Computer Aided Parallelisation Tools (CAPTools) User Manual

CAPTools Version 2.0Beta

S.P.Johnson, P.F.Leggett, C.S.Ierotheou, E.W.Evans, M.Cross

Parallel Processing Research Group
Faculty of Science and Technology
University of Greenwich
London

Email: captools@gre.ac.uk
Web: http://captools.gre.ac.uk

October 1998

Document: I:\Docs\CAPTools\manual\capman-2.0Beta-056.doc

Release Notes.

CAPTools Version 2.0Beta

23 October 1998

Introduction.

Version 2.0Beta of CAPTools represents a very significant improvement over the previous version 1.2Beta, both in terms of functionality, but also in accuracy, speed, robustness and the level of testing.

Major New Features.

Multi-dimensional structured mesh code partitioning and code generation.

The parallelisation process for a single dimension of the application code data structures can now be repeated for other dimensions, allowing, for example, the mapping of a 3D structured mesh onto a 3D processor topology.

Unstructured mesh code parallelisation.

Parallelisation of unstructured mesh code (such as Finite Element based codes) using a domain decomposition technique is now available in CAPTools. The mesh entities (e.g. finite elements and mesh nodes) are split into sub-domains and one sub-domain is allocated to each processor so that computation relating to entities in a sub-domain are executed only on the owning processor. The partition is determined using the **JOSTLE** partitioning tool aiming to maximise the load balance at the same time as minimising communication overheads between processors. The call to **JOSTLE** is automatically generated within the parallel application code along with calls to small number of CAPLib library routines that handle inter-processor communication etc. The memory reduction facility for unstructured mesh parallelisation is still under development and is not in a state to be included in this release.

Other New Features.

Omega based dependence testing.

In addition to the **Banerjee** dependence test using SIDA as the symbolic inequality proving algorithm, the **Omega** test is now also available. For codes that involve dimension mapping between routines or represent multi-dimensional data with a one-dimensional array, the **Omega** test can provide a more accurate dependence graph. The current implementation is slower that the **Banerjee** test and has undergone a far less rigorous test for this release, so we recommend the use of the **Banerjee** test unless deficiencies in the dependence graph involving mapped arrays are identified.

Automatic routine duplication during partitioning.

The partitioning process has been improved to account for routines that have multiple call sites where the data input differs in the nature of its partition between those call sites. In such cases, a copy of the called routine is automatically made to allow different partitioning, execution control masking, communication generation etc. to be applied as necessary to the copied version(s).

Time restricted dependence analysis.

The dependence analysis attempts to calculate a very accurate representation of the application code which can require a significant execution time. To allow analysis of large application codes, a timed dependence analysis feature is included to provide a dependence graph within a specified time limit. This is coupled with an "**ALERT**" browser to indicate statements whose dependence analysis has been restricted in accuracy and the severity of that restriction. Incremental analysis within a time limit for statements selected in the **ALERT** browser can then be made as necessary.

State Of Release Features.

Feature	State	Notes
Single dimension structured mesh parallelisation	Beta (advanced)	
Multi-dimensional structured mesh parallelisation	Beta	
Unstructured mesh code parallelisation	Alpha	
Structured mesh code memory reduction	Beta	Only for the last dimension that is partitioned using CAPTools
Omega based dependence testing	Alpha	
Overlapping calculation and communication	Beta	Only for the last dimension that is partitioned using CAPTools
Transformations	Alpha	

Contents

List of Figures

Glossary

Dependence	A dependence is frequently referred to within CAPTools. Dependencies can be separated into four different cases True, Control, Anti and Output. A true dependence (also known as a data flow dependence) refers to the usage of a variable after it has been previously assigned. A Control dependence refers to an executable statement whose execution depends on the result of a conditional statement, such as an IF. An Anti dependence refers to a re-assignment of a variable after a previous use, destroying the previously used value. An Output dependence refers to the assignment of a variable after it has been previously assigned, again destroying the old value.
Call Path	A call path is a list of routines and the associated routine reference commands (i.e. call statements and function references) that invoke a particular instance of a routine.
Partition	A partition is the strategy used to subdivide the computational domain into a number of sub-domains, each being assigned to a processor.
Execution Control Mask	An execution control mask is a logical condition (an IF statement) which prefixes an executable statement. The mask expression controls the execution of the statement in parallel and is usually based on the data partition limits assigned to a processor or the processor number itself.
Communication	A data communication is a copy of data from one process’s local memory to another process. This is necessary when data computed by one process is needed as part of a further computation on another process.

Introduction to CAPTools

1 Concepts Of Parallelisation.

In this chapter we examine some of the concepts of parallelisation and look at the fundamental steps involved in parallelising a simple example Fortran program.

1.1 What Is Parallel Processing?

Parallel processing is the use of a number of processors to co-operate in the execution of an application. By dividing the total problem between processors, the overall run time for the application can be dramatically cut, in the ideal case proportional to the number of processors employed. The possibility of exploiting parallelism is dependent on the chosen application code, however in reality almost all codes contain inherent parallelism. Independence (i.e. no data flow) between individual statements, between iterations of loops and between routine calls all provide enough parallelism for a selective choice of the best parallelism to be made. The parallelisation problem thus reduces to the problems of parallelism identification and implementation given the constraints of the parallel hardware available. The computations being performed on each processor must, for functional reasons interact throughout the applications execution. Computations performed by one processor will, for example, often require data calculated on another processor and thus cannot be performed until that data is available to this processor. Such interactions can have a significant effect on the overall performance of a parallel code with processors sitting idle awaiting data they need to use being calculated elsewhere. Access to data from another processor can also incur a significant penalty due to the latency in reaching the memory area concerned, often involving the use of shared hardware mechanisms that provide the connection between processors.

1.2 What Is A Distributed Memory System?

A Distributed Memory System (DMS) consists of a number of processors, each with its own local memory, connected together in a network. This network of processors can be connected in many ways, common examples being pipelines (also known as chains), rings, grids or hypercube. Each local memory is only directly accessible by the associated processor. To access another processor's local memory an indirect route must be taken involving the network interconnection hardware. Use of non-local memory therefore incurs a time penalty in transferring the data across the network thus keeping data local (i.e. placing data on the processor local memory where it will be used) is an important consideration in a DMS parallel application code. The parallel code must take into account the location of data in the processor network and collect required data from the relevant processors. If the program data can be distributed over the processor topology in an organised manner then the collection of required data from other processors should only involve a small local area of the full network, avoiding data movement over large distances and minimising traffic problems as many data requests try to use the same interconnection hardware at the same time.

This movement of data is typically performed by explicit message passing from one processor to another implemented by additional operations added to the executable code. The number of these message passing communications and the volume of data involved, along with the distance the messages have to traverse in the processor topology, have a direct and often disastrous effect on the performance of the parallel code. The hardware overhead incurred when performing a communication consists of two main components. The communication time is proportional to the amount of data being transferred and how remote the original location of that data is to the processor. The other overhead is the communication start up latency which is incurred by every communication regardless of the amount of data involved. In many current DMS the communication start up latency is very significant, thus performance improvements can be achieved by performing few bulk data communications as opposed to numerous small communications.

To alleviate the data transfer communication cost of a DMS, many of the hardware interconnects enable data transfer to be performed at the same time as computation. This allows data communications to be instigated before the data is needed where other computation code can be executed concurrently with the communication before the completion of the communication is ensured (i.e. synchronisation) and calculations using the received data commenced. This facility can dramatically reduce the cost of data transfer, sometimes removing its influence on the parallel code run time. The start up latency cannot be reduced in such a way.

Access to external devices on the host machine can be implemented in several ways. A unique processor can be connected to the host and all external accesses must then be performed by that processor. Alternatively, a subset of the processors can have access to the host where distributed files may be used to allow parallel I/O (i.e. a set of data files that each store a subset of the file that would be used in serial).

1.3 What Is The SPMD Model?

The Single Program, Multiple Data (SPMD) model is where each processor executes the same program on a different subset of the total program data. Typically, each data item (i.e. array element) in the program data is allocated to one processor and every processor is allocated approximately equal numbers of data items. Parallel execution is achieved by forcing each processor to only perform computations that are related to assignment of data elements allocated to that processor. An organised data partition across the processor topology is required to minimise the amount of communication required and allow simple code transformations to implement the SPMD parallel code. For many codes with an organised data partition, much of the data area that is used but not assigned on a processor (i.e. assigned on another processor) can be designated as an overlap area (also known as a halo region). The overlap area is where received data is placed ready for use in computations.

1.4 Example: Parallelisation Of The 1-Dimensional Jacobi Solver.

If we look at a simple diffusion Jacobi example where the problem being solved is

$$tnew(i) = \frac{t(i-1) + t(i+1)}{2} \qquad i = 2, ..., 999$$

with boundary conditions:

t(1) = 1   t(1000) = 100

and the convergence criteria is max[abs(tnew(I) – t(I)] < tolerance

1.4.1 Serial Code

The serial code for this problem could be as follows:

```
      PRINT*,'Enter N and TOL'
      READ*,N,TOL
      DO I=1,N
        T(I) = 0
      ENDDO
      T(1) = 1
      T(N) = 100
 40   CONTINUE
      DO I = 2, N-1
        TNEW(I) = (T(I-1)+T(I+1))/2
      ENDDO
      DIFMAX = 0
      DO I = 1,N
        DIFF= ABS(TNEW(I)-T(I))
        IF (DIFF .GT. DIFMAX) DIFMAX = DIFF
        T(I) = TNEW(I)
      ENDDO
      IF (DIFMAX .GT. TOL) GOTO 40
      END
```

Adhering to the SPMD model, the partition of the array tnew for the above Jacobi example, for 4 processors would require each processor to be allocated a data range of 250 array elements in order for each processor to obtain a balanced work load. In the calculation of the array tnew the array t is used and therefore this array will inherit the same partition range on each processor. Each processor will also require an overlap region because of data assigned on other processors required by the current processor. This will necessitate the communication of data assigned on one processor into the overlap region of their neighbouring processors. Due to the organised partition of the data the overlap areas need only be updated from their neighbouring processors. The data partition of the t partitioned array in comparison with the original unpartitioned array is shown in Figure 1 .

