Accurate Dependence Graph Construction For Finite Element Software[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, UK.

Technical Report PPRG-98-001

ABSTRACT

The effective automatic parallelisation of real application codes requires a very accurate dependence analysis. A particular example of this is Finite Element Analysis (FEA) codes where parallelism in all phases, including matrix setup, must be exploited. This paper highlights a set of techniques that, together with some essential user interaction, can lead to sufficiently accurate dependence analysis. We also show typical cases where direct user intervention is required to overcome deficiencies in the serial code. These deficiencies cause the serial code to lack robustness and although they may present no problem to a correct serial execution, they can have significant implications in the generation of equivalent parallel code. Finally, results for a real FEA code are presented to show that with the necessary minimal user interaction, an effective parallalelisation can be achieved.

1. INTRODUCTION

One of the keys to the automatic parallelisation of serial software is the construction of an accurate representation of variable interactions within the software. The accuracy of this representation has fundamental influences in many areas of parallelisation such as parallelism detection, communication requirement and communication placement. As a result, the heart of most parallelising compilers and tools is the dependence analysis that builds a dependence graph holding the interaction information, typically, on a statement by statement basis. The overriding constraint over such analysis, however, is the definite enforcement of all possible dependencies, forcing the nature of the analysis to be conservative i.e. set a dependence unless it can be proven non-existent. This has lead to numerous research programmes and resultant techniques aimed at proving the non-existence of dependencies, particularly between array references, where the investigation of solutions to sets of linear integer equations is typically used as a dependence test [1,2,3,4,5,6,7,8,9]. These tests range from the Banerjee inequality [2] to integer linear programming techniques such as the Omega test [8] and extensions such as the Symbolic Omega test [10].

The complexities of accurate dependence analysis have led to alternative approaches to parallelisation. The most common example of this is High Performance Fortran (HPF) [11] and its variants, where the user is required to place directives into the source code to explicitly specify, in particular, loop level parallelism. Similarly, many shared memory

[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

compilers rely heavily on user directives if efficient parallel code is to be produced. The dependence analysis typically employed in these cases has, at best, a fairly limited interprocedural capability (partly because separate compilation of modules is desirable for compilers) as well as restricted symbolic processing. This restricted analysis is largely enforced due to the commercial requirement of fast compilation for its acceptance amongst the user community. Unfortunately, the user's role in the parallelisation of placing directives in the source code shifts the burden of parallelism detection (and, implicitly, the related dependence analysis) onto the user. The expertise required to determine parallelism in a code section containing complex code and many calls to other routines is significant and the risk of incorrect assumptions being made by the user is invariably high. As a result, dependence analysis tools that can detect parallelism and therefore be used to place directives in source code are needed, so the dependence analysis problem is removed from the scope of the compilers.

In previous work [12], we have developed dependence analysis techniques aimed at real world application codes to enable parallelisation within an interactive environment, the Computer Aided Parallelisation Tools [13]. This analysis must be accurate in order to make the production of effective parallel code feasible without unnecessarily excessive user interaction. In this paper, we further extend the dependence analysis techniques to accurately build a representation of more complex application codes, such as archetypal Finite Element (FE) analysis codes, allowing them to be parallelised effectively.

2. VALUE BASED DEPENDENCE ANALYSIS AND USER INTERACTION.

The basic reference-reference dependence test, as mentioned in section 1, is only part of the range of techniques required for accurate dependence graph construction. These tests examine memory location accesses only and, in the case of data flow (true) dependencies, do not consider whether data can actually flow from source statement to sink statement without being overwritten in intermediate statements. The setting of such "false dependencies" prohibits effective parallelisation of almost all real world codes since these dependencies will often serialise loops and, in distributed memory parallelisations, cause the generation of large numbers of unnecessary communications. Common occurrences in real codes require such techniques to avoid the serialisation of loops, as shown in Figure 1.

```
L1      DO J=1,NJ
                DO I=1,NI
S1                      WORK(I) = . . .
                ENDDO
                . . .
                DO I=1,NI
S2                      . . . = WORK(I)
                ENDDO
        ENDDO
```

Figure 1 Typical example of the use of workspace arrays.

The case shown in Figure 1 is typical of the use of workspace arrays frequently found in real codes where the data used by array **WORK** in S_2 is defined in the same iteration of loop L_1 at S_1. Basic dependence analysis, however, must assume a dependence from statement S_1 to the usage in S_2 of array **WORK** from an iteration of L_1 to later iterations of loop L_1, serialising loop L_1.

To handle cases such as that in Figure 1, intermediate assignments of arrays must be considered (i.e. assignments of the carrier variable between the source and sink of a true dependence). If it can be proved that all data could be passed from assignment to usage of a dependence is overwritten by a set of intermediate assignments, then the dependence can be eliminated. The determination of the existence of a dependence therefore becomes the solution of a logical combination of integer equalities and inequalities representing source, sink and intermediate array references. This can be performed following a basic analysis as a 'pruning' phase (covering set calculation [12] and false dependence elimination [14]) and has also been implemented in initial analysis incrementally adjusting the constraints on currently unsatisfied usage ranges (as in lazy dependence analysis [15]).

All aspects of analysis must handle symbolic variable tracing through routine boundaries with all techniques applicable to interprocedural operation [12,16,17,18]. The dependence testing of array references must also operate interprocedurally to accurately reflect the actions of called routines [12,19,20,21]. Interprocedural dimension mapping of array references is widely used in real codes and therefore must also be handled. Analysis accuracy can be enhanced if dimensionality can be maximised, however, linearisation can be used to allow a slightly weakened analysis [12,20] where symbolic techniques for non-linear inequalities are required.

Another vital factor in accurate analysis of real codes is the ability to incorporate information provided by the code paralleliser [12,13]. This information usually consists of constraints on variable values that are well known and understood by the user, but are not explicitly stated within the code, forcing the conservative dependence analysis to take worst case assumptions. A frequent example of this type of interaction is the nature of variables that represent a computational domain e.g. the number of cells is significantly greater than zero. To maximise the effectiveness of such interaction, the analysis should also be able to ask questions in order to elicit important information from the code paralleliser that enables conservatively assumed dependencies to be eliminated [13]. User inspection of serial loops, for example, can enable the identification of specific information required for the removal of the serialising dependencies. Such a facility also provides explanation to the code paralleliser of why a code section is serial, indicating in some cases that the algorithm implemented is itself serial. Figure 2 shows a simple example, taken from a finite element code, of information required from the user to enable an accurate analysis. The node based array **A** is set up from 1 to the number of nodes in the mesh, **TOTNOD**, and then uses that array in calculating an element based array **B** by taking the values for each node used in that element (represented by the topology array **ELETOP**).

```
        DO INODE=1,TOTNOD
S1              A(INODE) = . . . .
        ENDDO
        DO IELEM=1,TOTELE
                DO IELNOD=1,NODES(IELEM)
S2                      B(IELEM)=B(IELEM)+A(ELETOP(IELNOD,IELEM))
                ENDDO
        ENDDO
```

Figure 2 Simple example of implicit information that is known to the user.

