Report on Project Benchmarks Rel.1

Deliverable:	D1.1a
Date:	March 1997
Status:	draft
Confidentiality:	public
Responsible:	ESI
Revised:	April/11/1997

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 corresponds to the deliverable D1.1a of the HPF+ project, describing the intermediate status of the project benchmark kernels using the HPF language and its extensions as proposed by the consortium. The report is part of task 1.1 of the project which focuses on the practical usage of HPF. Its contents reflect the need for important improvements of the language definition if it is to be used efficiently in advanced industrial parallel applications. The applications 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 this perspective and cover a very wide and representative range of algorithmic details. They are being analyzed for conversion to the HPF paradigm with maintainability and efficiency 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.

Contents

1 AVL

1.1 Introduction

The new version of the AVL kernel contains a standalone implementation of the iterative conjugate gradient equation solver as it is used inside the FIRE program to solve sparse linear equation systems of the form $A\Phi = S_\Phi$.

The solver extracted for the kernel is a Krylov subspace method which consists of a modified preconditioning and orthomin convergence acceleration. It can be applied to structured and to unstructured meshes. The method is designed for iterative sparse matrix computations whereby nonzero matrix elements occur only in the six side bands and in the main diagonal. The position of the elements is obtained from the indirect addressing array LCC(1:6,0:NNINTC), which is part of the input dataset. The LCC array reflects the geometrical connections of the computational cells in the transformed space. The first index contains direction information according to the compass notation, e.g. LCC(1,NC) contains the address of the neighbour cell in the direction south when looking from the pole cell NC. LCC(2,NC) refers to the neighbouring cell in east direction, LCC(3,NC) to north direction and so on. The logical directions corresponds to the general non-orthogonal coordinate direction in physical space.

Two different versions of the equation solver have been implemented: one under HPF+ and a message passing version under MPI.

1.2 Data Distribution

All arrays that contain cell center information are defined in a common block as it is implemented in the FIRE program and in the first version of the kernel. There are two different kind of arrays: arrays that are dimensioned over the number of internal cells (NNINTC) and arrays that are dimensioned over the number of internal and external cells (NNCELL).

Cell center information is stored in a specific order on the arrays: first the internal cells are stored on the array followed by all external (boundary) cells. The reason for this structure arises from the fact that most loops in the FIRE program are running over the internal cells and only the loops where also boundary information is required have loop indices from one to the total number of cells.

The first version of the benchmark has contained only one array that was dimensioned over the total number of cells (DIREC1) while all other arrays are dimensioned over the number of internal cells. For both types of arrays a block distribution has been used. Because of this different array distribution the amount of communication inside a loop that contains both array types can become very high.

In the second version of the kernel the arrays that contain only internal cell data are distributed in blocks, while the arrays over all cells are distributed indirectly by a map array that is read in at runtime. The map array can be computed for each dataset by the program genmap.f previous to the calculation. In order to minimise the communication the map array contains a block distribution for all internal cells, while the external cells a 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.

A one dimensional processor array is used for the data distribution (NRNODE is defined by the number of processors):

```
!HPF$ PROCESSORS :: PROC(NRNODE)

!HPF$ DISTRIBUTE(INDIRECT(MAP)) ONTO PROC :: DIREC1
!HPF$ DISTRIBUTE(BLOCK) ONTO PROC :: DIREC2
```

For the message passing version of the kernel the data distribution is performed during runtime. 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 for the HPF+ version.

This data distribution does not take any geometrical information of the mesh into account. For complex, unstructured geometries such a distribution will lead to disconnected subdomains and to large communication areas. A consequence of disconnected subdomains is that the number of processors that have to perform the local data exchange is becoming large. The implementation of better data distribution algorithms (like for example coordinate bisections or spectral bisection) could reduce the communication effort during the computation significantly, but this will require for the HPF+ version, that also the arrays that contain internal cell information are distributed indirectly by the use of a map array.

1.3 Communication

When performing matrix-vector products the matrix elements are already stored as functions of the pole index but the vector components are indirectly accessed via the LCC array. If one vector component does not belong to the processor which contains the pole cell, a data exchange must take place. In case of the HPF+ kernel this is automatically done by the compiler.

For the MPI version the data exchange is done by a separate routine, that has to be called before the elements are accessed inside the following loop in order to ensure no blockage of the computation. The addresses of the communicated cells are stored separately in form of send and receive lists, that are setup only once, previous to the calculation loops. The communication is performed by using an "all send - all receive" strategy, and is generally non-symmetric with respect to the amount of data exchanged between the processors. Artificial boundary layers, the interprocessor boundaries (IPB), have been introduced between any two neighbouring processors to avoid permanent data exchange of independent variables. Every cell accessed across the IPB is locally an external cell and acts as a receive cell for the dependent variables. The amount of data to be communicated strongly depends on the quality of the data distribution and on the number of compute nodes involved in the solution process. For the benchmark a data distribution has been implemented, which is equivalent to a HPF+ block distribution.

