Semi-automatic Parallelisation of Unstructured Mesh Codes Using Domain Decomposition[1]

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

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

Technical Report PPRG-98-002

ABSTRACT

In this paper we discuss enhancements to a suite of semi-automatic parallelisation tools to enable unstructured mesh (irregular) computational mechanics (CM) codes to be rapidly parallelised using SPMD domain decomposition techniques. This work draws upon the dependence analysis and code generation techniques that were originally developed for structured mesh (regular) FORTRAN codes and have been embedded within the Computer Aided Parallelisation Tools (CAPTools). Strategies abstracted from experience in the manual parallelisation of unstructured mesh codes have been combined with inspector loop ideas to develop a set of parallel code generation procedures. These procedures have been included in CAPTools and used to parallelise a number of significant finite element (FE) and finite volume (FV) FORTRAN codes. Parallel performance equivalent to that achieved by optimised hand parallelised versions of the code are demonstrated. However, where as hand parallelisations required many months of effort, those based upon the techniques and tools described here, only required a few days of work – where most is spent on various optimisations of the generated code.

1. INTRODUCTION

The insatiable demand for increased computational power, particularly in the scientific computing community, has led to the development of powerful computers with the ability to execute programs in parallel. For a number of years, techniques have been manually implemented into existing serial software to allow the efficient exploitation of parallelism. Although these parallelisations have generally been successful, the investment in code development to implement parallelism in both existing and newly developed serial software has proved to be a major inhibitor to the widespread exploitation of parallel machines in industry.

To attempt to relieve this problem, the development of automatic and semi-automatic parallelisation systems has been addressed by a number of research groups worldwide [1,2,3,4,5,6]. These systems range from new or significantly adapted languages with fully automatic compilers to interactive environments employing incremental parallelisation techniques, focussing on small code sections at a time. A major challenge comes from the parallelisation of real world codes and the complexities of algorithms developed, for example, to model complex physical processes. The Computer Aided Parallelisation Tools have been

[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

developed to handle such complex codes (both new and dusty deck) in a structured (regular) mesh context using mesh decomposition parallelisation techniques [7,8]. The success of this work has led to the extension of the structured mesh parallelisation capabilities to handle unstructured (irregular) mesh parallelisations where, rather than a regular structured data representation, the mesh interconnections are held in a data structure, typically read in at runtime. The aim is to extend the techniques that are currently successfully employed in structured mesh parallelisations for the parallelisation of unstructured mesh applications.

The main aims of CAPTools are to produce correct parallel codes with the following characteristics:-

- Parallel efficiency as good as that achieved by a manual parallelisation
- Parallel code generated in a small fraction of the time needed for a manual parallelisation
- Generic and scalable parallel code that is easily ported to a number of platforms
- Minimise changes to the serial code to enable user recognition and therefore allow for optimisation, and possibly future development and maintenance

To achieve these aims, the manual parallelisation of unstructured mesh CFD codes was firstly undertaken [9]. This work has culminated in parallel codes that exhibit high efficiency on a wide range of parallel machines, whilst also minimally altering the original serial code, leaving it easily recognisable to its original authors. These techniques were then abstracted and automated so that they could be applied to any unstructured mesh code. The programming model is based on the single program multiple data (SPMD) concept where each processor executes the same code but operates on its allocated data only. Data is communicated between processors via message passing routines and, although the distributed memory system is used as the base system, such an approach is equally applicable to shared memory systems where data location issues can still have a significant effect on parallel performance.

2 PARALLELISATION STRATEGY

The parallelisation strategy that has been automated is based upon that refined during the manual parallelisation of a number of unstructured mesh codes. The aim of the automation is to produce parallel code comparable in style and performance to the same strategy implemented manually. This should also ease any manual optimisation of the automated parallel code since the automatically generated code will still be recognisable and the strategy employed should be familiar to the programmer.

The basic approach is to partition the mesh into a number of sub-domains, allocating one sub-domain to each processor in the parallel machine. The partition is calculated by passing the mesh as an undirected graph with nodes relating to the primary mesh entity (e.g. finite elements) and edges representing relationships between primary entities, into a graph partitioner. The graph partitioner partitions the graph in an attempt to minimise both load imbalance across the processor topology and the number of graph edges cut between different sub-domains, where these cut edges will infer communications in the parallel code. Other mesh entity types (e.g. nodes) are then partitioned based upon their relationship to the primary entity where, again, the minimisation of load imbalance is an important consideration. Each partitioned entity is exclusively allocated to only one sub-domain (and therefore one processor), where that processor is said to own that entity. The set of entities owned by a processor is referred to as that processor's core set. Figure 1 shows an unstructured mesh where the primary partition is based on the elements and the secondary partition is based on the nodes.

Parallel execution is achieved by performing calculations that relate to the assigning of values to a data structure representing a particular entity only on the processor that owns that entity. Interprocessor interactions are predominantly required for the use of an entity that is owned on one processor where the data being assigned relates to another entity on another processor. The set of entities that are used on a processor but owned by others are referred to as this processor's overlap set (also known as halo or ghost entities). Typically, a set of such non-owned data items will be required and can be communicated in a single set of bulk communications. In figure 2, communication sets for elements are shown where all data items to be sent to the same processor are packed into a buffer to allow a single communication and only incur a single communication startup latency.

