An evaluation of HPF+ with crash simulation kernels: Intermediate Results

G. Lonsdale§, F. Zimmermann§, J. Clinckemaillie#, S. Meliciani#

§ C & C Research Laboratories
NEC Europe Ltd.
Rathausallee 10, D-53757 St. Augustin
Germany
{lonsdale, falk, petitet} @ccrl-nece.technopark.gmd.de

Engineering Systems International SA
20, rue Saarinen, Silic 270
Rungis Cedex
France
{jc, sm} @ esi.fr

1 Overview

This paper reports on the use of HPF language extensions for the irregular computational and communications requirements arising in the algorithms found in crashworthiness simulation codes. The HPF language extensions, forming the HPF+ language definition, have been designed to deal with the requirements of complex, scientific applications which involve irregular constructs (from meshes to data structures and access patterns). The HPF+ language development is the central result of the **HPF+** project (funded by the European Commission's ESPRIT Programme in the area of Long Term Research); for further information see [1].
A discussion of language requirements and implementation aspects of the HPF+ PAM-CRASH kernels will be proceeded by a brief introduction to the algorithms used within PAM-CRASH. The results include comparisons of kernel performances achieved with HPF / HPF+ compilers, the latter with the Vienna Fortran Compiler (VFC), together with hand-coded MPI implementations. The results presented here correspond to the penultimate set of kernel codes and corresponding performance evaluation. The workshop presentation will provide an update to the report including an overview of the final crash kernels and the performance obtained with the final HPF+ extensions.

2 PAM-CRASH Algorithms

A leading case of simulation as an industrial design tool, and the inter-dependence on leading-edge computing platforms, can be seen in the automotive industry with crashworthiness and occupant safety simulation - illustrated in this paper by the commercial software package PAM-CRASH™ from Engineering Systems International. PAM-CRASH is used for the numerical simulation of the highly non-linear, dynamic phenomena arising in short-duration contact-impact problems. The non-linear explicit finite element method employed uses a Lagrangian formulation of the equations of motion of the nodes of the unstructured mesh constructed by the replacement of the physical model by an interconnected set of mechanical elements. Modelling of the materials involved in the physical model - sheet metals, metals, plastics, carbon-fibre composites etc. - is done on an element level. This locality of discretisation enables all stress-strain calculations to be performed element-wise (giving a high degree of locality in parallel implementations !) and facilitates the use of a simple central difference time-marching scheme for the, thus diagonalised, equations of motion. For most industrial models, the majority of elements employed in the meshed object are 4 node thin shell (reduced integration) elements with elastic or elasto-plastic material properties. However, these are supplemented by a whole range of, in part highly specialised, mechanical elements: solids, beams, bars, spotwelds, springs, joints etc.

The two basic computational components are: the time-integration at nodal points, "force calculations" on the elements defined by those nodal points. The time-integration calculates the accelerations, velocities and finally new co-ordinates of the mesh based on the existing forces at the nodal points, created/generated by the movement of the points at the previous time-step. The major computational costs of the algorithm occur in the calculation of the forces at the nodal points. These force calculations can be broken down into (essentially) stress-strain calculations and contact-impact calculations - the two having very different levels of data locality, when considering parallel implementations. (A simplification is taken here, though the bulk of the computation is covered. The reader is referred to the above-mentioned references for a description of special features, such as the handling of nodal constraint sets, rigid-body interactions or airbag calculations.)
The stress-strain calculations are performed over the elements. The calculation on each element requires as input the latest co-ordinates and velocities from only those nodal points defining the element. Once calculated, the force on the element is distributed as individual forces at the nodal points. These calculations produce the largest contribution to the overall computational cost (between 60% and 80%, depending on the particular model).
In contrast to the stress-strain calculations, the contact-impact algorithms used within the 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 PAM-CRASH code, the latter is currently a penalty method, whereby contact forces are introduced at the impacting node and at the nodal points of the impacted segment. 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.

The reader is referred to [1] & [2], and the references therein, for further information about the PAM-CRASH code and its parallelisation.

3 The HPF+ Crash Kernels & Results

Within the HPF+ project, two levels of PAM-CRASH-related benchmarks have been developed to assist with both the definition of language requirements and the evaluation of the language via its implementation in the VFC compiler from the University of Vienna. The generic crash kernels are 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 was 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, we will give an introduction to the HPF+ parallelisation by showing examples of parallel code fragments from the generic kernels and performance results from both generic and full benchmark kernels. The results illustrate the comparative performance of HPF+ and HPF-1 parallel kernels (meaning kernels exploiting the features of the two language definitions) with respect to hand-coded message-passing (MPI) versions. Attention will be restricted to the simplest kernels, which *exclude* contact-impact algorithms.