For the message passing version the send and receive lists are computed once at the beginning of the calculation and they are reused inside the iteration loop many times since the communication pattern remains invariant. For the HPF+ version the reuse of a computed schedule will have the same effect and will reduce the communication time significantly.

1.4 Global Sum

Apart from the ordinary data exchange sum reduction operations can become a bottleneck because of the small amount of data (just one number) has to be communicated between all processors frequently. In the conjugate gradient equation solver up to five global sums are performed during one iteration cycle. Four of them are necessary for the orthogonalisation procedure and one is necessary for the convergence check. During the calculation of one time step several hundred of iterations have to by performed which results in the computation of several thousand global sums. A fast and efficient implementation for global sums is therefore an important requirement for the kernel and for the FIRE program.

For the HPF+ version the reduction clause was used to compute the sum:

```
!HPF$ INDEPENDENT, REDUCTION(RESREF)
      DO 10 NC=NINTCI,NINTCF
        RESREF=RESREF+RESVEC(NC)*RESVEC(NC)
      10 CONTINUE
```

For the message passing version the global sum is computed by an MPI routine immediately after the loop:

```
      DO 10 NC=NINTCI,NINTCF
        RESREF=RESREF+RESVEC(NC)*RESVEC(NC)
      10 CONTINUE
      CALL MPI_ALLREDUCE(RESREF,DVAL,1,MPI_DOUBLE_PRECISION,MPI_SUM,
      X    MPI_COMM_WORLD,IERR)
      RESREF=DVAL
```

1.5 Data Sets

All datasets were extracted from standard FIRE applications, where at a selected time step the coefficients of the discretized continuity equation, the source term and the whole addressing information is written to a formatted file. This file has to be converted to binary format by using the standalone routine form2bin.f. In particular for large datasets this becomes important in order to reduce the I/O time; e.g. the unformatted file for the dataset COJACK that contains 318044 internal cells has more than 27 MB.

Dataset	TJUNC	SWIRL	PENT	COJACK
Number of internal cells	13845	47312	108000	318044
Total number of cells	19061	60394	124928	359899
Mesh topology	struct.	unstruct.	struct.	unstruct.
Dimensionality	3D	3D	3D	3D

2 ECMWF

Three kernels have been developed for version 2 of the benchmark programs. These are named LG, LM and SL, and are described in the following sections. For each of the kernels both

HPF+ and message passing versions are supplied, and in the case of the LG and LM kernels a separate serial version is also provided. All kernels are written in F90 and include both a serial initialisation phase and a parallelised time-stepping phase in the same program. This differs from the structure used in the LG version 1 benchmark kernel where initialisation and time-stepping phases were performed in separate programs, written in F90 and F77 respectively. This was done for two reasons, firstly, allocatable arrays were not supported by VFCS at the time, and secondly, because the block sizes used in GEN_BLOCK distributions had to be known at compile time. These are no longer constraints in the new release of VFCS, and as a result the program development effort and execution procedures for these kernels is much improved. Schedule reuse is recognised as a key factor for the performance of HPF programs. Each of the ECMWF HPF+ kernels can take advantage of schedule reuse and have been coded to use the simple form REUSE(COND=LREUSE) on INDEPENDENT directives. It is planned that these HPF+ kernels will be run with and without schedule reuse to assess the performance gain from this capability.

2.1 LG Kernel

The original (version 1) LG kernel was written to perform IFS transpositions in grid-point space.

The LG kernel has now been enhanced to address more of the IFS program issues in grid-point space, namely,