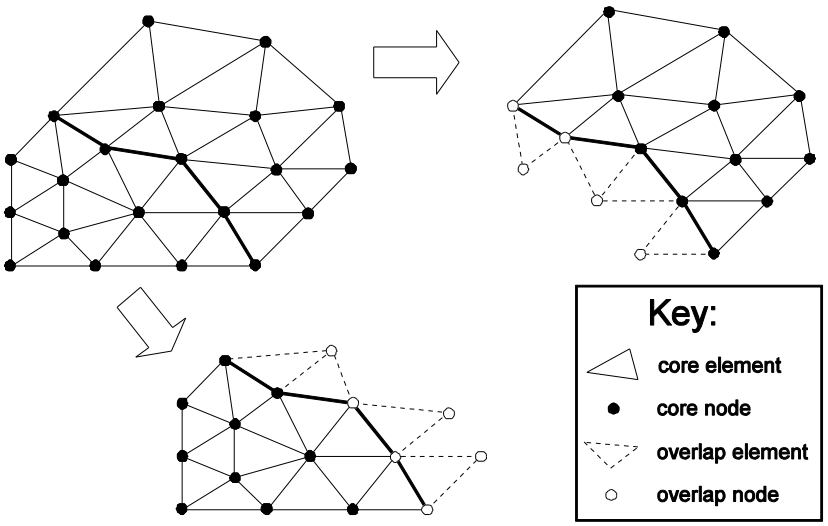
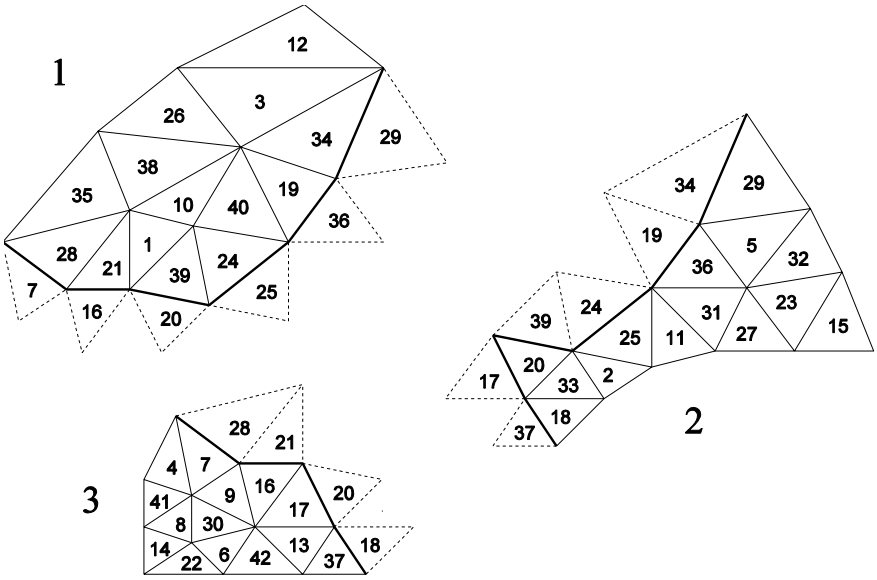


Figure 1. Data partition of elements and nodes in a simple mesh onto two processors.

A typical unstructured mesh code will contain many code sections where overlap data, and therefore communications, are required. Many communication sets will involve the same sets of entities (for example, corresponding to the overlap elements shown in figure 1) for a range of different data arrays (e.g. temperatures, velocities etc.). This allows only a small number of such sets to be required in an efficient parallel code where each set is used in numerous communications. The received overlap data can also often be used in many computations in many different code sections before an update of the data is required. Minimising the overlap update communications is obviously extremely important to the performance achieved by the parallel code. In a mesh such as that shown in figure 1, as few as four communications may be required:-

- Overlap elements required for element computations
- Overlap elements required for nodal computations
- Overlap nodes required for element computations
- Overlap nodes required for nodal computations

For simplicity in manual parallelisations, the superset of entities are often used to form a communication set e.g. all elements required for either element or nodal computations.



Topology representation in integer array ELETOP :-

Element 1 :	ELETOP(1,1) = 10	ELETOP(2,1) = 21	ELETOP(3,1) = 39
Element 2 :	ELETOP(1,2) = 25	ELETOP(2,2) = 33	
Element 3 :	ELETOP(1,3) = 12	ELETOP(2,3) = 26	ELETOP(3,3) = 34
Element 4 :	ELETOP(1,4) = 7	ELETOP(2,4) = 41	
Element 5 :	ELETOP(1,5) = 29	ELETOP(2,5) = 36	ELETOP(3,5) = 32
Element 6 :	ELETOP(1,6) = 22	ELETOP(2,6) = 42	ELETOP(3,6) = 30
Element 7 :	ELETOP(1,7) = 4	ELETOP(2,7) = 9	ELETOP(3,7) = 28

. . . . . .

Communication sets for the update of element based overlap areas:-

Processor 1	Processor 2	Processor 3
Send 21 and 28 to processor 3	Send 18 and 20 to processor 3	Receive 21 and 28 from processor 1
Receive 7 and 16 from processor 3	Receive 17 and 37 from processor 3	Send 7 and 16 to processor 1
Send 19, 24,34 and 39 to proc 2	Receive 19,24,34 and 39 from proc 1	Receive 18 and 20 from processor 2
Receive 20,25,29 and 36 from proc 2	Send 20, 25, 29 and 36 to proc 1	Send 17 and 37 to processor 2

Figure 2. Three processor decomposition with an element overlap updating communication set

A set of generic routines were developed during the manual implementation of the parallelisation technique detailed above. They take as input a simple data structure abstracted from the data structures used in the application code being parallelised. These utilities cover a range of functions including primary partition calculation, secondary partition calculation, communication set calculation, and communication of data to update the overlap area for a particular variable.

Figure 2 also shows a possible data structure for representing the unstructured mesh. Obviously, there are numerous alternative data structures, which, as discussed in the following sections, have a significant impact on the automation of the parallelisation process. For simplicity, the data structure shown in figure 2 is used in the examples in section 3.

3 AUTOMATION OF THE PARALLELISATION PROCESS

The automation of the unstructured mesh parallelisation process involves a number of stages following the same path as that devised for structured mesh code parallelisation within the CAPTools environment.

