Inspector Loop Determination to Reduce Communication Overheads in Unstructured Mesh Code Parallelisation[1]

S.P.Johnson, C.S.Ierotheou and M.Cross

Parallel Processing Research Group
Centre for Numerical Modelling and Process Analysis
University of Greenwich
London SE18 6PF, U.K.

Technical Report PPRG-98-003

Abstract
Effective automatic parallelisation of unstructured mesh (UM) codes requires that the code generation algorithms used must be able to generate efficient parallel code given complex real-world application codes. To achieve this, compile-time algorithms have been developed and implemented within the Computer Aided Parallelisation Tools (CAPTools) that operate in an interprocedural, symbolic framework, aiming to reduce communication overheads. Using the Inspector-Executor code generation technique, a key is to ensure that as few inspector loops as possible are generated, where many of these loops are used for numerous communications. The techniques developed here to address these issues are presented and also demonstrated through the parallelisation of two complex application codes. A Finite Element (FE) code is used to show the effectiveness of these techniques and that the resultant parallel code exhibits significant parallel performance. A more complex multi-physics UM code is also parallelised and compared to an equivalent manual parallelisation. The quality of the code produced, in combination with the continued user recognition of that code, allows further optimisation, leading to efficient parallel code created in a greatly reduced timeframe when compared to a fully manual parallelisation.

1 Introduction

The demand for the raw computational power provided by high performance parallel computing systems is significant, at least within the science and engineering community. Despite the availability of massively parallel processing (MPP) hardware and robust low level tools to enable their use, the exploitation of such high performance computing systems has been much slower than anticipated a decade ago. One key reason for this has been the difficulty associated with the implementation of the significantly large-scale scientific/engineering analysis codes onto MPP systems. Although a number of projects, aimed at porting such software to parallel architectures, have been sponsored by public agencies (for example, the EC funded EUROPORT programme [1]), and have demonstrated the feasibility of achieving efficient parallel implementations, they have merely scratched the surface of the huge volume of codes that should be parallelised. The only

[1] This work has been funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under the Portable Software Tools for Parallel Architectures (PSTPA) programme, Grant No. GR/K40321

feasible solution to this problem is to make available tools to parallelise existing sequential software to efficiently exploit the potential of parallel hardware, whilst making the job of parallelisation both simple and fast.

Considerable effort has be focussed upon the development of tools and compilation systems by many groups world-wide [2,3,4,5,6]. Most of this work in recent years has been in relation to High Performance Fortran (HPF) where many of the parallelisation decisions are made by the user, who inserts parallelism and data placement directives to instruct the compiler on how to parallelise the application code.

In previous work [7,8,9], we have developed a parallelisation system, the Computer Aided Parallelisation Tools (CAPTools), aimed at structured mesh (also known as regular computation) Fortran-based codes. This system can produce parallel versions of real codes that are as good as those parallelised manually, but in a tiny fraction of the time and without the inevitable introduction of human errors. User interaction is a vital component of this parallelisation process, however, the interaction is typically restricted to simple information that is known to the user, and basic parallelisation decisions. Parallelism detection and exploitation are then performed automatically following domain decomposition techniques that have been successfully employed in many manual parallelisations. The algorithms used in the automation of the parallelisation process have been enhanced to cater for the complexities of real world application codes such as those used in computational fluid dynamics, structural analysis etc. It is essential that the parallelisation system can handle "unfashionable" programming techniques since these application codes are written in unrestricted Fortran, often having origins in Fortran IV for legacy codes. Beyond legacy codes, the tools aim to remove the burden of parallelisation details from the code author, who will often specialise in a non-computation field, allowing them to continue development without regard to later parallelisation.

The aims of this current work are to produce unstructured mesh code parallelisations achieving comparable performance to manually parallelised code, where the generated code is recognisable to the user and parallelised in as non-intrusive a manner as possible [10]. The parallelisation strategies employed are those used by many successful manual parallelisations and should therefore reap effective performance from the automatically generated code. The strategy employed for the parallelisation is a domain decomposition of the entities that comprise an unstructured mesh across the processor topology, applying the owner computes rule to attain parallelism.

2 Unstructured Mesh Parallelisation Using CAPTools

The Computer Aided Parallelisation Tools (CAPTools) have been developed for the parallelisation of numerical software [7]. They were previously targeted at structured mesh (regular) based application codes using block, cyclic and block/cyclic array distributions. The extension to the domain decomposition parallelisation of unstructured mesh (irregular) based codes requires many of the same techniques as developed for structured mesh parallelisations. Additionally, it was advantageous to follow similar stages to structured mesh parallelisations so that the user process was the same and familiar to existing users of CAPTools. The following sections briefly highlight

the implications of the various stages that are common to structured and unstructured mesh parallelisations.

2.1 Dependence Analysis

The dependence graph constructed by the tools holds data flow information for every statement in every routine in the code being parallelised. The success of the parallelisation is fundamentally reliant on the accuracy of the dependence graph since the quantity and frequency of communication, in particular, is determined by the graph and this directly affects parallel performance. As a result, the dependence analysis exploits symbolic information and proceeds interprocedurally, requiring significant computational effort in graph calculation [8,11].

2.2 Data Partition Calculation

The data partition calculation determines a set of arrays and the nature (although not the precise details) of their mapping onto the processor topology. For unstructured mesh parallelisations, the partition details are stored as a list, where each entry indicates the owning processor of that partitioned array element. The actual processor numbers are not set until runtime, thus the processing during the remainder of the parallelisation uses the list as a symbolic integer array. The run-time details are calculated using a mesh partitioning tool, in this case JOSTLE [12] (see section 5). This takes the hardware processor topology and mesh topology as input and returns the owning processor list. Typically, for a finite element or control volume code, there will be only a few partitions i.e. one for all element based arrays, another for all node based arrays and so on (see section 6).

2.3 Execution Control Calculation

Execution control mask addition is the next stage in the parallelisation. These masks determine if a particular instance of a statement should be executed on a particular processor. The rules employed for unstructured mesh mask addition are based on the accessing of partitioned arrays and dependence relations with other previously masked statements [9]. The masks for an unstructured mesh parallelisation take the form :-

IF (P_{A}(expr).EQ.PROCNUM) A(expr) = ...

where $\mathbf{P\_{A}}$ is the processor ownership list for the entries in array **A** (where array **A** caused the mask in this case through the enforcement of the 'owner computes' rule), and PROCNUM is this processor's unique identification number. Every processor will therefore compare its number with the owner of this element of array **A** and only one processor will perform the subsequent computation. Obviously, the final code generated involves block masks, where many individual statement masks are merged into a block IF statement. The transformation of masks onto loop limits requires the mesh to be renumbered into a local numbering scheme in the same way as for manual unstructured mesh parallelisations [10].