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

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

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

DO JFLD=1,NGTO
  IB=0
  DO J=1,NGPTOTG,NPROMA
    IBEG=1
```

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

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 VFCS/HPF+ Rel.2) 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) = RESHAPE(PROCS)
!HPF$ DISTRIBUTE(GEN_BLOCK(MAPGLA),INDIRECT(MAPFLDO)) ONTO PROCSAB :: ZGL
      REAL ZGL(NPROMAG,NGT0)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZGA
      REAL ZGA(NGPTOTG,NGT0)

!HPF$ INDEPENDENT(JFLD,J),REUSE(COND=LREUSE)
        DO JFLD=1,NGT0
          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. This is an enhancement from the version 1 LG kernel where only (GEN_BLOCK,*) could be used.

The use of an INDIRECT distribution to distribute (NGT0) fields over NPROCB processors is required as both 3D variables (i.e. those with a vertical dimension) and 2D variables (surface fields) are included in this dimension. In this case we need to keep the same set of vertical levels for all 3D variables distributed to the same NPROCB processor. 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) = RESHAPE(PROCS)
!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(JVAR,JLEV,J),REUSE(COND=LREUSE)
        DO JVAR=1,NG3D0
          DO JLEV=1,NFLEVG
            DO J=1,NGPTOTG
              ZGL3(INDL(J),JLEV,JVAR)=ZGA3(J,JLEV,JVAR)
            ENDDO
          ENDDO
        ENDDO
!HPF$ INDEPENDENT(JVAR,J),REUSE(COND=LREUSE)
        DO JVAR=1,NG2D0
          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 (yet to be seen). 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).

2.2 LM kernel

The purpose of the LM kernel is to test IFS transpositions in fourier space. In these transpositions, fourier data is redistributed between a latitude distribution and a wave distribution.

For example, the serial code fragment for an LTOM transposition is coded as follows,

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

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

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

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

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

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

2.3 SL kernel

The semi-Lagrangian (SL) calculation in the IFS consists of two parts, the computation of a trajectory from a grid point backwards in time, and the interpolation of various quantities at the departure point and at the mid-point of the trajectory. The 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. The SL kernel computes such a halo, and in the case of the HPF+ version uses a HALO specification with a INDIRECT re-distribution directive to obtain the halo data, as shown below,

```
!HPF$ PROCESSORS PROCS(NPROC), PROCSAB(NPROCA,NPROCB) = RESHAPE(PROCS)
!HPF$ DISTRIBUTE(INDIRECT(MAPGP ),*) ONTO PROCS   :: ZSLBUF1,ZSLBUF2
!HPF$ DYNAMIC :: ZSLBUF1
      REAL ZSLBUF1(NGPTOTG,NFLDSLB1)
      REAL ZSLBUF2(NGPTOTG,NFLDSLB1)
      ...
      DO I=1,NSTOP
        ...
!HPF$ REDISTRIBUTE(INDIRECT(MAPGP,HALO=NSLHALO),*) :: ZSLBUF1
        ...
!HPF$ INDEPENDENT(J,JLEV),
!HPF$*NEW=(I1,I2,I3,I4,I5,I6,Z1,Z2,Z3,Z4,Z5,Z6)
        DO JLEV=1,NFLEVG
          DO J=1,NGPTOTG
            I1=NSLIND(1,J)
            I2=NSLIND(2,J)
            I3=NSLIND(3,J)
            I4=NSLIND(4,J)
            I5=MAX(1,JLEV-1)
            I6=MIN(NFLEVG,JLEV+1)
            Z1=ZSLBUF1(I1,JLEV)
            Z2=ZSLBUF1(I2,JLEV)
            Z3=ZSLBUF1(I3,JLEV)
            Z4=ZSLBUF1(I4,JLEV)
            Z5=ZSLBUF1(J,I5)
            Z6=ZSLBUF1(J,I6)
            ZSLBUF2(J,JLEV)=(Z1+Z2+Z3+Z4+Z5+Z6)/6.
          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 REDISTRIBUTE 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.

2.4 Summary

In summary, three ECMWF kernels (LG, LM, SL) are provided for version 2 of the benchmark programs, to exploit the new features provided by HPF+ Release 2. The HPF+ features used are :-

- HPF_LOCAL (LG kernel)
- Schedule reuse by specification of REUSE(COND=LREUSE) on INDEPENDENT directives. (LG, LM, SL kernels)
- Generalised distributions (i.e. having more than one dimension of an array distributed). (LG, LM kernels)
- User specified HALO for INDIRECT distributions. (SL kernel)

3 ESI

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

The HPF+ benchmarks were derived from release V1995 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 dependencies and inter-processor communication structures. 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 first project year. Next, a more detailed description of the second version of the kernels, treating contact-impact algorithms, is presented. The main difference between both sets of benchmarks lies in the data locality, which in the 2nd kernel effects parallelization because of the pseudo-global search used by the contact algorithm. Finally, some considerations are given about the use of F90/F95 language constructs as a means to improve the efficiency of HPF as a programming technique.

3.1 Outline of the PAM-CRASH benchmarks Rel.1

PAM-CRASH is an explicit time-marching Finite Element program used for then 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 (Benchmark Version 1)
- contact-impact calculations (Benchmark Version 2)

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

3.2 PAM-CRASH kernel Version 1

The first 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 define the strain rates to which the element is subject. The element stresses are obtained from the strains by means of the constitutive law. 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 exhibit a high degree of data locality.

3.2.1 Data 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 list of nodes are reordered such as to reduce the communication volume. Thus, to be compliant with the initial definition of the HPF language, the interesting INDIRECT directive of the VFCS was not used.

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
```

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. For the first kernels, all optimizations used to vectorize the code had to be removed because they introduce an additional level of indirection the compilers cannot cope with at this time. The code was completely rewritten to enable the introduction of PUREST procedures as provided in the Vienna Fortran Compilation System. This implies to have only local variables in the subroutine.