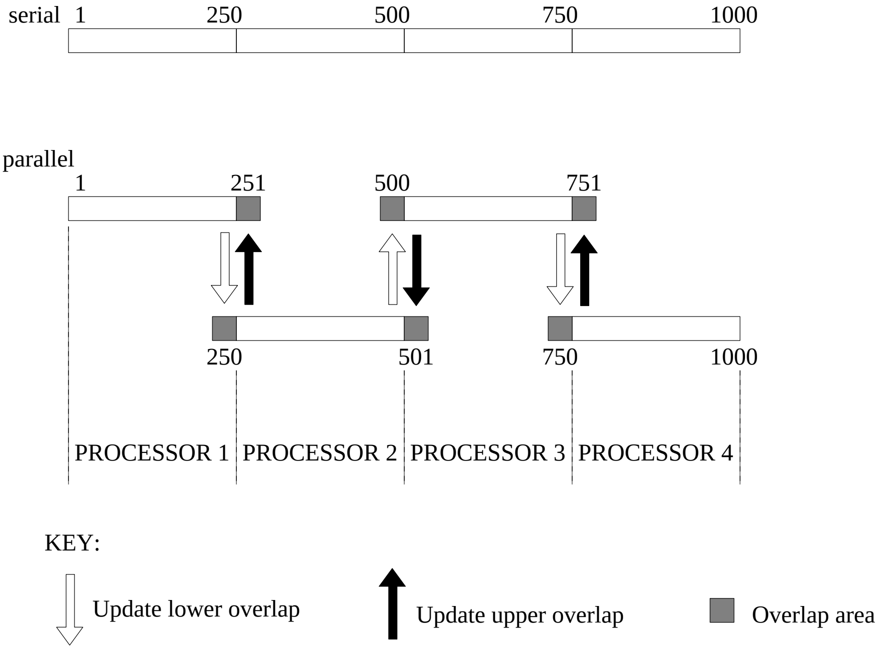


Figure 1 Comparison of an unpartitioned and partitioned 1-D array.

1.4.2 Parallel Code

The parallel code for this problem (executed on every processor) could be as follows :

```
C     Assign unique id number to each processor
      PRINT*,'Enter N and TOL'
      READ*,N,TOL
C
C     Broadcast N and TOL to all processors.
C
C     Set up data partition range details for this processor.
C     DO I= data range of this processors partition
        T(I) = 0
        ENDDO
      IF (processor owns element 1) T(1) = 1
      IF (processor owns element N) T(N) = 100
 40   CONTINUE
C
C     Update the lower overlap from previous processor.
C     Update the upper overlap from next processor.
C     DO I = data range of this processors partition
         TNEW(I) = (T(I-1)+T(I+1))/2
      ENDDO
      DIFMAX = 0
      DO I = data range of this processors partition
        DIFF= ABS(TNEW(I)-T(I))
        IF (DIFF .GT. DIFMAX) DIFMAX = DIFF
        T(I) = TNEW(I)
      ENDDO
C     Collect and find the maximum value of DIFMAX from every processor.
      IF (DIFMAX .GT. TOL) GOTO 40
```

```
      END
```

Comparing the above parallel code against the original serial code it can be seen that the parallel code is still very similar to the original serial code. The only difference in the two codes are additional statements added to the code for communication of data and the alteration of the loop limits so that each processor will only work on the partition range allocated to each processor. There is also additional code at the start of the program to set up the parallel network and the data partition for the number of processors available. The following section describes these additional statements in some detail.

1.4.3 Parallelisation Statements Added In The Jacobi Example

The first stage is the initialisation of a network for the program to execute upon. The initialisation of this network will return to each processor a unique identity number. This will allow each processor to know its position in the network and assist in the communication process. The initialisation of the network is invoked by a call to the CAPTools set up primitive CAP_INIT.

The next stage in the parallel code is the layout of the data range of each processor. A call to the CAPTools set up primitive CAP_SETUPPART given the minimum and maximum range of the serial data range and the number of processors will return to each processor in the network its own unique value for the minimum and maximum value for the partitioned data range. If the above Jacobi example was partitioned onto 4 processors then CAP_SETUPPART would return to processor 1 the partition range 1 to 250, processor 2 the partition range 251 to 500, processor 3 the partition range 501 to 750 and processor 4 the partition range 751 to 1000.

As mentioned earlier each processor has an overlap region for data that was assigned on another processor. These overlap areas are updated when necessary. This overlap region is updated by invoking a call to the CAPTools communication primitive CAP_EXCHANGE. This communication primitive will send data to its neighbouring processor's overlap area while also receiving data into its own overlap region from the neighbouring processors, e.g.

```
      CALL CAP_EXCHANGE(t(501),t(251),1,2,CAP_RIGHT)
      CALL CAP_EXCHANGE(t(250),t(500),1,2,CAP_LEFT)
```

The first statement exchanges the data of t(251),on processor 2, of length 1 and type 2 (i.e. REAL) by sending it in the direction of processor 1 and places it into t(251), an overlap location on processor 1. Also, at the same time processor 2 will receive from the right (using CAP_RIGHT direction) into t(501), an overlap location. A similar procedure follows for the second statement.

Also, in the Jacobi example it is necessary to calculate the convergence criteria based on DIFMAX. Since the data range has been split onto 4 processors then each processor will calculate its own local value for DIFMAX, however, it is necessary to calculate the global value of DIFMAX. This may be done by using the CAPTools communication primitive CAP_COMMUTATIVE. For example,

```
      CALL CAP_COMMUTATIVE(DIFMAX,2,CAP_RMAX)
```

This will take the input of the local value DIFMAX (which is a REAL i.e. type 2) of each processor and calculate the global convergence criteria using CAP_RMAX (which is a function to calculate the maximum of two real numbers). The CAP_COMMUTATIVE call then returns the global value of DIFMAX to all processors.

Other CAPTools communication primitives that would be used in the Jacobi example are CAP_SEND and CAP_RECEIVE. The use of these in the program would be after the values N and TOL were entered by the user. Assuming only processor 1 has input and output facilities, then processor 1 would have to communicate the values of N and TOL to all other processors. This would be done in the following way,

```
      CALL CAP_RECEIVE(TOL,1,2,CAP_LEFT)
      CALL CAP_SEND(TOL,1,2,CAP_RIGHT)
```

The CAP_RECEIVE call would receive TOL of message length 1 and type 2 (REAL) from the left (LEFT), while the CAP_SEND call would send TOL of length 1 to the right (CAP_RIGHT).

1.4.4 The Complete Parallel Code For The Jacobi Example.

```
      INTEGER CAP_LEFT,CAP_RIGHT
      PARAMETER (CAP_LEFT=-1,CAP_RIGHT=-2)
      REAL T(0:1001),TNEW(1000),TOL
      INTEGER N,CAP_LT,CAP_HT
      EXTERNAL CAP_RMAX
      REAL CAP_RMAX
      INTEGER CAP_PROCNUM,CAP_NPROC
      COMMON /CAP_TOOLS/CAP_PROCNUM,CAP_NPROC
      CALL CAP_INIT
      IF (CAP_PROCNUM.EQ.1)PRINT*,'ENTER N AND TOL'
      IF (CAP_PROCNUM.EQ.1)READ*,N,TOL
      CALL CAP_RECEIVE(TOL,1,2,CAP_LEFT)
      CALL CAP_SEND(TOL,1,2,CAP_RIGHT)
      CALL CAP_RECEIVE(N,1,1,CAP_LEFT)
      CALL CAP_SEND(N,1,1,CAP_RIGHT)
      CALL CAP_SETUPPART(1,N,CAP_LT,CAP_HT)
      DO I=MAX(1,CAP_LT),MIN(N,CAP_HT),1
        T(I)=0.0
      ENDDO
      IF (1.GE.CAP_LT.AND.1.LE.CAP_HT)T(1)=1
      IF (N.GE.CAP_LT.AND.N.LE.CAP_HT)T(N)=100
40    CONTINUE
      CALL CAP_EXCHANGE(T(CAP_HT+1),T(CAP_LT),1,2,CAP_RIGHT)
      CALL CAP_EXCHANGE(T(CAP_LT-1),T(CAP_HT),1,2,CAP_LEFT)
      DO I=MAX(2,CAP_LT),MIN(N-1,CAP_HT),1
        TNEW(I)=(T(I-1)+T(I+1))/2.0
      ENDDO
      DIFMAX=0.0
      DO I=MAX(1,CAP_LT),MIN(N,CAP_HT),1
        DIFF=ABS(TNEW(I)-T(I))
        IF (DIFF.GT.DIFMAX) DIFMAX=DIFF
        T(I)=TNEW(I)
      ENDDO
      CALL CAP_COMMUTATIVE(DIFMAX,2,CAP_RMAX)
      IF (DIFMAX.GT.TOL) GOTO 40
      DO 50  I=1,N,1
        IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=*,FMT=*)TNEW(I)
50    CONTINUE
      END
```

2 CAPTools Parallel Primitives

This chapter describes the set of parallel primitives that CAPTools uses in generated code. The set of routines can be divided into five categories

- Initialisation primitives
- Synchronous communication primitives
- Asynchronous communication primitives
- Parallel commutative operation primitives
- Global communication primitives

All CAPTools primitives are generic. The user links their parallel object code with a specific version of the CAPTools library for the machine and parallel environment they wish to run on. Depending on the user's choice the CAPTools library make calls to the required underlying communication library, e.g. Transtech i860 Toolset library, PVM 2.4+, PVM 3.2+, MPI ,CRAY SHMEM, etc.

