
    1 CONTINUE
```

The fast implementation of reduction operations is essential for the scalability of the code. Sum reduction operations had to be implemented several times in the conjugate gradient equation solver and in the calculation of the residual. Within each time step several hundret reduction operations have to be performed.

There are up to three sum reductions within one loop:

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

In the loop above the direction vector DIREC1 is indirectly accessed over the connectivity array LCC. All coefficients are stored in the one dimensional array ACOEFF where the offset to the main directions are defined by the variables NSFR01, NSFR02, .. NSFR06.

The datasets that have been extracted for the kernels version 1 and 2 can not be used for the final benchmark version, because several input data (e.g.: cell face transformation coefficients) are missing. For benchmark version 3 a dataset from a FIRE application was extracted: the 3 dimensional, block-structured grid of a duct, with inlet velocity and outlet static pressure boundary conditions. The grid consists of 26400 internal and 4320 external cells (30720 total cells). The formatted input data (the size is 8.3 MB) contains the following arrays: the connectivity arrays LCC and LCF, the cartesian velocity components in x, y and z-direction, cell volumes weighting factors into the 6 main directions, cell face transformation coefficients and all necessary boundary condition data.

1.5 MPI benchmark

The MPI version is based on the same serial kernel as the HPF+ version. Several additional routines had to be developed for the communication, e.g.: routines for the generation of the send and receive lists and for the local data exchange. All kernels (MPI, HPF+ and serial) read from the same input data set.
The data distribution is performed during runtime, where the master node reads the input dataset and distributes the data in blocks of equal size to the other processors. This distribution is equivalent to the block distribution of the HPF+ kernel. The external cells belong to the same processor as the adjacent internal cell, in the same way as it is implemented in the HPF+ version.
Local data exchange is necessary at the end of every loop where array elements are updated that are accessed indirectly over one of the connectivity arrays. The communication is performed by using an "all send - all receive" strategy, and is generally non-symmetric. The size of data packages that have to be communicated, strongly depends on the quality of the data distribution and on the number of compute nodes. Global data exchange is performed by standard MPI reduction operations.

1.6 Summary

Within version 3 of the AVL benchmark the following HPF+ features are used: The data distribution is implemented by the DISTRIBUTE(BLOCK), DYNAMIC and REDISTRIBUTE statements. For all independent loops the INDEPENDENT directive together with the REUSE clause are used. The ON HOME clause is used as an additional directive within independent loops. The RESIDENT statement is used for loops with indirect data access, where no communication is required. Sum reduction operations are implemented with the REDUCTION statement inside of INDEPENDENT loops.

2. ECMWF

2.1 Introduction

This section describes the final kernels provided by ECMWF in the framework of the HPF+ Project. These kernels which are described in the subsequent sections have been coded to use Fortran 90 modules to allow for greater portability to other HPF compilers. In particular, the use of POINTER arrays that were in common blocks, have been converted to ALLOCATABLE arrays in modules. A new kernel called TS has been developed which performs the data transformations of an IFS time-step, from physical (or grid-point) space to fourier space and then to spectral space.
This kernel was developed from the version 2 LG and LM kernels and by including fourier and legendre transforms as used in the IFS. The version 2 kernel LG (now coded for Fortran 90 modules) is provided specifically to evaluate a variant that could not be run in the corresponding evaluation due to constraints applied by the VFC release 0.91 compiler. Likewise, the version 2 kernel SL (also coded for modules) is provided in an enhanced form to utilise the new halo capabilitiy of VFC.
For each of the kernels both HPF+ and message passing versions are supplied, and in the case of the TS and LG kernels a separate serial version is alsoprovided. All kernels are written in F90 and include both a serial initialisation phase and a parallelised time-stepping phase in the same program.

2.2 TS Kernel

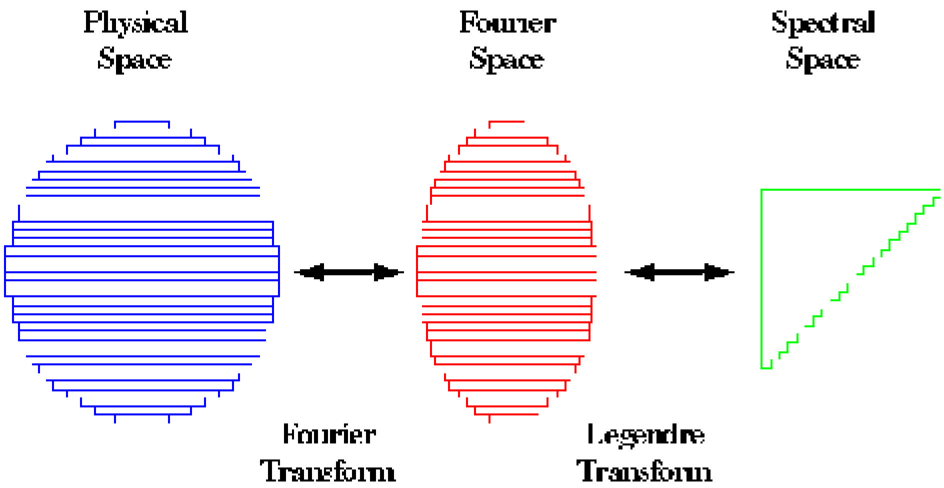

Figure 1: Spectral transform method

The TS kernel uses fourier and legendre transforms to transform data between grid-point space and spectral space, as shown in Figure1. In a single time-step data is transformed from grid-point space to spectral space and back to grid-point space. These transformations are a fundamental part of the IFS spectral model and are non-trivial for HPF compilers as the above data spaces cannot be efficiently represented by BLOCK or CYCLIC distributions. Instead, GEN_BLOCK and INDIRECT are needed.
Two HPF+ variants of the TS kernel have been coded, with the fourier and legendre transforms being performed by using the extrinsic HPF_LOCAL or by using INDEPENDENT loops.

The use of HPF_LOCAL for the transforms is expected to provide good performance, however, the variant using INDEPENDENT loops will require some existing VFC compiler constraints to be relaxed to obtain performance, in particular, use of non-invariant bounds for independent loops (i.e. bounds of inner loops dependent on outer loops) subscript expressions of distributed arrays referencing more than one loop variable, e.g. A(IND(i)+j) is required where i and j are loop control variables.

2.3 LG Kernel

The LG kernel is provided to address the IFS program issues of grid-point space, namely, Indexed access to latitude data structures. This is required as the LTOG and GTOL transpositions need to redistribute grid-point space where each latitude includes some extra space required by the FFT package used at ECMWF. E-W data dependencies of Fast Fourier Transforms (SIMFFT). Vertical dependencies of physics calculations (SIMPHYS). Blocking of grid-point space to conserve memory usage in physics calculations and exploit locality of reference.

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

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

      DO JFLD=1,NGT0
        IB=0
        DO J=1,NGPTOTG,NPROMA
          IBEG=1
          IEND=MIN(NPROMA,NGPTOTG-J+1)
          IB=IB+1
          DO JJ=IBEG,IEND
            ZTMP(J+JJ-1)=ZGA(JJ,JFLD,IB)
          ENDDO
        ENDDO
        DO JGL=1,NDGLG
          DO JL=1,NLOENG(JGL)
            ZGL(NSTABUFL(JGL)+JL,JFLD)=ZTMP(NSTABUFG(JGL)+JL-1)
          ENDDO
        ENDDO
      ENDDO
```