```
!HPF$ INDEPENDENT, ON HOME(IY(2,I)), NEW (IL,XL,VL,RL, ... )
      DO I = 1, N4NODE
         DT2AUX = DT2AR(I)
         XL(1)  = X(1,IY(2,I))
         XL(2)  = X(2,IY(2,I))
         XL(3)  = X(3,IY(2,I))
         CALL FORCEI(I, XL, FORCE ...)
         DT2AR(I) = DT2AUX
         FORCE1(1,I) =-FORCE(1)
         FORCE1(2,I) =-FORCE(2)
         FORCE1(3,I) =-FORCE(3)
      END DO
```

The INDEPENDENT directive contains:

- The On Home clause with limited restriction
- The NEW statement defining the iteration-private variables: scalars and arrays of various shapes.

3.2.2 Local subroutine

The PUREST directive defined by the HPF+ language specification is used only in a subroutine called within an independent parallel loop. 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.

```
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)....
```

3.2.3 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 four distributed automatic arrays indexed by the connectivities.

```
!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
```

3.2.4 Reduction

The stability of the explicit algorithm is dictated by a Courant-Friedrichs-Levy type of criterion, expressed by a maximum time step by which the integration may proceed. To compute this stable time step, one needs to find the minimum time step over all elements, requiring a special global reduction operation. In the corresponding kernel, this reduction loop has to be modified in order to comply with the available compilers. An array of elements was allocated, storing the individual element time steps. A min-reduction is then performed on this array. In a later stage, alternate strategies will be considered in an attempt to improve overall parallel performance.

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

If the "inspector/executor" paradigm is used by the compiler to generate communication schemes, it is highly desirable that these schemes be reused in each cycle because of their static nature. Therefore, directives described in the new HPF+ language specification will be used in the next benchmark to avoid to recompute the same communication patterns and thus reduce the inspector costs.

3.3 PAM-CRASH kernel Version 2 : Self-Impact with 3D bucket search

Within the crashworthiness simulation problems resolved by the PAM-CRASH code, 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, and the development of such algorithms is still going on. 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 disproportionally large amount of time. In fact, vectorizing contact algorithms is difficult because of the conditional branching associated with the search algorithms. 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 intensify the complexity of the HPF constructs needed.

3.3.1 Self-Impact algorithm

In contrast with the stress-strain calculation, the contact-impact algorithm uses more complicated data structures. The high level of indirection arises as a consequence of the need to loop over subsets of nodes and elements, coupled via the connectivity and inverse connectivity lists. Moreover, these nodal lists are not mutually exclusive. The peculiarity of the self-impact algorithm is that slaves node belong to the master segments. Moreover, these nodal lists are not mutually exclusive. While in surface-to-surface contact, nodes of one surface are guarded against penetrating the other surface, self-impacting contact algorithms use only one surface, meaning that the slave nodes are the nodes of the master segments.

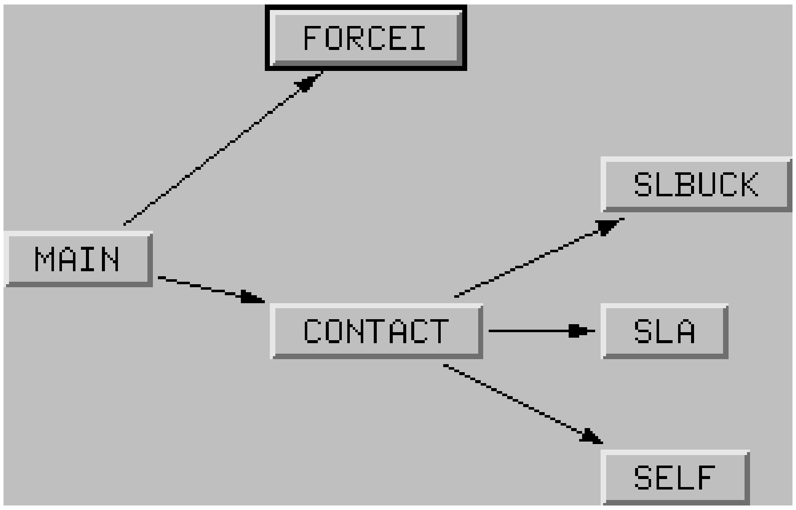

Figure 1: Call Graph of the Benchmark V2.0

The algorithm is divided in three parts :

1) the pseudo-global search, for which the so-called Bucket Algorithm will be used

2) Close search

3) The application of contact forces

   - at the impacting node
   - at the vertices of the impacted segment Contact force application

3.3.2 HPF+ benchmark version 2

The sequential kernel is taken from the PAM-CRASH solver written in F77. However, because of of the highly optimized and vectorized constructs of the standard version, the most important loops of the contact search algorithm as well as the penalty method were rewritten in a simplidied form so as to comply with the actual HPF compilers. A more simplified form was rewritten for the contact search algorithm and the penalty method to be compliant with

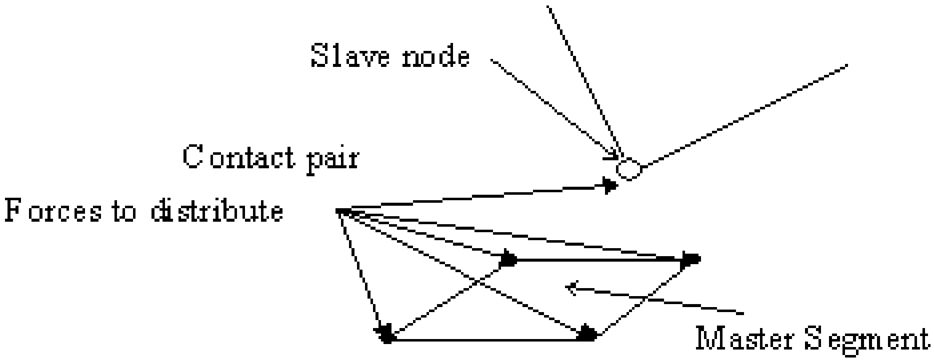

Figure 2: Definition of one Contact Pair

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 well defined. Some investigations about different constructs based on the F90/5 language will be envisaged.

3.3.3 Global search : 3D-bucketing algorithm

The 3D space occupied by the sliding interface is divided into NBUCKX*NBUCKY*NBUCKZ 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. The data dependencies can be classified in two groups. First, one needs to gather the coordinates of the slave and master nodes. Next, a list of nodes is constructed which will be used to determine the contact candidates. In the sequential kernel, the access to this list is made by arrays of pointers.