A dependence on statements prior to S_1 may be set for the usage of array **A** in statement S_2. For all legal executions of the finite element code, however, all values of array **A** used in S_2 are assigned in S_1 since the **ELETOP** array contains node numbers which must always range between 1 and **TOTNOD**. The array **ELETOP** is read at runtime from an input file with no check within the code for legal values since the input is assumed to be correct. The only way a dependence analysis can obtain this information is through user interaction, where the user may either volunteer this information, as it is well known to them, or provide it as a response to a question posed by the analysis system. The information required in this case is simple :-

1 <= **ELETOP** <= **TOTNOD**

indicating that all values in the **ELETOP** array are within that range. The assignment of array **A** in S_1 can then be proven to cover the usage in S_2, allowing dependencies on previous assignments of **A** to be removed.

The vital question facing a code paralleliser when examining an analysis of an application code is "why has a dependence been set ?". Obviously, there are a range of answers to this question as discussed in Table 1 below:-

Reason for dependence	Action
Set because the dependence actually exists as part of the implemented algorithm.	None – transformations or algorithm replacement can be used if problems in parallelisation are caused by this dependence.
Set because implicit restrictions on the application input data that would remove the dependence are not set in analysis.	Provide information to analysis system in form of constraints etc.
Set due to error in serial code.	Correct serial code and re-analyse
Set due to restrictions on dependence testing algorithm due to analysis runtime.	Relax limits in analysis and allow greater runtime.
Set due to complex algebra and/or indirections in array indices or control statements.	Manual deletion of false dependencies.

Table 1. Reasons for a dependence being set.

In the following sections, we shall demonstrate the need for all the analysis techniques mentioned above in the context of an archetypal finite element code.

3. FINITE ELEMENT CODE STRUCTURE

A common finite element implementation technique involves the construction of a system matrix followed by the solution of the linear system $\mathbf{A}\underline{x} = \underline{b}$ where **A** is the system matrix representing interactions in the mesh and $\underline{b}$ is the vector representing loadings on the mesh entities. The algorithm is often implemented within a time-dependent framework where the solution in each time step is used in the system matrix construction in the following time step. Typically, the two phases of a finite element code that dominate the runtime are the system matrix construction and the subsequent solution of the matrix system. The solution to the matrix system can either be performed using implicit (direct) solvers or by iterative solvers, where the linear equation solver used may be fundamental to the choice of parallelisation strategy [22,23]. In terms of dependence analysis, however, these solvers do not typically represent any extra problems as compared to structured mesh based codes addressed in previous work [12]. Our experience has been that the system matrix construction phase, however, requires a level of analysis beyond that required by many structured mesh codes, although similar problems may exist in some structured mesh codes.

The building of the system matrix is typically implemented by the construction of matrices for each element in turn and adding contributions for the element matrix into the full system matrix. This process is complicated by the range of element types available in the mesh construction such as triangular, quadrilateral, cuboidal elements etc., the varieties of analysis types possible such as plain strain, plain stress, axis-symmetric analysis etc. and even the dimensionality of the domain. In addition, the coding styles used may present obstacles to accurate dependence analysis although they do not cause any problems in serial code execution. As we shall show later, assumptions about the nature of code input data during code authoring are often made since the input will usually involve some form of mesh generation software providing only certain forms of input. Such assumptions cannot be exploited by an analysis system without explicit user assertions.

One of the keys to accurate analysis of such a FE algorithm is the proving of independence between consecutive element stiffness matrices. Consider the code section in Figure 3.

```
        DO ITIME=1,NUM_TIMESTEPS
$L_1$            DO IE=1,NUM_ELEMENTS
$S_1$                 Calculate element stiffness matrix **E**
$S_2$                 Add contributions from **E** into system matrix **A**
                 ENDDO
$L_2$            DO IE=1,NUM_ELEMENTS
                      Calculate element loading matrix **C**
                      Add contributions from **C** into system matrix **A** and loading vector $\underline{\mathbf{b}}$
                 ENDDO
                 Solve linear system $\mathbf{A}\underline{\mathbf{x}}=\underline{\mathbf{b}}$ for displacements vector $\underline{\mathbf{x}}$
        ENDDO
```

Figure 3. Basic code structure of element matrix and system matrix construction

An obvious and profitable source of parallelism is between iterations of loop L_1 (as well as between iterations of the similar loop L_2). The array **E** holding the element stiffness matrix is implemented as workspace and is re-used for every element. To allow such parallelism, all the data in the usage of matrix **E** in statement S_2 must have been assigned earlier in the same iteration of L_1. The implications of failing to prove that each element stiffness matrix is calculated within the same iteration of the element loop are very significant. Firstly, dependencies will be set on the calculation of element stiffness matrices in earlier iterations of the element loop. Additionally, dependencies may also be set between the assignment and usage of element stiffness matrices in different iterations of other loops surrounding the system matrix construction code section, for example, from one time step to another. In a shared memory parallelisation, the dependence on earlier iterations of the element loop serialises that loop. In a distributed memory message passing parallelisation using mesh decomposition, each element will be owned by a specific processor and the corresponding element stiffness matrix is calculated only on that processor. Any dependencies on previous iterations of the element loop (and earlier time steps etc.) therefore refer to element stiffness matrices that will often have been calculated on a different processor. This requires the communication (often a broadcast) of each element stiffness matrix to every processor to ensure correct parallel code.

In the following examples and discussion, the Finite Element code referred to solves for displacements and stress using 2D or 3D meshes with triangular, quadrilateral and cuboidal element types with 3, 4 and 8 nodes respectively. Four simulation modes are also available, Plain Stress, Plain Strain, Axis Symmetric and Three Dimensional. All these factors are involved in the construction of element stiffness matrices and therefore have implications in the proving of parallelism between iterations of the loop L_1 as illustrated in Figure 4. As well as the element stiffness matrix itself, all other workspace arrays involved in its construction can cause serialisation of the element loop and excessive communication. The following code section in Figure 4 shows key features of the system matrix construction process, although greatly simplifies the actual FE code. A typical dependence graph for routine **ELEINFO** is shown in
Figure 5. The graphs represent statements using the numbered oval nodes where the node labelled **Start** and **Stop** indicate the inputs and outputs of the routine. The directed arcs between nodes represent dependencies where, in these diagrams, they indicate the flow of data from a statement (assignment) to another statement (usage).

```
  SUBROUTINE ELEINFO( TYPE,NNODES,NDOF, NSTRS)
C SET UP BASIC ELEMENT DIMENSIONS
  IF (TYPE.EQ.1) THEN
C     TRIANGLE
      NNODES=3
      NDOF=6
      NSTRS=4
  ENDIF
  IF (TYPE.EQ.2) THEN
C     QUADRILATERAL
      NNODES=4
      NDOF=8
      NSTRS=4
  ENDIF
  IF (TYPE.EQ.3) THEN
C     CUBOID
      NNODES=8
      NDOF=24
      NSTRS=6
  ENDIF
  END
```

```
        SUBROUTINE INVERT( MAT, INVMAT, N)
C       INVERT A MATRIX
        DET=. . .
        IF (ABS(DET).GT.0) THEN
                INVMAT(1,1)=. . .
                INVMAT(1,2)=. . .
                INVMAT(1,3)=. . .
                . . . .
                INVMAT(N,N)=. . .
        ENDIF
        END
```

```
        SUBROUTINE MATMUL(A,B,C,N1,N2,N3)
C       MULTIPY MATRICIES C=A*B
        DO J=1,N3
           DO I=1,N1
                C(I,J)=0
                DO K=1,N2
                        C(I,J)=C(I,J)+A(I,K)*B(K,J)
                ENDDO
           ENDDO
        ENDDO
        END
```