To clarify the automation algorithms described below, the example in figure 3 is used. The arrays **NTEMP**, **OTEMP**, **NUM_NEIGHBOURS** and **ELETOP** all relate to element based data. **ELETOP** represents the mesh topology, where for each element it lists the connected elements as shown in figure 2. The code example in figure 3 calculates **NTEMP** for each element to be the average of the values of **OTEMP** for each neighbouring element.

```
DO IELEM=1,NUM_ELEMENTS
    NTEMP(IELEM) = 0.0
    DO INEIG=1,NUM_NEIGHBOURS(IELEM)
            NTEMP(IELEM)=NTEMP(IELEM) + OTEMP(ELETOP(INEIG,IELEM))
    ENDDO
    NTEMP(IELEM)=NTEMP(IELEM) / NUM_NEIGHBOURS(IELEM)
ENDDO
```

Figure 3. Simple code section using data structures from figure 2.

3.1 Dependence Analysis

The dependence graph constructed by CAPTools holds dataflow information for every statement in every routine of 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. The dependence analysis exploits symbolic information and is performed interprocedurally (i.e. from one routine to another), requiring significant computational effort to produce an accurate dependence graph [10]. The significant implications of unstructured mesh codes on dependence analysis are related to the use of workspace data and complexities of algorithms used, for example, in system matrix construction [11]. The complex nature of such codes also requires some user interaction in the analysis process to provide information not explicitly stated in the application source code. Such information can relate to the nature of the input data for the code, particularly when implicit restrictions are placed on this data for correct operation [11].

3.2 Data Partition Implementation

The data partition calculation determines a set of arrays and the nature of their mapping onto the processor topology. For unstructured mesh codes, two alternative partitioning strategies are commonly applied. When a direct solver (e.g. a frontal solver) dominates execution time, a cyclic decomposition of the system matrix may be required [12]. When no such obstacle exists, a geometric decomposition of the unstructured mesh usually enables an effective parallelisation [9]. Both partitioning approaches are available within CAPTools.

For a mesh decomposition, 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 run-time when the size and construction of the mesh is known, 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 graph partitioning tool called JOSTLE [13]. This takes the hardware

processor topology and the graph based representation of the mesh topology as input and returns the owning processor list. For example, when the element-based array **NTEMP** is partitioned, JOSTLE will return the following processor-entity relationships (as shown in figure 2): **P_elem**(1)=1, **P_elem**(2)=2, **P_elem**(3)=1, **P_elem**(4)=3, … this means that element 1 is owned by processor 1, element 2 is owned by processor 2, element 3 is owned by processor 1 etc. Typically for an unstructured mesh code, there will be only a few partitions (one for element-based arrays, another for node-based arrays and so on). The partitioning algorithm in CAPTools sets the same processor ownership array to a number of different arrays so that all element based arrays will use the same processor ownership array. The determination of these sets of arrays, including interprocedural partition inheritance is calculated for an initial user selection of a single array in a user selected routine following the same algorithm as devised for structured mesh code partition calculation [7]. In figure 3, for example, arrays **NTEMP**, **OTEMP**, **NUM_NEIGHBOURS** and **ELETOP** all use the processor ownership array for elements (**P_elem**).

3.3 Masking Statements For Parallel Execution

Execution control masks are added after the data partitioning stage. 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 [7] and use the rules developed for structured mesh parallelisation. The masks for an unstructured mesh parallelisation take the form:-

```
IF (P_A(EXPR).EQ.PROCNUM) A(EXPR) = ...
```

where **P_A** is the processor list for 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, as shown in figure 4 for the example of figure 3.

The transformation of masks onto loop limits requires the mesh to be renumbered so that owned entities are packed together in all related arrays allowing the loop to only iterate over the owned set [9]. This allows the element loop in figure 4 (L_1) to iterate for locally renumbered elements from 1 to the number of locally owned elements, where no execution control mask is required. The renumbering and subsequent memory reduction phases are performed after communication generation [14].

```
L1      DO IELEM=1,NUM_ELEMENTS
            IF (P_elem(IELEM).EQ.PROCNUM) THEN
                NTEMP(IELEM) = 0.0
                DO INEIG=1,NUM_NEIGHBOURS(IELEM)
S1                  NTEMP(IELEM) = NTEMP(IELEM) + OTEMP(ELETOP(INEIG,IELEM))
                ENDDO
                NTEMP(IELEM)=NTEMP(IELEM)/NUM_NEIGHBOURS(IELEM)
            ENDIF
        ENDDO
```

Figure 4. Block execution control mask in example of figure 3.

3.4 IDENTIFICATION AND PLACEMENT OF COMMUNICATIONS

Communication requirement calculation involves comparing the location of an up-to-date value of a data item with where that data is required within the processor topology. A communication request is generated unless it is certain that the data already resides on the processor where it is needed.

For partitioned arrays, the usage location as determined by the execution control mask on the usage statement (i.e. where the data is needed) is compared with the owning processor of the array element as determined in the partition definition. This involves symbolic variable comparisons since the processor ownership array is not setup until runtime. For the example in figure 4, the used item of array **NTEMP** in statement S_1 is owned on processor **P_elem(IELEM)** and the execution control mask forces the statement to execute on processor **P_elem(IELEM)**, therefore guaranteeing that the used data is available on that processor. The usage of array **OTEMP**, however uses data available on processor **P_elem(ELETOP(INEIG,IELEM))** only, which may not be the same processor as that executing statement S_1. A communication request is therefore issued by statement S_1 for an overlap update of values stored in array **OTEMP**.

For unpartitioned data (such as workspace arrays), the execution control masks on statements related by a true (data flow) dependence are compared [7]. The execution control mask on the source (assigner) of the dependence indicates the processor on which the data is available whilst the execution control mask on the dependence sink (usage) indicates where that data is required.

In most cases, the communication required will be an update of the appropriate overlap area. This requires the generation of a communication statement but also requires the construction of a communication set to indicate the data movement patterns. To enable efficient communications to be generated, emulating the manual parallelisation process, two requests are generated for every identified communication. The technique used here is derived from the inspector-executor method of constructing a run-time graph [15,16]. One request is for the data to be communicated (executor) so that a communication statement will be generated. The other request is for the construction of a communication set of all items from that structure that need to be communicated (inspector) [17].