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

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),INDIRECT(MAPFLD0)) ONTO PROCSAB :: ZGL
      REAL ZGL(NPROMAG,NGT0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZGA
      REAL ZGA(NGPTOTG,NGT0)
      ...
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JFLD=1,NGT0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL(INDL(J),JFLD)=ZGA(J,JFLD)
          ENDDO
        ENDDO
```

As can be seen in the above fragment, the LG kernel uses (GEN_BLOCK,INDIRECT) distributions for ZGL. The use of an INDIRECT distribution to distribute (NGT0) fields over NPROCB processors is required as both 3D variables (i.e. those with a vertical dimension) and 2D variables (surface fields) are included in this dimension. In this case we need to keep the same set of vertical levels for all 3D variables distributed to the same NPROCB processor. However, if we hold 3D and 2D variables in separate arrays we can avoid the use of an INDIRECT distribution and use a much more efficient BLOCK distribution for both, as follows,

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK,*) ONTO PROCSAB :: ZGL3
      REAL ZGL3(NPROMAG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),BLOCK) ONTO PROCSAB :: ZGL2
      REAL ZGL2(NPROMAG,NG2D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*,*) ONTO PROCS :: ZGA3
      REAL ZGA3(NGPTOTG,NFLEVG,NG3D0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS :: ZGA2
      REAL ZGA2(NGPTOTG,NG2D0)
      ...
!HPF$ INDEPENDENT,NEW(JLEV,J),REUSE(LREUSE)
        DO JVAR=1,NG3D0
!HPF$ INDEPENDENT
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT
            DO J=1,NGPTOTG
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT,NEW(J),REUSE(LREUSE)
        DO JVAR=1,NG2D0
!HPF$ INDEPENDENT
          DO J=1,NGPTOTG
            ZGL2(INDL(J),JVAR)=ZGA2(J,JVAR)
          ENDDO
        ENDDO
```

This is at the expense of doubling the size of such code sections, but acceptable given the potential gain in performance. In the HPF+ kernel we have used HPF_LOCAL as a solution to the memory usage/blocking issue of grid-point space, however, it is also interesting to explore whether HPF can still be used efficiently in physics routines should we choose to accept the memory overhead. A version of SIMPHYS has therefore been coded using HPF+ directives to explore this issue.
In summary, 3 variants of the LG/HPF kernel are provided, selectable at runtime,

- Single array/distribution for 3D and 2D grid-point space variables. Use of HPF_LOCAL for SIMFFT and SIMPHYS.
- Separate arrays/distributions for 3D and 2D grid-point space variables. Use of HPF_LOCAL for SIMFFT and SIMPHYS.
- Separate arrays/distributions for 3D and 2D grid-point space variables. Use of pure subroutine for SIMFFT. Use of HPF+ version of SIMPHYS (with high memory requirement).

2.4 SL Kernel

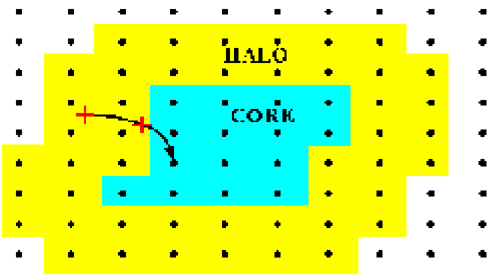

Figure 2: Semi Lagrangian method, showing computation of trajectory, requiring interpolation of grid-point data held on neighbouring processors (halo area)

The semi-Lagrangian (SL) calculation in the IFS consists of two parts, the computation of a trajectory from a grid point backwards in time, and the interpolation of various quantities at the departure point and atthe mid-point of the trajectory. The issue here for distributed memory systems is that both these parts require access to grid-point data held on neighbouring processors. The approach used in the IFS today is to construct a halo of all grid columns from these processors based on the global maximum wind likely to be encountered as shown in Figure 2.
The SL kernel computes such a halo, and in the case of the HPF+ version uses a HALO specification with a update_halo directive to obtain the halo data, as shown below,

```
!HPF$ PROCESSORS PROCS(NPROC)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP,HALO=NSLHALO),*) ONTO PROCS :: ZSL1
      REAL ZSL1(NGPTOTG,NFLDSLB1)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP),*) ONTO PROCS :: ZSL2
      REAL ZSL2(NGPTOTG,NFLDSLB1)
      ...
      DO I=1,NSTOP
        ...
!HPF$ UPDATE_HALO(ZSL1)
        ...
        DO JSL=1,ISL
!HPF$ INDEPENDENT,NEW(J),REDUCTION(ZSL2)
          DO JLEV=1,NFLEVG
!HPF$ INDEPENDENT,NEW(Z1,Z2,Z3,Z4,Z5,Z6)
            DO J=1,NGPTOTG
              Z1=ZSL1(NSLIND(1,J,JSL),JLEV)
              Z2=ZSL1(NSLIND(2,J,JSL),JLEV)
              Z3=ZSL1(NSLIND(3,J,JSL),JLEV)
              Z4=ZSL1(NSLIND(4,J,JSL),JLEV)
              Z5=ZSL1(NSLIND(2,J,JSL),MIN(NFLEVG,JLEV+1))
              Z6=ZSL1(NSLIND(3,J,JSL),MAX(1,JLEV-1))
              ZSL2(J,JLEV)=ZSL2(J,JLEV)+
     +          (Z1+Z2+Z3+Z4+Z5+Z6)/(6.*ISL)
            ENDDO
          ENDDO
        ENDDO
        ...
      ENDDO
```

The SL interpolation routines in the IFS are very complex, and for this reason are only approximated in this kernel, by a simple average of neighbouring grid-points. In the above code fragment we would expect to see all the SL communication being performed by the UPDATE_HALO directive and no communication in the following INDEPENDENT loop. Furthermore, we should expect to see the benefit of reduced communication as we run the kernel with square-like partitions over running the kernel with long thin partitions, by specifying appropriate values for NPROCA and NPROCB at run time.

3. ESI

3.1 Introduction

This report discusses the HPF port of the final release of the PAM-CRASH benchmarks in the framework of the Esprit Project HPF+. The different benchmark kernels are representative of two principal functional units of the PAM-CRASH solver. The goal of the kernels is to incorporate some program features needed to create a more complete HPF benchmark with a better expressiveness, maintainability and optimized in terms of performance. New kernels of different levels of complexity w.r.t. HPF language features were produced and are to be adapted following the different releases of the tools, as they become available during the course of the project.

The HPF+ benchmarks were derived from release V1996 of the standard, commercial PAM-CRASH code. The major difficulty was to extract the salient parts of the solver in order to create a realistic set of benchmark kernels, containing all representative data-structures, data dependencies and inter-processor communication structures. For each of the kernels both HPF+ and message passing versions are supplied. All HPF+ kernels are written in F90 and include both an input phase and a parallelised time-stepping phase in the same program. The message-passing versions are written in F77. All kernels use the same input files pre-computed by a modified version of the standard PAM-CRASH code. Besides, the message-passing kernels including the contact-impact routines use the same kind of data-structures than the production code and will be necessarily more optimized than the HPF+ kernels in term of CPU and memory requirements. First, a brief review of the PAM-CRASH modules used in the kernels is given, followed by a description of the different kernels as defined during the second project year. Next, a more detailed description of the contact kernels, treating contact-impact algorithms, is presented. The main problems with this set of benchmarks lie in the data locality and complex statements in independent loops.

3.2 Outline of the benchmarks

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

- stress-strain calculations
- contact-impact calculations

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

3.3 HPF+ kernel Version 3

The final HPF+ version of the benchmark is a subset of the finite element solver and can be represented by implementing the stress-strain calculations of a typical element formulation. The computations, executed on an element level, require as input the updated coordinate and velocity vectors at the nodal points defining the element and defines the strain rates to which the element is subject. The resultant forces on the element are then computed from the stresses through a consistent integration over the element volume. These forces are then transferred to the individual nodal points or vertices of the element. The computation of the internal forces represents the most important contribution to the overall computational cost (between 60% and 80%, depending on the particular problem set) and exhibits a high degree of data locality. In the second part, the contact-impact algorithms play a very important role because crash phenomena are largely influenced by intricate contact events. Much work has therefore gone into the elaboration of efficient and reliable contact-impact algorithms. Beside, contact adds a significant cost to the overall solution even when the ratio of the number of contact points to the number of elements is small. The number of floating point operations performed in the contact calculations is not large as compared with the element calculations, but the contact treatment takes a disproportional large amount of time. For a parallel implementation, the data-dependencies are more difficult to resolve because of parallel loops with conditional branching and bounds, which depend of distributed variables. The use of pointers on distributed arrays intensifies the complexity of the HPF constructs needed. Besides, we need to construct the list of nodes that determine which master node is closest to the slave node. Actually, the HPF compilers (VFC, PGI) used does not handle correctly all features needed to have a complete version of the contact algorithm described in the previous reports. Thus, in close collaboration with the compiler partners from the University of Vienna, a version without the bucket search and others features about the dynamically defined lists was written to be compliant with the VFC.