2.4 Communication Requirement Calculation and Generation

Communication requirement calculation involves comparing the location of a data item with where the data is required within the processor topology. A communication request is generated if it is not certain that the data already resides where it is needed. The basic techniques for communication requirement identification in structured meshes are applicable to unstructured meshes. For partitioned arrays, comparing the usage location with the owning processor of the array element as determined in the partition definition (in both cases involving symbolic variables) indicates whether a communication is required. For unpartitioned data, the comparison of the execution control masks on statements related by a true dependence [9] (i.e. the mask on the assigner indicates data location and the mask on the usage indicates where that data is required) determines the need for a communication. Communication request migration moves requests to as early a point in the code as is legal, i.e. migrate as far as possible without preceding the assignment of any data that is required to correctly perform the communication [9]. This migration moves communication requests out of loops and through routine boundaries to minimise the frequency of communication calls allowing bulk communications. Migration also enables many communications to be merged since many requests will be placed at the same point in the code that also communicate the same data. The following sections detail the communication generation for unstructured mesh codes.

3 Inspector Loops and Their Determination to Increase Migration.

The code generation adopted in CAPTools follows the popular inspector-executor technique [13, 14, 15,16]. The generation of a request for communication of unstructured mesh data involves the generation of two requests, one for the communication (executor) and the other for the construction of a list of all items from that data structure that are to be communicated (inspector).

Both requests are migrated up the control flow graph and through routine boundaries until a barrier to legal migration is encountered. Assignments of data involved in the communication form barriers to the migration of both requests, however, assignments of the data being communicated only form barriers to the executor, ensuring that inspectors will always precede the related executor.

The inspector-executor technique has been developed over a number of years and is used in the parallelisation of irregular codes, including unstructured mesh codes. It is being incorporated into many High Performance Fortran compilers implementing the HPF2 standard [17]. The generality of application codes targeted for HPF parallelisation requires the implementation of the technique to be aimed at any code and therefore cannot exploit the particular features of a class of codes as is done for unstructured mesh codes in this work. As a result, the executor phase in these compilers usually consists of significant code generation to perform communications into specially created workspace and to ensure that the data usage that required the communication refers to that workspace. This often produces code that is not easily recognisable to the user, although since it is generated by a compiler, this is not perceived as a significant problem. The focus upon unstructured mesh codes in this work with a parallelisation strategy targeted specifically at that class of codes allows the executor to merely be a communication statement since the overlap area for the array concerned can be used to store the received data. Another difference between the HPF parallelisation and the CAPTools parallelisation is that strict rules are placed on legal HPF code

that do not apply to the Fortran 77 code that is parallelised by CAPTools. As an example, many real world codes use dimension mapping between routines. Whilst CAPTools can cope with this complication through its interprocedural capabilities, HPF requires the dimension mapping to be removed through code re-authoring.

3.1 Basic Inspector Loop Determination

In this work, the inspector loop is used to build a run-time dependence graph of the interdependencies of an unstructured mesh based array. It provides the graph of data access patterns for a given computation (often based on the mesh topology used in the code) in a standard data structure. This allows partition calculation and communication list calculation to be performed by generic utilities. Two utilities provide the basic inspector and executor functions, **CAP_OVERLAP** takes in a run-time graph and internally stores a communication schedule whilst **CAP_SWAPOVER** performs an overlap update for the input variable following the specified communication schedule. The run-time graph is represented by pairs of entity numbers that are related in a computation implicitly identifying the location where a data item is needed and the location of an up to date value of the used array entry. The execution control mask on the statement using the partitioned array data provides information as to where the data is needed, whilst the array index expression of the used array indicates where the data exists. Consider **A** and **B** as partitioned arrays in the following example :-

IF (P_{A}(expl).EQ.PROCNUM)A(expl)=B(exp2) --> inspector pair(expl,exp2)

The inspector loop for a communication consists of a number of statements that are involved in calculating the values of the expression in the execution control mask on the requesting statement and the index expression in the partitioned array. Obviously, the aim is to minimise the code in the inspector loop whilst at the same time maximising the migration of the inspector request outside as many loops as possible, thereby minimising the number of times it will be executed. The process of inspector loop identification uses the dependence graph to identify statements that are transitively connected via true (or control) dependencies to the expressions in the run-time dependence graph being constructed. The process of identifying such statements must be carefully controlled to avoid the extremes of either inclusion of no other statements (i.e. no possible migration) or inclusion of every statement in the code. Restriction of the volume of code in an inspector loop involves initially restricting eligible statements to be only those whose loop nesting is a subset of that of the requesting statement.

The migration of inspector requests involves several stages. Firstly, all statements that are to be copied to form the generated inspector loop are marked. Statements that form barriers to migration are then identified using dependencies of all statements that have been marked for inclusion in the inspector loop. Any assignment to data used in a marked statement can potentially represent a barrier to migration. Barriers must also be detected from dependencies of the execution control mask and the index expressions of array usage's that are to be copied in the construction of that inspector loop. Any barrier statement that is also marked as being part of the inspector loop does not prohibit migration. This is legal since a copy of that statement is used in the generated inspector loop, satisfying the related dependencies, and not the original statement that would have been a

barrier. Consider the following example in Figure 1 after the addition of execution control masks, where arrays **A** and **B** are both partitioned based upon the same processor ownership array :-

```
S1          IE = 1
S2          NPTR = 1
S3 10       DO I=1,TOP(NPTR)
S4                  IF (P_A(IE).EQ.PROCNUM) THEN
S5                          A(IE) = B(TOP(NPTR+I))
S6                  ENDIF
S7          ENDDO
S8          IE = IE + 1
S9          NPTR = NPTR + TOP(NPTR) + 1
S10         IF (TOP(NPTR).GT.0) GOTO 10
```

Figure 1 Original code section.

```
            IE = 1
            NPTR = 1
20          DO I=1,TOP(NPTR)
                    IF (P_A(IE).EQ.PROCNUM) THEN
                            ADD_EDGE (IE,TOP(NPTR+I))
                    ENDIF
            ENDDO
            IE = IE + 1
            NPTR = NPTR + TOP(NPTR) + 1
            IF (TOP(NPTR).GT.0) GOTO 20
```

Figure 2 Generated inspector loop.