Figure 4a. Element stiffness matrix construction routines (continued on next page).

```
   SUBROUTINE ELESTRS(STRESS,APPROX)
C  CALCULATE STRESS MATRIX
   IF (APPROX.EQ.'PLAIN_STRESS') THEN
            STRESS(1,1)=. . .
            STRESS(1,2)=. . .
            STRESS(2,1)=. . .
            STRESS(2,2)=. . .
   ENDIF
   IF (APPROX.EQ.'PLAIN_STRAIN') THEN
            STRESS(1,1)=. . .
            . . . .
            STRESS(2,2)=. . .
   ENDIF
   IF (APPROX.EQ.'AXIS_SYM') THEN
            STRESS(1,1)=. . .
            . . . .
            STRESS(2,2)=. . .
   ENDIF
   IF (APPROX.EQ.'THREED') THEN
            STRESS(1,1)=. . .
            STRESS(1,2)=. . .
            . . . .
            STRESS(6,6)=. . .
   ENDIF
   END
```

```
            SUBROUTINE LOCDERIV( LOCAL, TYPE)
C           CALCULATE LOCAL DERIVATIVE
            IF (TYPE.EQ.1) THEN
                        LOCAL(1,1)=. . .
                        LOCAL(1,2)=. . .
                        LOCAL(1,3)=. . .
                        . . . .
                        LOCAL(2,3)=. . .
            ENDIF
            IF (TYPE.EQ.2) THEN
                        LOCAL(1,1)=. . .
                        LOCAL(1,2)=. . .
                        . . . .
                        LOCAL(2,4)=. . .
            ENDIF
            IF (TYPE.EQ.3) THEN
                        LOCAL(1,1)=. . .
                        LOCAL(1,2)=. . .
                        . . . .
                        LOCAL(3,8)=. . .
            ENDIF
            END
```

```
   SUBROUTINE LOCXYZ(CORD,XYCORD,ELETOP,
 &                             NNODES,DIMENS)
C   SET UP LOCAL COORDINATE MATRIX
   DO I=1,NODES
            DO J=1,DIMENS
                        CORD(I,J)=XYCORD(ELETOP(I),J)
            ENDDO
   ENDDO
```

```
   SUBROUTINE MATTRAN(A,B,N1,N2)
C  TRANSPOSE MATRIX A TO GIVE MATRIX B
   DO I=1,N1
            DO J=1,N2
                        B(J,I)=A(I,J)
            ENDDO
   ENDDO
   END
```

```
            SUBROUTINE STRAIN(DERIV,STRAIN,APPROX,
         &                             NNODES,NSTRS)
C           CALCULATE STRAIN MATRIX
            IF  (APPROX.EQ.'PLAIN_STRESS'.OR.
         &       APPROX.EQ.'PLAIN_STRAIN'.OR.
         &       APPROX.EQ.'AXIS_SYM') THEN
                        STRAIN(1,1) = . . .
                        STRAIN(1,2) = . . .
                        . . . .
                        STRAIN(NSTRS,2*NNODES) = . . .
            ENDIF
            IF (APPROX.EQ.'THREED') THEN
                        STRAIN(1,1) = . . .
                        STRAIN(1,2) = . . .
                        . . . .
                        STRAIN(NSTRS,3*NNODES) = . . .
            ENDIF
            END
```

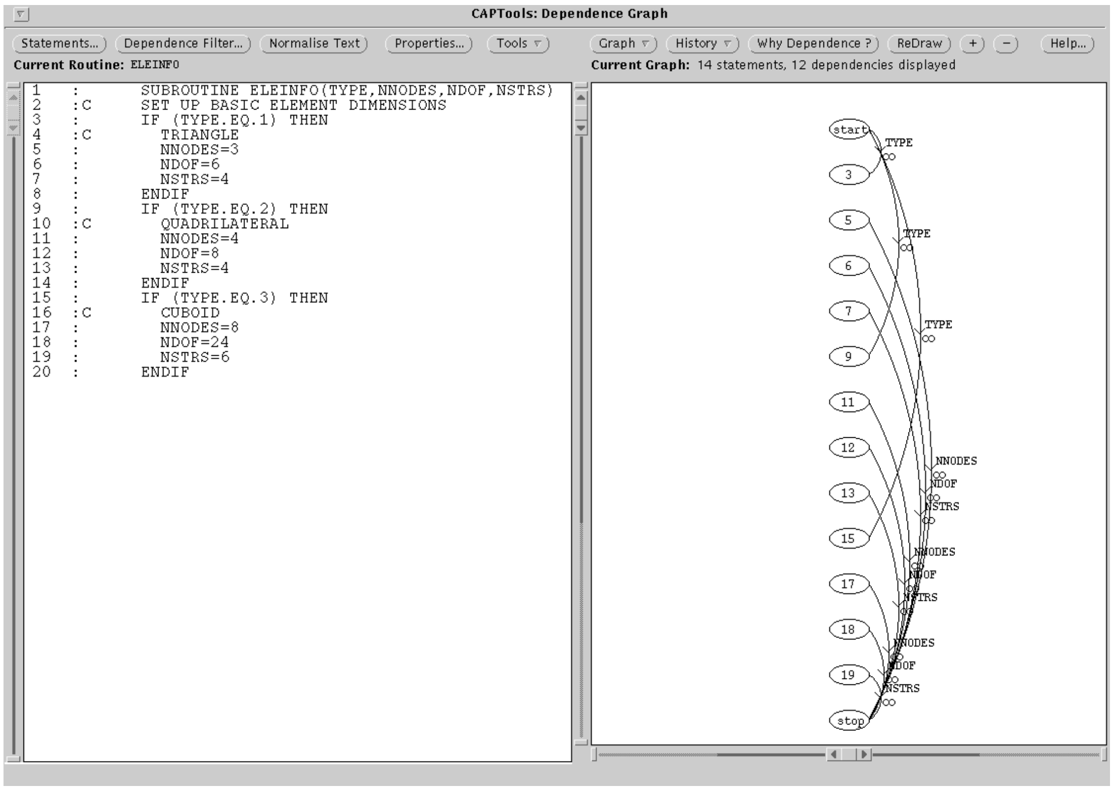
```
            SUBROUTINE SYSTEMMATRIX
C           CONSTRUCT THE SYSTEM MATRIX TAKING CONTRIBUTIONS FROM EACH FINITE ELEMENT
            . . . .
            IF (APPROX.EQ.'THREED') THEN
                        DIMENS=3
            ELSE
                        DIMENS=2
            ENDIF
            . . . .
L1          DO IE=1,NUM_ELEMENTS
                        CALL ELEINFO( ELETYP(IE), NNODES, NDOF, NSTRS)
                        CALL LOCXYZ(COORD,ELETOP(1,IE),NNODES,DIMENS)
                        CALL ELESTRESS( STRESS, APPROX)
                        CALL LOCDERIV( LOCAL, ELETYP(IE))
S1                      CALL MATMUL( LOCAL, COORD, JACOBIAN, DIMENS, NNODES, DIMENS)
                        CALL INVERT( JACOBIAN, INVJAC, DIMENS)
S2                      CALL MATMUL( INVJAC, LOCAL, DERIV, DIMENS, DIMENS, NNODES)
                        CALL STRAIN( DERIV, STRAIN, APPROX, NNODES, NSTRS)
S3                      CALL MATMUL(STRESS, STRAIN, FORCE, NOSTRS, NOSTRS, NNODES)
                        CALL MATTRAN(STRAIN, TSTRAIN, NOSTRS, NNODES)
S4                      CALL MATMUL(TSTRAIN, FORCE, CONTRIB, NNODES, NOSTRS, NNODES)
                        . . . . . . .
```

Figure 4b. Element stiffness matrix construction routines (continued).