2.1 CAPTools Initialisation Primitives.

The CAPTools initialisation primitives must be used before the call to any other primitives. These are required for the initialisation of a network of processes/processors for the parallel program to execute on. There is also the requirement for the layout of the data range of each processor. A call to the CAPTools set up primitive CAP_INIT will set up the required network of processors. The set up primitive CAP_SETUPPART will partition the data for a pipe or ring network given the minimum and maximum range of the serial data range and the number of processors and will return to each processor in the network its own unique value for the minimum and maximum value for the partitioned data range.

Summary of the CAPTools set up primitives

CAP_INIT	This will set up the required network of processors: either a pipe, ring, 2-dimensional grid or 3-dimensional grid. The default set up is for a pipe or ring.
CAP_SETUPPART	This will set up the minimum and maximum range of the partitioned variables for each processor in a pipe or ring network.
CAP_SETUPDPART	This will set up the minimum and maximum range of the partitioned variables for each processor in a grid network for each dimension of a multi-partitioned code.

2.2 CAPTools Communication Primitives.

There are three basic process to process communication primitives in CAPTools, SEND, RECEIVE and EXCHANGE. Variants of these implement synchronous (blocking), asynchronous (non-blocking), cyclic, and buffered communications.

2.2.1 Synchronous (Blocking) Communications.

The routines CAP_SEND, CAP_RECEIVE, and CAP_EXCHANGE will all wait for the communication to complete before continuing execution of a program. Each communication primitive will automatically detect if a processor is on the end of a pipe or on the edge of a grid and will not execute a communication. The parameters for these communication primitives are as follows,

```
CAP_SEND(Send_Address, Length, Type, Direction)
```

```
CAP_RECEIVE(Receive_Address, Length, Type, Direction)
CAP_EXCHANGE(Receive_Address, Send_Address, Length, Type, Receive_Direction)
```

Where Send_Address is the start address of the data to send; Receive_Address is the start address to receive the data; Length is the data length to be communicated; Type is the data type; Direction is the direction defined for communication between two processors.

Type is an integer which is defined as one of the following:- 1 - INTEGER   2 - REAL   3 - DOUBLE   4 - COMPLEX   5 - LOGICAL   6 - CHARACTER

Figures and illustrate the three basic communication primitives for and array A between two processors labelled i and i+1.

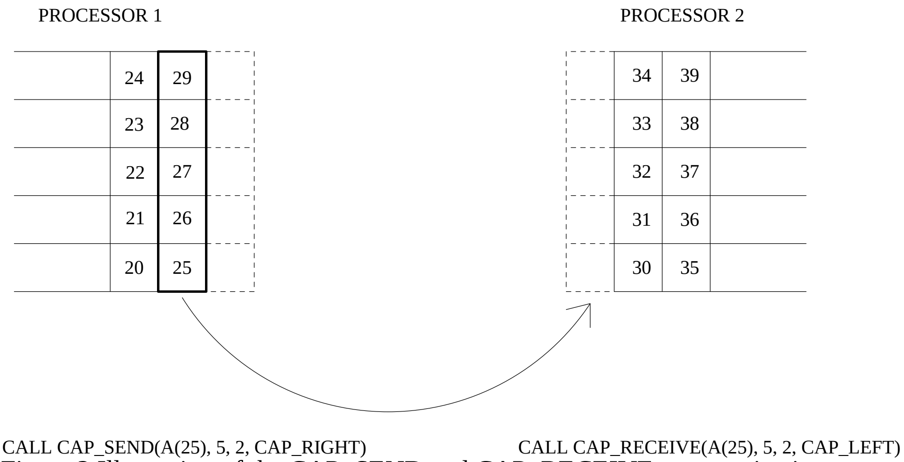


Figure 2 Illustration of the CAP_SEND and CAP_RECEIVE communications.

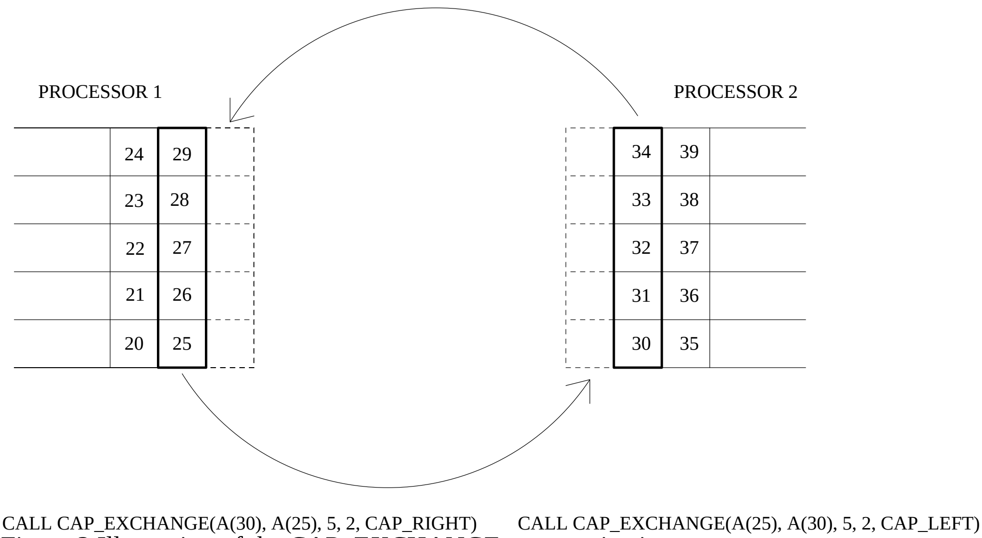


Figure 3 Illustration of the CAP_EXCHANGE communication.

2.2.2 Asynchronous (Non-Blocking) Communication.

The routines CAP_ASEND, CAP_ARECEIVE and CAP_AEXCHANGE are the asynchronous versions of synchronous communication, which allow the program to initialise the communication then returning to continue execution of the program while the communications are being performed. These asynchronous communications may be used in conjunction with the synchronisation communication primitives

CAP_SYNC_SEND, CAP_SYNC_RECEIVE and CAP_SYNC_EXCHANGE to ensure the communication has completed. The parameters for these communication primitives are as follows:

```
CAP_ASEND(Send_Address, Length, Type, Direction, Send_Synchronisation)

CAP_ARECEIVE(Receive_Address, Length, Type, Direction,  Receive_Synchronisation)

CAP_AEXCHANGE(Receive_Address, Send_Address,   Length, Type, Direction,
              Send_Synchronisation, Receive_Synchronisation)endtrivlist
```

Where Send_Synchronisation is the synchronisation value for the send and Receive_Synchronisation is the synchronisation value for the receive. These values are used in calls to CALL_SYNC_SEND, CALL_SYNC_RECEIVE and CAP_SYNC_EXCHANGE to verify the completion of a particular asynchronous communication.

2.2.2.1 Example of Synchronous and Asynchronous Send Communication.

The following example shows the difference in coding synchronous and asynchronous SEND:

```
C     Synchronous code
      DO I=1,NITER
        CALL CAP_SEND(A(I),5,2,CAP_LEFT)
        Code which does not overwrite communicated data
        ...
        ...
        Calculation overwriting communicated data
        A(I) = A(I) + 1
      ENDDO

C     Asynchronous code
      DO I=1,NITER
        CALL CAP_ASEND(A(I),5,2,CAP_LEFT,CAP_SYNCS1)
        Code which does not overwrite communicated data
        ...
        ...
        CALL CAP_SYNC_SEND(CAP_LEFT,CAP_SYNCS1)
        Calculation overwriting communicated data
        A(I) = A(I) + 1
      ENDDO
```

In the case of the synchronous SEND the execution is as follows:-

1. Carry out `CAP_SEND` execution to completion
2. Execute code which does not overwrite communicated data
3. Execute code which does overwrite communicated data

In the case of the asynchronous SEND the execution order is as follows:-

4. Carry out `CAP_ASEND` execution in background.
5. Execute code which does not overwrite the data communicated by `CAP_ASEND`.
6. The execution of `CAP_SYNC_SEND` ensures that the communication started by `CAP_ASEND` is completed.
7. Execute code which does overwrite communicated data.

The CAP_SYNC_SEND call verifies the completion of a particular asynchronous communication by means of the parameter CAP_SYNCS1.

2.2.2.2 Example of Synchronous And Asynchronous Receive Communication.

The following example shows the difference in coding synchronous and asynchronous receives,

```
C     Synchronous Code
      DO I=1,NITER
        CALL CAP_RECEIVE(A(I),5,2,CAP_RIGHT)
C       Code which does not overwrite communicated data
       ...
       ...
C       Calculation overwriting communicated data
        B = A(I)
      ENDDO

C    Asynchronous Code
     DO I=1,NITER
       CALL CAP_ARECEIVE(A(I),5,2,CAP_RIGHT,CAP_SYNCR1)
C      Code which does not overwrite communicated data
       ...
       ...
C      CALL CAP_SYNC_RECEIVE(CAP_RIGHT,CAP_SYNCR1)
       Calculation overwriting communicated data
       B = A(I)
     ENDDO
```

In the case of the synchronous RECEIVE the execution is as follows:-

1. Carry out `CAP_RECEIVE` execution to completion.
2. Execute code which does not use communicated data.
3. Execute code which does use communicated data.

In the case of the asynchronous RECEIVE the execution order is as follows:-

1. Carry out `CAP_ARECEIVE` execution in background.
2. Execute code which does not use data being communicated.
3. The execution of `CAP_SYNC_RECEIVE` ensures that `CAP_ARECEIVE` is completed.
4. Execute code which does use communicated data.