The request is generated by the reference to array **B** in statement S_5 and migration commences from that location. The expressions in the inspector loop pair are (**IE, TOP(NPTR+I)**) with the associated processors **P_A(IE)** and **P_A(TOP(NPTR+ I))**. Statements S_1, S_2, S_8 and S_9 are related to the inspector pair by true dependencies and are thus included in the inspector loop. Statement S_4 is also added to the inspector loop as it has a control dependence on the requesting statement S_5. Additionally, the statements required to implement the surrounding loops (and any related statements) are also marked, in this case S_3, and S_{10}. All these statements are potential barriers to communication migration since they provide information for the inspector pair. However, since they are in the inspector loop, copies of those statements will be used, so no such restrictions apply. The inspector loop can therefore migrate out of the surrounding loops (starting from the requesting statement S_5), to a point in the code immediately after the setting up of the only variable defined externally to the inspector loop, i.e. array **TOP**. The inspector loop for Figure 1 is shown in Figure 2.

3.2 Inclusion of Non-Common Loops in Inspector Loops

When a barrier is found and migration is therefore restricted, inclusion of statements in loops not surrounding the requesting statement can be considered. The criteria for such inclusion is that the inspector loop including non-common loops must migrate out of the loop within which the original inspector was blocked. This allows both the number of statements in the inspector loop and the

frequency with which the inspector loop will be invoked to be minimised. In Figure 3, both arrays A and B are partitioned and a communication request is generated for array B in S_8

```
S1              IE = 1
S2              NPTR = I
S3        10    DO I=1,TOP(NPTR)
S4                    WORK(I)=TOP(NPTR+I)
S5              ENDDO
S6              DO I=1,TOP(NPTR)
S7                    IF (P_A(IE).EQ.PROCNUM) THEN
S8                          A(IE) = ... B(WORK(I))
S9                    ENDIF
S10             ENDDO
S11             IE = IE + 1
S12             NPTR = NPTR + TOP(NPTR) + 1
S13             IF (TOP(NPTR).GT.0) GOTO 10
```

Figure 3 Inclusion of non-common loops in an inspector loop.

Statements S_1, S_2, S_6, S_7, S_{11}, S_{12} and S_{13} will initially be included in the inspector loop. Statement S_4, although related through a true dependence to the inspector loop, is not included since it is nested within a loop that does not surround the statement that requested the inspector. As a result, statement S_4 is a barrier to migration, forcing the inspector request to remain inside the iterative loop (loop 10), immediately after the loop setting up array **WORK**. Since the inspector is still inside some loops the process is repeated, this time allowing loops not common to the requesting statement. This allows statements S_3 and S_4 to be included in the inspector loop, therefore removing S_4, as a barrier to migration. The extra-loop version of the inspector is accepted since it enabled additional loops to be exited, in this case the iterative loop.

3.3 Full Index Range and Linearised Reference Inspector Loops

Another common case that can inhibit inspector loop migration involves the un-partitioned components of a communication requesting array reference. In Figure 4, the inspector pair involves the partitioned component of the usage of array **B** only (i.e. (**IE**, **ELTOP(IE, IG)**)) and will not include loop S_3, which is required to identify the elements of **B** that need to be communicated. If the **K** loop upper limit were constant then the loop would be irrelevant to the inspector loop and only be involved in the generation of the communication itself where simple regular buffering could be used (a communication length could be set to the declared index range if the indices of **B** were interchanged). If the upper limit is not constant then the use of explicit loops surrounding the communication could be used, however this would lead to unacceptably high communication start-up latencies being incurred (Figure 5). To avoid these problems, one of two alternative rules may be employed.

If a definite declared range can be determined for the un-partitioned components of the array then that entire range can be used in the communication leading to similar communications to those generated when the **K** loop upper limit is constant. However, this will potentially involve the communication of data not actually required.

```
S1      DO IE=1,NELEM
S2              DO IG=1,NNODES(IE)
S3                      DO K=1,NGAUSS(IE)
S4                              IF (P_A(IE).EQ.PROCNUM) THEN
S5                                      A(IE)=A(IE)+B(ELTOP(IE,IG),K)
S6                              ENDIF
S7                      ENDDO
S8              ENDDO
S9      ENDDO
```

Figure 4 Migration barrier caused by non-partitioned array component.

Alternatively, the index expression can be mapped to be one-dimensional and that mapped expression used in the inspector loop to specify the memory locations to be communicated (Figure 6). To identify owning processors for the data, modulus and divisor functions are used in utilities to extract the partitioned component for use with the processor ownership array. The inspector pair would then be (**IE**, **ELTOP(IE, IG) + (K-1) * DIM1**), where **DIM1** is the declared size of the first dimension of array **B**. The number of inspector pairs will then be greatly increased as each memory location will be explicitly stored in the graph and resultant communication set. Inside the utility routines, the modulus expression **DIM1** is used to extract the required component of the expression, relating to the relevant entity number, which can then be used to identify the owning processor. This mapping allows the loop S_3 to be included in the inspector loop allowing further migration.

```
DO IE=1,NELEM
        DO IG=1,NNODES(IE)
                ADD_EDGE(IE,ELTOP(IE,IG))
        ENDDO
        CALL CAP_OVERLAP(. . . ., 0, 0, ID)
        DO K=1,NGAUSS(IE)
                CALL CAP_SWAPOVER(B(1,K),1,ID)
        ENDDO
ENDDO
```

Figure 5: Simple inspector loop generating poor parallel code with many communication start-up latencies.

```
DO IE=1,NELEM
        DO IG=1,NNODES(IE)
                DO K=1,NGAUSS(IE)
                        ADD_EDGE(IE,ELTOP(IE,IG)+(K-1)*DIM1)
                ENDDO
        ENDDO
ENDDO
CALL CAP_OVERLAP(. . . ., 0, DIM1, ID)
. . . .
CALL CAP_SWAPOVER(B,1,ID)
```

Figure 6: Linearised inspector pairs generating explicit memory location inspector loop with simple and efficient communication, also passing modulus expression into CAP_OVERLAP.

3.4 Interprocedural Inspector Loop Construction

To allow efficient parallel code generation in real application codes, the process of inspector loop identification must also proceed interprocedurally, incorporating loops and assignments from several routines into a single inspector. In Figure 7, the execution control masks are all set in the caller routine based on variable **IE**. The calls to routines **SETUP** and **UPDATE** are masked by the execution control mask addition algorithm, therefore not requiring any masks in the routines themselves. Consider the requests generated by the usage of partitioned nodal based array **T** in routine **UPDATE**. Initially, the communication requests pass through the routine boundary into the caller routine, but are blocked by the assignment of the index array **WORK** in the call to routine **SETUP**.