```
!HPF$ SCHEDULE, SAVE :: S1
!HPF$ INDEPENDENT,ON HOME(NSV(I)), NEW(XI), GATHER(X::S1)
!HPF$&REDUCTION(BUBEGX,BUBEGY,BUBEGZ,BUENDX,BUENDY,BUENDZ)
     DO 7 I=1,NSN            ! NSN = Subset of the number of nodal points
        XI(:)= X(:,NSV(I))   ! list of slave nodes
        BUBEG(:) = MIN(BUBEG(:),XI(:))
        BUEND(:) = MAX(BUEND(:),XI(:))
     7 CONTINUE
NALL = NBUCKX*NBUCKY*NBUCKZ ! total Number of nodes/bucket
C  put nodes in LSBUCK array : Create list of slave nodes a bucket
     DO 80 I=1,NSN
       J = IWRK1(I)       ! J : Id of the bucket for the slave node I
       IPOS = IBUCKT(J+1) ! IPOS : Position of the node in the bucket
       LSBUCK(IPOS) = I   ! List of slave nodes / buckets
       IBUCKT(J+1) = IPOS + 1
     80 CONTINUE
```

In the next section, the bounding box for each segment needs to be found. In a loop over the buckets spanning the segment, all nodes inside this bounding box are found. Nodes connexted to the master segment are eliminated. The remaining candidates (slave nodes, master segment) are called the contact pairs.

```
!HPF$ INDEPENDENT ,NEW(N1,N2,N3,N4),
!HPF$& GATHER(X:: S2=PATTERN(IRECT(II,1 :4)))
      DO 5 II=1,NRT      ! Loop on Number of segments (subset of elements)
        N1 = IRECT(1,II) ! Gather connectivities of segments
        N2 = IRECT(2,II) ! N1,N2,N3,N4 : Id of nodes (1 :NUMNP)
        N3 = IRECT(3,II)
        N4 = IRECT(4,II)
C  * Search box
        XYZMAX(:,II) = MAX(X(:,N1),X(:,N2),X(:,N3),X( :,N4)) + THK
        XYZMIN(:,II) = MIN(X(:,N1),X(:,N2),X(:,N3),X(:,N4)) - THK
C<...>
      5 CONTINUE
C     * Determine buckets BEGIN-END containing the searchbox of the II segment
      DO 9 II=1,NRT
        NRB(:,II) = 1 + INT((XYZMIN(:,II)-BUBEG(:)/BUSTP(:))
        NRE(:,II) = 1 + INT((XYZMAX(:,II)-BUBEG(:)/BUSTP(:))
      9 CONTINUE
```

The HPF+ implementation of the two previous loops and the next loop are not yet fixed. Some better optimized constructs could be found, depending on compiler support. This loop construct the list of contact pairs.