The CAP_SYNC_RECEIVE call verifies the completion of a particular asynchronous communication by means of the parameter CAP_SYNCR1.

2.2.2.3 Example of Synchronous and Asynchronous Exchange Communication.

The following example shows the difference in coding synchronous and asynchronous EXCHANGE,

```
C     Synchronous Code
      DO I=1,NITER
        CALL CAP_EXCHANGE(A(CAP_LA-1),A(CAP_HA),25,2,CAP_LEFT)
C       Code which does not use communicated data
        ...
        ...
C       Calculation using communicated data
        DO J=MAX(1,CAP_LA),MIN(200,CAP_HA)
```

```
          C = A(J) + A(J-1)
        ENDDO
      ENDDO

C     Asynchronous Code
      DO I=1,NITER
        CALL CAP_AEXCHANGE(A(CAP_LA-1),A(CAP_HA),25,2,CAP_LEFT,CAP_SYNCS1,CAP_SYNCR1)
C       Code which does not use or overwrite communicated data
        ...
        ...
        CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SYNCS1,CAP_SYNCR1)
C       Calculation using communicated data
        DO J=MAX(1,CAP_LA),MIN(200,CAP_HA)
          C = A(J) + A(J-1)
        ENDDO
      ENDDO
```

In the case of the synchronous EXCHANGE the execution is as follows:-

1. Carry out CAP_EXCHANGE execution to completion.
2. Execute code which does not use the data being communicated.
3. Execute code which does use communicated data.

In the case of the asynchronous EXCHANGE the execution order is as follows:-

1. Carry out CAP_AEXCHANGE execution in background.
2. Execute code which does not use the data being communicated.
3. The execution of CAP_SYNC_EXCHANGE ensures that the CAP_AEXCHANGE above has completed.
4. Execute code which does use communicated data.

The CAP_SYNC_EXCHANGE call verifies the completion of a particular asynchronous communication by means of the parameters CAP_SYNCS1 and CAP_SYNCR1.

2.2.3 Buffered Synchronous (Blocking) Communications.

The routines CAP_BSEND, CAP_BRECEIVE and CAP_BEXCHANGE are the buffered versions of the synchronous communications. These place the data into a buffer before sending and extract the data from the buffer after receiving.

```
CAP_BSEND(Send_Address, Length, Stride, NumStrides, Type, Direction)
CAP_BRECEIVE(Receive_Address, Length, Stride, NumStrides, Type, Direction)
CAP_BEXCHANGE(Receive_Address, Send_Address, Length, Stride, NumStrides, Type,
              Receive_Direction)
```

Where Stride is the distance between each block of data to be buffered and the NumStrides is the number of strides to buffer.

2.2.4 Cyclic Communications

The routines CAP_CSEND and CAP_CRECEIVE provide communications for use on ring networks. They are the same as the CAP_SEND and CAP_RECEIVE apart from the fact the end of a pipe or the edge of a grid are actually connected together thus allowing communication from one edge to another. This allows data to be passed a shorter distance than otherwise could be possible. The parameters are as follows,

```
CAP_CSEND(Send Address, Length, Type, Direction)
CAP_CRECEIVE(Receive_Address, Length, Type, Direction)
```

2.2.5 Buffered Asynchronous (Non-Blocking) Communications

The routines CAP_ABSEND, CAP_ABRECEIVE and CAP_ABEXCHANGE perform the combination of asynchronous and buffered communication. The parameters are,

```
CAP_ABSEND(Send_Address, Length, Stride, NumStrides, Type, Direction,
           Send_Synchronisation)
CAP_ABRECEIVE(Receive_Address, Length, Stride, NumStrides, Type, Direction,
              Receive_Synchronisation)
CAP_ABEXCHANGE(Receive_Address, Send_Address, Length, Stride, NumStrides, Type,
               Receive_Direction, Send_Synchronisation, Receive_Synchronisation)
```

2.2.6 Synchronisation Primitives

The routines CAP_SYNC_SEND, CAP_SYNC_RECEIVE and CAP_SYNC_EXCHANGE provide synchronisation points that can be used in association with the asynchronisation communications CAP_ASEND, CAP_ARECEIVE, CAP_AEXCHANGE, CAP_ABSEND, CAP_ABRECEIVE and CAP_ABEXCHANGE to ensure that the communication has completed before allowing the execution of the program to continue.

```
CAP_SYNC_SEND(Direction, Send_Synchronisation)
CAP_SYNC_RECEIVE(Direction,  Receive_Synchronisation)
CAP_SYNC_EXCHANGE( Direction, Send_Synchronisation, Receive_Synchronisation)
```

2.2.7 Dual Exchange Communication

This routine CAP_DUAL_EXCHANGE allows an exchange of data to be accomplished in both directions of the network. The parameters are as follows,

```
CAP_DUAL_EXCHANGE(Receive_Address1, Send_Address1, Receive_Address2, Send_Address2,
                  Length, Type, Receive_Address1_Direction)
```

2.2.8 Parallel Commutative Operations

The routine CAP_COMMUTATIVE is used to calculate the global value of each processors commutative operations such as max, min, sum and product. The routine CAP_DCOMMUTATIVE also calculates the global value of each processor commutative operations but for a particular dimension in a multi-dimensonally parallelised code. The parameters are as follows :

```
CAP_COMMUTATIVE(Value, Type, Function)
CAP_DCOMMUTATIVE(Value, Type, Function, Direction)
```

Where Value on input is the processor local value of the variable and the global Value on output. Function is the commutative function to be used such as CAP_RADD (real summation), CAP_IMAX (integer maxima calulation) etc.

2.2.9 Global Communications.

Finally there is the CAP_BROADCAST communication primitive which broadcasts global data from the input/output processor to all other processors. The parameters for this are:

```
CAP_MBROADCAST(Address, Length, Type)
```

Where Address is the start address of the data to send and receive.

2.3 Summary Of The CAPTools Primitives

CAP_SEND	Synchronous (blocking) SEND
CAP_RECEIVE	Synchronous (blocking) RECEIVE
CAP_ASEND	Asynchronous (non-blocking) SEND
CAP_ARECEIVE	Asynchronous (non-blocking) RECEIVE
CAP_BSEND	Buffered synchronous (blocking) SEND
CAP_BRECEIVE	Buffered synchronous (blocking) RECEIVE
CAP_CSEND	Cyclic SEND for use with ring networks
CAP_CRECEIVE	Buffered Cyclic RECEIVE for use with ring networks
CAP_ABSEND	Buffered asynchronous (non-blocking) SEND
CAP_ABRECEIVE	Buffered asynchronous (non-blocking) RECEIVE
CAP_SYNC_SEND	Wait for completion of asynchronous SEND
CAP_SYNC_RECEIVE	Wait for completion of asynchronous RECEIVE
CAP_EXCHANGE	Synchronous (blocking) EXCHANGE
CAP_AEXCHANGE	Asynchronous (non-blocking) EXCHANGE
CAP_BEXCHANGE	Buffered synchronous (blocking) EXCHANGE
CAP_ABEXCHANGE	Buffered asynchronous (non-blocking) EXCHANGE
CAP_DUAL_EXCHANGE	Synchronous EXCHANGE in both directions
CAP_SYNC_EXCHANGE	Wait for completion of asynchronous exchange
CAP_COMMUTATIVE	Perform specified global COMMUTATIVE operation
CAP_DCOMMUTATIVE	COMMUTATIVE operation for a particular dimension
CAP_BROADCAST	Global broadcast of data to all processors from first processor

3 Influences On The Performance Of A Parallel Code.

This chapter describes some of the influences that can effect the performance of parallel code. These include pipelines, load balance and communication latency.

3.1 Pipelines

A pipeline in a parallel code involves each processor performing the same set of operations on a successive stream of data. Pipelines occur due to the recurrence of data within a loop. The main disadvantages are that during the pipeline process some processors will be idle at the start up and shut down stages. Also, if one processor in the pipeline has a higher computation load due to a conditional, then this bottleneck causes a degradation in performance. However, pipelines are commonly used and are often essential when it is desirable to ensure correctness of the parallel code. We now show a simple example of a loop which has been parallelised using a pipeline:

```
C     Serial code           C    Parallel code
      DO I=2,NI                  DO I=2,NI
        DO J=2,NJ                  CALL CAP_RECEIVE(A(I,CAP_LA-1),1,2,CAP_LEFT)
          A(I,J) = A(I,J-1)        DO J=MAX(2,CAP_LA),MIN(NJ,CAP_HA)
        ENDDO                        A(I,J) = A(I,J-1)
      ENDDO                        ENDDO
                                   CALL CAP_SEND(A(I,CAP_HA),1,2,CAP_RIGHT)
                                 ENDDO
```

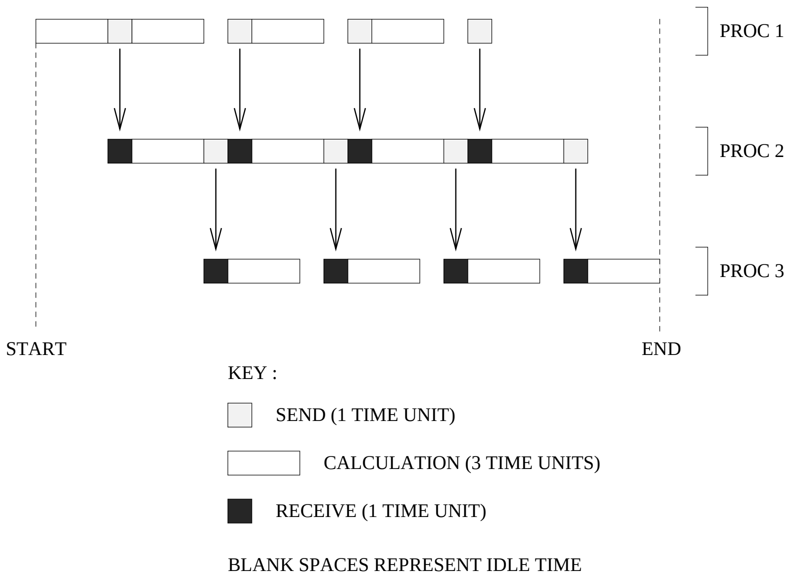


Figure 4 Execution of a synchronous pipeline.