The inspector request serves two purposes for a communication. It must construct the communication set using one of the utilities developed in the manual parallelisation work described in section 2. Additionally, however, since this utility takes as input a standard data structure, the application code data structure (and precise use of that data structure in the application code) must be abstracted into this standard form. As a result, the inspector request generates an inspector loop to construct a graph represented by pairs of entity numbers that are related in a computation, identifying the location where a data item is needed and the location of the latest value of that data item. The execution control mask on the statement using the partitioned array data provides information as to where the data is needed, whilst the index expression of the used array (or the execution control mask of an assigning statement for unpartitioned arrays) indicates where the data exists. Consider **A** and **B** as partitioned arrays in the following example where a communication request is issued due the usage of array **B** :-

```
IF (P_A (EXP1).EQ.PROCNUM) A(EXP1)=B(EXP2) --> INSPECTOR_PAIR (EXP1,EXP2)
```

A set of these inspector pairs are calculated within inspector loops where these pairs form the edges of a graph that can then be interpreted by the utility routines.

The inspector loop will then involve copies of all statements from the original code required to

calculate the consecutive inspector pairs, including surrounding loops and controlling conditional statements. These statements are identified using the dependencies in the original source code for the inspector pair with a set of rules to determine when a statement is to be included in the inspector loop to enhance the quality of the resultant code. The inspector loop for the communication of **OTEMP** in statement S_1 in figure 4 is shown in figure 5.

```
DO IELEM=1,NUM_ELEMENTS
        IF (P_elem(IELEM).EQ.PROCNUM) THEN
                DO INEIG=1,NUM_NEIGHBOURS(IELEM)
                        INSPECTOR_PAIR (IELEM,ELEPTS(INEIG,IELEM))
                ENDDO
        ENDIF
ENDDO
```

Figure 5 Inspector loop for the communication of **OTEMP** in S_1 from figure 4.

The communication requests are migrated upwards in the routine control flow graph to as early a point in the code as is legal to improve the quality of the generated code [7]. This migration moves communication requests out of loops and through routine boundaries to attempt to minimise the frequency of communication calls allowing bulk communications with few communication startup latencies. Migration also improves the potential for communication merger since many communications may migrate to the same point in the source code. Migration is blocked by assignments to any variables involved in the communication, either at the assigning statement itself, or at a loop head (e.g. immediately after a DO statement) that contains the assigning statement. The identification of barriers to migration for the inspector loop request first requires the determination of all statements to be included in the inspector loop. Any statement required to calculate the inspector pair (or assignments of data used in any statement to be included in the inspector loop) that is not itself in the inspector loop then forms a barrier to migration. The barriers to migration for the communication request are the same as those of the related inspector request, but the statements that assign the array to be communicated now also represent a barrier. This ensures that the inspector loop and therefore the communication set will precede the associated communication call. A vital requirement of this process is that migration must be maximised whilst inspector loop code volume is minimised. A number of algorithms and heuristic rules that attempt to achieve this for real application codes have been developed in this work [18].

3.5 Merger Of Inspector Loops And Communications

Once all requests for communications and inspector loops have been identified and migrated, they can be merged to significantly reduce the number of communications and the number of inspector loops. The overriding aim of this process is to reduce the number of communications in the code, however, additional aims of reducing inspector loop execution time, reducing communication set memory requirements and reducing code volume are also considered. Since communications can only be merged when the associated inspector loops are either merged or one is proven to produce a graph that is a subset of another, the comparison of inspector loops is a pre-requisite. From the experience of parallelising real codes manually, it is clear that coding styles and minor algorithmic variations force the test for subset inspector loops to be non-trivial [18]. The process of merging compares all inspector loops throughout the code with the tests performed in the three stages listed below:

1. First, attempt to delete subset inspectors when the related communication requests have themselves been placed at the same point in the code and related to the communication of

the same subset of the same array. Deletion of an inspector loop also allows the deletion of the associated communication since all data items moved in the deleted set will be moved by the other communication.

2. Once the first stage is complete, the second phase reduces the number of communications and their volume by creating 'unioned' inspector loops, where all the related communications involve the same subset of the same array at the same point in the code. The unioned inspector loop is merely a number of inspector loops generated consecutively in the code where the edges added by them all are used to construct a single communication set. The related communications can then be merged to be a single communication where each data item is transferred only once instead of the duplication that was almost certainly present amongst the inspector loops in the union.

3. Finally, all inspector loops are checked, regardless of the details of the related communications, and any subsets are then deleted. Although no communications are removed in this phase, the other considerations of inspector loop execution time overhead, memory requirements and code volume are reduced. This process is performed after the union of inspector loops. This is because when an inspector loop is used by many communications, the possibility of it legally being included in a further union is minimal since all related communication must have already been considered.

3.6 Interprocedural Communication Generation Example

To demonstrate the complexity of operation required for even simple real codes, figure 6 shows an extract from a computational mechanics code. The arrays **U**, **T**, **GPTIEL** and **ELETOP** are element based arrays, where **ELETOP** stores the element mesh topology and **GPTIEL** the number of adjacent elements. A workspace array **WORK** is assigned in routine **SETUP** for each element in turn, with these values used in routine **UPDATE** during the same iteration of the element loop (L_1). The number of element types is stored in **NETYPE**, where the number of elements of each type is stored in the **NTYPE** array so that the input data may, for example, contain a number of triangular elements followed by a number of quadrilateral elements etc. This example illustrates the process of inspector loop identification, migration, merger and code generation. The inspector loop will include code copied from loops and assignments in several routines.

```
        IEND = 0
        DO I = 1,NETYPE
            ISTART = IEND + 1
            IEND = IEND + NTYPE(I)
L1          DO IE = ISTART , IEND
                NPTS = GPTIEL(I)
                . . .
S1              CALL SETUP(NPTS,IE,ELETOP,WORK)
                . . .
                CALL UPDATE(NPTS,IE,U,T,WORK)
                . . .
            ENDDO
        ENDDO
        . . .
```

```
SUBROUTINE SETUP(NPTS,IE,ELETOP,WORK)
. . .
WORK(1) = ELETOP(NPTS,IE)
DO K = 2,NPTS
        WORK(K) = ELETOP(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 6 Extract from a computational mechanics code.

The execution control masks are all set in the caller routine based on variable **IE** and the element based processor ownership array. The calls to routines **SETUP** and **UPDATE** are masked, 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 at S_1 in figure 6 due to the assignment of the index array **WORK** in the call to routine **SETUP**. Allowing loops not common to the requesting statement [18] enables the assignment to **WORK** to be included in the inspector loop and thus not be a barrier to communication migration. Migration of the inspector loop is then only blocked by the assignments to arrays **ELETOP**, **GPTIEL** and **NTYPE**, along with scalar **NETYPE**, allowing both requests to migrate out of all of the loops shown in figure 6. Figure 7a shows the inspector loop generated.