Some separate versions of this kernel of increasing levels of complexity w.r.t HPF language features including or not the contact routines with different options were also produced.

3.3.1 Data distribution

In the first version of the HPF kernel, all data structures used the BLOCK directive to distribute nodes and elements. This is the first distribute directive provided by the HPF-1 specifications. In the final version, all mapped arrays are dynamically allocated. We always have to distribute elements in such way that all nodes that belongs to local elements reside as mush as possible on the same processor. Besides, variants of the HPF kernel include the GEN_BLOCK distribute directive for elements and a weight function defined by an input file was included to mimic the behavior of different elements which can produce some load-imbalance during computation. Besides, the INDIRECT and the ALIGN directives are now used to distribute the nodal arrays. The DYNAMIC attribute is also used the first time to evaluate restrictions in some parallel loops. Unfortunately, it was not yet possible to use correctly the DYNAMIC attribute because of constraints in independent loops generated by compilers. However, in the meantime the RANGE facility was developed. Thus, this attribute in complement of the DYNAMIC and INHERIT directive will permit to generated a more efficient code and will remove some old restrictions. Another variant was also written including the HALO directive. The Halo function or the BHALO data structure was written to return an integer array of the non-local nodes per processor. The use of this Halo will alleviate the storage allocation for non-local data and will reduce the preprocessing overhead generated by the inspector during the time-stepping phase.

Distribute directives used for the stress-strain calculations:

```
SUBROUTINE PCRASH(MAPELM,MAPNOD,BHALO)
INTEGER MAPELM(:),MAPNOD(:)
TYPE(THALO),POINTER,DIMENSION(:) :: BHALO
C
C HPF DISTRIBUTE DIRECTIVES
C
!HPF$ PROCESSORS :: R(NUMBER_OF_PROCESSORS())
C    elements
!HPF$ DYNAMIC, RANGE(BLOCK) :: INDEX
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPELM))   ONTO R  :: INDEX
!HPF$ ALIGN WITH INDEX          :: ROEL,DT2AR
!HPF$ ALIGN (*,I) WITH INDEX(I) :: IY,STRS,FORCE1,FORCE2,FORCE3,FORCE4
C     nodes
!HPF$ DYNAMIC , RANGE(*,BLOCK):: XYZC
!HPF$ DISTRIBUTE(*,GEN_BLOCK(MAPNOD)) ONTO R  :: XYZC

!HPF$ ALIGN (*,I) WITH XYZC(*,I):: XMSI,V,A,XMS,BLAMB,X1,X,VGLOB
C
C HALO DIRECTIVE
C
!HPF$  REDISTRIBUTE(*,BLOCK), HALO(*,BHALO(I)%INDHALO) ONTO R(I)  :: XYZC
```

Distribute directives used for the contact-impact calculations. In the first variant of the contact routines, a block distribution was used for segments and nodes. The Halo function returns the list of non-local slave nodes per processor.

```
C             C O N T A C T    D I S T R I B U T I O N
C     Segments
!HPF$ DISTRIBUTE(BLOCK) ONTO R    :: LSEMO
!HPF$ ALIGN WITH LSEMO            :: NPENP
!HPF$ ALIGN (*,I) WITH LSEMO(I)   :: XYZMIN,XYZMAX
!HPF$ ALIGN (*,*,I) WITH LSEMO(I) :: DIRN,DIST1
C     Slave nodes
!HPF$ DISTRIBUTE(BLOCK) ONTO R    :: NSV
!HPF$ ALIGN WITH NSV              :: NPENO,LSBUCK,IWRK1
!HPF$ ALIGN (*,I) WITH NSV(I)     :: XVAL
C     Halo function
!HALO halo_func(I,NUMBER_OF_PROCESSORS(),NSN) ONTO R(I)  :: NSV
```

3.3.2 Independent loops

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 previous reports.
The first version of HPF+ kernels were completely rewritten to enable the introduction of PUREST procedures provided by the VFC as well as for the computing of internal forces that for the contact-impact routines. This implies to have only local variables in the subroutine. Nevertheless, the PURE attribute is also used to be able to use other HPF compilers.

The REUSE directive recognized as a key factor was used to avoid re-executing the inspector phase that describes the same communication patterns at each time-step loop. Also, the COND clause of the INDEPENDENT directive is used in the contact routines to take advantage of some algorithms features.
The INDEPENDENT directive also contains (in the most general case):
- The On Home clause
- The NEW statement defining the task-private variables: scalars and arrays of various shapes.
- The REDUCTION
- The RESIDENT clause which exploits the updated halo.

The combination of these features with the UPDATE _HALO directive is still in progress because of possible conflicts during the compiler analysis and/or the execution of kernels. Consequently, it is important to have the complete final language specification to know whether some features have potential for changing incorrectly the meaning of the program.

To compute internal forces, the main independent loop over elements could be changed to have an inner loop over sub-block of elements and the outer loop over a list of sorted elements. The aim is to group elements with the same properties and to call the vectorized f77 routines. Until now, restrictions in independent loops prevent us to use such constructs.

The stability of the explicit algorithm dictated by a Courant-Friedrichs-Levy type of criterion, expressed by a maximum time step needs to find the minimum time step over all elements, requiring a global reduction operation. In the previous kernel, additional storage was needed and the min-reduction was performed later thanks to a buffer array. Now, this reduction is also computed in the main independent loop thanks to an improvement of the VFC abilities.

```
!HPF$ UPDATE_HALO :: X,V,VGLOB
!HPF$ INDEPENDENT,ON HOME (IY(:,I)), NEW(…),REUSE,REDUCTION(DT2),RESIDENT
  DO I=1,N4NODE
    IL(1:5) = IY(1:5,I)
    XL(1)   = X(1,IY(2,I))
    XL(2)   = X(2,IY(2,I))
…
    CALL FORCEI(IL,XL,FORCE,DT,…) ! PURE subroutine
…
    FORCEX(1,1,I) = FORCE(1)
    FORCEX(2,1,I) = FORCE(2)
…
    DT2 = MIN(DT2,DT)
  END DO
```

3.3.3 PURE routines

The PUREST directive defined by the HPF+ language specification is used in subroutines called within an independent parallel loop. By example, this subroutine computes internal forces for one element. All distributed variables used by this routine, even though read-only, need to be duplicated into local variables. In the final kernel, this routine was enhanced to mimic the CPU time of elements with different properties which can change the load-balance during computation and thus demonstrate why the GEN_BLOCK directive and the REDISTRIBUTE directive are useful with a more complete benchmark. Besides, the contact kernel was also rewritten to call a PURE routine that computes the new forces to apply at every contact pair.

```
PURE 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)....
```

3.3.4 Gather-Scatter

The scattering 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 one big mapped arrays indexed by the connectivity. For the moment, the array syntax was not used because of compiler restrictions in independent loops.

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

In the case of the contact routines, independent loops are too complicated with actual HPF compilers, because of the highly level of indirection, in particular due to the non-invariant bounds. Consequently, some features were temporary removed and more simpler loops were written. An intermediate kernel was also written to handle only one contact pair per segment and non-invariant bounds were removed.

```
!HPF$ INDEPENDENT, ON HOME(IRECT(:,II)), REDUCTION(A), NEW(...), REUSE
 DO 100 II=1,NRT
C   DO 101 ICONT=1,LSEMO(II)
  DO 101 ICONT=1,MAX36
    IF (ICONT.LE.LSEMO(II) THEN
      XSNX  = X(:,NSV(IMNOD(ICONT,II)))
      RDIST = DIST1(:,II)
      XMASI = XMS(1,IRECT(1:4,II))
      CALL SELF(XNSNX, ,XMASI ,RDIST , ....)
      A(1:3,NSV(IMNOD(ICONT,II))) = A(1:3,NSV(IMNOD(ICONT,II)))+FORCES
      A(1:3,IRECT(1:4,II))        = A(1:3,IRECT(1:4,II)) + FORCEM
    END IF
  ENDDO
 ENDDO
```

An optimized implementation of reduction operations over group of processors is essential for the scalability of the contact routines.