It can be seen in the above example that the element stiffness matrix (**CONTRIB** in S_4) is constructed using a number of matrices of differing dimensions dependent on the element type (**ELETYP**), dimensionality (**DIMENS**) and analysis type (**APPROX**). The accurate analysis of such codes is a complex process. In the following section, the techniques used for dependence analysis are applied to the above code.

4. ANALYSIS OF ELEMENT MATRIX CODE

Proving that no dependencies are carried by the element loop requires multi-dimensional, interprocedural covering set calculation taking control information into account [12]. An initial dependence analysis is first performed based predominantly on memory location access as opposed to data flow. Some data flow rules are used in this analysis, such as the exact dependence [12] of a usage on a single assignment where all used values are satisfied in that assignment. Routine inputs and outputs are indicated as dependencies with a source of the start node for each routine or with a sink of the routine stop node. The start and stop nodes are then used in interprocedural operations.



Current Routine: ELEINFO

Current Graph: 14 statements, 12 dependencies displayed

```
1   :      SUBROUTINE ELEINFO(TYPE,NNODES,NDOF,NSTRS)
2   :C     SET UP BASIC ELEMENT DIMENSIONS
3   :      IF (TYPE.EQ.1) THEN
4   :C       TRIANGLE
5   :        NNODES=3
6   :        NDOF=6
7   :        NSTRS=4
8   :      ENDIF
9   :      IF (TYPE.EQ.2) THEN
10  :C       QUADRILATERAL
11  :        NNODES=4
12  :        NDOF=8
13  :        NSTRS=4
14  :      ENDIF
15  :      IF (TYPE.EQ.3) THEN
16  :C       CUBOID
17  :        NNODES=8
18  :        NDOF=24
19  :        NSTRS=6
20  :      ENDIF
```

Figure 5. Dependence Graph for routine ELEINFO in Figure 3.

After this initial analysis, all dependencies are re-examined to identify all intermediate assignments, constructing a control set containing control and index range information for all relevant array references. The initial analysis is required to determine all possible assignments for each usage which can then be used to determine the set of intermediate assignments relevant for the testing of each dependence. If it can be proven that all data that could be passed between source (assignment) and sink (usage) of a dependence is overwritten in intermediate assignments then that dependence carries no values and can be eliminated from the dependence graph. The control flow graph and the dependence graph are used to ensure that the intermediate assignments occur after the assignment and before the usage of the dependence being tested. Additional tests for anti-dependencies (i.e. usage to the assignment that destroys the used value) are also used to ensure that any intermediate assignment provides data for the usage, rather than overwriting data previously used in the usage. This allows loop carried dependencies to be used in the detection of intermediate assignments, guaranteeing that all data assigned precedes its usage in the dependence being tested.

Consider the usage of array **A** within routine **MATMUL** as called from statement S_1 in routine **SYSTEMMATRIX** in Figure 4. The initial analysis sets routine input dependencies for array **A** to the start node of routine **MATMUL**. This in turn leads to dependencies being set for the related input array **LOCAL** in the call at S_1 in routine **SYSTEMATRIX** from both before the element loop L_1 (carried , for example, by a time step loop) and also from previous assignments to array **LOCAL** in earlier iterations of L_1. For each of these dependencies, examining all other dependencies of the usage in S_1 can identify the set of intermediate assigners of array **LOCAL**. In the case of the loop carried dependence, the assignment to **LOCAL** in the call to **LOCALDERIV** in routine **SYSTEMMATRIX**, identified using a loop independent dependence, is an intermediate assignment. The actual assignments are then identified using the dependencies of the stop node of routine **LOCALDERIV** (i.e. the routine outputs), where each such dependence represents one assignment. This leads to 38 intermediate assignments, all in routine **LOCALDERIV**, being identified, where all must be considered if an accurate dependence graph is to be produced.

Program variables are defined in terms of their symbol, defining statement, defining routine and the call path that caused the current instance of the defining routine. Forward substitution is used to store symbolic expressions in terms of the earliest set of definers possible, using the dependence graph to identify statements that assign data and that can therefore be used in substitution. The routine start and stop nodes allow this substitution to proceed into called routines and also into caller routines. On entering a called routine, the call statement is added to the call path of any subsequent variable definitions. When entering a calling routine, either a unique caller statement or a caller statement specified by the variable defining call path must be identified. This substitution is blocked by the usage of a variable that has multiple definitions, either due to conditionally controlled assignments or several callers of a routine. In such cases, the statement that includes the usage represents the defining statement. Control information is extracted using the control dependence graph [24] for control of the usage and assignment of a dependence being tested along with the control of all the intermediate assignments. Additionally, control information for all statements in the call paths of these statements is extracted and intersected with other control information to give the overall condition under which the instance of a statement will execute. This control information is then intersected with the index range information for the relevant array access to provide the full control set for the array access.

Consider the first assignment of array **LOCAL** in routine **LOCALDERIV**. The assignment range is represented in terms of a dummy variable for each index of the array, referred to as **INDEX1** and **INDEX2** in the following examples, to allow a comparison of array references in any loop in any routine. The assignment range of **LOCAL** in the first assignment of **LOCALDERIV** is :-

```
(1 <= INDEX1 <= 1) and (1 <= INDEX2 <= 1)
```

This is combined with the control of this assignment to give the full control set of :-

> (1 <= INDEX1 <= 1) and (1 <= INDEX2 <= 1) and (TYPE = 1)

Forward substitution traces the symbolic variable **TYPE** through the routine boundary into routine **SYSTEMMATRIX** where it is substituted with **ELETYP(IE)**, which is in turn traced so that **ELETYP** is defined at a READ statement (not shown in Figure 4) and **IE** is defined at statement L_1.

Similarly, the original dependence usage of array **A** in the call to routine **MATMUL** as called at S_1 produces the control set information :-

> (1 <= INDEX1 <= N1) and (1 <= INDEX2 <= N2)

Forward substitution is then used to trace variable **N1** to the defining statement S_1 where further substitution is prevented since two alternative assignments exist for variable **DIMENS**. Variable **N2** is also traced from within routine **MATMUL** to become variable **NNODES** at statement S_1, which in turn is traced using a unique true dependence to the assignment within the call to **ELEINFO**. Substitution within routine **ELEINFO** is then prevented since 3 alternative assignment statements exist for variable **NNODES**. As a result, the defining statement used in the control set is the stop node of routine **ELEINFO**, with the call to **ELEINFO** in routine **SYSTEMMATRIX** also being stored to define the definite call path that defines this instance of variable **NNODES**. This gives the control set of :-

> (1 <= INDEX1 <= DIMENS) and (1 <= INDEX2 <= NNODES)

The process of covering set calculation then continues by intersecting the control set information for all intermediate assignments of the dependence usage to create the control set representing all potential intermediate assignments. This set is then negated so that it represents the set of data not assigned by the intermediate assignments. This negated set is then intersected with the control sets for the assignment and usage of the dependence being tested to represent data carried by this dependence. This set can then be tested for contradictions, where any contradiction indicates that the set of data carried by the dependence is empty and therefore the dependence can be deleted.