Allowing loops not common to the requesting statement enables the assignment to **WORK** to be included in the inspector loop and thus not be a barrier to communication migration. The inspector loop is only blocked by the assignments to arrays **ETOP**, **GPTIEL** and **NTYPE**, along with scalar **NETYPE**. The inspector loop generated is shown in Figure 8.

```
IEND = 0
DO I = 1,NETYPE
      ISTART = IEND + 1
      IEND = IEND + NTYPE(I)
      DO IE = ISTART , IEND
            NPTS = GPTIEL(I)
            . . .
            CALL SETUP(NPTS,IE,ETOP,WORK)
            . . .
            CALL UPDATE(NPTS,IE,U,T,WORK)
            . . .
      ENDDO
ENDDO
. . .
END
```

```
SUBROUTINE SETUP(NPTS,IE,ETOP,WORK)
. . .
WORK(1) = ETOP(NPTS,IE)
DO K = 2,NPTS
      WORK(K) = ETOP(K-1,IE)
ENDDO
. . .
END

SUBROUTINE UPDATE(NPTS,IE,U,T,WORK)
. . .
DO K = 1,NPTS
      U(IE) = U(IE) + f (T(WORK(K)))
ENDDO
. . .
END
```

Figure 7 Typical code extract for a computational mechanics code.

```
IEND = 0
DO I = 1,NETYPE
      ISTART = IEND
      IEND = IEND + NTYPE(I)
      DO IE = ISTART,IEND
            NPTS = GPTIEL(IE)
            WORK(1) = ETOP(NPTS,IE)
            DO K = 2,NPTS
                  WORK(K) = ETOP(K-1,IE)
            ENDDO
            DO K = 1,NPTS
                  ADD_EDGE (IE,WORK(K))
            ENDDO
      ENDDO
ENDDO
```

Figure 8 Inspector loop for communication from Figure 7.

Although this inspector loop appears fairly complex and perhaps time consuming, its location is outside all loops in the code and will therefore only be computed once. Without the interprocedural and non-common loop inclusion features, the inspector and executor would remain nested in the two loops in the caller routine (plus any other loops in the caller routines). This would obviously require very frequent computation and numerous small communications, causing significant overheads detracting from the efficiency of the parallel code.

4 Inspector Loop and Communication Merging.

As with structured mesh codes, the migration of inspector and executor requests leads to many requests being placed at the same point in the code. This provides the possibility of merging or deleting communications (executors) to optimise parallel performance, which in turn requires the merger or deletion of the associated inspector loops.

4.1 Inspector Loop Merging and Deletion Detection

The comparison of inspector loops may be complicated, for example, by coding differences such as that in Figure 9 where, although the inspector loops appear different, they do in-fact generate identical graphs. Figure 10 shows two inspector loops placed at different points in the code due to the use of array **SOLID** (indicating only certain values are required) forming a barrier to the second inspector loop, where the second forms a graph that is a subset of that constructed by the first. The interprocedural components of inspector loops as shown earlier also complicate the merge process.

```
NXTIND=0
DO Il=l,TOTNOD
        POSIND=NXTIND+1
        NXTIND=POSIND+NTEINX(POSIND)
        DO Jl=POSIND+1,NXTIND
                ADD_EDGE(NTEINX(J1),I1)
        ENDDO
ENDDO
```

```
NPTR=1
DO I2=1,TOTNOD
        DO J2=1,NTEINX(NPTR)
                ADD_EDGE(NTEINX(J2+NPTR),I2)
        ENDDO
        NPTR=NPTR+NTEINX(NPTR)+l
ENDDO
```

Figure 9: Two inspector loops that generate identical graphs.

```
DO I=1,TOTELE
        DO J=1,NNODES(I)
                ADD_EDGE(ELETOP(J,I),I)
        ENDDO
ENDDO
```

```
DO I=TOTELE
        DO J=1,NNODES(I)
                IF (SOLID(ELETOP(J,I)) THEN
                        ADD_EDGE(ELETOP(J,I),I)
                ENDIF
        ENDDO
ENDDO
```

Figure 10 Two inspector loops where the second creates a graph that is a subset of the first.

To overcome these difficulties, the test for identical or subset inspector loops, and subsequent deletion, is implemented using an induction proof algorithm, taking the symbolic expressions in the inspector pair and tracing initial values and incremental changes as determined by the dependence graph.

Consider the example in Figure 9. The initial values of the inspector pairs (during the first iteration of each surrounding loop and following only loop independent dependencies) are :-

(NTEINX(2),I) and (NTEINX(2),I)

where both use the array **NTEINX** defined by the same defining statement (for example, a READ statement). For the incremental comparison, each surrounding loop that adjusts the values of the inspector pair between iterations must be considered. Firstly, consider the outer **I1** and **I2** loops comparing the first inspector pairs added. Assuming equality in an iteration, the test for equality in the next iteration involves :-

Knowing $NTEINX(NXTIND_1+2) = NTEINX(NPTR_1+1)$ and $I1_1 = I2_1$
Test $NTEINX(NXTIND_2+2) = NTEINX(NPTR_2+1)$ and $I1_2 = I2_2$

where $\mathbf{NXTIND_1}$ and $\mathbf{NPTR_1}$ are the values of variables **NXTIND** and **NPTR** at the start of the earlier iteration with $\mathbf{NXTIND_2}$ and $\mathbf{NPTR_2}$ the values at the start of the later iteration.

Substitution of the later iteration variables in terms of the earlier iteration variables gives :-

Test $NTEINX(NXTIND_1+NTEINX(NXTIND_1+1)+3) = NTEINX(NPTR_1+NTEINX(NPTR_1)+2)$
and $I1_1+1 = I2_1+1$

It is also necessary to restrict the test by assuming that the integer arrays involved are permutation arrays where the array entries can only be equal if the indices are equal. This gives added information inferred from the previous known information involving integer array **NTEINX** :-

$NXTIND_1 = NPTR_1-1$

Since the test also involves the comparison of two accesses to the **NTEINX** array, that can also be proven true by a comparison of the index expressions alone :-

Test $NXTIND_1+NTEINX(NXTIND_1+1)+l = NPTR_1+NTEINX(NPTR_1)$

With this information, the two references to **NTEINX** in the test can be proven equal and subsequently the full test can be proven true. Additional tests on any control information determining whether the inspector pair is added to the graph and on the number of loop iterations are needed to prove one or other is a subset or both are equivalent. When this is complete, the two inspector loops are found to be equivalent and the test can continue to the next loop. The comparison of loops **J1** and **J2** involves a simple test that also proves that the two inspector loops generate identical graphs. Since all surrounding loops have been considered, the overall test proves that one of the inspector loops can be deleted and the graph generated by the other loop used accordingly.