3.3.5 Dummy argument mapping

In the final kernel, interface blocks were generated to help the F90/HPF compiler and to increase the portability across platforms at procedure boundaries. Besides, routines that are described later have been recoded to use Fortran 90 modules to allow for greater portability to other HPF compilers. The use of the INHERIT and/or ALIGN directives have been used in all subroutines which compute the internal forces and the contact algorithm. All routines were rewritten to use the assumed-shape array syntax and reshaping was avoided by clean constructs. Usually, we pass whole arrays every time with always the same distribution. The RANGE attribute is also specified in supplement of the INHERIT directive.

3.3.6 Self-Impact algorithm

A more simplified form was rewritten for the contact search algorithm and the penalty method to be compliant with actual compilers and to focus on the most important loops used in the algorithm in term of the HPF requirements. The sequential source code uses dynamically defined lists and pointers to constructs buckets and lists of contact pairs. Actually, the HPF implementation of these dynamic lists is not possible. Some investigations about different constructs and the use of the HPF intrinsic functions are another possible alternative if they are implemented. Moreover, workarounds have been found to avoid the flaws of the back-end compiler.

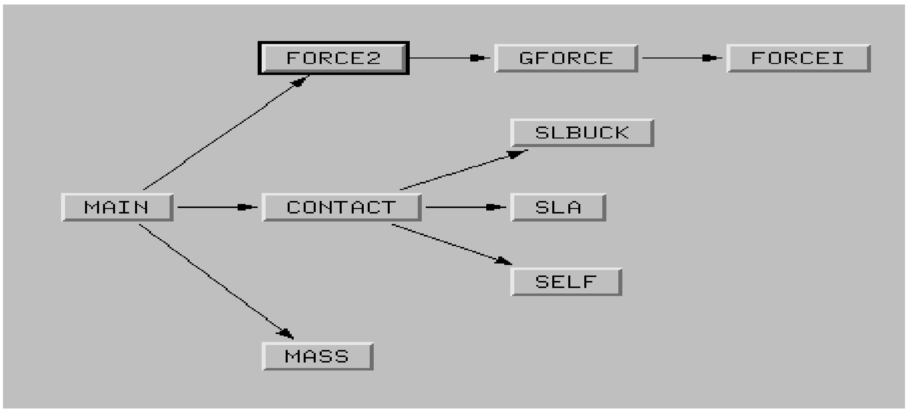

Figure1: Call Graph of the HPF+ Kernels.

3.3.7 Global search

In the initial f77 kernel version, the 3D space occupied by the sliding interface is divided into buckets. So, the goal is to distribute the number of buckets. Optimally, the number of buckets in each coordinate direction is such that the buckets have the shape of a cube. In a second part, a list of nodes is constructed which will be used to determine the contact candidates. All nodes inside this bounding box are found. Nodes connected to the master segment are eliminated. Therefore, remaining candidates (slave nodes, master segment) are the contact pairs.

At the time of writing, The HPF+ implementation of the bucket search was not possible because of complex statements in independent loops and the heavy use of array pointers. A simpler form using a loop on the list of slave nodes was construct. Nevertheless, constraints due to the inspector/executor strategy prevent us to construct the list of slave nodes. By example, the bounds-expressions of loops that are nested within INDEPENDENT loop reference outer loop variables that are distributed arrays. As a consequence, the strategy for collection of references in the inspector fails. In consequently, the bucket sort included in the final benchmark is deactivated. Another strategy with non-mapped arrays to construct this list is envisaged.

```
!HPF$ INDEPENDENT,ON HOME(LSEMO(:)),NEW(NMNOD,…),REDUCTION(NBPTT),REUSE(LREUSE)
DO II=1,NRT
  DO J=1,NSN
    IF ((XVAL(:,J).LT.XYZMAX(:,II)) .AND.
     .(XVAL(:,J).GT.XYZMIN(:,II)) .AND.
     .IRECT(1:4,II).NE.J .AND. DISTV.LT.THK) THEN
      IF (NMNOD.LE.MAX36) THEN
        MNOD = NMNOD+1
        IMNOD(NMNOD,II) = J
      END IF
    END IF
  END DO
  LSEMO(II) = NMNOD
  NBPTT     = NBPTT + NMNOD
END DO
```

Moreover, at the time of writing other routines using INDEPENDENT loops will require some existing VFC compiler constraints to be relaxed to obtain performance, in particular, use of non-invariant bounds for independent loops (i.e. bounds of inner loops dependent on outer loops). The subscript expressions of distributed arrays referencing more than one loop variable, e.g. X(1,A(I,II)) is also required where I and II are loop control variables.

```
!HPF$ INDEPENDENT, ON HOME(DIST1 (:,:,II)),NEW(ICONT),REUSE(LREUSE),RESIDENT
      DO II=1,NRT
        DO ICONT=1,LSEMO(II)
          DIST1(1,ICONT,II)=(X(1,NSV(IMNOD(ICONT,II)))-X(1,IRECT(1,II)))*
     .                      DIRN(1,1,II) + (X(2,NSV(IMNOD(ICONT,II)))- …
          DIST1(2,ICONT,II)=(X(1,NSV(IMNOD(ICONT,II)))-X(1,IRECT(1,II)))*
     .                      DIRN(1,2,II) + (X(2,NSV(IMNOD(ICONT,II)))- …
        END DO
      END DO
```

3.4 MPI benchmarks

The message-passing kernel is based on the second revision of the HPF+ kernel used for the time-measurements on the QSW CS-2 machine. This implementation was written with the MPI library and always uses the host/node paragdim including a more powerful parallel contact algorithm similar to the HPF+ kernels. Nevertheless, the contact routines were derived from the F77 sequential kernel using the most optimized constructs to have the best performance with F77 compilers without to take in account the restrictions imposed by the HPF language. Besides, the parallel routines are based on the bucket sort and it will not be possible to compare the performance with the HPF+ kernels which have to follow the HPF language definition, the compiler restrictions and use the sun F90 compiler. Nevertheless, a slight modified version without the bucket sort could be produced to compare, but we will expect to have very poor scalability on several processors with the current data-set. Another solution is to redefine the contact search to a subset of elements.

Host code:

- Very simple input code (ASCII input files: Geometry, material properties, contact interface)
- It reads the same input data as the others kernels
- Construction of mailing-lists for the exchange of information at processors boundaries
- Distribution of all information to the nodes

Node code:

- Receiving all necessary information from host performed initialization
- Mailing-lists used for the contact routines are locally generated.
- Optimized F77 constructs for the contact routines were written (not HPF compliant)
- Routines using MPI non-blocking calls were written to parallelize the contact algorithm
- One global broadcast to compute the minimal time-step
- Exchange Forces and update velocities of the boundaries nodes for the neighboring processors.

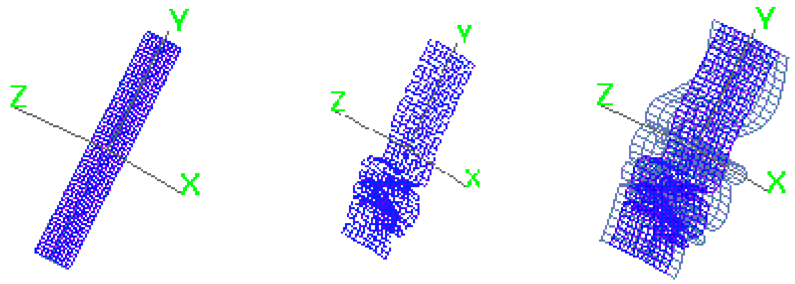

Figure 2: Box data-set used

3.5 Conclusion

In this section, the parallelization strategy was described for generating the PAM-CRASH kernels of the HPF+ project. Benchmark kernels of increasing degree of complexity, containing programming constructs that constitute essential HPF language requirements, were developed using the new HPF directives. Moreover, the contact-impact algorithm introduced previously was modified to stick to the HPF language and compiler constraints. Nevertheless, the writing of this benchmark is still in progress because of the need to combine more judicious the HPF construct to take in account the implementation of the redistribution libraries and thus to control the dynamic behavior of the contact.