The dominant computational section of the PAM-CRASH calculations, as discussed in Section 2, handles the stress-strain calculations in the elements. Within the generic kernels, this is emulated via calculation over 4-node elements as illustrated in the HPF+ code fragment in Figure 1.

```
!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
```

Figure 1: Generic crash kernel code fragment for HPF+ handling of elemental force calculations.

This HPF+ parallelisation uses standard BLOCK distributions of the arrays used in the computation and focuses on the use of the **INDEPENDENT** directive, supplemented by constructs to:

- specify the distribution of the computation over the processors, **ON HOME**; such that element-wise force calculations are performed;
- indicate that a reduction operation takes place, **REDUCTION**;
- specify that the subroutine execution neither influences other do loop iterations nor introduces communication (via a **PUREST** directive within the routine MFORCE);
- indicate to the compiler, that the communication pattern which it may generate is invariant (i.e. it will not change for *all* executions of the particular loop), **REUSE**.

For the inspector-executor compilation strategy employed in the VFC compiler, the result of the **REUSE** directive is that the (computationally expensive) inspector-phase need only be executed once. Whereas the former options may be found (perhaps in a modified form) in the HPF-2 developments, the latter belongs to the *work-in-progress* category.

The combination of indirect addressing and the reduction operation poses problems for the commercial HPF compilers tested. However, an alternative to the second loop in the above code fragment is presented by use of the HPF library function **SUM_SCATTER**, giving the replacement code shown in Figure 2. The implementation of the full PAM-CRASH benchmark kernel for the Portland Group compiler also used such a replacement.

```
do ic = 1, 6
    f = SUM_SCATTER(aforce(ic,:,:), f, ic, ix)
end do
```

Figure 2: Generic crash kernel code fragment employing the HPF Library function, SUM_SCATTER.

The basic shell-element force calculation shown above is embedded within a time-stepping loop - which provides the mechanism by which **REUSE** can provide performance gains - and is followed by the node-based acceleration-velocity-displacement update. The latter should not contribute to communication overhead (given that these quantities are all distributed in the same way). The modelling of these operations in generic kernel is illustrated in Figure 3.

The code section in Figure 3 provides an example of possible drawbacks in moving from F77 to F90: while code clarity is indeed greatly improved, certain F90 compilers are unable to generate code which is as efficient as the original F77 version (as used, incidentally, in the MPI kernel). One reason for this is that the F77 version implements the assignments to the six components of the array **f** in a single DO loop which enables improved cache-use since it avoids single loops with stride-6 access (which could be generated by an F90 compiler with the code as in Figure 3).

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

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

v = v + dt1 * f

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

Figure 3: Generic crash kernel code fragment for the nodal calculations.

One additional source of significant overhead seen with F90 compilers is the copy-in / copy-out problem at subroutine boundaries. Some F90 compilers do not check for contiguous memory layout of actual arguments, copying instead those structures to temporary contiguous memory locations even though this may be unnecessary. Thus, programs with many subroutine calls will show significant performance degredation.

The scalability of the HPF+ version of the generic kernel, as realised by the VFC compiler (release 0.92), in comparison with the hand-coded MPI code is illustrated in Figure 4. The platform used was a 128 processor NEC Cenju-3, a distributed-memory machine employing VR 4400 (75 MHz) processors and a multi-stage interconnect network. The model problem was a simple box with mesh of size $4 \times 40 \times 160$, with 1000 (time-step) iterations being performed. The **REUSE** feature is critical in allowing the performance to approach that of the MPI code - without this, the simulation times are an order of magnitude higher. Nevertheless, the achievement of scaling similar to that exhibited by the MPI code is at a cost of a large overhead for the case of a single processor - only part of which can be attributed to a performance loss when moving from Fortran 77 to Fortran 90.