```
NMNOD = 0
!HPF$ INDEPENDENT , NEW(...), REUSE(COND=MOD(NCYCLE,NACC))
     DO II=1,NRT          ! List of segment to search contact pair
C  * loop over buckets (in z-direction buckets are blocked)
       DO 15 NX=NRB(1,II),NRE(1,II)           ! Nested loops (3-D)
         DO 16 NY=NRB(2,II),NRE(2,II)
C   * calculate number of first bucket in xy domain
           NBUCKT = (NX-1)*NBUCKY*NBUCKZ+(NY-1)*NBUCKZ
C  * calculate first and last entry in ibuck for bucket
           NBEG = IBUCKT(NBUCKT+NRB(3,II))
           NEND = IBUCKT(NBUCKT+NRE(3,II)+1)-1
           DO 20 J=NBEG,NEND
             IF FUNCTION(J,XMAX(II)) GOTO 20
             NMNOD = NMNOD+1
             IRTL (NMNOD) = LSBUCK(J)        ! store List of slave nodes in contact
             LSEM(NMNOD) = II                ! store the corresponding segment
           20 CONTINUE
         16   CONTINUE
      15  CONTINUE
```

3.3.4 Apply penalty forces

For each contact pair, we apply a function that compute the new forces to add to the existing forces of the touched nodes of the impacted segment and the slave node.

```
!HPF$ INDEPENDENT ,NEW(ISEG,ISLV,ND1,ND2,ND3,ND4), REDUCTION(A)
!HPF$ REUSE (COND=MOD(NCYCLE,NACC))
! Reuse the different schedules previously calculated
     DO 32 I=1,NMNOD            ! Number of contact pairs
       ISEG = LSEM (I)          ! Master segment found
       ISLV = NSV(IRTL(I))      ! Id of slave node found
       ND1 = IRECT(1,ISEG)      ! Connectivities of the segment
       ND2 = IRECT(2,ISEG)
       ND3 = IRECT(3,ISEG)
       ND4 = IRECT(4,ISEG)
       <...>
C * Distribution of contact forces to the slave node
       A (1:3, ISLV) = A(1:3, ISLV) + FN(1:3,I)
C * Distribution of contact forces to 4 master nodes of the segment
       A(1 :3,ND1) = A(1 :3,ND1) -FN(1 :3,I)*F1
       A(1 :3,ND2) = A(1 :3,ND2) -FN(1 :3,I)*F2
       A(1 :3,ND3) = A(1 :3,ND3) -FN(1 :3,I)*F3
       A(1 :3,ND4) = A(1 :3,ND4) -FN(1 :3,I)*F4
     32 CONTINUE
```

3.4 MPI benchmarks

The hand-written message-passing versions written in MPI are based on the benchmarks Rel.1 used for the time-measurements on the CS-2 machine provided by the VCPC. They use the same kernel as for the HPF+ benchmarks.

- Host code:
  - Very simple input code
  - It was constructed to read the same input data as the others kernels
  - Distribution of all information to the nodes
- Node code:
  - Initialization by receiving all necessary information from host
  - Same kernel as the HPF+ and sequential implementation
  - One global broadcast to compute the minimal time-step
  - Exchange Forces and update velocities of the boundaries nodes for the neighboring processors
  - Output is redirected to local files for validation of the results

The same input file is used by all different kernels.

3.5 Summary

In this section, the parallelization strategy was described for generating the PAM-CRASH kernels of the HPF+ project. Two classes of benchmark kernels were written using the new HPF directives. The first class was evaluated with VFCS on the Meiko CS-2 and compared with the corresponding MPI version. This kernel revealed the necessity to reuse the communication schedules computed in the inspector phase. The second version will use these new HPF constructs to improve the scalability and performance of the algorithms. Moreover, the contact-impact algorithm was introduced based on the bucket-list. This important new feature needs more attention than the previous kernels because of the pseudo-global search which could reduce performance of the algorithm. The writing of the second benchmark is still in progress because of the need to combine the new HPF construct and F90/5 syntax. Moreover, different compilers need to be evaluated for the ease of programming and language restrictions. The best suited directives, providing a maximum of information to the compilers must be found in order to obtain the most optimized code.

4 NEC - 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. The computational sections of these kernels are intended to reflect the type of computations being performed - through the simplifications included, the calculations are neither optimised for performance nor produce physically realistic results. The aim in producing the generic kernels is to assist 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. In the following, salient features of various kernels, both completed and under development, will be described. The increasing complexity of the kernels within the current set can be summarised by 3 levels: element-node interactions for 4-node shell elements; explicit time-marching built around this shell-kernel; the additional inclusion of contact-impact routines. Release 1 of the project benchmarks included various implementations of the first level kernel. While the HPF+ treatment of the first two levels is now, following the first year of the project development, quite well understood, this is by no means the case for the final level.

4.2 K1 Kernel set: Generic Shell Element Calculation

This set of kernels (or rather implementations of the K1 kernel) represents the basic computational operation within the PAM-CRASH solution algorithm: stress-strain calculations on elements, driven by the changing nodal points, and the accumulation of the resulting forces as nodal quantities. An explanation of the kernel and details on implemented versions used in the language/compiler evaluation (including listings of key subroutines) was given in the deliverable D1.2a and thus will not be repeated in full here. The origin of all the K1 kernel implementations is the F77 kernel K1SEQ. The computational section of interest within this kernel is a single call to the subroutine kforce which contains the following central loop:

```
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
```

The crucial components are: the gather operation for nodal co-ordinates; a call to a routine which uses only loop local data; the final scatter operations for the computed forces. 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. HPF+ evaluations to date have focused on the use of regular decompositions (using BLOCK distributions) - since the options of run-time defined irregular distributions, which would be expected to be necessary in practice for general meshes, has not been available. This feature will be covered in future investigations & alternative K1 implementations. The HPF+ features investigated with the K1 kernel have to date been: INDEPENDENT loop involving a call to a PUREST routine; INDEPENDENT loop with REDUCTION.

Overview of K1 Kernels:

Name	Implementation for:	Comments
K1SEQ	F77	
K1HPF1_a_PGI	HPF-1	Uses HPF-library function SUM_SCATTER Tested with PGI compiler
K1HPF+_b1_vfcs	HPF+	Tested with VFCS HPF+-compilation, Dec. '96
K1MPI_a	MPI	Hand-coded MPI implementation using non-blocking communication

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

The extension of the K1 kernel to include the time-marching updates to (first velocities and then) co-ordinates is a fairly minor one, with the inclusion of an outer loop to simulate multiple time-steps and a routine to perform an update based on the calculated forces. The latter provides an additional component which is fully paralleliseable:

```
!HPF$ INDEPENDENT
     DO IN = 1, NUMNP
       F(1:3,IN) = F(1:3,IN) * MASS(1)
       F(4,IN) = (F(4,IN) + (MASS(3)-MASS(4))*V(5,IN)*V(6,IN))*MASS(2)
       F(5,IN) = (F(5,IN) + (MASS(4)-MASS(2))*V(4,IN)*V(6,IN))*MASS(3)
       F(6,IN) = (F(6,IN) + (MASS(2)-MASS(3))*V(4,IN)*V(5,IN))*MASS(4)
       V(:,IN) = V(:,IN) + DT1 * F(:,IN)
       X(1:3,IN) = X(1:3,IN) + DT2 * V(1:3,IN)
     END DO
```

The outer loop, though a trivial addition, enables the investigation of the schedule reuse facility within the HPF+ language: the inspector costs within the KFORCE subroutine can be amortised over repeated subroutine calls. The HPF+ version of the (computational part of the) KFORCE routine thus becomes:

```
!HPF+ INDEPENDENT, NEW(XN,FORCE), ON HOME (IX(1,IEL)), REUSE
     DO IEL = 1, NUMEL2
       XN(1:3,1:4) = X(1:3,IX(1:4,IEL))
       CALL MFORCE(FORCE,XN)
       AFORCE(:,:,IEL) = FORCE
     END DO

!HPF+ INDEPENDENT, ON HOME (IX(1,IEL)), REDUCTION(F), REUSE
     DO IEL = 1, NUMEL2
       F(:,IX(:,IEL)) = F(:,IX(:,IEL)) + AFORCE(:,:,IEL)
     END DO
```

Overview of K2 Kernels:

Name	Implementation for:	Comments
K1SEQ	F77	
K2HPF+_a	HPF+	Implementation completed. Testing dependent on VFCS extensions & handling of F90 constructs. An F77-based version (closer to the previous HPF implementations of K1) could be developed as required.
K2MPI_a	MPI	Implementation requires only minor modification of K1MPI_a - communications overhead will be unchanged.

Future investigations of irregular data distributions could also be performed on the basis of the K2 kernels. The same is true for investigations of F90 use.

4.4 CK1 Kernel set: Segment-based Contact-impact

In contrast to the stress-strain calculations, 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 disjunct) 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). Even if this means that the relative amount of contact calculations is disproportionate, this is not important: the aim of the kernels is to investigate HPF treatment of generic code constructs (and the same computational load will be evident in the MPI kernels used for performance comparisons).

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 which are checked for penetration into a much smaller box (the contact cube) around the segment. This is illustrated in Figure 1.

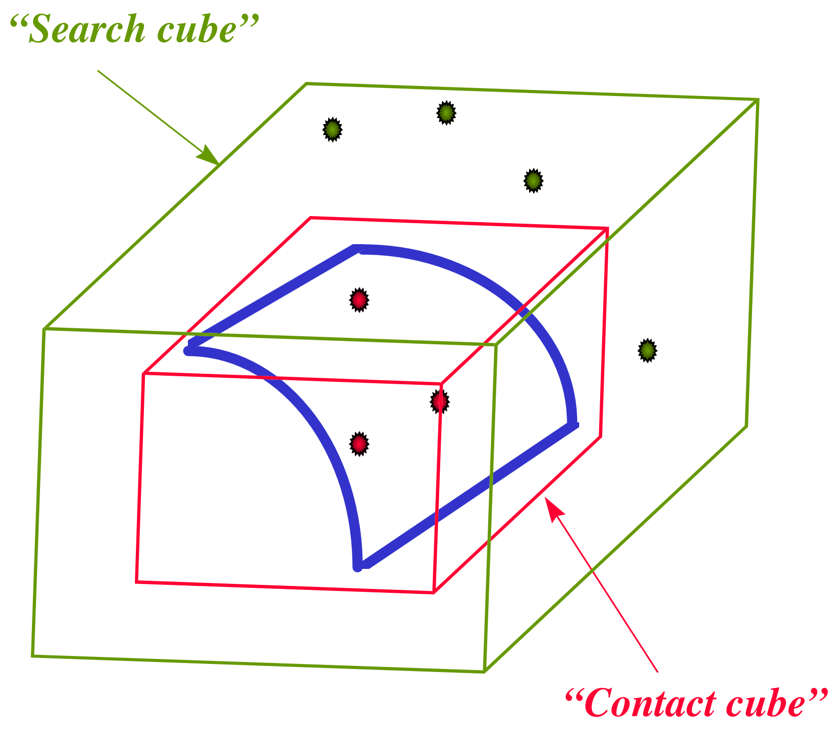

Figure 3: Segment-based bounding-boxes

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 below by a code fragment from CK2SEQ modelling the search within the contact cube - the list schn holds the nodes within the search cube, 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.