4. NEC

4.1 Introduction

This report provides an update to the previous deliverables describing the generic crash kernels used for the evaluation of HPF+ language developments. The generic crash kernels produced by NEC/CCRLE supplement the full PAM-CRASH benchmark kernels from ESI - the language requirements match those of the full kernels, but the data structures and some algorithmic features have been greatly simplified. Their purpose has been to facilitate an accelerated understanding of the basic problems to be encountered in the full kernels from ESI.
Deliverable D1.1a gave an overview of the three levels of kernels to be used for the language development and evaluation and the “Kernel sets” (K1, K2, CK1) reflecting these levels: element-node interactions for 4-node shell elements; explicit time-marching built around this shell-kernel; the additional inclusion of contact-impact routines.
Within the evaluation report of deliverable D1.2b, modifications leading to several HPF variants of the K2-kernels were also described. We recall here that the hierarchy of computational tasks indicated is indeed reflected in an encapsulation within the kernels: the K2-kernels include a time-marching loop and an additional subroutine around the central computation routine of the K1 kernels; the CK1 kernels extend the K2-kernels with contact-impact routines. For the final evaluation of the HPF+ language extensions within the project, only the K2- and CK1-kernel sets will be employed.

In the following sections, details will be given of an additional HPF+ kernel to the K2-set, together with the newly constructed MPI and HPF (both HPF1.1 & HPF+) contact-impact kernels in the CK1-set. The basic algorithm employed within the CK1set was described in deliverable D1.1a, but will be repeated here for clarity. Although the basic algorithms from the K2-set will not be repeated, the reader is referred to deliverable D1.1a, the HPF+ implementation of the core computational task will be repeated to enable a comparison of the two approaches for user control of, or provision of information relating to, communication schedules.

4.2 K2 Kernel set: Time-marching with Generic Shell Elements

In the following we will discuss only an addition to the K2 kernel set which makes use of the newly developed HALO facility. As a reminder of the previously developed approach exploiting the schedule reuse facility (REUSE) we first repeat the computational core of the KFORCE routine in Figure 1.

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

      real, dimension(:,:), intent(in) :: x;      real, dimension(:,:), intent(inout) :: f
      integer, dimension(:,:), intent(in) :: ix
      integer, intent(in) :: numnp,numel2
      real, dimension(6,4,numel2) :: aforce
      real, dimension(3,4) :: xn;      real, dimension(6,4) :: force
      logical lreuse;      common /bkreuse/lreuse

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

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

!HPF$ INDEPENDENT, NEW(xn,force), ON HOME (ix(1,iel)), REUSE(lreuse)
      do iel = 1, numel2
       do i=1,3
         xn(i,1) = x(i,ix(1,iel));      xn(i,2) = x(i,ix(2,iel))
         xn(i,3) = x(i,ix(3,iel));      xn(i,4) = x(i,ix(4,iel))
       end do
       call mforce(force,xn)
```

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

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

  return;  end
```

Figure 1: Subroutine KFORCE in the K2HPF+_a Kernel (Implementation for VFC release 0.92)

We recall that the major feature exploited in the K2HPF+_a kernel was the schedule REUSE facility (combined with the ON HOME and REDUCTION options and PUREST subroutine). This facility indicates to the compiler that the communication schedule that ***it*** produces is invariant - thus the overhead for the communication schedule may be amortised over many calls to the subroutine.
An alternative strategy - and one which whose behaviour was investigated via a form of hand-coded approach with the Adaptor compiler as reported on in deliverable D1.2b - is for the user to take control of the communication for non-local data by defining and specifying the time of update of generalised halos: indices of arrays which will be accessed but which will have been assigned to other processes. The addition to the K2-kernel set, the kernel K2HPF+_b, incorporates such a mechanism for the gather operation required for the co-ordinates, stored in the nodal variable X, within the first INDEPENDENT loop of Figure 1 and for the force array, f, over which the sum-scatter reduction operation is performed in the second loop. The modifications in comparison with the K2 kernel occur in both the main calling program and the KFORCE subroutine as illustrated in Figures 2 and 3. In order to allow the halo function to be evaluated just once, the X & F arrays have been re-defined as DYNAMIC arrays whose re-distribution in specified once the halo function can be calculated. Given that the X array's halo is updated before the call to the KFORCE routine, the first of the two INDEPENDENT loops can exploit the RESIDENT clause and should not involve any communication. However, at the time of writing, some open questions, which will be addressed during modifications to be performed during the final evaluation phase of the project, remain:
Is the REUSE clause for the first INDEPENDENT loop actually necessary ?
Given that a halo region has been specified for F, could the REDUCTION loop be replaced by an INDEPENDENT loop with RESIDENT followed by an UPDATE of its halo region ?
Should the original distributions within the KFORCE routine be replaced by a combination of INHERIT and RANGE(*,BLOCK) statement to guarantee (independently of compiler implementations) that the HALO is visible ?

In Figures 4&5, we illustrate the mechanism used for the construction of the HALO function. It should be noted that the basis for the "process boundary" calculations is the definition of the distribution of arrays as given by the HPF standard. This would lead to erroneous execution should particular compiler implementations not adhere to the standard (a feature not usually of importance to the programmer). An alternative would be to use the enquiry function HPF_SUBGRID_INFO defined within the approved extensions of HPF-2.
Following the addition of the K2HPF+_b kernel, the final K2-kernel set becomes:

Name	Implementation for:	Comments
K2SEQ	F77	Provides the basis of elapsed time comparisons
K2F90	F90	Allows the evaluation of HPF overhead w.r.t. Fortran 90
K2HPF_a	HPF 1.1	Implementations tuned to particular compilers
K2HPF+_a	HPF+	Employs schedule reuse
K2HPF+_b	HPF+	Employs HALO facilities
K2MPI_a	MPI	Basis for parallel scaling & absolute performance evaluation - employs F77

It is to be noted that there has been a concentration on features addressing communication and overhead minimisation for the irregular computational access patterns, rather than on irregular distributions, since these are viewed as posing the greatest challenge for efficient compilation. This is also (and indeed to a greater extent) true for the contact-impact kernels.

```
program K2HPF

                         Lines omitted

   TYPE Boundary
       INTEGER lb, ub
   END TYPE Boundary
   TYPE (Boundary), allocatable :: Bound_x(:)
   TYPE (Boundary), allocatable :: Bound_ix(:)


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

    nprocs = NUMBER_OF_PROCESSORS()
   Allocate (Bound_x(nprocs))
   Allocate (Bound_ix(nprocs))
   call CalcBoundary(ubound(x,dim=2), Bound_x(:), nprocs)
   call CalcBoundary(ubound(ix,dim=2), Bound_ix(:), nprocs)

                         Lines omitted


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

!HPF$ REDISTRIBUTE (*,BLOCK), &
!HPF$ HALO(*, MAP(ix, Bound_x, Bound_ix, I)) ON PROCS (I) :: f, x

                         Lines omitted

   do 100 it = 1, ittot

     f = 0.0

!HPF$ UPDATE_HALO :: x

     call kforce(f,x,ix,Bound_x(:),Bound_ix(:),numnp,numel2)

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

100   continue
```

Figure 2: K2HPF+_b - Modified main program using the run-time (but invariant) defined halo function MAP for the arrays f & x together with the UPDATE_HALO for x

```
subroutine kforce(f,x,ix,bound_x,bound_ix,numnp,numel2)

                                Lines omitted

!HPF$ INDEPENDENT, NEW(xn,force), ON HOME (ix(1,iel)), &
!HPF$              REUSE(lreuse), RESIDENT
   do iel = 1, numel2
    do i=1,3
     xn(i,1) = x(i,ix(1,iel)); Lines omitted end do
    call mforce(force,xn)
    do i=1,6
     aforce(i,1,iel) = force(i,1); Lines omitted end do
   end do
```

Figure 3: K2HPF+_b - Use of RESIDENT within KFORCE, exploiting the updated halo.

```
SUBROUTINE CalcBoundary (axis_length, bound, np)

   TYPE Boundary;      INTEGER lb, ub;    END TYPE Boundary
   TYPE (Boundary), INTENT(OUT) :: bound(:)
   INTEGER, INTENT(IN) :: axis_length, np;    INTEGER :: block, remainder, ub, lb

   block = (axis_length+nproc-1)/nproc;    ub =  axis_length;    lb =  1
   do I=1, np
     bound(I)%lb = -1    ;         bound(I)%ub = -1
     if (I > 1) then
         bound(I)%lb = (I-1) * block + lb
    else
         bound(I)%lb = lb;      end if
    if (I * block <= ub) then
         bound(I)%ub = n * block
    else
         remainder = ub - n * block
         if (remainder > 0) then;     bound(I)%ub = remainder;        end if
    end if
   end do

   END SUBROUTINE CalcBoundary
```