The final control set for the loop carried dependence of array **LOCAL** as used in statement S_1 consists of the negated 38 intermediate assignment sets (one for each routine output assignment in routine **LOCDERIV**), along with the source and sink control sets for the dependence being tested :-

```
(INDEX1 <> 1 or INDEX2 <> 1 or ELETYP(IE) <> 1) and
(INDEX1 <> 2 or INDEX2 <> 1 or ELETYP(IE) <> 1) and
. . . . . .
(INDEX1 <> 2 or INDEX2 <> 8 or ELETYP(IE) <> 3) and
(INDEX1 <> 3 or INDEX2 <> 8 or ELETYP(IE) <> 3) and
(1 <= INDEX1 <= DIMENS) and (1 <= INDEX2 <= NNODES)
```

This large set is first simplified, applying simple logic to remove many literals, using, for example, the fact that **INDEX1 >= 1** to remove components from other clauses. The test of the remaining control set is performed using logical substitution [12] to trace variables through conditional statements and using the inference engine based on conflict resolution [12] on every fully substituted control set. This will involve many inference engine tests, all of which must return a false result if the dependence is to be removed.

In the above example, the logical substitution will create three alternative control sets due to the three routine output assignments of scalar **NNODES** in routine **ELEINFO** that have dependencies on the routine stop node which is the current definition location of **NNODES**. Similarly, the values of **DIMENS** will be substituted. The first substituted set follows the dependence of **DIMENS** to the assignment **DIMENS = 2** along with the associated control of **APPROX <> 'THREED'** and the dependence to the assignment **NNODES = 3** in routine **ELEINFO** with the associated control **TYPE = 1** which is then also substituted as discussed earlier to give :-

CASE 1:-

```
(INDEX1 > 1 or INDEX2 > 1 or ELETYP(IE) <> 1) and
(INDEX1 <> 2 or INDEX2 > 1 or ELETYP(IE) <> 1) and
. . . . . .
(INDEX1 <> 2 or INDEX2 <> 8 or ELETYP(IE) <> 3) and
(INDEX1 <> 3 or INDEX2 <> 8 or ELETYP(IE) <> 3) and
(1 <= INDEX1 <= 2) and (APPROX <> 'THREED') and
(1 <= INDEX2 <= 3) and (ELETYP(IE) = 1)
```

This control is then simplified, eliminating literals when they are false and entire clauses when any included literal is definitely true to leave the control set :-

```
(INDEX1 > 1 or INDEX2 > 1) and                          C1
(INDEX1 < 2 or INDEX2 > 1) and                          C2
. . . . . .
(INDEX1 > 1 or INDEX2 < 2 OR INDEX2 > 2) and            C3
(INDEX1 < 2 or INDEX2 < 2 OR INDEX2 > 2) and            C4
. . . . . .
(INDEX1 > 1 or INDEX2 < 3) and                          C5
(INDEX1 < 2 or INDEX2 < 3) and                          C6
(1 <= INDEX1 <= 2) and (APPROX <> 'THREED') and
(1 <= INDEX2 <= 3) and (ELETYP(IE) = 1)
```

Conflict resolution is then used in the inference engine to attempt to prove that no solution to the control set exists :-

CONFLICT : **INDEX1 > 1** with **INDEX1 < 2** giving

```
(INDEX2 > 1) and                          C7 (from C1 and C2)
(INDEX2 < 2 or INDEX2 > 2) and            C8 (from C3 and C4)
(INDEX2 < 3)                              C9 (from C5 and C6)
```

CONFLICT : **INDEX2 > 1** with **INDEX2 < 2** giving

```
(INDEX2 > 2)                              C10 (from C7 and C8)
```

CONFLICT : **INDEX2 > 2** with **INDEX2 < 3** giving

No solution from C9 and C10.

This false result from the inference engine indicates that no values are carried by the dependence being tested given the specified conditions. This allows the next logical substitutions to occur and another test performed. Another alternative set will involve the logical substitution to the assignments **DIMENS = 2** and **NNODES = 8**, a false result could, again, be proven in the following way:-

CASE 2:-

```
(INDEX1 > 1 or INDEX2 > 1) and                                  C1
(INDEX1 < 2 INDEX1 > 2 or INDEX2 > 1) and                       C2
(INDEX1 < 3 or INDEX2 > 1) and                                  C3
. . . . .
(INDEX1 > 1 or INDEX2 < 2 or INDEX2 > 2) and                    C4
(INDEX1 < 2 or INDEX1 > 2 or INDEX2 < 2 or INDEX2 > 2) and      C5
(INDEX1 < 3 or INDEX2 < 2 or INDEX2 > 2) and                    C6
. . . . .
(INDEX1 > 1 or INDEX2 < 8) and                                  C7
(INDEX1 < 2 or INDEX1 > 2 or INDEX2 < 8) and                    C8
(INDEX1 < 3 or INDEX2 < 8) and                                  C9
(1 <= INDEX1 <= 2) and (APPROX <> 'THREED') and
(1 <= INDEX2 <= 8) and (ELETYP(IE) = 3)
```

CONFLICT : **INDEX1 > 1** with **INDEX1 < 2**

```
(INDEX1 > 2 or INDEX2 > 1) and                                  C10 (from C1 and C2)
(INDEX1 > 2 or INDEX2 < 2 or INDEX2 > 2) and                    C11 (from C4 and C5)
. . . . .
(INDEX1 > 2 or INDEX2 < 8)                                      C12 (from C7 and C8)
```

CONFLICT : **INDEX1 > 2** with **INDEX1 < 3**

```
(INDEX2 > 1) and                                                C13 (from C3 and C10)
(INDEX2 < 2 or INDEX2 > 2) and                                  C14 (from C6 and C11)
. . . . .
(INDEX2 < 8)                                                    C15 (from C9 and C12)
```

CONFLICT : **INDEX2 > 1** with **INDEX2 < 2** gives (INDEX2 > 2)
CONFLICT : **INDEX2 > 2** with **INDEX2 < 3** gives (INDEX2 > 3)

. . . . .

CONFLICT : **INDEX2 > 6** with **INDEX2 < 7** gives (INDEX2 > 7) which contradicts C15 proving no solution exists.

Consider, however, the substitution of **DIMENS = 3** and **NNODES = 3** :-

CASE 3:-

```
(INDEX1 > 1 or INDEX2 > 1) and
(INDEX1 < 2 or INDEX1 > 2 or INDEX2 > 1) and
. . . . . .
(INDEX1 < 2 or INDEX1 > 2 or INDEX2 < 2 or INDEX2 > 2) and
. . . . . .
(INDEX1 < 2 or INDEX1 > 2 or INDEX2 < 3) and
(1 <= INDEX1 <= 3) and (APPROX = 'THREED') and
(1 <= INDEX2 <= 3) and (ELETYP(IE) = 1)
```

In this case, a false result is not returned from the inference engine since no conflict for the literal **(INDEX1 > 2)** can be found and the test terminates. This is due to the range 1-3 being included in the usage set for the first index whilst many of the intermediate assignments relate to 2D elements and therefore only cover the range 1-2.

The full set of alternative substitutions and the result of application of the inference engine are shown in Figure 6.