The pipeline can also be shown using a diagram, see Figure 4. The first processor (PROC 1) starts the pipeline by executing its calculations and then sending data to the next processor (PROC 2), before proceeding with its next calculation. The next processor will then receive its data and proceed to execute its calculations before passing its data along to the next processor, After N iterations all the processors are working in parallel, where N is the number of processors.

3.2 Load Balancing

If a parallel application code is "load balanced", this means that the computational workload assigned to each processor is exactly the same in solving the given problem. For example, if a three-dimensional structured grid problem of dimensions 10x10x50 was to be solved in parallel using a partition in the third dimension of each array of this definition, then the execution on 5 processors would mean the following assignment ranges:-

```
                          ASSIGNMENT RANGE
              Index 1    Index 2    Index 3    Cells
processor 1:  1 - 10     1 - 10      1 - 10    1000
processor 2:  1 - 10     1 - 10     11 - 20    1000
processor 3:  1 - 10     1 - 10     21 - 30    1000
processor 4:  1 - 10     1 - 10     31 - 40    1000
processor 5:  1 - 10     1 - 10     41 - 50    1000
```

In this case each processor has the same number of cells on which to perform its relevant computation. In general the equal distribution of computational effort amongst the processors gives rise to a load balanced execution. However, if the same problem was solved using 4 processors, then the distribution of effort is as follows:-

```
                          ASSIGNMENT RANGE
              Index 1    Index 2    Index 3    Cells
processor 1:  1 - 10     1 - 10      1 - 13    1300
processor 2:  1 - 10     1 - 10     14 - 26    1300
processor 3:  1 - 10     1 - 10     27 - 38    1200
processor 4:  1 - 10     1 - 10     39 - 50    1200
```

Here the effort is not evenly distributed and it is said to be "load unbalanced" or "not load balanced". The parallel execution in this case will generally mean that processor 3 and processor 4 will be idle for part of the execution because the respective workloads are less than those assigned to processors 1 and 2.

A cautionary note when discussing load balancing - it is possible that the characteristics of some computational mechanics codes means that the computation performed for a single cell may depend on the properties or the nature of that cell (in the context of the application being solved). For example, it may be that any cell with a porosity factor greater than liquid_max is treated more like a solid and anything less than liquid_min is treated more like a gas. Other factors could include boundary condition equations, the use of unstructured elements, etc, in all these cases the weighting related to the computation for all cells may be different.

3.3 Communication Of Data

For some parallel hardware platforms the way in which the communication of data between processors is carried out is crucial to a successful parallel performance. In its simplest form the operations for the communication of data are:- a)The "set up" or preparation of the data elements to be communicated, and b)The actual sending (or receiving) of data elements from another processor.

3.3.1 Communication start up latency.

In the preparation of data to be communicated, there is often a communication start up latency associated with this operation, this appears as an overhead in the total communication time since no data is actually communicated at this stage. In some heterogeneous parallel systems this overhead may be significant (of the order of hundreds of microseconds). It is possible, through experimentation, to determine at which point the total communication time is dominated by the actual sending of data rather than the setting up operation.

3.3.2 Communication speed

The data transfer speed determines the effectiveness of the actual data communication. The faster the data communication the lower the cost in executing a data communication. Therefore, if the communication latency is not significant a large number of small messages can be communicated efficiently.

It may be possible to hide some or all of the communication transfer of data if there is software and hardware available to allow asynchronous communications to be performed.

In summary, parallel hardware systems which have a high communication start up latency need to be programmed such that:- as few a number of communications are executed; the communications are bulk in nature - the message lengths are as long as possible taking advantage of array elements which are contiguous in data memory; or, the communications are buffered - the messages are made as long as possible by placing the data elements to be communicated in a buffer and then sending the buffer as a single communication; and whenever possible, the use of asynchronous communications can be of benefit in hiding the communications overheads.

Parallel hardware systems which have a low communication latency can be programmed such that messages with small lengths can be used to good effect since the cost of setting up a message is not significant. In this case the communications can be point (or single data) in nature. Although it may be equally advantageous to either buffer the data to be communicated or if possible, to prepare the communications in a bulk form.

The choice of transferring data as a series of single data items, as a block of contiguous elements or as an accumulated buffer is very much hardware dependent. The cost of gathering data into a buffer and scattering the buffer at the final destination may be more effective than a series of much smaller single messages if the start up latency is a critical factor. Again, the choice should be made through experience and a series of experiments.

4 The Philosophy Of CAPTools

Converting serial software to exploit a distributed memory machine architecture is an extremely complicated task. For an efficient parallel implementation the entire code must be transformed to map both the program data and the related computations across the distributed memory architecture. This can often take months to accomplish manually with an extensive period of implementation of the mapping strategy to the code invariably followed by a considerable amount of time debugging the parallel version. Mapping strategies, while now available, require expert knowledge and experience on behalf of the code paralleliser to maximise the efficiency of the parallel implementation. Having implemented and debugged the parallel code, this must now be maintained and possibly migrated to other hardware systems, magnifying the task of the code paralleliser.

The crucial requirement for the parallelisation of an application code is to satisfy the need of the application code user. Such users do not want to know their code is running in parallel, they just want the improvement in performance gained by efficient parallel execution. A measure of the success of a parallel implementation is the usability of the code. An alteration to the user input requirements of the parallel version may well inhibit acceptance of that parallel code. Inefficient parallelisations will be superseded by more efficient versions preferred by the application code user regardless of the route used to create each version. Therefore, it is clear that sacrificing parallel efficiency to simplify the parallelisation process will be unsatisfactory to the end user, and will soon render the inefficient parallel version obsolete.

A major reason for the limited use of parallel processing is the number of parallel application codes available. The initial cost of mapping an application code onto a parallel system manually is one major reason for the reluctance of application code authors in creating parallel versions. Perhaps even more significant is that the application code author needs to have a framework whereby the code can be maintained, and updated throughout its life. This would often need only simple alterations to the serial code, so an incremental update of the parallel code would be an advantage, rather than a complete rewrite. However, in some cases the changes to the serial code may require a parallelisation of the whole code in order to effectively parallelise new sections of the code. For a manual parallelisation, the maintenance problem becomes almost insurmountable. The release of a parallel code should not follow a year after the release of the serial version, nor can the serial release be held back months until the parallel code is available. Without tools to aid the initial parallelisation and subsequent maintenance, the proliferation of parallelised application codes will not occur.

The ideal solution for the code paralleliser (whether that be the code author, a code user or a third party) would be to use a fully automatic parallelising compiler. In the short term, such compilers are unavailable and even in the longer term, the inherent difficulty of DMS parallelisation and complex coding techniques, along with user insight in the parallelisation process indicate that code from such a route would not be of the highest quality and would therefore soon become superseded. The other extreme of a manual parallelisation is also unacceptable, particularly to the code author. The solution would therefore appear to be a compromise, utilising the best qualities of both the compiler technology and human insight.

Having established the need for tools to aid the above problems, what is not clear is the nature of these tools. The Computer Aided Parallelisation Tools (CAPTools) were conceived in recognition of the fact that the process of source code parallelisation could not in general, be fully automated giving an acceptable level of speed up, whilst minimising the code parallelisers effort in the parallelisation. The code parallelisers role is made one of inspection. The results of the automatic parallelisation, may be altered by the answering of simple questions posed by the system, and by making fundamental decisions on the parallelisation strategy etc. These user decisions are an integral part of the parallelisation, vital to the success of a subsequent parallel version. For example, the answering of simple questions can help to improve the quality of the overall data flow representation of the serial code, and hence having a direct effect on almost all stages of the parallelisation. Effective interaction with the code paralleliser is dependent on their understanding of what has been done, and precisely what they are being asked at any stage. The tools therefore enable a two way dialogue, between computer and user, forming an essential component of the whole activity with the system providing explanations at every stage of the parallelisation.

The tools are intended to aid the user by automating the code conversion in both the detailed decisions that need to be made and the actual code transformation process. The user can then concentrate their effort on optimisation of the parallel code, or investigation of new parallel solver techniques to further improve code performance. The information provided by the tools also greatly aid the optimisation process by indicating, for example, why a communication is needed, or by providing information on why a particular algorithm exhibits poor performance in parallel.

The code produced by CAPTools must be familiar to the user at every stage in order to maximise their ability to take an effective part in the two way interaction. Questions posed about code transformed beyond the recognition of the user, may as a result, be deemed unanswerable. To ensure recognition, the appearance of the code is kept as close to that of the serial as possible with any significant transformation under the control of the user.