For the example in Figure 10, the induction test is relatively straightforward. The test on control information proves that the second inspector loop produces a graph that is a subset of the first (i.e. whenever an edge is added by the second inspector loop, that edge will also be added by the first). Since this result is consistent for all loops, the overall result is that the second inspector can be deleted and any communications use the first inspector loop and the related communication set.

4.2 Ordering Inspector Loop Merger to Ensure Communication Merger

The most important role of the communication merger process is the reduction in the communication overhead with all other factors taking secondary importance. The merging of communications (executors) is possible when the related inspectors are either unioned (merged) or one is deleted. To reduce the volume of code generated, the number of inspectors (and thus memory requirements) and the execution time overhead presented by inspector loops, whilst ensuring communication reduction remains the priority, the merger process is performed in the following three stages :-

Stage 1

Attempt to delete inspectors when the related executors are placed at the same point in the code and refer to the communication of the same variable. Legal deletion allows the removal of one communication since all data is also transferred by the other communication. Code volume, execution time and memory requirements are obviously also reduced in this stage. The effect of this stage is shown in Figure 11.

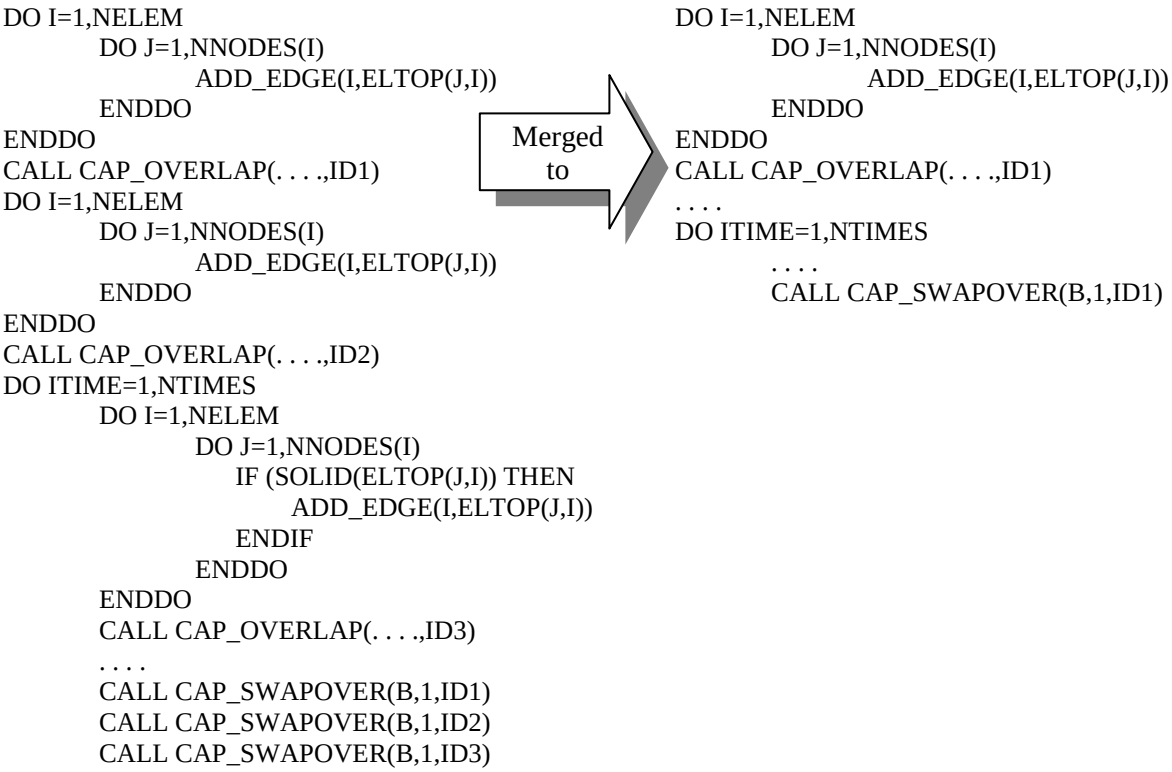

Figure 11 Three inspectors and communications before and after merging.

Stage 2

For all executors of the same variable at the same point in the code, create a graph that is the union all related inspector loops (i.e. add the edges of each runtime graph into a single runtime graph). This allows one communication to be generated for every set of unioned inspector loops. Also, since the same edges may be added by several of the unioned inspector loops, the memory requirements are also reduced. An example of two inspector loops being combined to produce a single communication set, and therefore a single communication, is shown in Figure 12

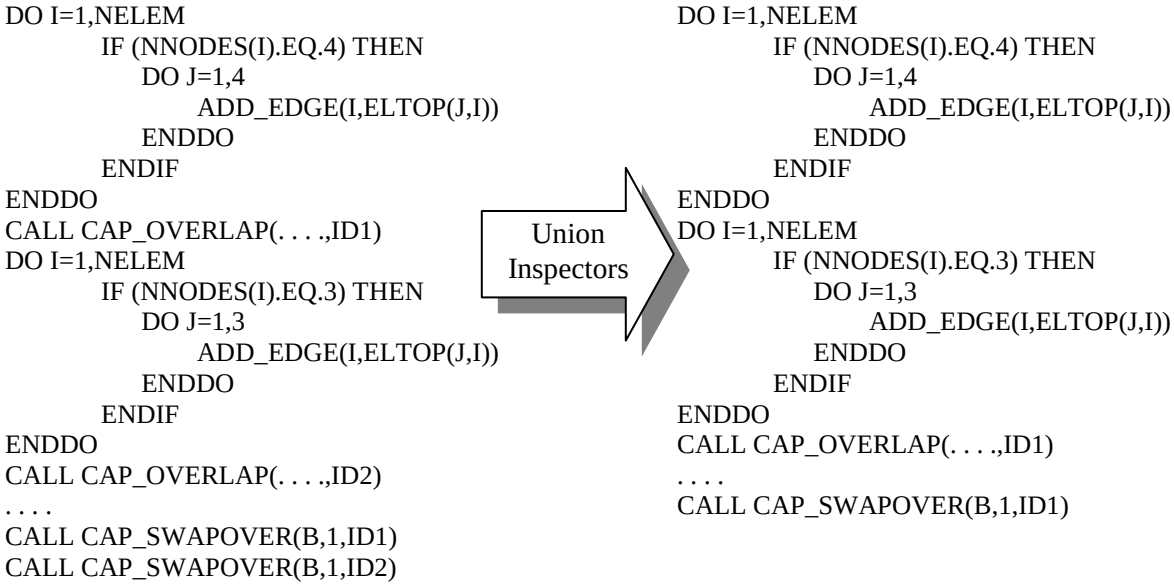

Figure 12 Two inspectors before and after unioning.