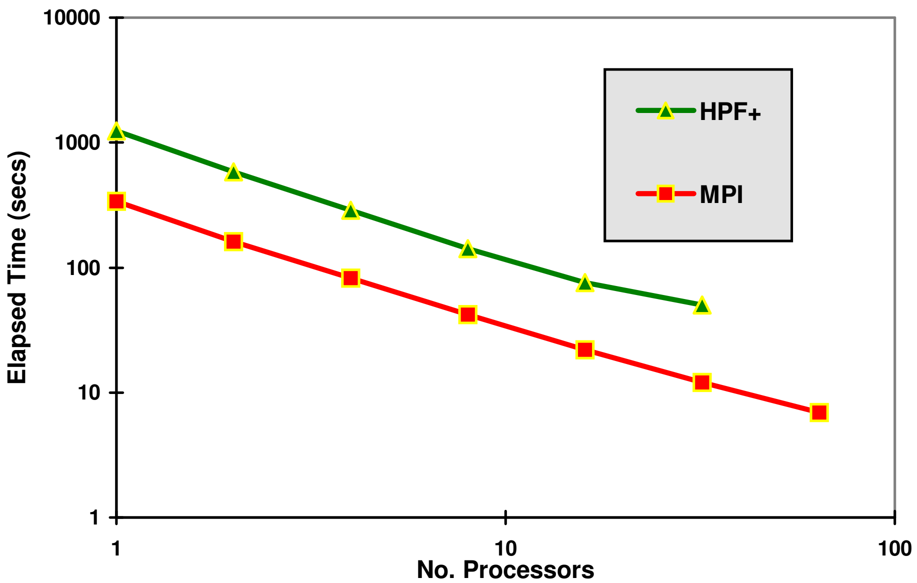

Figure 4: Generic crash kernel - HPF+ & MPI comparison on the NEC Cenju-3.

The problems faced by commercial compilers when dealing with such irregular codes are exemplified by the performance results achieved with the full PAM-CRASH benchmark code on a Cray T3E (DEC 300 MHz processors) using the Portland Group compiler (release R2.3). “Internal speed-ups” achieved with both HPF and MPI kernels for a box-beam model (with approximately 35000 elements) showed very similar scaling behaviour up to 64 processors (speed-ups around 40 in both cases). With 128 processors, the gain with the HPF code was either much reduced or negligible (depending on compiler flags) whereas the MPI kernel still exhibited a speed-up of over 80. However, the term “internal speed-up” has been used here to indicate parallel speed-ups calculated with respect to the single processor version of *each* individual code. This allows the scaling with processor number to be most clearly demonstrated. In fact, an investigation of elapsed times again reveals an enormous overhead being generated for the HPF code: on one processor the HPF code was approximately 7 times slower than the native F90 compiler (and thus also the MPI code).

4 Concluding Remarks

The advances made with the HPF+ language extensions and their implementation in the VFC compiler within the HPF+ project showed that a large potential for optimization is still available within the HPF-language. The interaction of both new language concepts and their efficient realisation within compiler and/or run time system could lead to competitive alternatives in the future.

The results presented for crash kernels with contact-impact routines excluded already highlight the overheads introduced by HPF compilation when executing on a single processor. Only a fraction of this overhead can be apportioned to the use of Fortran 90 constructs and compilers. When including the contact-impact routines, with increased indirection and the necessity for conditional statements, the single processor overheads are greatly increased. Nevertheless, even though the current overhead in terms of elapsed times would not justify the use of HPF in performance critical applications today, the achieved performance with certain language constructs are very promising. In particular, the re-use of communication schedules for INDEPENDENT loops showed extremely good speed-ups. Furthermore, results included in the workshop presentation show that further gains, beyond those presented above, can be achieved by enabling the user to specify and control the updating of run-time defined HALO regions. On the other hand, the HALO concept assumes a detailed knowledge of the underlying data-distribution policy of the compiler in order to calculate all non-local data elements of each processor by the programmer. Thus, a language feature like this is only suited for advanced programmers.

Acknowledgements

This work has been partly supported by the European Commission via the ESPRIT Programme within the Long Term Research Project HPF+. It is our pleasure to acknowledge the support and assistance from our research partners in the project.

References

[1] HPF+ ESPRIT LTR Project No. 21033. *Optimizing HPF for Advanced Applications.* World-Wide Web document: http://www.par.univie.ac.at/hpf+

[2] J Clinckemaillie, B Elsner, G Lonsdale, S Meliciani, S Vlachoutsis, F de Bruyne and M Holzner, Performance issues of the parallel PAM-CRASH code, *Int. J. Supercomputer Applications and High Performance Computing*, **11(1),** 3-11,1997

[3] ESI Group, Proceedings of the PAM '97 Conference, Oct 16-17, Prague, *available as CD-ROM from ESI*, 1997