4.1 CAPTools - Is It A Compiler ?

CAPTools has two distinct differences from a traditional parallelising compiler. The first, mentioned previously, is the essential requirement for user interaction. The second factor is the constraint on analysis time placed on a compiler as opposed to a parallelising tool. To be commercially acceptable, a compiler must perform a parallelisation in a fairly short space of time (typically the order of minutes). However, in an interactive system, the key restriction is that of human time where a low quality analysis will greatly increase the required human time. Since the analysis will camouflage key information the user will require considerable effort to optimise the generated parallel code, whose quality has been directly diminished by the poor analysis. Our experience has been that parallelisation with a poor analysis is often disastrous with little the user can do to redeem the slow down that is invariably exhibited by the parallel code produced. "Inaccuracies", false relationships assumed in order to ensure correct code in the data flow representation, can be fatal. This can cause serialisation, false implications in code generation and reduce code clarity to the user. It can also cause extra communication of data and so slow the parallel execution of the code. We therefore employ an in depth interprocedural, symbolic, data-flow based analysis to give an accurate representation of actual data movement within the code. This analysis can take a number of hours to perform for large codes or codes with complex control flow. However, the user time required to complete the parallelisation is then not excessive. CAPTools allows the user to analyse a large code in a few hours that can subsequently be saved to a database. The database allows the user the reassurance of a backup should a restart of the session be needed, due to fundamental alterations in the parallelisation strategy that may be specified by the user.

4.2 User Interaction Within CAPTools

The process of user interaction within the CAPTools is two-way. The user is able to add knowledge that relates to program variables and inquire as to why dependencies are set and if possible, answer any related questions proving the dependence does not exist. Additionally, editing at all stages can be performed by the user to adjust every stage of the parallelisation process. CAPTools presents the user with a set of questions identified during various stages of the parallelisation process whose truth could not be determined from the current knowledge base. CAPTools also provides explanations for decisions made during all stages in the parallelisation to promote user understanding improving the quality of the user interaction. This is vital to the optimum production of the best possible parallel code. The user interaction in CAPTools can best be described by highlighting three ways of user interaction.

4.2.1 User Volunteered Information

The volunteering of information by the user can be carried out in a number of ways. Some information may be obvious to the user although not held implicitly in the code. Common examples of such information are the ranges that variables can take, such as dimensions of meshes in computational domains where the user knows that the number of cells in any direction is positive and usually a significantly high number (i.e. non-trivial). Many variables are read in at runtime either from the user or from input files thus these variables may be familiar to the user or described in the application code literature. This knowledge will aid in the initial analysis of the source code often significantly reducing the number of dependencies set by eliminating assumed dependencies that would previously have been set. A case in point is shown below. This code section involves a dependence between statements s1 and s2.

```
s1:   B(2)= ......
s2:   ...= B(IN) .......
```

Without user information B(IN) could be the same memory location asB(2) since IN could be equal to two. If the user knew for a fact that IN would always be positive and non-trivial (for example greater than five) in this code then no relationship exists and the dependence could be removed.

4.2.2 User Questions

During the dependence analysis, CAPTools generates a list of questions based on unresolved inequalities encountered during the analysis that led to dependencies being assumed. These questions are presented to the user after the analysis for inspection and, if possible answers to be provided allowing dependencies to be eliminated in a subsequent incremental analysis. The questions are sorted into occurrence order so that the most frequent and thus most significant questions are asked first. Browsing facilities are available to help the user to answer any questions asked by CAPTools.

4.2.3 Key Questions

To identify important questions, the user can browse serial loops and identify serialising dependencies. Following the selection of such a dependence, CAPTools will present the user with details of that dependence along with any questions caused by unresolved inequalities in the dependence test for that dependence. These key questions can then be answered if possible, allowing the dependence to be removed and possibly proving that the loop is parallelisable following an incremental analysis.

4.3 Code Development Cycle

The following is a simple walk through the development cycle of parallel code using CAPTools. It is important to realise that the cycle is an iterative process, this is necessary to produce effective parallel code.

1. CAPTools reads in semantically correct FORTRAN serial source code.
2. The user volunteers information to CAPTools.
3. A level of analysis is selected by the user and sets it running.
4. Questions generated by CAPTools are answered by the user.
5. The user browses the dependence graph produced, identifying the key questions to answer.
6. An incremental analysis is performed if necessary (return to stage 4).
7. Based on the dependence graph browser and the other available browsers an array index is chosen.
8. The partition is based on the index of an array in a routine. This general partition is inherited throughout the code to all relevant arrays.
9. The partition is then browsed and accepted.
10. Parallel code is generated and inspected for communication placement.

User adjustment to the partition details or a fundamental alteration to another index or another array.(return to stage 7 or 8).

Either :

Generate code, run and identify key sections to interrogate.

Or :

Set up for a further partitioning (return to stage 7)

13. Examine and adjust the code if necessary.

After all stages of the parallelisation, the current state can be saved in a database which can be returned to if the subsequent parallelisation decisions did not produce satisfactory code. The success of each stage is dependent on the success of all previous stages, thus communication generation could, for example identify key dependencies that caused much communication leading to a re-load of the dependence graph database (from after stage 6) with additional information being added and an incremental analysis.

4.4 Dependencies

To be able to use many of the facilities provided by CAPTools it is important to understand what a dependence is in terms of the source code for a program. The dependence analysis performed by CAPTools builds a dependence graph where the nodes in the graph represent executable statements and the arcs of the graph represent relationships between statements. For the purposes of code generation and parallelism, the arcs (dependencies) show the required execution order of the connected nodes (FORTRAN statements) to achieve semantically valid code. Any generated code, parallel or serial, that does not violate any of the dependencies is valid. There are four basic types of dependence:-

4.4.1 True dependence

caused by actual data flow from an assignment to a usage of a particular variable. Obviously the sink of the dependence cannot execute until the source has assigned the required value. For example,

```
s1: A(I)= .....
s2: ....= A(I) .....
```

true dependence between source s1 and sink s2 caused by A(I) being assigned in s1 and used in s2.

4.4.2 Control dependence

set when a control statement (usually an 'IF') controls the execution of other statements, either for a true or a false result. The controlled statements cannot execute until the controlling statement has executed. For example,

```
s1:  IF (A(I).EQ.5) THEN
s2:    A(I) = A(I) + B(J)
```

control dependence between source s1 and sink s2. Obviously the execution of s2 depends on the boolean expression in s1 being calculated.

4.4.3 Anti dependence

caused by the re-assignment of a used value, i.e. the value used in the source of an anti dependence is overwritten in the sink statement. For example,

```
s1:  ...= A(I)
s2:  A(I)= ......
```

anti dependence between source s1 and sink s2 caused by A(I) being reassigned in s2. s2 therefore cannot execute until after s1 has used the value s2 will overwrite.

4.4.4 Output dependence

caused by the re-assignment of previously assigned data. The order of assignment to the memory location must be maintained and the value in that location after the execution of both statements must be that provided by the sink statement. For example,

```
s1:  A(I)= ....
s2:  A(I)= ....
```

output dependence between source s1 and sink s2 caused by A(I) being reassigned in s2.

True and control dependencies are actual algorithmic dependencies caused by information flow. Anti and output dependencies (jointly referred to as pseudo dependencies) are caused by re-use of memory locations (often performed to minimise memory requirements) and are not inherent to the code.

In addition to dependencies having a type, they are also marked to indicate whether they exist between iterations of a particular loop, or if they exist during a single iteration of all loops surrounding both source and sink statements.

4.4.5 Level one dependence

A dependence between two statements (or two instances of the same statement) between iterations of the outer most loop surrounding both statements. For example,

```
      DO 10 J=2,NJ-1
        DO 20 I=2,NI-1
          T(I,J)=T(I,J-1)
 20     CONTINUE
 10   CONTINUE
```

Each iteration of the J loop (except in this case the first iteration) uses a value of array T assigned in the previous iteration of the J loop.

4.4.6 Level two dependence

A dependence between two statements (or two instances of the same statement) caused by the iteration of the second outer most loop that surrounds both statements.

```
      DO 10 J=2,NJ-1
        DO 20 I=2,NI-1
          T(I,J) =....
          ...    = T(I-2,J)
 20     CONTINUE
 10   CONTINUE
```

Each iteration (except in this case the first two iterations) uses a value of T calculated in previous iterations of the I loop.

4.4.7 Loop independent dependence

A dependence between two statements that exists during a single iteration of all loops that surround both statements. Loop independent dependencies are marked with a level of infinity.

```
      DO 10 J=2,NJ-1
        DO 20 I=2,NI-1
          T(I,J) =....
          ...    = T(I,J)
 20     CONTINUE
 10   CONTINUE
```

The value of T passes from assignment to usage during a single iteration of all loops that surround both statements.

Additional dependence attributes are :-

4.4.8 Exact true dependence.

This is defined for a true dependence when every usage of the variable causing the dependence in the sink statement is matched by an assignment in the source statement. This indicates that the assigning statement covers the range of data required in the sink statement.

4.4.9 Exact output dependence.

This is defined for an output dependence when the sink of the output dependence overwrite all values of the variable causing the dependence assigned in the source statement. The sink assignment therefore "kills" the assignment in the source statement.

Scalar dependencies are therefore always exact, but for array dependencies an examination of the data accesses is necessary to determine if the dependence can be marked as exact.

4.4.10 Uncertain dependence.

This is defined as a dependence that was set in the dependence analysis with unresolved inequalities in the dependence tests. Such dependencies are assumed and may be proved non-existent with additional user interaction.

The most important dependencies for the user to understand when parallelising a code are those that serialise loops and prevent the generation of parallel code. These are true (data flow) dependencies of any scalar or array reference between iterations of a loop.

4.5 Symbolic Variable Definition

The techniques used to eliminate uncertain dependencies from the dependence graph constructed by CAPTools involve performing symbolic algebra calculations on a program's variables. In this process, as well as in user interaction, it is essential to clearly specify the particular instance of a variable being considered. To achieve this, the concept of a variables defining statement and that statement's instance of execution are introduced.

A particular statement may be executed many times in the course of a program running. This may be because it is in a nesting of loops and/or in a routine that is called via many different call paths. In dependence analysis we always focus on one particular instance of execution at a time.