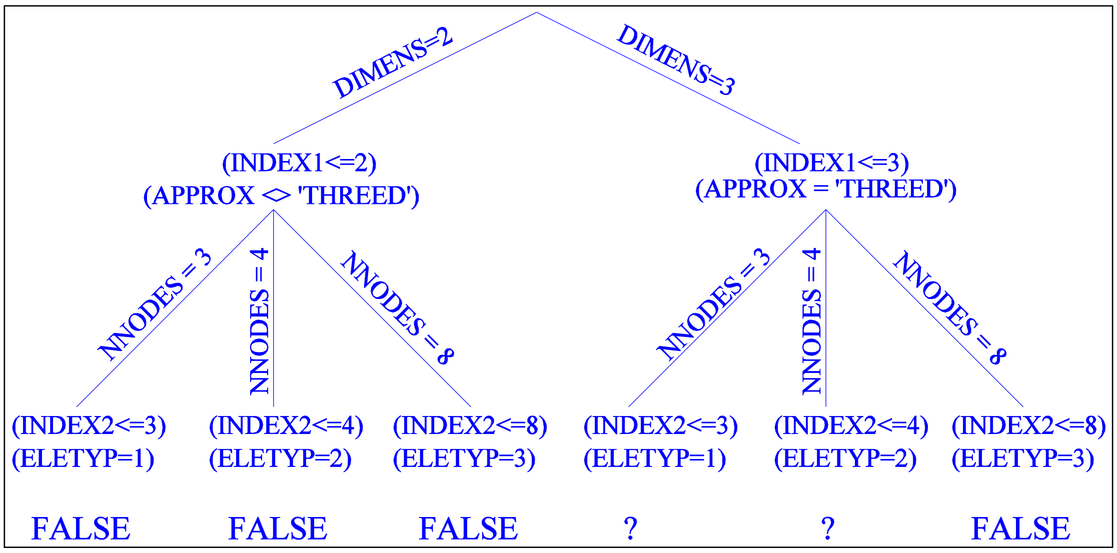


Figure 6. Result for all substitution paths.

Accurate information for the interaction between the factors of the finite element analysis is essential for a high quality dependence analysis. Unfortunately, it is often the case that such information is not explicitly stated within the source code but is implicitly enforced by the input data. For example, in the above case, the cuboidal element type can only be used with a three dimensional analysis. Without such information, the dependence tests will be left with unresolved inequalities such as that shown above in CASE 3, having potentially serious consequences on the parallelisation.

In the code section shown in Figure 4, a number of cases similar to that discussed above exist involving workspace arrays **COORD**, **STRESS**, **JACOBIAN**, **INVJAC**, **DERIV** etc., all of which require the same information from the user. Another analysis problem also exists since the array **STRESS** is used in the call to **MATMUL** in S_3, however, the assignments to array **STRESS** within the call to routine **ELESTRS** are all controlled by the value of string **APPROX** as read in from an input file. The dependence analysis, without any other information, must assume that **APPROX** can take any value and not necessarily be any of the four strings used to control the four sections in routine **ELESTRS**. This forces the dependence analysis to assume that the value of the array **STRESS** may not have been calculated during the same iteration of loop L_1, causing assumed dependencies to be set.

5. USER INTERACTION IN DEPENDENCE ANALYSIS.

The constraints on program input data highlighted in section 4 on the factors of the finite element analysis must therefore be introduced through user interaction since the user has knowledge of the relevant information. In this case, the two sets of information required are :-

APPROX = 'PLAIN_STRESS'.or.APPROX = 'PLAIN_STRAIN'.or. C1
APPROX = 'AXIS_SYM'.or.APPROX = 'THREED'

IF and only IF (DIMENS = 3) then ELETYP = 3 C2

The first item of information (C1) is placed directly in the knowledge base and can be used in inference engine proofs. This allows cases such as that involving **STRESS**, as discussed in section 4, to be correctly handled since it guarantees that the one of the four sections in routine **ELESTRS** will be entered in every call.

The if and only if constraint (C2) indicates that cuboidal elements can only legally used in three dimensional meshes and that three dimensional meshes must consist of only cuboidal elements. C2 adds two sets of information to the knowledge base i.e.

```
((DIMENS <> 3) or (ELETYP = 3)) and                          C3
((DIMENS = 3) or (ELETYP <> 3))                              C4
```

With this information, the previous unresolved dependence tests can be proven false. Consider CASE 3 that was previously left unresolved :-

```
(1 <= INDEX1 <= 3) and (APPROX = 'THREED') and               C5
(1 <= INDEX2 <= 3) and (ELETYP(IE) = 1)
```

The substitution with conditional assignments for CASE 3 also substitutes the reference to **DIMENS** in clauses C3 and C4 to be equal to 3, leading to the simplified set :-

```
(ELETYP = 3)                                                 C6
```

A contradiction can then be identified between the set C6 and the original clause in C5 **(ELETYP(IE) = 1)** proving the set false. This process has used the fact that triangular elements are only legally used in two dimensional problems. A similar process can be used to prove that no solution is possible in all the cases shown to originally fail in Figure 6.

The information required from the user can be input in terms familiar to the user in respect to the program input variables about which the implicit restrictions exist. The process of identifying which information is required to eliminate a particular dependence (or set of dependencies) and asking the appropriate question(s) to the user is not a straightforward task. The questions that CAPTools currently presents to the user are based on index equality during the memory location based dependence tests. This provides a concise question for user inspection. As was shown in the cases in section 4, the unresolved control sets can consist of very large numbers of clauses, each involving several literals presented in clausal form. Additionally, the problems only occurred under certain substitution paths potentially requiring numerous sets for each false dependence that has been assumed. This, added to the fact that many dependencies may involve large control sets when a dependence should actually be set, makes the presentation of a concise question to the user problematic. The filtering of the control sets to present the user with only information that relates to program input variables, can greatly simplify interaction. This may, however, in many cases prevent key information being presented, making the any user response insufficient to allow dependence removal.

Consider the unresolved control set from CASE 3 in section 4. The variables **INDEX1** and **INDEX2** are internal dependence analysis variables and have no meaning to the user. Removing all clauses that contain literals using these variables, the only remaining information is :-

```
(APPROX = 'THREED')  and  (ELETYP(IE) = 1)
```

To ease the users task, this information could be re-configured in several ways to ask the user for information that proves this control set is false :-

```
(APPROX <> 'THREED') or  (ELETYP <> 1)                      C7
                or
IF (APPROX = 'THREED') THEN (ELETYP <> 1)                   C8
                or
IF (ELETYP = 1) THEN (APPROX <> 'THREED')                   C9
```

One or all of the above representations may aid correct user interpretation. A user assertion that the information as presented is definitely true allows that information to be added to the knowledge base. This piece of information, however, relates only to one of the substitution paths shown in Figure 6.