```
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; ENDDO

  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
```

The computational features of consequence for the HPF 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, this loop should correspond to an INDEPENDENT REDUCTION whose communication schedule 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 and the calculation can be included in an INDEPENDENT loop. However, the corresponding gather operation corresponds to (possibly, dependent on slide line definition) all-to-all global communication of the co-ordinates.

An initial HPF+ implementation, CK1HPF+_a, has been completed. The use of lists and pointers has been replaced by logical masks. One immediate drawback is the need for a 2-D mask of size (number of contact nodes) (number of contact segments). However, the distribution of these masks is then straightforward and complex IF constructs can be replaced by F90 WHILE constructs. The contact box search section, shown below, exploits the HPF+ conditional schedule reuse clause.

```
!HPF+ INDEPENDENT, ON HOME(CSEGS(1,ISEG)), REDUCTION(F), &
!HPF+ NEW(NSEG,XSEG,CMIN,CMAX,FC), REUSE( COND = .NOT.RECALC)

SEGMENTS: DO ISEG = 1, NCS
NSEG = CSEGS(:,ISEG);           XSEG = X(:,NSEG)

NUMNOD = COUNT (MaskSum(ISEG,:))
IF (NUMNOD.EQ.0) CYCLE SEGMENTS

CMIN = MINVAL(XSEG,2) - THKCONT
CMAX = MAXVAL(XSEG,2) + THKCONT

Mask1 = .FALSE.
WHERE (MaskSum(ISEG,:)
+ .AND. (CMIN(1) <= X(1,CNODES).AND. CMAX(1) >= X(1,CNODES))
  + .AND. (CMIN(2) <= X(2,CNODES).AND. CMAX(2) >= X(2,CNODES))
  + .AND. (CMIN(3) <= X(3,CNODES).AND. CMAX(3) >= X(3,CNODES)))
  + Mask1 = .TRUE.
END WHERE

DO I=1,NCN
IF (Mask1(I)) THEN
     CALL CORRECT(X(:,CNODES(I)),XSEG,FC,THKCONT)
  F(1:3,CNODES(I)) = F(1:3,CNODES(I)) + FC(:,5)
  F(1:3,NSEG(:)) = F(1:3,NSEG(:)) + FC(:,1:4)
END IF
END DO

END DO SEGMENTS
```

Overview of CK1 Kernels:

Name	Implementation for:	Comments
CK1SEQ	F77	Completed - possible requirements for modifications to improve model used
CK1HPF+_a	HPF+	Initial implementation completed
CK1MPI_a	MPI	Implementation not yet commenced

4.5 Future Kernels

Further kernel development will focus on the contact-impact kernels, since these pose the greatest challenge to achieving efficient HPF performance. Currently only at a design stage, an alternate implementation of the bucket-list based contact algorithm within the ESI benchmark should be included in the final set of kernels.As implied by the current form of the HPF+ implementation of CK1, F90 useage will play a role in the continued development and extending the features employed (to include, for example, derived types) will be an area of investigation. Irregular data distributions in conjunction with run-time defined mesh partitions can be integrated into new variants of the above kernels.