Figure 4: K2HPF+_b - construction of the process bounds for use by the HALO function

```
FUNCTION MAP (ix, bound_x, bound_ix, n)

    TYPE Boundary;      INTEGER lb, ub;      END TYPE Boundary
    TYPE (Boundary) :: bound_x(:), bound_ix(:)
    INTEGER, POINTER :: MAP(:)
    INTEGER :: ix(:,:), n, start, stop, remainder, ub, lb, I, K
    INTEGER, ALLOCATABLE :: TMP_MAP(:)

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

       start = bound_ix(n)%lb;   stop  = bound_ix(n)%ub
       lb_x  = bound_x(n)%lb;    ub_x  = bound_x(n)%ub

       ALLOCATE (TMP_MAP(4*(stop-start+1)))

       K = 1
       DO I = start, stop
         DO J=1,4
           IF (ix(J,I) < lb_x .OR. ix(J,I) > ub_x) THEN
             TMP_MAP(K) = ix(J,I);          k = k+1
           END IF
         END DO
       END DO

       ALLOCATE (MAP(K-1))
       MAP(1:K-1) = TMP_MAP(1:K-1)

       RETURN
   END FUNCTION MAP
```

Figure 5: K2HPF+_b - construction of the HALO function

4.3 CK1 Kernel set: Segment-based Contact-impact

In order to introduce the final generic contact-impact kernel - and its simplifications in comparison with actual contact algorithms found in the PAM-CRASH code - we repeat for clarity the description of the design presented in the deliverable D1.1a. In contrast to the stress-strain calculations (represented by the K1- & K2-kernel sets), the contact-impact algorithms used within the PAM-CRASH code have, in terms of data access, a pseudo-global nature. These contact algorithms serve to detect and correct penetration of structural components. This is achieved by first performing a proximity and penetration search, followed by a penetration correction procedure. Within the code, the latter is a penalty method, whereby contact forces are introduced at the impacting node and at the nodal points of the impacted segment.

Computational performance within PAM-CRASH is maintained by having optimised search routines employing sorting procedures. An implementation (or practical usage) issue which affects parallelisation is that the contact calculations are performed only within user-defined (and not necessarily distinct) areas, referred to as *slide-lines*. Both these aspects will be neglected in the CK1 kernels: no sorting of nodes will be performed; only one slide-line will be considered (which may, nevertheless, be defined to be a sub-set of the model). These simplifications should assist the compiler analysis by removing *additional* levels of indirection and dynamically defined lists. One consequence of the lack of sorting is that the contact kernels become computational expensive (performing $O(N^2)$ operations in the search). This means that careful attention will be paid to the measurements employed within the evaluation process. We will see that a two-level approach will be taken - the focus for evaluation will be on the most often used level within which searching is limited to dynamically created lists which should not exhibit such disproportionate computational load. That is, the evaluation will consider how well the HPF+ language can deal with the dynamically changing computational phase wherein efficient communication handling will be important.

Before continuing with the description of the algorithms and kernel implementations, the question of whether or not the calculations performed are representative of actual crash simulations will be addressed. It should be recalled that the actual computations within the generic kernels are *models* of the computational tasks and are in no way physically relevant. Since the contact-impact phase relies on geometrical searching, we need to be careful that the performance behaviour is representative. Figure 6 illustrates the deformed box mesh (as used in the performance evaluation) after 30 iterations. Although clearly non-physical, a locality of deformation and thus contact penetration is exhibited which does in some way represent the locality of contact detection in the early stages of an actual simulation. Continued iterations have been seen to be much less representative (with enormous overlapping of a small edge of the model with the remaining domain). Thus, the evaluation with the kernel will be (for that model size) restricted to 30 iteration runs.

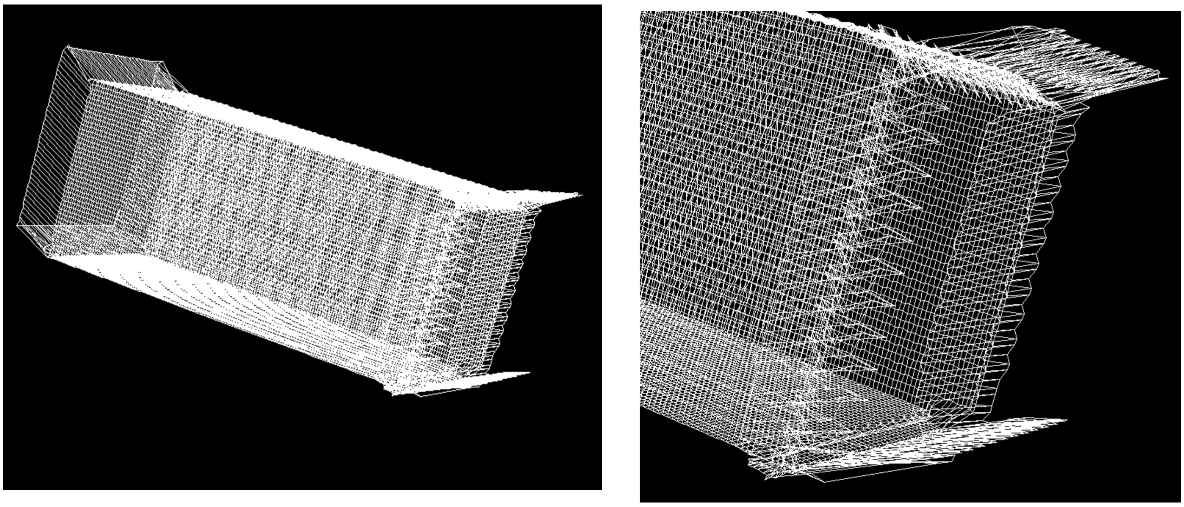

Figure 6: The (non-physical) deformation produced by the CK1-kernels - 30 iterations

The CK1 contact-impact kernels are based on the use of two-levels of fixed-size bounding-boxes around each segment. The larger box (which we will refer to as the “search cube”) is used to define a set of (non-connected) nodes that are checked for penetration into a much smaller box (the “contact cube”) around the segment. This is illustrated in Figure 7.

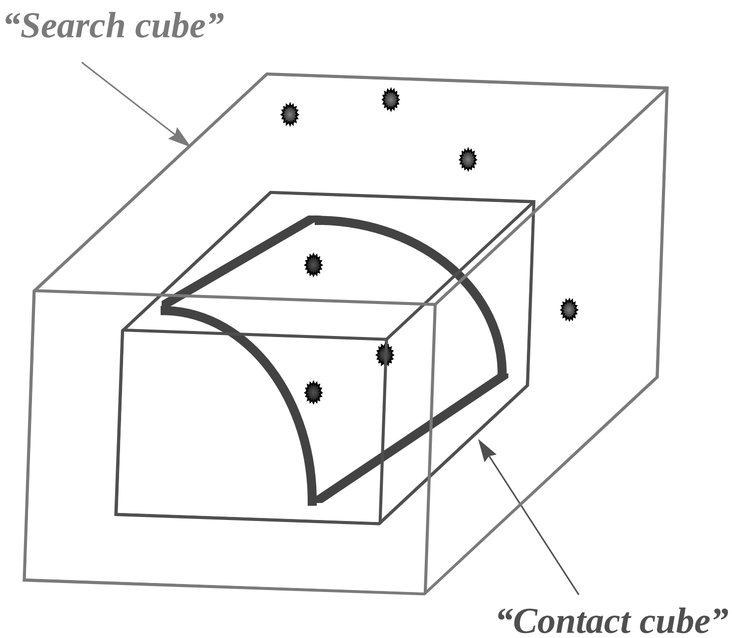

Figure 7: Two-level bounding-boxes for segments

The search box is used to minimise the cost of the search in that it is defined only at fixed intervals within the time-marching loop - this will also reduce the necessary communication since only searches to determine the nodes in each segments search box are global. The sequential F77 kernel, CK1SEQ, employs dynamically defined lists and (F77) pointers to perform the two levels of search. This is illustrated in Figure 8 by a code fragment from CK2SEQ modelling the search and penetration correction (in the routine CORRECT) within the contact cube. The list ***schn*** holds the nodes within the search cube, which will be constant for a number of iterations and will have been produced (together with its F77 pointer ***pnt***) by a similar search pattern within the larger bounding box.