6. IMPLICATIONS OF SERIAL CODE ALGORITHM ERRORS IN CODE PARALLELISATION

The techniques in section 4 enable an accurate dependence graph to be generated and the user interaction in section 5 can ease the impact of the assumptions inherent in the serial code. There are, however, some dependencies still remaining between iterations of the element loop of the **SYSTEMMATRIX** routine. These dependencies remain since they actually exist due to error checking within the loop. Most of these error checks set error flags which are followed by loop exits that terminate the loop (and often terminate the application code) and can be handled in parallel code generation so that loop carried dependencies do not inhibit parallelism [25]. A few dependencies, however, were caused by potentially incorrect serial code in the matrix inversion routine to cater for matrices with a zero determinant as highlighted in Figure 7 for a more complete version of the **INVERT** routine from 4.

```
DET=MAT(1,1)*MAT(2,2)-MAT(2,1)*MAT(1,2)
IF (DET.NE.0) THEN
        INVMAT(1,1)=. . .
        INVMAT(2,1)=. . .
        INVMAT(1,2)=. . .
        INVMAT(2,2)=. . .
ENDIF
END
```

Figure 7. Matrix inversion code with check for zero determinant

The **INVMAT** array is not set if a zero determinant is found, however, no loop exiting error trap is set and execution continues. This is obviously a fault in the serial code but forces the dependence analysis to set a loop carried true dependence for later usages of **INVMAT** in an iteration of the element loop since values from the previous iteration may be used. The user must then intervene to either re-write the code section to add an error trap, or alternatively, to delete the loop carried dependence in the knowledge that any input data that operates correctly in serial will still operate correctly in parallel. Experience shows that such cases are fairly common in real application codes identifying problems with the original serial code. In this case, two dependencies need to be deleted to overcome the above problem in three routines, all involving assignments to **INVMAT** in calls to routine **INVERT**.

The dependence graph produced is now accurate enough to allow effective parallelisation, with parallelism within the element loop discussed here being exploited. In this case, a loop split transformation was used to split the element matrix construction code from the system matrix setup to allow effective parallelisation. If any loop carried dependencies were set between iterations of the element loop, such a transformation would not be legal unless a significant amount of user interaction were exerted.

7. RESULTS OF ANALYSIS ON THE FINITE ELEMENT CODE

The measure of success of dependence analysis is the performance of the resulting parallel code. The code generation algorithms extensively use the dependence graph for data partition determination, execution control masking, inspector loop generation and communication generation [26], thus the parallel performance is fundamentally linked to the dependence graph accuracy. The measures used here are the number of communications generated, the number of parallel loops and the number of true dependencies. Four levels of graph accuracy are considered :-

Low Power	:	No covering set calculation
Full Power	:	Full covering set calculation
Full + Information	:	Full covering set calculation with user information as described in sections 5.
Full + Information + Manual Deletion	:	All of above + manual deletion of dependencies as described in section 6.

Table 2 shows that using the full power of the dependence analysis (as describe in section 4) dramatically decreases the number of dependencies even without any user information. The further addition of a small amount of user information (as described in section 5) eliminates another fifty dependencies, many of which are key to the success of the parallelisation of this code. The parallel code produced using the dependence graph constructed prior to this user information contained an extremely large number of communications, making that code impractical as a starting point for a manually tuned, efficient parallel code. With the user information, the code produced contained a large number of communications concerned with broadcasting the various finite element based arrays between iterations of the element loop in the system matrix construction routines. The cause of these broadcasts was easily determined (using the CAPTools communications browser [13]) to be the remaining serialising dependencies within those loops caused by the incorrect serial code as discussed in section 6. Manually deleting these dependencies enabled all the loops involved in element stiffness matrix calculation to execute fully in parallel with no communications, significantly reducing the number of communications within the parallel code. Figure 8 shows the loop carried dependencies within the system matrix building routine (similar in structured to that shown in Figure 4). The numbered oval nodes represent statements in the application codes with the directed arcs representing dependencies with the level of the carrier loop and the associate variable name also indicated. The improved accuracy of the dependence graph can clearly be seen as the power of the analysis is increased. An important consideration in dependence graph construction is the amount of interaction that can be demanded from the user. The manual dependence deletion needed to produce the final accurate dependence graph involved the deletion of only 8 dependencies throughout the application code. If this process were to be performed using the dependence graph produced using a lower power analysis (or without user information), the number of dependencies requiring investigation and subsequent manual deletion would be significantly high and perhaps impractical. The communications generated and performed for the lower power analyses are not shown since they are excessive, exceeding the high numbers given for the *Full Power+Interaction* column. The massive increase in communication number when no manual deletion is used is caused by the need to either broadcast to every processor, or calculate on every processor, the values for an element matrix that are involved in the loop carried dependencies. This indicates how unforgiving the code generation algorithms can be of extra dependence edges, whether they are due to the dependence analysis algorithms used or deficiencies in the serial code. The communications browser in CAPTools [13] does, however, allow the user to determine the dependencies that caused excessive communication, therefore, with this number of broadcast communications, the offending dependencies are easily identified. The importance of the detection and exploitation of parallelism in all sections of the code can be seen in that the number of overlap (halo) updating communications in the final parallel code. There are 4 communications performed once (outside the time step loop); 16 performed once per timestep and 1 performed for each iteration of the Conjugate Gradient solver (within the timestep loop). The remaining communications consist of reductions (primarily for summations in the conjugate gradient routine) and the communication of data as it is read into the code at the start of the execution.

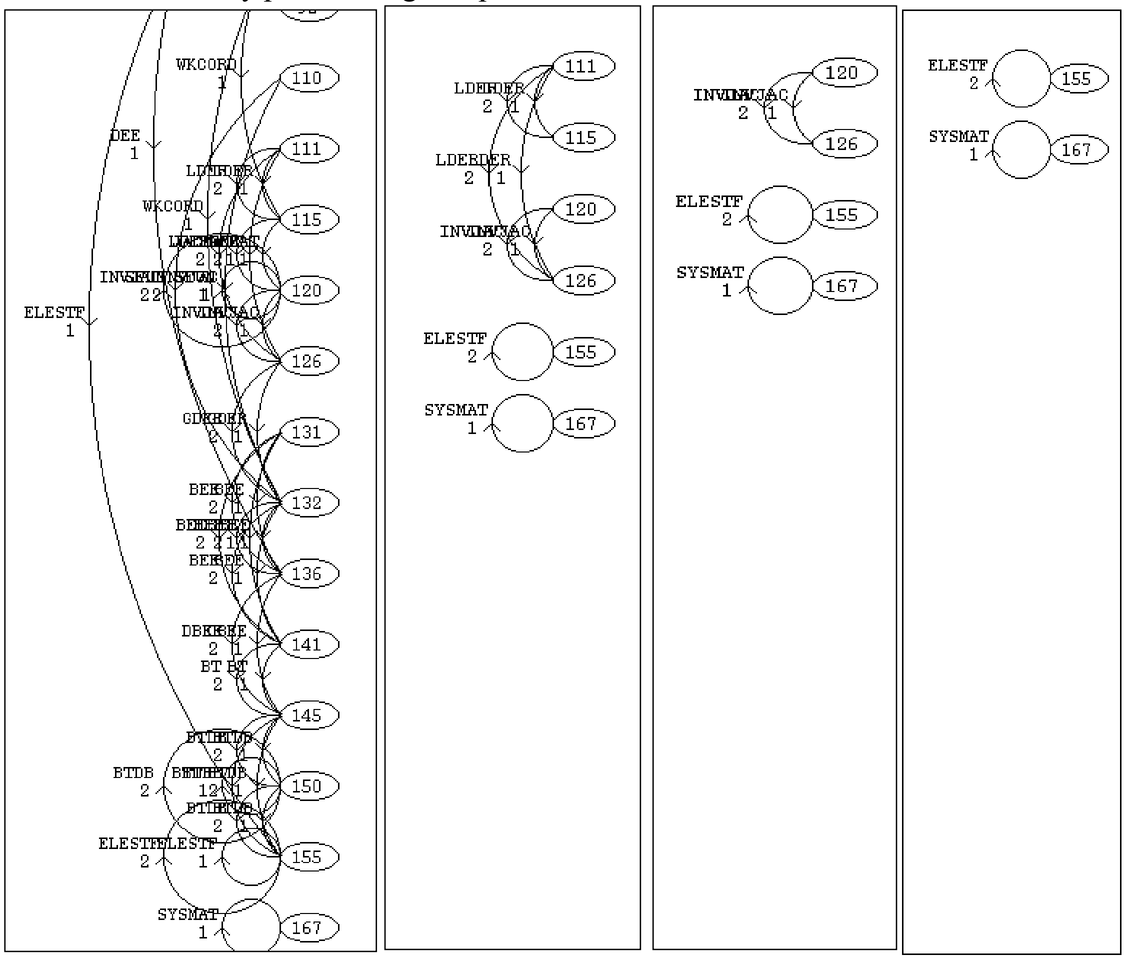
	Low Power	Full Power	Full Power + Interaction	Full + Interact + Manual Del
Number of dependencies	2756	2329	2279	2271
Number of parallel loops	79	84	86	90
Number of comms generated	----	----	298	39