```
IEND = 0
DO I = 1,NETYPE
        ISTART = IEND + 1                          S1
        IEND = IEND + NTYPE(I)                     S2
        DO IE = ISTART , IEND                      S3
                NPTS = GPTIEL(I)                   S4
                CAP_WORK(1) = ELETOP(NPTS,IE)      S5
                DO K = 2,NPTS
                   CAP_WORK(K) = ELETOP(K-1,IE)    S6
                ENDDO
                DO K = 1,NPTS
                   CALL  ADDEDGE (IE,CAP_WORK(K))
                ENDDO
        ENDDO
ENDDO
CALL CAP_OVERLAP(GRAPH,ID1)
```

Figure 7 (a) Inspector loop from Figure 6.

```
DO IT=1,NTIME
        CALL CAP_SWAPOVER(U,ID1)
        CALL CAP_SWAPOVER(V,ID1)
        CALL CAP_SWAPOVER(W,ID1)
        CALL CAP_SWAPOVER(P,ID1)
        CALL CAP_SWAPOVER(T,ID1)
        . . .
        CALL FLOW(U, V, W,  P, T,…)
ENDDO
```

(b) Usage of communication set calculated from inspector loop after merger.

The call to **ADDEDGE** adds an edge to a graph of edge pairs representing the interaction between usage location and data location. Although this inspector loop appears fairly complex, 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 in figure 6 (plus any other loops in the main program such as the time step loop). This failure to migrate would obviously require very frequent calculation of small communication sets and numerous small communications, causing significant overheads that detract from the efficiency of the parallel execution. Figure 7b shows a further extract from generated parallel code that follows the code section in figure 7a, which uses the inspector loop shown there. The call to routine **FLOW** is within the time step loop and contains the code that requires communications that use the inspector loop in figure 7a. The call to **CAP_OVERLAP** generates the communication set from the inspector loop generated graph that make up an overlap of data needed by the processor; this list is labelled as **ID1.** In this case, the overlap list is generated and executed only once as a result of the legal migration of all inspector loop requests and their merger outside the time step loop. CAPTools was firstly able to determine that the contents of the list (i.e. the elements) did not alter from one time step to the next, allowing the inspector requests from within the call to routine **FLOW** at statement S_6 to exit the time step loop. Secondly, CAPTools was able to merge a number of inspector loop requests into a single inspector loop and a single call to **CAP_OVERLAP**. The actual communications are performed for each remaining communication request in statements S_1-S_5 using a call to the library routine **CAP_SWAPOVER** which takes as input the communication set identification number generated in the related call to **CAP_OVERLAP** to access the internally stored communication set. In the above example, these communications are barriered by the time step loop since the quantities of velocity (**U**,**V**,**W**), pressure (**P**) , temperature (**T**), etc. are time dependent with their values changing from one time step to the next.

4 RESULTS FOR THE PARALLELISATION OF A FINITE ELEMENT BASED STRESS CODE

As an example of the parallelisation process, a dusty-deck unstructured mesh application code that was written without any consideration for parallelisation, was parallelised within the CAPTools environment. The code uses standard finite element (FE) techniques to solve for displacement, stress and strain, in either a two or three-dimensional mesh. The FE simulations are performed over a number of time steps. Typically, the algorithm constructs stiffness matrices for each finite element in turn and incorporates these into a full system matrix and boundary condition vector. The resulting linear system is solved using a Conjugate Gradient scheme to determine displacements. The stress and strain values are then derived from the displacements. The system matrix construction and the solution of the linear system dominate the overall runtime.

The dependence analysis of this code is a complex process with some information required from the user due to the restrictions on input needed to allow correct execution [11]. The data partition was specified within the CAPTools environment by firstly selecting an element based array in any routine, indicating the index of that array that is to be partitioned and selecting the *UNSTRUCTURED* partitioning option. All element based arrays are then automatically detected and marked to be partitioned in accordance with the initial user selection. The nodal based arrays were then partitioned, also based upon a single array selection by the user.

The need for code transformations in the system matrix setup routine was identified after an initial pass of the parallelisation. Each element stiffness matrix (**ELEMAT** in figure 8a) is owned by the element owning processor array as defined by the data partition. The system matrix however, involves rows of data whose location is based on the nodal-based partition ownership array. As a result, a broadcast of every element stiffness matrix was generated to ensure data was available for the appropriate rows of the system matrix (since the nodes of an element can be shared amongst several owning processors). To overcome this, a series of CAPTools transformations were applied to enable the communication **ELEMAT** to be outside the element based loop, thus requiring only one communication instead of large number of small communications. Firstly, a number of variables (including **ELEMAT)** were expanded to have an extra dimension representing the associated finite element. For this to be legal, there must be no use of data calculated in an iteration of the loop (i.e. for a finite element) that is used in the calculation of a later iteration of that loop. The sophisticated CAPTools dependence analysis correctly proved that every element stiffness matrix calculated is independent of all others [11], allowing automatic application of the transformation. The second transformation was then applied, splitting the element loop so that all element stiffness matrices are calculated and stored in the now expanded **ELEMAT** array before their inclusion into the full system matrix **SYSTEM** in a separate element based loop (figure 8b). This allows the communication of element matrix data to the relevant system matrix row owning processors to be performed as an overlap update between the two copies of the element loop (figure 8b). To reduce storage requirements, this process can be performed in sections so that sets of element stiffness matrices can be calculated, communicated and incorporated into the element stiffness matrix one after another.