Each variable referenced in a particular instance of a statement will have a particular defining statement from which it was assigned a value. Therefore, we define a particular instance of a symbolic variable in terms of :-

i) A symbol of the variable.

ii) The defining statement assigning the value of the variable for that particular instance.

iii) The call path to the defining statement. This is a list of routines and the associated call statements that lead to this instance of the routine containing the defining statement.

Consider the following example,

```
   PROGRAM MAIN
   REAL A(100)
   INTEGER M
s1 CALL READA(A,M)
   CALL SOLVE(A,M)
   END

   SUBROUTINE READA(A,N)
   REAL A(100), N
```

```
      INTEGER N
   s2 CALL INPUTMESH(A,N)
      END

      SUBROUTINE INPUTMESH(B,K)
      REAL B(100), K
   s3 READ*, K, (B(I),I=1,K)
      END

      SUBROUTINE SOLVE(C,N)
      REAL C(100), N
      A(1) = 1.0
   s4 A(N) = 100.0
      ....
      END
```

In the example above the symbolic definition of N in statement s4 is

Symbol K

Defining statement s3

Call path s1, s2

During many of the stages of parallelisation two symbolic variables will be compared for equality as part of a test for dependence between instances of two statements. Two symbolic variable references can be equal if they have the same symbol, defining statement and call path. For example,

```
        SUBROUTINE SOLVE
s1      CALL INIT(N)
s2      CALL PROC(A,N,NI)
        END

        SUBROUTINE INIT(K)
s3      K=...
        END

        SUBROUTINE PROC(A,M,NI)
        ....
s4      A(M,1) = ...
        DO I=1,NITER
s5        CALL INIT(M)
          DO J=1,NI
s6          A(M,J)=...
          END DO
       END DO
```

In statement s4, symbolic variable M is defined as:-

Symbol: K, Defining statement: s3, Call path: s1

In statement s6 symbolic variable M is defined as :-

Symbol K; Defining statement s3; Call path: s5

For a dependence test between s4 and s6 the symbolic variable M must be assumed to have different values in s4 and s6 due to the differing call paths indicating that different instances of the defining statement provided the two values.

In tests that involve loop carried dependencies, i.e. dependencies between statements caused by the iteration of a loop, additional tests must be made for equality of symbolic variables involved in, for example, array indices. These are :-

a) If there is no call path (i.e. the defining statement of the variable is in the same routine as the carrier loop) and the defining statement is not in the carrier loop then the variable instances are equal.

b) If there is a call path then the head of the call path is examined and if this is not in the carrier loop then the variable instances are equal.

For a dependence test of s6 on itself (a self-dependence test) between iterations of the I loop, M will have different values because the head of the call path, statement s5, is inside the I loop and is obviously called once for every iteration. However, for the self dependence test of s6 between iterations of the J loop, M will have the same value since the head of the call path, s5, is outside the J loop and thus a single instance of the defining statement provided both variable proving that they are equal.

4.6 Partitioning.

The model of parallel processing used in the current version of CAPTools is a static data partition of structured data. A set of program arrays are divided over the processor topology, with each array item being allocated to a unique processor, usually as a consecutive range. At a later stage the code is transformed to allow every processor to assign only those partitioned array elements that have been allocated to that processor. The semantic validity of the code is maintained using communication, whenever it is necessary, to receive required data, which has been assigned on another processor.

Each partitioned array has associated with it two CAPTools generated variables, CAP_L<array_name> and CAP_H<array_name> that define the local lower and upper bounds for the array items stored on each processor. For example, for an array called AP the CAPTools variables for local lower and upper index values would be CAP_LAP and CAP_HAP. Partitions are aligned so that related arrays are partitioned with the same partition range variables, therefore other array may well be partitioned in terms of, for example, CAP_LAP and CAP_HAP. These variables are used in the final generated code to control the execution of statements (see next section) and in communication statements. As the values of the range are calculated at run time, this makes the code scalable to any number of processors (with certain restrictions due to the size of the problem being processed in the application code).

For a multi-dimensional parallelisation the low and high variables for the first dimension will have the same naming convention as described above. For subsequent dimensions the variables are defined as CAP<x>_L<array_name> and CAP<x>_H<array_name> where <x> is an integer representing the order of the partition from the second dimension. For example, for the second dimension for an array called AP the CAPTools variables for local lower and upper index values would be CAP2_LAP and CAP2_HAP.

4.7 Execution Control Masks

The model used by CAPTools to produce parallel code is one in which processors only execute statements that assign data to array elements that they own as defined by the data partition. To enforce this, some statements are masked so they are only executed if the array element being assigned is owned by that processor. A mask is an IF statement prepended to a statement with the boolean expression based usually on the values of an array index and the lower and upper bounds of an array, for example,

```
IF (I.GE.CAP_LAP.AND.I.LE.CAP_HAP) AP(I) = B(I)
```

To generate parallel code using a specified partition CAPTools first places a mask on every statement that assigns partitioned data. Obviously, running code in this form would have a high inefficiency due to the number of IF statements that would have to be executed. So, CAPTools will optimise the placement of these masks by merging masks where possible into block IF masks and DO loop range alterations so that the loop only iterates over a subset of it total iteration range. Consider the following code :-

```
DO I = 3, N-2
  AP(I) = (AP(I-2)+AP(I+2))/2.0
END DO
```

The execution control mask is firstly added to the statement assigning array AP and then inherited by the DO loop itself. The mask is then removed and replaced by altering the loop limits :-

```
DO I = MAX(3,CAP_LAP),MIN(N-2,CAP_HAP)
  AP(I) = (AP(I-2)+AP(I+2))/2.0
END DO
```

Consider CAP_LAP and CAP_HAP on processor 1 are 1 and 5, thus the low limit for the loop is 3 (since this is the first iteration in the original loop). On processor 2, CAP_LAP is obviously 6 and, since this is greater than the original loop low limit, the loop in processor 2 commences with I equal to 6.

Using CAPTools

5 Installing And Running CAPTools

5.1 Installing CAPTools

See the separate installation notes on how to install CAPTools.

5.2 Before Running CAPTools

To start CAPTools you need to do two main things. The environment variable CAPHOME needs to be set and the CAPTools executable must be in the path. Either type the following commands or place them in your .cshrc or .login file. Depending upon which machine you are running CAPTools on there may be a few other commands you need to type, see the README.RELEASE file in the CAPTools installation directory for more information.

```
% setenv CAPHOME <directory where CAPTools is installed>
% set path = ( $CAPHOME/bin $path )
```

5.3 Running CAPTools

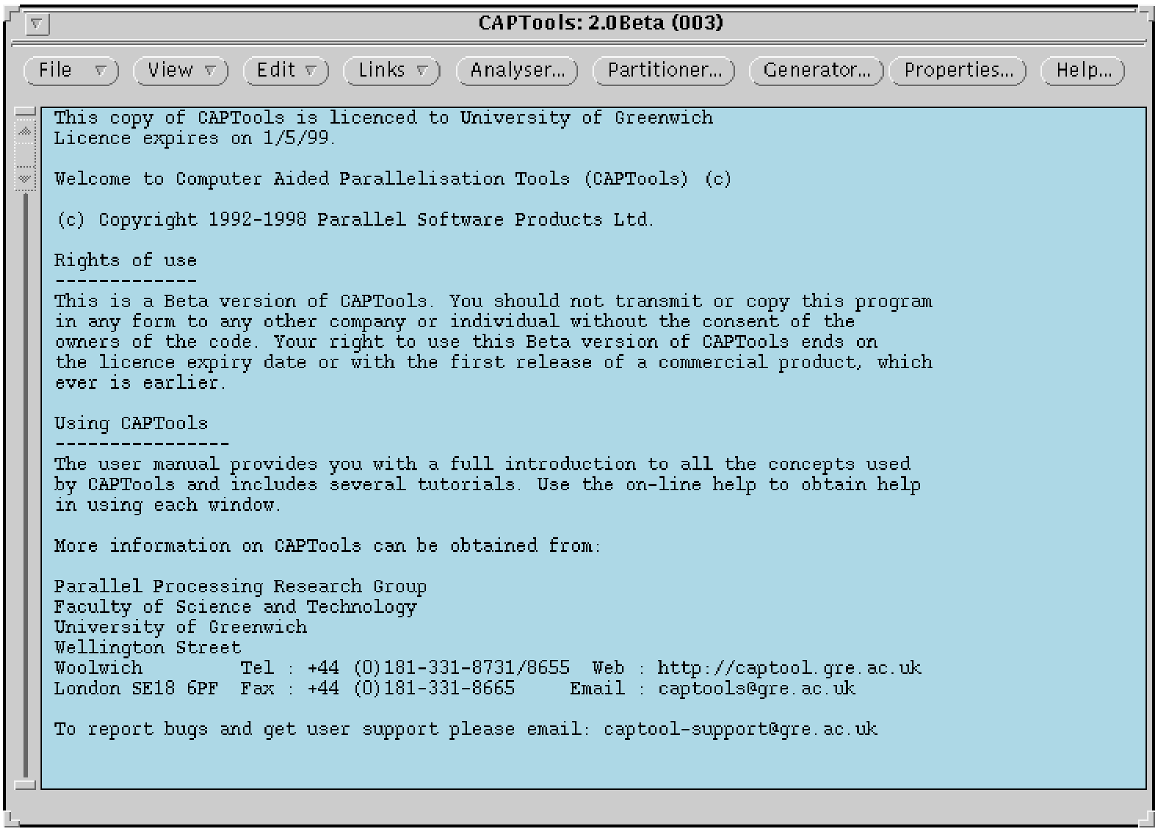
To run CAPTools type

```
% captool
```

Once you execute the CAPTools command the CAPTools base window is displayed. You are now ready to proceed with the parallelisation of a FORTRAN code.