Table 2. Effect of dependence analysis and user interaction on the parallelisation process.

The code generated with full power and a small amount of user interaction contains a far smaller communication overhead in both execution time and code volume than the other analyses. The code generated is close to that of an efficient manual parallelisation since the techniques and strategies used in the automation attempt to mimic an effective manual parallelisation process. The code generated is also changed as little as possible from the original serial to maintain user recognition to allow manual optimisation of the generated code, achieving parallel performance very close to that of a good manual parallelisation. The entire process of parallelising this 7100 line FE code and optimising the generated code can be performed in a few hours using the full power dependence analysis. As an example of performance, this code achieves a speedup of around 20 on 32 Cray T3D nodes for a relatively small 3000 element mesh.

8. CONCLUSIONS

Real application codes, in particular Finite Element codes, require in-depth interprocedural, value based dependence analysis, along with minimal, if fairly sophisticated, user interaction. The knowledge required is usually known to the code paralleliser and relates to information implicit in input data and not explicitly stated in the code itself. Although it can be argued that a "correct" serial code should test input data, and therefore hold such information explicitly in the source code, the re-authoring of serial code is not a task that can be demanded by parallelising compilers or tools. Such



Low Power Full Power Full Power + Information Full Power + Information + Deletion

Figure 8. Dependence graph for element matrix construction routine for different levels of analysis.

situations must therefore be catered for in dependence analysis techniques. The example code used here requires more user involvement than any other code we have experience of to date, including many that consist of far larger code volumes, however, for the purposes of this paper it is the most interesting example.

For the Finite Element code investigated here, the techniques developed enabled an effective parallelisation of the code in a reasonably short period of time achieving significant speedup.

8. REFERENCES

1 D.J.Kuck, R.H.Kuhn, B.Leasure, D.A.Padua and M.J.Wolfe, Compiler Transformations of Dependence Graphs. pp 207-218 in Conference Record of the 8th ACM Symposium on Principals of Programming Languages, 1981.

2 U.Banerjee, Speedup of Ordinary Programs. PhD Thesis, University of Illinois At Urbana Champaign, 1979.

3 J.R.Allen and K.Kennedy, PFC - A Program to Convert Fortran Programs to Parallel Form. in Proceeding of the IBM Conference on Parallel Computation and Scientific Applications, 1982.

4 U.Banerjee, Dependence Analysis for Supercomputing. Kluwer Academic Publishers, 1988.

5 M.Wolfe and C-W Tseng, The Power Test for Data Dependence. pp 591-601 in IEEE Transactions on Parallel and Distributed Systems,3,5, 1992

6 D.E.Maydan, J.L.Hennessy and M.S.Lam, Efficient and Exact Data Dependence Analysis. pp 1-14 in ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991.

7 G.Goff, K.Kennedy and C-W Tseng, Practical Dependence Testing. pp 15-29 in Proceedings of The ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991.

8 W.Pugh, A Practical Algorithm for Exact Array Dependence Analysis. pp 102-114 in Communications of the ACM, Vol 35, 8, 1992.

9 T.Fahringer, Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. The Journal of Supercomputing, 12, pp 1-29, 1998.

10 C.S.Ierotheou, S.P.Johnson and M.Cross, An Extension of the Standard Omega Test for the Parallelisation of Computational Mechanics Codes Containing Arrays with Mapped Indices. University of Greenwich Technical Report, CMS Press 98/IM/34, 1998.

11 H.P.F.Forum, High Performance FORTRAN Language Specification Version 2.0, Rice University, Houston, Texas, 1996.

12 S.P.Johnson, M.Cross, M.G.Everett. Exploitation of Symbolic Information in Interprocedural Dependence Analysis. Parallel Computing, 22, pp 197-226, 1996

13 P.F.Leggett, A.T.J.Marsh, S.P.Johnson and M.Cross, Integrating user Knowledge with Information from parallelisation Tools to Facilitate the Automatic Generation of Efficient Parallel FORTRAN code. Parallel Computing, Vol 22, 2, pp197-226, 1996.

14 W.Pugh and D.Wannacott, Eliminating False Dependencies Using the Omega Test. pp 140-151 in ACM SIGPLAN'92 Conference on Programming Design and Implementation, San Francisco, CA, July 1992.

15 V.Maslov, Lazy Array Data-Flow Dependence Analysis. pp 311-325 in Proceeding of 21st Annual ACM Sigplan-Sigact Symposium on Principals of Programming Languages, January 1994.

16 D.Callahan, K.D.Cooper, K.Kennedy and L.Torczon, Interprocedural Constant Propagation. In Proceedings of the ACM Symposium on Compiler Construction, pp 152-161, 1986.

17 M.Haghighat and C.Polychronopoulos, Symbolic Dependence Analysis for High-Performance Parallelizing Compilers. pp 310-330 in Advances in Languages and Compilers for Parallel Processing, Pitman, 1990.

18 B.Creusillet and F.Irigoin, Interprocedural Analyses of Fortran Programs. Parallel Computing Vol. 24, pp 629-648, 1997.

19 P.Havlak and K.Kennedy, An Implementation of Interprocedural Bounded Regular Section Analysis, in IEEE Transactions on Parallel and Distributed Systems, 2.3, 1991.

20 Z.Li, P-C Yew, Program Parallelisation with Interprocedural Analysis. pp 225-244 in Journal of Supercomputing, 2, 1988.

21 M.Burke and R.Cytron, Interprocedural Dependence Analysis and Parallelisation. pp 162-175 in ACM SIGPLAN'86 Symposium on Compiler Construction, June 1986.

22 S.P.Johnson, F.Ali, and M.Cross, Parallelising Of The FAMCALC FEA Code. University of Greenwich Technical Report, 1992.

23 K.McManus, A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs Onto Distributed Memory Parallel Architectures. PhD Thesis, University Of Greenwich, 1995.

24 J.Ferrante, K.J.Ottenstein and J.D.Warren, The Program Dependence Graph and its use in Optimisation. ACM Transactions on Programming Languages and Systems, 9, pp 319-349, 1987.

25 S.P.Johnson, E.W.Evans and M.Cross, Generation of Efficient Parallel Code Using Iteration Space Alignment. University of Greenwich Technical Report, PPRG-98-012.

26 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.