```
DO IE=1,TOTELE
   DO IGAUSS=1,NGAUSS
      CALL STIFFNESS(ELEMAT,IE,IGAUSS,…)
   ENDDO
   CALL UPSYSTEM(ELEMAT, SYSTEM,IE,…)
ENDDO
```

```
DO IE=1,TOTELE
   DO IGAUSS=1,NGAUSS
      CALL STIFFNESS(ELEMAT(IE),IE,IGAUSS,…)
   ENDDO
ENDDO
CALL CAP_SWAPOVER(ELEMAT,...)
DO IE=1,TOTELE
```

```
        CALL UPSYSTEM(ELEMAT(IE),SYSTEM,IE,…)
      ENDDO
```

Figure 8: (a) System matrix code (b) Transformed code.

The system matrix, as accessed in the conjugate gradient solver routine, involves all degrees of freedom for every node (i.e. 2 or 3 rows per mesh node depending in the dimensionality of the mesh). The execution control masks therefore use divisor expressions of **DIMS** to identify the relevant node and thus the owning processor for an entry in the system matrix (Figure 9a) where **DIMS** is the dimensionality of the mesh. The division operation in this mask represents a significant overhead at runtime. To overcome this, such loops were partially unrolled and the division therefore removed (Figure 9b). The remainder of the code, including the conjugate gradient solver, was parallelised automatically requiring no user involvement.