Stage 3

Attempt to delete any inspectors regardless of the related executors. This will not allow deletion of any communications, but can greatly reduce the volume of code, execution time overhead and the memory requirements. Two unioned inspector sets can only be merged if every component inspector loop is matched. An example of this stage of inspector loop merging is shown in Figure 13 where, although other factors are reduced, the amount of communication remains unchanged.

In this way, the amount of data communicated, the memory requirements, the volume of inspector loop code and the compute time for inspector loops are all minimised. Stage 1 is performed before stage 2 to avoid identical inspector loops being needlessly placed together in a union. Stage 2 must precede stage 3 since the merge of inspectors with differing executor locations and/or arrays will prevent any legal union of inspector loops and the associated reduction in communications.

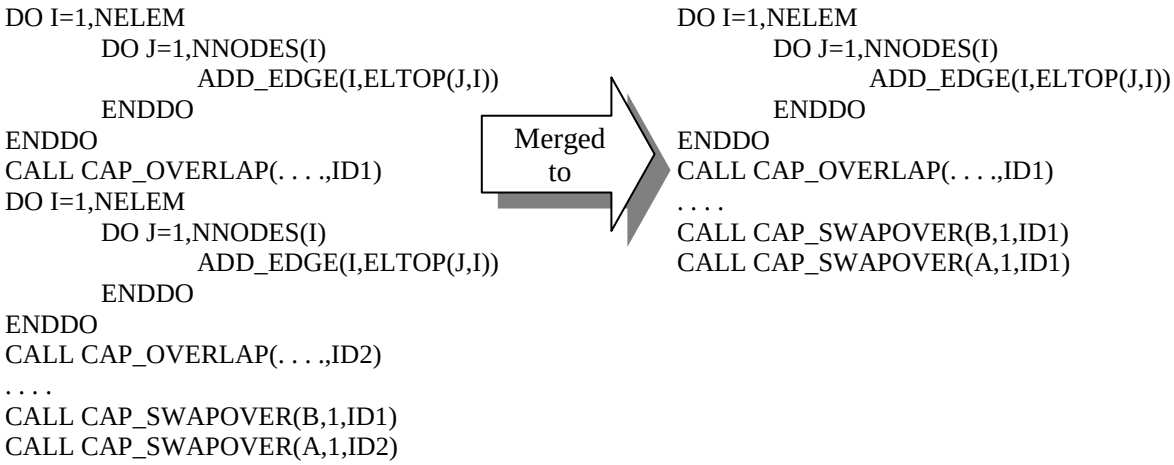

Figure 13: Merging of two inspector loops to reduce memory overheads etc. but not reducing communication.

5 Inspector Loop Generation

The inspector loop generates a list of connected pairs that form the runtime graph that is used in the utilities to determine the partition details and communication sets. Since the inspector loop requests have been migrated out of loops and through routine boundaries, the generated loop must rename variables and loops as required to avoid conflicts with data in the routine in which they are generated. Generation begins by of marking all statements that are to be included in the inspector loop. The process continues by performing a traversal down the control flow graph from the point of generation of the inspector loop, visiting all statements that are accessible on route to the requesting statement. Any statements marked as being in the inspector are copied. This process also continues into routine calls that are marked as being active in the inspector loop with traversal in the called routine commencing from the routine start. This obviously also includes traversal of all routines in the inspector request's call path. The expression pair values (i.e. edges in the runtime graph) are stored in a list at the appropriate point in the inspector loop. This location is identified when the requesting statement is reached during the traversal since this is where all required values have been calculated.

To complete the inspector loop, initialisation is placed prior to it and calls to the appropriate utilities are placed after the full runtime graph is constructed. For a set of unioned inspector loops, a single initialisation precedes all the inspector loops generated consecutively with the calls to utility routines using the graph edges from the set of inspectors.

Calls to the partition calculation utility (JOSTLE [6]) and subsidiary partition calculations (e.g. a nodal partition based on a prior element partition) are automatically generated following the first inspector loop encountered in the code.

The communication sets for each inspector loop are calculated by the **CAP_OVERLAP** utility. Each call of the **CAP_OVERLAP** utility constructs a list of processor and entity numbers that this processor must send or receive, together with an integer that uniquely identifies each

communication set. The executor is performed by a call to the **CAP_SWAPOVER** utility, this performs the communication of data between processors. The **CAP_SWAPOVER** utility uses the integer identifier defined from the call to **CAP_OVERLAP** in order to send or receive the correct data.

5.1 Order of Communication and Inspector Loop Generation

Typically, the inspector loops will be generated before the communications if they were all migrated to the same point in the code. There are cases, however, where some communication must be performed to collect data required in a subsequent inspector loop. The example in Figure 14 uses the values of array **B** for all nodes of all neighbouring elements to the element currently being processed. Array **ADJELE** stores the list of neighbouring elements with array **ELTOP** storing the list of nodes used for each element. When the element based arrays (**A**, **NNEIG**, **ADJELE**, **NNODES** and **ELTOP**) and the nodal based array **B** are partitioned, three requests for communication will be issued, one for **NNODES** in S_4, one for the use of **ELTOP** in S_5 and another for the use of **B** in S_5. If all the requests for both inspector loops and communications for these three usages migrate to the same point in the code, the order of generation becomes vital. The inspector loop for the request generated by the reference to **B** in statement S_5, for example, must follow the communication of the overlap data of **NNODES** as requested by S_4, and **ELTOP** as requested in S_5 as these references are included in that inspector loop. This inter-dependence can be detected by checking for inclusion of array references that requested communication in the inspector loops of other communications. The inspector loops and communications generated for the example in Figure 14 are shown in Figure 15 where the maximum number of nodes per element (**MAXNODE** as extracted from the declaration of **ELTOP** as discussed in section 3.3) is used for the **ELTOP** communication.

```
S1  DO IE=1,NELEM
S2      DO J=1,NNEIG(IE)
S3          IADJ=ADJELE(J,IE)
S4          DO K=1,NNODES(IADJ)
S5              A(IE)=A(IE)+B(ELTOP(K,IADJ))
S6          ENDDO
S7      ENDDO
S8  ENDDO
```

Figure 14 Usage of transitively connected mesh data.