The computational features of consequence for the parallelisation are:
The contact forces calculated in the routine CORRECT (a simplified version of the PAM-CRASH routine SELF6, from Version 12.0 MPP) are added to the existing forces at the segment nodal points and located node. Thus, the communication schedule required for this loop is fixed as long as the search cube lists are not recalculated; this is controlled in the kernel by a logical variable ***RECALC*** whose value is determined by the iteration number.
The communication schedule for the search cube nodal list calculation will be static. However, the corresponding gather operation corresponds to (possibly, dependent on slide line definition - but **actually** in the model used within the evaluation) all-to-all global communication of the co-ordinates.

```
Subroutine contk1(......)

do iseg = 1, ncs

    do in = 1,4;    nseg(in) = csegs(in,iseg)
        do ic = 1,3;        xseg(ic,in) = x(ic,nseg(in));       end do
    end do

    numnod = pnt(iseg+1) - pnt(iseg)
    if (numnod.eq.0) goto 200

    do ic = 1,3;    cmin(ic) = min(xseg(ic,1),xseg(ic,2),xseg(ic,3),xseg(ic,4)) - thkcont
                    cmax(ic) = max(xseg(ic,1),xseg(ic,2), xseg(ic,3),xseg(ic,4)) + thkcont; end do

    do ip = pnt(iseg),pnt(iseg+1)-1
        nn = schn(ip)
        do ic=1,3; xn(ic) = x(ic,nn);       end do

        if (        (cmin(1).le.xn(1).and.cmax(1).ge.xn(1)) .and.
                    (cmin(2).le.xn(2).and.cmax(2).ge.xn(2)) .and.
                    (cmin(3).le.xn(3).and.cmax(3).ge.xn(3)) ) then

          call CORRECT(xn,xseg,fc,thkcont)
          do ic = 1,3;      f(ic,nn) = f(ic,nn) + fc(ic,5);       end do
          do ic = 1,3;      do in = 1,4;      f(ic,nseg(in)) = f(ic,nseg(in)) + fc(ic,in);     end do; end do

        end if
    end do
200   continue

end do
```

Figure 8: CK1SEQ kernel - contact box search & correction

Before proceeding to discuss the HPF implementation possibilities in more detail, in addition to an overview of communication constructs employed within two variants of hand-coded MPI implementations, we will conclude this section with an overview of the final CK1-kernel set:

Name	Implementation for:	Comments
CK1SEQ	F77	Basis for performance comparisons
CK1HPF_a	HPF 1.1	Implementations tuned to particular compilers
CK1HPF+_a	HPF+	Employs schedule reuse
CK1HPF+_b	HPF+	Employs HALO facilities
CK1MPI_a	MPI	Two variants with differing ordering of contact
CK1MPI_b	MPI	calculations & communications. - both in F77

4.4 HPF Parallelisation

The initial HPF parallelisation described in D1.1a employed an extremely large mask array and the use of F90 WHILE constructs within INDEPENDENT loops. This approach has now been discarded since: (a) the memory requirements were far too high; (b) the combination of INDEPENDENT-WHILE was unlikely to be feasible in any compiler within the lifetime of the project. The new CK1HPF and CK1HPF+ kernels have been re-cast to resemble to a large extent the sequential code. The only functional, but minor, modification has been to use a static upper bound on the possible contact nodes per segment to facilitate a simplified array distribution (dependent on the availability of the F90-intrinsic *maxval*).

For the CK1HPF_a kernel based on HPF-1 (with possible inclusion of some HPF-2 extensions) the major features of the parallelisation were:

Replicate the X array *within* the subroutine CONTK1 - this forces an all-to-all communication at the subroutine boundary, but allows the INDEPENDENT loops performing gather operations on X to be executed with high efficiency.

Replicate the list of contact nodes since it is used across all processes for the search cube phase. The list of contact segments is distributed (in the form (*,BLOCK)).

Split the contact cube search and correction loop within subroutine CONTK1 (cf. Figure 8) in a manner similar to that employed for the KFORCE routine (cf. Figure 1). Contact forces are stored in temporary arrays and the summation of contact forces is performed in a separate loop (with ON HOME and REDUCTION) or via the use of a SUM_SCATTER library routine.

Two versions of the kernel have been produced to exploit the features offered by HPF+. Both kernels have the same computational features and loops as the CK1HPF_a kernel.

The first, CK1HPF+_a, makes use of schedule reuse for the search cube phase (which will nevertheless involve all-to-all communication - so that a subroutine boundary redistribution to a replicated x array may be considered as a replacement at a later stage) and conditional schedule reuse for the penetration detection within the contact cube phase. As described above for the HPF-1 implementation, a splitting of the contact cube search loop is carried out: (a) penetration detection and calculation of corrective forces, (b) corrective contact force summation. It should be noted that the communication schedule for (b) is not possible. These features of the CK1HPF+_a kernel are illustrated in Figures 9 & 10.

The second HPF+ kernel, CK1HPF+_b, exploits the HALO facility and builds on the K2HPF+_b kernel. Thus, there are DYNAMIC distributions for the x & f arrays. The basic form of the CONTK1 routine will be as in Figures 9 & 10, and as in that kernel there will be the option to replace the search cube phase with a routine using a replicated x array. Analogous to the treatment in the KFORCE routine (c.f. Figure 3), is the introduction of a RESIDENT statement for the INDEPENDENT loop in the contact cube phase and there is also the open question regarding the replacement of INDEPENDENT-REDUCTION loops with INDEPENDENT-RESIDENT loops followed by UPDATE_HALO directives. The major difference to the K2HPF+_b kernel is the requirement for a new halo in the contact cube phase and the suppression of halo usage for the search cube phase - implying that frequent REDISTRIBUTION statements will be required. Rather than introduce separate halos for the KFORCE & CONTK1 routines - implying the need for REDISTRIBUTION in/before each call of KFORCE - the policy has been to use a union of halo regions, with the contact routine adding to that required for the “partition boundaries” in the shell calculations. Note that this means that all halo region updates will involve more communication than is absolutely necessary for the particular computational task to follow. The resulting REDISTRIBUTION and HALO_UPDATE sequence within subroutine CONTK1 is illustrated in Figure 11.

```
subroutine contk1 (f,x,cnodes,csegs,pnt,schn,numnp,ncn,ncs,schdim,..., RECALC,..)

      real, dimension (:,:), intent(in) :: x                      ! (3,numnp)
      real, dimension (:,:), intent(inout) :: f                   ! (6,numnp)
      integer, dimension (:,:), intent(in) :: csegs               ! (4,ncs)
      integer, dimension (:,:), intent(inout) :: schn             ! (schdim)
      integer, dimension (:), intent(inout) :: pnt                ! (schdim)
      integer, dimension (:), intent(in)    :: cnodes             ! (ncn)

            Lines Omitted

! Additional Work arrays for parallelisation:
      real, allocatable, dimension (:,:,:) :: loc_f
      real, allocatable, dimension (:,:,:,:) :: loc_f2
      integer, allocatable, dimension (:,:) :: active_index

!HPF$ processors procs(number_of_processors())

!HPF$ ALIGN (*,I) WITH csegs(*,I) :: active_index,schn
!HPF$ ALIGN (*,*,I) WITH csegs(*,I) :: loc_f
!HPF$ ALIGN (*,*,*,I) WITH csegs(*,I) :: loc_f2
!HPF$ ALIGN (I) WITH csegs(*,I)  :: pnt
!HPF$ DISTRIBUTE (*,BLOCK) :: csegs, x, f

            Lines Omitted

c                       Re-calculate list of nodes in segment search cube
      if (RECALC) then

!HPF$ INDEPENDENT ,NEW (......),ON HOME(csegs(1,iseg)), REUSE
        do iseg = 1, ncs

                        Lines Omitted
          (search cube calculation to produce schn(.., iseg) & pnt(iseg) entries)

        end do
      end if

!     c_points = maxval(pnt)
      c_points = 20
      allocate (loc_f(3,c_points,ncs));      allocate (loc_f2(3,4,c_points,ncs))
      allocate (active_index(c_points,ncs))
```