```
DO I=1,TOTGPT*DIMS
   IF (P_NODE((I-1)/DIMS+1).EQ.PROCNUM) THEN
```

```
DO NODE=1,TOTGPT
   IF (P_NODE(NODE).EQ.PROCNUM) THEN
      DO I=(NODE-1)*DIMS+1,NODE*DIMS
```

Figure 9: (a) Mask with divisor (b) Transformed code

Speedup results for the 3,000 element simulation taken from the NAFEMS benchmark set [19] shown in Figure 10 are presented in Figure 11a for the initial CAPTools generated code, manually optimised code and overlapped communication version using an i860 based Transtech Paramid system. Figure 10a shows the decomposition of the mesh onto 16 processors, as calculated by the automatically inserted call to JOSTLE, with the Von-Mises stress solution calculated using the serial or any parallel execution shown in figure 10b. The manual optimisations included adjusting one communication in the system matrix setup and the re-arrangement of the conjugate gradient solver implementation to reduce the number of individual global operations involved since these incur a high communication startup latency cost [9]. The final set of results show the effect of performing the communication into overlap areas at the same time as the calculations that do not access the communicated data. Communication synchronisation is enforced before the communicated data is used in the communication/computation overlapping version to ensure correct operation. Figure 11b shows the performance of the optimised code executed on a CRAY-T3D and an IBM SP2. These results show that, with minor manual modification, the generated code provides high efficiency even on this small mesh. The loss of efficiency is largely due to essential communications as indicated by the overlapping communication results in figure 11a. The lack of scalability onto 16 processors is due to the small amount of computation that is independent of the communication, preventing the complete overlap of the communication time.

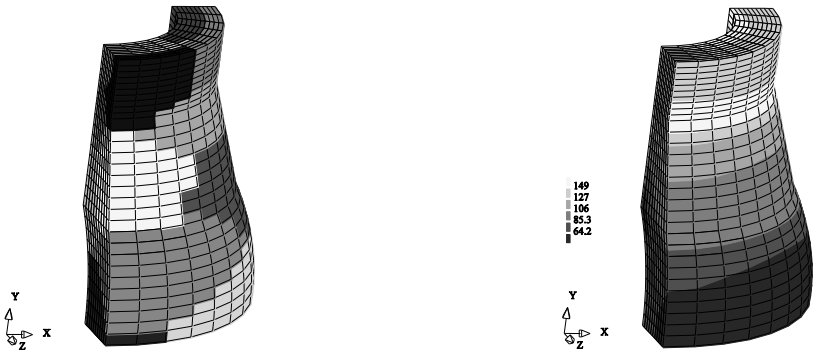


Figure 10 (a) Partition for 16 processors (b) Von-Mises solution

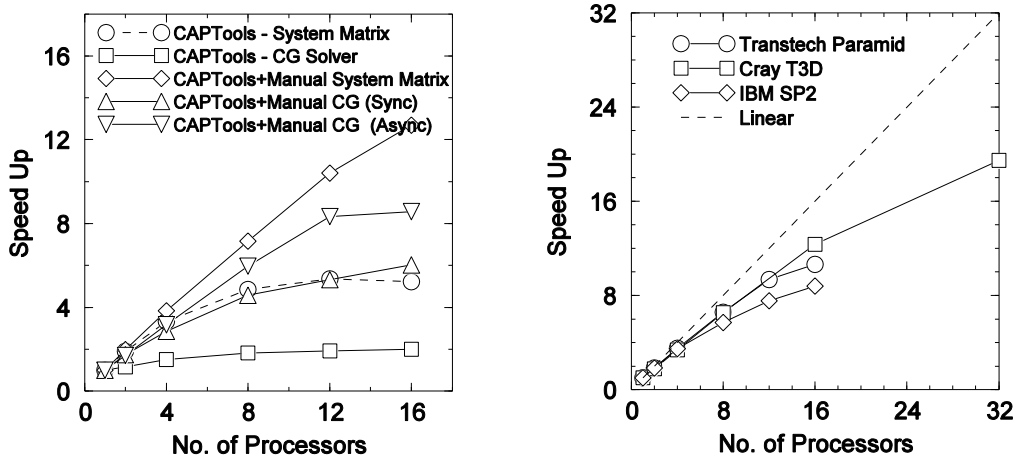


Figure 11: (a) Results for Transtech Paramid, (b) Results for T3D, SP2 and Paramid

The basic parallelisation process of this FE code within the CAPTools environment can be completed in a few hours. To generate efficient code, however, required a number of passes through the partitioning and code generation phases along with some effort for the manual optimisations mentioned earlier. The entire parallelisation process was nonetheless completed in a few days.

5 RESULTS FOR THE PARALLELISATION OF A FLOW AND STRESS CODE

The Unstructured Incompressible Flow and Stress (UIFS) code is written in Fortran and contains 170 subroutines and approximately 19,000 lines of source. It uses a control volume discretisation of the domain to solve for the incompressible Navier-Stokes equations on an unstructured mesh. This code employs the semi-implicit SIMPLE solution procedure and solves coupled equations for heat, solidification, displacement, stress, strain and other physical quantities [20] shown in the flowchart (figure 12a). The flow components are solved using the Jacobi and SOR iterative schemes, the stress computations are solved using the conjugate gradient technique with diagonal preconditioning. The UIFS code has also been manually parallelised [9] using the same SPMD technique that has been embedded within the CAPTools

environment. This allows a direct comparison of code appearance and performance between fully manually parallelised code and the automatically generated version.

The dependence analysis of this code was relatively straightforward as compared to that for the FE code, requiring minimal user interaction. As with the FE code, the partition specification required two user selections, one of an element based array and another of a nodal based array.

Table 1 compares the manual parallelisation process with the CAPTools parallelisation process in a range of key areas.

Manual Parallelisation	CAPTools Parallelisation
Time required ~6 months	Time required ~1 week
Correct for all tested cases	Correct as serial (except errors introduced in manual optimisation)
5 different communication sets	31 different communication sets
Communication of redundant data :- - solid cells when only liquid required - liquid cells when only solid required - extra nodes in overlaps not needed in actual data usage (supersets)	No communication of redundant data
No inspectors in time step loop - ignore possibility of moving meshes	10 inspector loops inside time step loop - liquid/solid dependent inspector loops - moving mesh based inspector loops
70 overlap area updates	68 overlap area updates
Total exploitation of element and node based parallelism	Total exploitation of element and node based Parallelism

Table 1 Comparison of manual parallelisation and CAPTools parallelisation

Examining the code generated by CAPTools, it was noted that inspector loops were placed appropriately for different calculation modules. For example, inspector loops were generated specifically for the flow computations and another set generated specifically for the stress computations. This distinction was correct since the overlap data requirements for the two computations were not the same. In general, computations in the flow module require just those values relating to liquid cells/nodes in the overlap areas, and for computation in the solid mechanics modules, only solid cells/nodes are required in the overlap areas.

The time taken to parallelise this code manually was about six months of effort from a fairly experienced code paralleliser, whereas the time taken to parallelise the same code using CAPTools was less than one week for an experienced CAPTools user. The results for the parallel execution of CAPTools generated code for a thin plate problem using 60,000 elements are shown in figure 12b. Although, the speed up figures are not as high as those obtained for a fully optimised manual parallelisation [9] they do follow similar trends. The CAPTools generated code has a similar appearance to the manually generated parallel code because the strategies used by CAPTools were extracted from the best manual parallelisations. As a result, the generated code can be optimised to achieve the same performance. A significant difference between the two parallel versions is the mesh renumbering and subsequent memory reduction in the manual version. The automated renumbering algorithms and related code transformations are discussed in [14], but have not been used here. The renumbering also has very beneficial effects on parallel performance