6 Parallelisation of a Finite Element Code

The finite element code considered here uses fairly standard finite element (FE) techniques to solve for displacement, stress and strain over a number of time steps, in either a two or three dimensional mesh. The code consists of around 7100 lines of source in approximately 80 subroutines. As with many such codes, the algorithm constructs stiffness matrices for each finite element in turn and incorporates the components of these matrices into a full system matrix and boundary condition vector. The resulting linear system is then solved by a diagonally pre-conditioned Conjugate Gradient solution procedure to determine displacements. These displacements are then used to evaluate stress and strain values throughout the problem domain. The two processes that dominate the overall runtime are the system matrix construction and the solution of the linear system, although the entire code has been parallelised.

```
DO IE=1,NELEM
        DO J=1,NNEIG(IE)
                ADD_EDGE(IE,ADJELE(J,IE))
        ENDDO
ENDDO
CALL CAP_OVERLAP(. . . .,ID1)
CALL CAP_SWAPOVER(NNODES,1,ID1)
CALL CAP_SWAPOVER(ELTOP(1,1),MAXNODE,ID1)
DO IE=1,NELEM
        DO J=1,NNEIG(IE)
                IADJ=ADJELE(J,IE)
                DO K=1,NNODES(IADJ)
                        ADD_EDGE(IE,ELTOP(K,IADJ))
                ENDDO
        ENDDO
ENDDO
CALL CAP_OVERLAP(. . . .,ID2)
CALL CAP_SWAPOVER(B,1,ID2)
```

Figure 15 Inspector loops and communications for the example in Figure 14.

The effect of the inspector loop determination and merging techniques developed in sections 3 and 4 is demonstrated in tables 1 and 2. The results in Table 1 indicate the number of inspector loop requests and communication requests issued against the number finally generated in the parallel code selectively using the techniques introduced. The loop nesting of the generated inspector loops and communications indicates how many loops they are generated within. The higher the loop nesting of an inspector loop or a communication, the more frequently it will be performed in a code execution. Communications not nested within any loops, for example, will only be performed once in an execution of the parallel code. The one communication nested within two loops for the full power of the inspector loop heuristics relates to an overlap update during iterations of the conjugate gradient solution algorithm that is performed in every time step, therefore being a communication fundamental to the correct operation of the parallel code.

	Inspector / Comms requests	Inspectors generated					Communications generated				
		Loop Nesting					Loop Nesting				
		0	1	2	3	4+	0	1	2	3	4+
Basic (see 3.1)	77	4	1	2			10	2	3	4	
Extra loop (3.2)	77	4	1	2			10	2	3	4	
1D Mapping(3.3)	77	7					11	7	1		
Full	77	7					11	7	1		

Table 1 Effect of inspector loop heuristics in parallel code generation.

	Inspectors generated	Communications generated
No merger	77	77
Subset Merger (4.1)	15	27
Full Merger (4.2)	7	19

Table 2 Effect of inspector loop merging in parallel code generation.

All inspector loops are migrated and generated outside all other loops in the code and so are executed only once. Additionally, only 7 inspector loops are generated due to the merger, each only executed once prior to the start of the time step loop, with the resultant communication sets being used frequently. This is a common occurrence in unstructured mesh codes since, for example, the element overlap communication set will be used frequently for a range of different variables.

The reduction in inspector loops when using subset merger alone in Table 2 shows the significant duplication of inspector loops with the same schedule being usable for a large number of communications. With the addition of the combining of inspector loops to form a unioned inspector, the generated number falls further and, very importantly, the number of related communications significantly decreases.

The resulting parallel code exhibits good performance, attaining a speedup of round 20 on 32 Cray T3D nodes for a 3000 element mesh [18,19]. The generation of recognisable parallel code using parallelisation strategies derived from well understood manual techniques, makes it amenable to further manual optimisation. An inspection of the generated code reveals that the placement of communications and inspector loops matches that of an efficient manual parallelisation with no obvious manual optimisations possible.

7 Parallelisation of a Multi-Physics Unstructured Mesh code.

This code couples the solution of fluid flow equations with those for heat transfer and stress/strain calculations to model, for example, solidification processes. It consists of around 19,000 lines of code in 170 routines. The complex nature of this code represents a far sterner test of the unstructured mesh code generation algorithms developed in this work. Results are shown in tables 3 and 4 to indicate the effect of the inspector loop generation heuristics and the inspector loop merger. These results are also compared with the number of inspector loops and communications in a manually parallelised version of this code [10].

	Inspector / Comms requests	Inspectors generated					Communications generated				
		Loop Nesting					Loop Nesting				
		0	1	2	3	4+	0	1	2	3	4+
No Interpro	418		1	34	9	17		1	49	12	37
Basic (see 3.1)	418	9	2	14		7	5	6	44	3	8
Extra loop (3.2)	418	12	5	12	1	5	5	10	48	4	5
1D Mapping (3.3)	418	11	3	12	9	1	6	8	42	13	1
Full	418	14	9	8			8	11	45	3	
Manual	---	5					25	13	29	3	

Table 3 Effect of inspector loop heuristics in parallel code generation.

Code produced without interprocedural inspector loop determination and migration includes a very large number of deeply nested inspector loops. All the other results operate interprocedurally and demonstrate a significant improvement in the loop nesting and number of inspector loops. The use

of the extra-loop heuristic rule has a significant influence on removing highly nested inspector loops and communications. This is vital since the frequency of execution of such communications will have a drastic effect on the performance of the resultant code and, almost inevitably, create very significant slow-down.

	Inspectors generated	Communications generated
No merger	418	418
Subset Merger (4.1)	37	91
Full Merger (4.2)	31	67
Manual	5	70

Table 4 Effect of inspector loop merging in parallel code generation.