Figure 9: CK1HPF+_a - Data distributions and Search Cube phase

```
                      Lines Omitted

!HPF$ INDEPENDENT ,NEW (l_nn,.....),ON HOME(csegs(1,iseg)), REUSE(COND=.not. RECALC)
    do iseg = 1, ncs
                      Lines Omitted
      numnod = pnt(iseg)
      if (numnod <> 0) then
                      Lines Omitted
        l_nn = 0
        do ip = 1,numnod
          nn = schn(ip,iseg)
           x1 = x(1,nn);.....
          l_nn = l_nn + 1
          if ( (cmin(1).le.x1.and.cmax(1).ge.x1) .and. .....)     then
            call correct(xn,xseg,fc,loc_thkcont)
            active_index(l_nn,iseg) = nn
            do ic = 1,3;     loc_f(ic,l_nn,iseg) = loc_f(ic,l_nn,iseg) + fc(ic,5);     end do
            do ic = 1,3;  do in = 1,4
                   loc_f2(ic,in,l_nn,iseg) = loc_f2(ic,in,l_nn,iseg) +  fc(ic,in)
            end do; end do
          end if
        end do
      end if
   end do


!HPF$ INDEPENDENT, NEW(l_nn,ip,nn,ic), ON HOME (loc_f(1,1,iseg)), REDUCTION(f)
    do iseg = 1,ncs
       l_nn = 0
       do ip = 1, pnt(iseg)
         nn = schn(ip,iseg)
         l_nn = l_nn + 1
         if (active_index(l_nn,iseg) == nn) then
           do ic = 1,3;    f(ic,nn) = f(ic,nn) + loc_f(ic,l_nn,iseg);  end do
         endif
      end do
    end do

!HPF$ INDEPENDENT, NEW(l_nn,t_nn,...), ON HOME (loc_f(1,1,iseg)), REDUCTION(f)
    do iseg = 1,ncs
       l_nn = 0
      do ip = 1,pnt(iseg)
         nn = schn(ip,iseg)
         l_nn = l_nn + 1
         if (active_index(l_nn,iseg) == nn) then
           do ic = 1,3;  do in = 1,4
              t_nn = csegs(in,iseg);    f(ic,t_nn) = f(ic,t_nn) + loc_f2(ic,in,l_nn,iseg)
           end do;  end do
         endif
      end do
    end do
                      Lines Omitted
```

Figure 10: CK1HPF+_a - Contact Cube Phase

```
Subroutine CONTK1(....

!HPF$ processors procs(number_of_processors())
!HPF$ DYNAMIC, RANGE (*,BLOCK), DISTRIBUTE (*,BLOCK) :: f, x
! Alternatively:  !HPF$ INHERIT, RANGE (*,BLOCK) :: f,x


              Lines Omitted

c             Re-calculate list of nodes in segment search cube

      if (RECALC) then

!HPF$ REDISTRIBUTE (*,BLOCK) ON PROCS (I) :: f, x

!HPF$ INDEPENDENT ,NEW (......),ON HOME(csegs(1,iseg)), REUSE
        do iseg = 1, ncs

              Lines Omitted
           (search cube calculation to produce schn(.., iseg) & pnt(iseg) entries)

        end do

!HPF$ REDISTRIBUTE (*,BLOCK),      &
!HPF$ HALO(*,CMAP(schn, csegs, Bound_cnodes, Bound_csegs, ix, Bound_x, Bound_ix, I)) &
!HPF$   ON PROCS (I) :: f, x


     end if

                                   Lines Omitted

!HPF$ UPDATE_HALO:: x
!HPF$ INDEPENDENT ,NEW (l_nn,.....),ON HOME(csegs(1,iseg)), REUSE(COND=.not. RECALC)
      do iseg = 1, ncs
                                          Lines Omitted: Penetration detection & Contact force calculation
      end do


!HPF$ INDEPENDENT, NEW(l_nn,ip,nn,ic), ON HOME (loc_f(1,1,iseg)), REDUCTION(f)
      do iseg = 1, ncs
                                          Lines Omitted
      end do

!HPF$ INDEPENDENT, NEW(l_nn,t_nn,...), ON HOME (loc_f(1,1,iseg)), REDUCTION(f)
      do iseg = 1,ncs
                                          Lines Omitted
      end do
```

Figure 11: CK1HPF+_b: redistributions and halo updates within the CONTK1 routine.

4.5 MPI Parallelisation

A hand-code parallelisation of the CK1 kernel - based on the sequential contact-impact routines of CK1SEQ and building on the K2MPI kernel - has been carried out, including two minor variants, included in kernels CK1MPI_a & CK1MPI_b, of the parallel version of the CONTK1 routine. Since MPI parallelisation is not the topic of the project developments - the MPI kernels are for evaluation *comparisons* - only a sketch of the features within the MPI kernels will be given. It should be noted, however, that the development times for these kernels were significant; a feature to be taken into account during the evaluation. One major source of difficulty lies in the handling of local and global numbering systems employed within the SPMD code. The construction of dynamic mailing lists involves a large amount of such local-to-global numbering bookkeeping calculations and leads to very complex coding.

The search cube phase is implemented by performing a series of broadcasts from each process to all others, whereby each process broadcasts its local set of contact nodes. On receipt, nodes received are compared with local segments in order to construct the contributions to the local versions of the *schn* array, in addition to lists of nodes to be received. The dynamic mailing lists (to be used for the provision of data for subsequent contact cube phase calculations) are completed using an MPI_ALLTOALLV call.

The two MPI implementations differ only in their treatment of the contact cube phase.

CK1MPI_a uses the dynamic mailing list to define a **set** of non-blocking communications to be performed in advance of the contact cube phase for exchange of contact nodes and subsequent to the computation for the communication and correction of remote contact forces. This communication is similar to that exhibited in the partition boundary update schemes given in D1.1a (routine xa2b, for example). The contact cube computation itself is as illustrated in Figure 8 – it is modified only by an outer loop over communicating processes (as determined in the search cube phase) and the extraction of the node co-ordinates from the receive buffer. In the current implementation, the processes that have to exchange contact nodes for the contact cube phase also return contact forces for **every** node in their mailing list, the contact forces having been initialised to zero. This unnecessary communication of zero force could be reduced in a later optimisation.

CK1MPI_b uses a procedure within which *pairs of processes:* exchange contact nodes; check penetration of local segments by contact nodes received; exchange contact forces; correct forces at local nodes. All exchange communications are implemented using non-blocking point-to-point functions. The selection of process pairs is currently implemented by running over the diagonal and all super-diagonals of a process-process adjacency matrix and checking for inclusion in the process list created in the search cube phase, that is:

```
do k=0,nprocs-1; do jp=k, nprocs-1; ip = jp-k
        process pair is (ip,jp) – check if process belongs to pair
                                   & if exchange necessary.
        data exchange – computation – data exchange
end do; end do
```

Although introducing a level of sequentialisation (which can be extreme in cases where the contact communication patterns are complete), there are some advantages to the approach: (a) reduced network contention; (b) reduced memory requirements for message buffering, in comparison with CK1MPI_a. Exchange of full lists of contact forces (including the unaltered zero forces) is as in CK1MPI_a.

4.6 Concluding Remarks

The most important features of the HPF+ language extensions for the irregular computations occurring in codes such as PAM-CRASH are likely to prove to be the communication schedule re-use directives (REUSE and its conditional form) and the general halo facility (HALO directive and UPDATE_HALO). The HPF+ usage within the current generic crash kernels reflects this. The choice of evaluation model (the very regular box structure with a simplified correspondence between element and node orderings) enables a focus on these features while using standard distributions. For the general case, with element and node orderings dependent only on the sequence of operations carried out when generating the complex mesh, the employment of run-time defined GEN_BLOCK distributions will be essential – a full INDIRECT distribution is deemed to be unnecessary.

Given the time constraints of the final evaluation phase, an investigation of the effect of load-imbalance within the contact-impact phase is, though foreseen within the construction of all the CK1-set kernels, unlikely to be feasible. This will be addressed at a later date. Looking beyond such features to future requirements for an extension to the current HPF+ language definition, one would address the problems raised by adaptive meshing features.