since cache usage is improved and execution control mask evaluation overheads are greatly reduced.

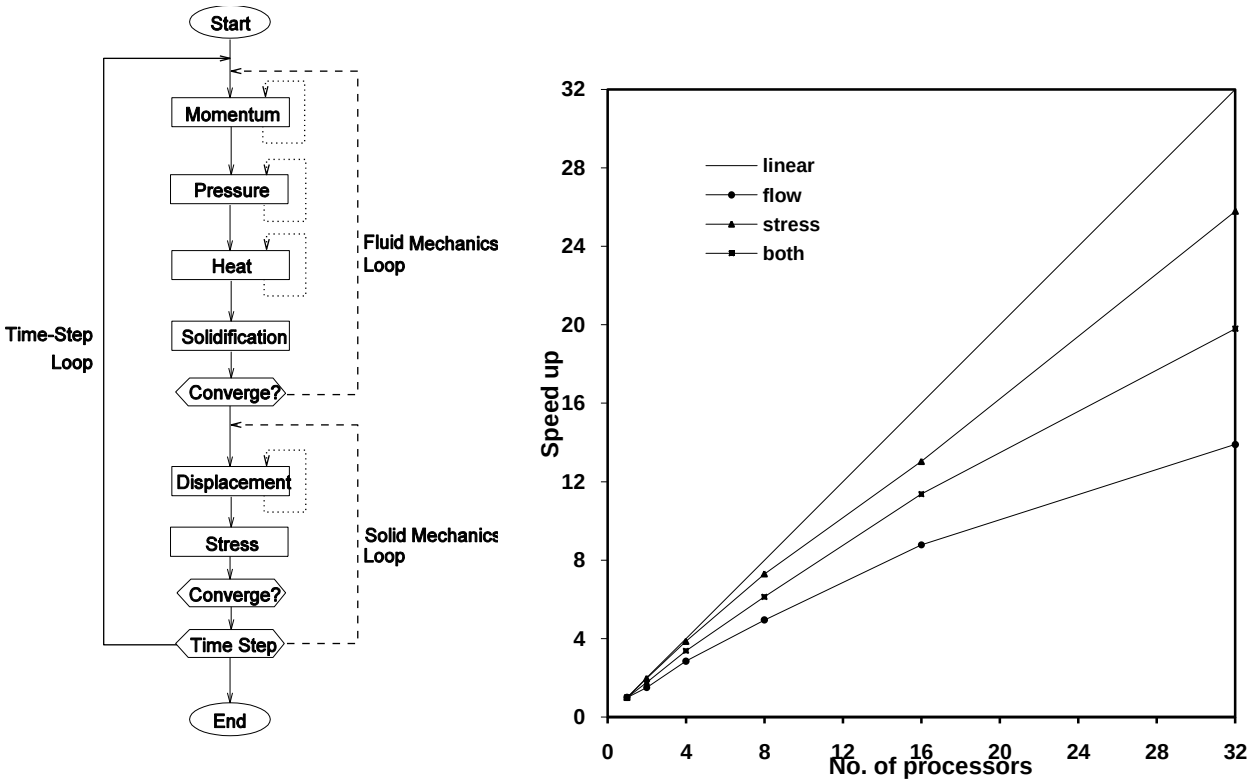


Figure 12: (a) Flow graph of flow/stress code (b) Performance of CAPTools generated code (60000 element problem) on a Cray-T3E.

6 RELATED WORK

The parallelisation of irregular application codes, including unstructured mesh codes, has also been attempted within HPF compilers [21,22,23]. As with regular HPF, the specification of the partitioning strategy is required from the user as part of the starting point source code. The inspector/executor technique is typically used to construct communication sets and perform inter-processor communications, but also to alter the statements that use non-owned data to access workspace. Many of the implementations of irregular HPF involve only intra-procedural algorithms for inspector and executor determination and placement, subsequently suffering significant communication duplication at runtime. Some effort into communication set re-use has been made, but the lack of interprocedural operation is a major disadvantage when considering real-world application codes. Agrawal and Saltz [24] address interprocedural inspector and executor movement using interprocedural partial redundancy elimination (IPRE) as a basis for the determination of, for example, loop invariance of an inspector loop. This can allow "hoisting" of inspector loops out of surrounding loops and subsequently into caller routines, also performing some inspector loop merging after this migration. Areas not addressed include the interprocedural construction of inspector loops and symbolic comparison algorithms that have proved essential in this work when addressing real-world application codes [18].

7 CONCLUSIONS

In this paper, the general parallelisation strategy based on a decomposition of the mesh for unstructured mesh-based codes has been presented. The strategies embedded within the parallelisation tools, CAPTools, have evolved through experience in manual parallelisations. The use of CAPTools instead of a manual parallelisation significantly reduces the time and effort required to parallelise such codes, typically from months to hours/days. In addition, the SPMD code generated by CAPTools can be optimised since the code is very similar to the original serial code and therefore easy to understand. Although the code generated is portable and exhibits a high degree of parallel efficiency, it can also be optimised further for a particular architecture.

ACKNOWLEDGEMENTS

The authors wish to thank Chris Bailey (University of Greenwich) and Peter Chow (Fujitsu European Centre for Information Technology) for the use of their application codes in this work.

REFERENCES

1 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.

2 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.

3 S.P.Amarasinghe and M.S.Lam, Communication Optimisation and Code Generation for Distributed Memory Machines. pp 126-138 in ACM Conference On Programming Languages Design And Implementation, 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 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.

6 Z.Bozkus, A.Choudhary, G.Fox, T.Haupt, S.Ranka and M-Y.Wu. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. Journal of Parallel and Distributed Computing, 21(1), pp 15-26, 1994.

7 S.P.Johnson, C.S.Ierotheou, M.Cross. Automatic Parallel Code Generation for Message Passing on Distributed Memory Systems. Parallel Computing 22(2), pp 227-258, 1996.

8 E.W.Evans, S.P.Johnson, P.F.Leggett, M.Cross. Automatic Generation of Multi-Dimensionally Partitioned Parallel CFD Code in a Parallelisation Tool. Proceedings of Parallel CFD 97', Manchester, England, pp 531-538 1997.

9 K.McManus PhD Thesis: A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures, Univ. Greenwich, 1996.

10 S.P.Johnson, M.Cross, M.G.Everett. Exploitation of Symbolic Information in Interprocedural Dependence Analysis. Parallel Computing, 22, pp 197-226, 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 S.P.Johnson, F.Ali, and M.Cross, Parallelising of The FAMCALC FEA Code. University Of Greenwich Technical Report, 1992.
13 C.H. Walshaw, M. Cross and M.G. Everett. A Localised Algorithm for Optimizing Unstructured Meshes. International Journal of Supercomputing Applications, 9(4),1995.
14 S.P.Johnson, V.Aravinthan, K.McManus and M.Cross, Techniques and Tools for Effective Implementation of Memory Reduction and Dynamic Load Balancing in Parallel Unstructured Mesh Software. University of Greenwich Technical Report, PPRG-98-006, 1998.
15 J.Saltz, R.Mirchandaney, K.Crowley. Run-Time parallelisation and scheduling of loops. IEEE Transactions on computers, 40, 4, pp 603-612, May 1991.
16 R.V. Hanxleden, K. Kennedy and J. Saltz. Value Based Distributions in Fortran D: A Preliminary Report. *CRPC-TR93365-S*, Rice University, December 1993.
17 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.
18 S.P.Johnson, C.S.Ierotheou and M.Cross, Inspector Loop Determination to Reduce Communication Overheads in Unstructured Mesh Code Parallelisation. University Of Greenwich Technical Report PPRG-98-003, 1998.
19 NAFEMS - Background to benchmarks, Editors G.A.O. Davies, R.T. Fenner and R.W.Lewis, Nafems, 1993.
20 C.Bailey, P.Chow, M.Cross, Y.Fryer and K.Pericleous, Multiphysics Modelling of the Metals Casting Process. Proc. Royal Society London A, vol 452, pp 459-486, 1996.
21 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, pp 152-168, 1993.
22 A.Muller and R.Ruhl. Extending High Performance Fortran for the Support of Unstructured Computations. CSCS TR-94-08. CH-6928, Manno, Switzerland 1994.
23 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.
24 G.Agrawal and J.Saltz. Interprocedural Compilation of Irregular Applications for Distributed Memory Machines. in Proceedings of Supercomputing 95', San Diego, California, 1995.