At first sight, the comparison of the final generated code against a manual parallelisation of the same code does not appear encouraging. Numerous factors, however, have a significant influence upon this comparison. Firstly, and very importantly, the manual parallelisation did not consider that the mesh might move between time steps, making the manual version incorrect for such cases. Obviously, the automatically generated code must cater for such instances and has therefore left many inspector loops within the time step loop. Another assumption made in the manual parallelisation is that all overlap communications fit into only five communication sets. This is incorrect, particularly since the application code implements solidification algorithms, where some communications require movement of only values relating to solid or liquid cells. This will often lead to no communication being required, but the manual parallelisation will perform a full overlap update. Many of the other inspector loops in the automated parallelisation are caused, for example, by complicated boundary conditions that require human intervention to improve the parallel code. Since the generated code is recognisable and directly comparable to the original code, and the parallelisation techniques employed are well known, optimisation is relatively straightforward, almost comparable to the manual tuning of a manually parallelised code. After a small amount of manual optimisation, the generated parallel code exhibits some speedup (over 11 on 16 Cray T3E nodes), with a significant overhead being incurred in the computationally dominant overlap updates in the linear equation solvers [19].

7 Related Work

The HPF standard has been extended to include irregular array accesses through indirection arrays. This has led to a number of research projects aimed at extending existing compiler technology to handle communications of irregularly accessed data using some form of inspector-executor technique. Many of these are based in an intra-procedural framework with inspector loop construction and placement along with the associated communication location all being within a single routine [14,15,16,20]. The necessity of inter-procedural communication and inspector loop has been recognised and some areas have been addressed by Agrawal and Saltz in [21] where they use the interprocedural partial redundancy elimination (IPRE) framework to “hoist” inspector loops

out of surrounding loops and through routine boundaries. In simple cases, they also check for redundant communication schedules to remove those that are duplicated. Inspector loop construction with components from any number of routines along with the independent migration and merger of inspector loops in a symbolic framework has not, to our knowledge, been addressed elsewhere.

8 Conclusions

The parallelisation of unstructured mesh software in a semi-automatic environment has been investigated and the code generation algorithms implemented. The realities of dealing with complex real-world application codes have been addressed leading to symbolic, interprocedural techniques that can greatly reduce communication and inspector loop overheads in the generated code. The parallelisation process fits easily within the existing CAPTools structured mesh framework using similar concepts requiring user interaction in a similar manner. The resultant codes produced indicate the potential of parallel code of comparable efficiency to manually produced parallel code being attained in a tiny fraction of the time. The non-intrusive nature of the parallelisation maintains the appearance of the serial code throughout the parallelisation, allowing further optimisations by the user to be identified. With the future addition of mesh renumbering facilities, the quality of the generated code should be directly comparable to that of a fully manual parallelisation.

9 References

1 H. Mierendorff, K. Stuben, C. Thole and O. Thomas. Europort-1: Porting Industrial Codes to Parallel Architectures. HPCN 1995

2 S.Hiranandani, K.Kennedy and C-W. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. Communications of the ACM, 35(8) 66-80, 1992.

3 S.Hiranandani, K.Kennedy, J.M.Crummy and A.Sethi, Advanced Compiler Techniques For Fortran D. Technical Report CRPC-TR93338, Centre for Research on Parallel Computation, Rice University, Houston, Texas, 1993.

4 E.Su, J.Palermo and P.Banerjee, Automating Parallelization of Regular Computations for Distributed-Memory Multi-computers in the Paradigm Compiler. International Conference on Parallel Processing, St. Charles, Illinois, August 1993.

5 H.P.Zima, H.J.Bast and M.Gerndt, SUPERB : A Tool for Semi Automatic MIMD/SIMD Parallelisation. pp 1-18 in Parallel Computing, 6, North-Holland, 1988.

6 H.Zima and B.Chapman. Compiling for Distributed Memory Systems. Proceedings of IEEE Special Section on Languages and Compilers for Parallel Machines, pp 264-287, 1993.

7 C.S. Ierotheou, S.P Johnson, M. Cross and P.F. Leggett. Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes. Parallel Computing, 22(2),1996.

8 S.P.Johnson, M.Cross and M.G.Everett. Exploitation of symbolic information in interprocedural dependence analysis. Parallel Computing, 22(2),1996.

9 S.P. Johnson, C.S. Ierotheou and M. Cross. Automatic Parallel Code Generation for Message Passing on Distributed Memory Systems. Parallel Computing, 22(2),1996.

10 K.McManus. A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures. PhD Thesis, University of Greenwich,1996.
11 S.P.Johnson, C.S.Ierotheou and M.Cross, Accurate Dependence Graph Construction for Finite Element Software. University of Greenwich Technical Report PPRG-98-001, 1998.
12 C.H.Walshaw, M.Cross and M.G.Everett. A Localised Algorithm for Optimizing Unstructured Meshes. International Journal of Supercomputing Applications, 9(4),1995.
13 J.H. Saltz, R. Mirchandaney and K. Crowley. Run-time Parallelisation and Scheduling of Loops - IEEE Transactions on Computers, 40(4)5, 1991.
14 R.V.Hanxleden, K.Kennedy and J.Saltz. Value Based Distributions in Fortran D: A Preliminary Report. CRPC-TR93365-S, Rice University, December 1993.
15 A.Muller and R.Ruhl. Extending High Performance Fortran for the Support of Unstructured Computations. CSCS TR-94-08. CH-6928, Manno, Switzerland 1994.
16 M.Ujaldon, E.L.Zapata, B.Chapman and H.P.Zima. Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation. TR 95-5, Institute for Software Technology and Parallel Systems, University of Vienna, 1995.
17 H.P.F.Forum, High Performance FORTRAN Language Specification Version 2.0, Rice University, Houston, Texas, 1996.
18 S.P.Johnson, C.S.Ierotheou, M.Cross. Computer Aided Parallelisation of Unstructured Mesh Codes. Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA) Conference, Las Vegas, volume 1, CSREA, pp 344-353, 1997.
19 S.P.Johnson, K.McManus, C.S.Ierotheou and M.Cross. Semi-automatic Parallelisation of Unstructured Mesh Codes Using Domain Decomposition. University of Greenwich Technical Report PPRG-98-002, 1998.
20 R.Das, J.Saltz, R.von Hanxleden. Slicing Analysis and Indirect Access to Distributed Arrays. In Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing, p 152-168, 1993.
21 G.Agrawal and J.Saltz. Interprocedural Compilation of Irregular Applications for Distributed Memory Machines. in Proceedings of Supercomputing 95', San Diego, California, 1995.