6 The CAPTools User Interface

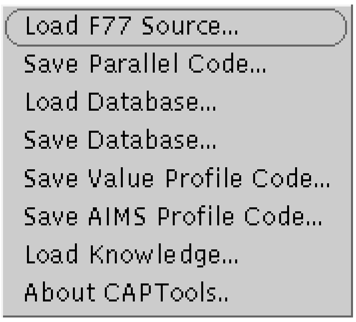
6.1 CAPTools Base Window.



The CAPTools base window contains the menus and buttons you need to operate CAPTools. The various windows for loading, analysis, partitioning, browsing, code generation can be started from this main window. Usually within the course of a working session the buttons lead the user from left to right, following a general parallelisation. However the nature of CAPTools allows, in fact encourages, the user to step back and forward between windows. For instance, the entry of information and answering of questions within the knowledge base facilities might precede the browsing of the code. This is a very deliberate feature as the background of users may vary considerably, from an expert parallel code author, to an end user understanding only the physics of the original serial code.

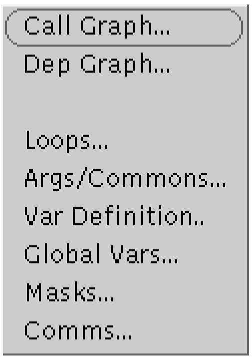
The legend of the first four buttons are followed by a triangle, this denotes a sub-menu. A submenu is activated by the right hand mouse button. The legend of the next five buttons contains trailing punctuation marks these indicate a pop-up child window.

6.1.1 The File Menu



The loading and saving of Fortran source code speaks for itself. The other options are related to CAPTools specific files. Load and Save Database apply specifically to the CAPTools generated database. A database can be saved at any time during the run of CAPTools and can be reloaded at a later date, on entry to CAPTools. This allows the extensive analysis of complex source code to be made only once and to be stored and used whenever required. The **Load Knowledge...** facility is not available at present. The **About CAPTools...** feature pops up a window displaying the version number and creators of CAPTools, contact address and bug information.

6.1.2 The View Menu



The View menu contains menu items to display the browsing windows in CAPTools. With the exception of the About CAPTools... option all are browsing engines of one form or another.

The **Call Graph...** allows the user to view the overall structure of their code via a number of filters.

The **Dependency Graph...** feature is displayed alongside a copy of the source code. The Dependence Graph is made up of numbered ellipses representing individual statements, linked by arcs describing the dependencies between the statements. The four dependence types are data flow dependence (described in CAPTools as a true dependence), control dependence, anti dependence, and output dependence. A number of filters are available to limit the size and volume of information displayed in the dependence graph.

The **Loops...** browser generates information about all the loops within the source code. The user can filter for serial, parallel, partitioned, or unpartitioned loops. The browser contains a number of other reduction filters and the iteration count profile data for each loop chosen indicates the possible importance of a specific loop.

The **Args/Commons...** browser enables the user to view the arguments and common block variables and to allow them to be traced through the code.

The **Var Definition...** browser is a very powerful feature allowing the user of CAPTools to single out specific variables on specific lines of source code. These specific variables may then be traced back through the code to their earliest possible definition, thereby helping in the understanding of the original code and final optimisation of the target code. The user can select a routine and an individual statement. This then presents all the variables used in that statement in a scrolling list. Selecting one of the variables will give a scrolling list of possible definitions for that variable. To traverse the possible definitions the user selects one of the possible definitions in the scrolling list, and presses the **Traverse To Definitions** button. This refreshes the routine scrolling list with the present routine, and the variables scrolling list with the variable on this statement line. Also a history of the traversal is kept via the Current Definition Path, allowing the user to step back to a previous definition to view another branch.

The **Global Vars...** browser allows the user to find any variable(s) of a specified type. There are a number of filters that allow the user to tailor the search to a small subset of all the variables used and assigned in the code. For example, the user can view all REAL variables in a selected routine that are assigned but not used.

The **Masks...** browser works in a similar way to the **Communications...** browser and allows the user to view and edit the execution control masks for individual statements.

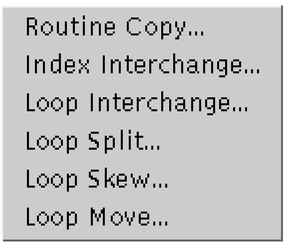
The **Comms...** browser allows the user to browse the communication statements generated by CAPTools. Features are available to view, move, and edit these communications, as well as an indication of importance via their profile features of CAPTools.

6.1.3 The Edit Menu

The Edit menu contains items relating to the input or editing of information in CAPTool:

The **Analyser...** option allows the user to define the level of dependence analysis to be performed and then to perform that analysis.

6.1.4 The Transformations Menu



The Transformations menu is a sub menu of the Edit menu.

The **Routine Copy...** option supplies the CAPTools user with the ability to copy or replace subroutine calls so creating duplicate copies. This can be useful when it is necessary to use a general purpose routine to behave slightly differently in various calls.

The **Index Interchange...** option allows the user to alter indices and, for example, enable contiguous data for communication.

The **Loop Interchange...** option allows the user to interchange the order in which the loops are executed within the code.

The **Loop Split…** option allows the user to selected execution lines within a loop to be within its own separate loop.

The **Loop Skew…** option allows the loop to be skewed.

The **Loop Move...** option allows the loops to be moved.

The **Partitioner...** option allows the user to define a data partitioning strategy. The partitioning strategies available are initially intended for structured mesh (single block) codes. The partitioning stage is essentially the first stage in the parallel code generation.

The **Code Generator...** option allows the user to generate parallel code. Firstly through execution control masking, followed by communications generation and finally to parallel code output.

The **Knowledge Base...** window displays all the questions generated by the analysis of the code. These questions can be examined, answered, or removed via profile information.

The **READ Knowledge...** window displays all the read statements within the source code that assign integer data. The entry of any information about the variables in these READ statements can greatly reduce the number of dependencies generated by the CAPTools analysis, since CAPTools is always conservative in nature to ensure the generation of correct code. An example of information that could be entered would be that a specific variable could never be less than zero. Profiled data is presented about any variable that appeared in the profile run.

The **User Knowledge...** window allows information to be added for non-READ variables, showing also their defining statements and possible call path.

The **Value Profiler...** used within CAPTools generates a value based profile for sample profiling runs, and can pool the information over a number of runs to be used within various windows of CAPTools. It is not necessary to make a complete run of a program during a profiling session as CAPTools can terminate the run and use a sample of its information. This option is not available in this version.

6.1.5 Links

The Links menu has one menu item which is not available in this version :

Start **AIMS VK…** allows the user to view trace files generated by CAPTools, using the Automated Instrumentation and Monitoring System (AIMS) VK tool.

6.1.6 The Analyser Button

The **Analyser...** allows the user to define the level of dependence analysis to be performed and then to perform that analysis.

6.1.7 The Partitioner Button

The **Partitioner...** option allows the user to define a data partitioning strategy. The partitioning strategies available are initially intended for structured mesh (single block) codes. The partitioning stage is essentially the first stage in the parallel code generation.

6.1.8 The Generator Button

The **Generator...** option allows the user to generate parallel code. Firstly through execution control masking, followed by communications generation and finally to parallel code output.

6.1.9 CAPTools Properties.

The user can choose to have source lines clipped in all CAPTools windows.

6.1.10 Online Help.

If you want information about any of the windows, buttons or lists in CAPTools you can get online help by pressing the help button in any window.

6.2 List Menus.

Most of the scrolling lists in the CAPtools GUI environment have their own menu attached to them. You can pop up this menu using the right hand mouse button when over a list. There are items in the menu to print the list to a file or printer and also to hot link to another window in context using the currently selected list entry. Using this facility enables you to quickly go from one window to another without having to go through the setup of a window each time to display the item(s) of interest.

6.3 Textpane Menus

The textpanes that display source code also have their own menus which can be accessed using the right hand mouse button. These provide hot link facilities as well. By highlighting a section of text in the textpane such as a routine name, variable name or statement line you can hot link quickly to another window.

7 Overview Of Using CAPTools

This chapter provides information on using the main parts of CAPTools, this includes the following:

- Loading a program
- Volunteering knowledge to the system
- Analysing a program to build a dependence graph
- Partitioning the program by mapping certain arrays across processors.
- Generating execution control masks to all possible statements.
- Communication calculation and generation.
- Generating scalable, generic parallel code.
- Compiling and running the parallel code.

7.1 Loading A Program

Use the file chooser activated from the **Load F77 Source** option of the **File** menu in the CAPTools base window to load a FORTRAN 77 program into CAPTools. At this point a basic parsing of the source code is performed and a symbol table and parse tree along with a control flow graph are constructed. The input code should be syntactically correct, however any errors are detected and reported to the user, however, use of a standard serial compiler should be used first to ensure correct code is presented to CAPTools.

7.2 Volunteering Knowledge

At this point you have the option of volunteering knowledge about program input variables to the system. This will aid in the initial analysis by, for example, minimising possible control flow paths in the code and proving many dependencies non-existent that would otherwise have to be assumed. By selecting the **READ knowledge** from the **Edit** menu of the main base window you can pop up the Read Knowledge window (Figure 5). This window shows a list of all READ statements defined in the code, one entry for each integer variable on a READ statement. You can enter information about the range of a program variable by selecting the entry in the list and hitting one of the buttons below it. In Figure 5 the system is told that the variable IN is Positive Non-trivial (by default ≥ 5) by selecting the IN entry in the list and then hitting the Positive Non-trivial button. If IN was the run time limits for the size of a mesh for instance, this would tell CAPTools that some loops were certain to execute at least once. Other information can be specified after selection of a read variable from the scrolling list using FORTRAN syntax i.e. ITURB.EQ.0.

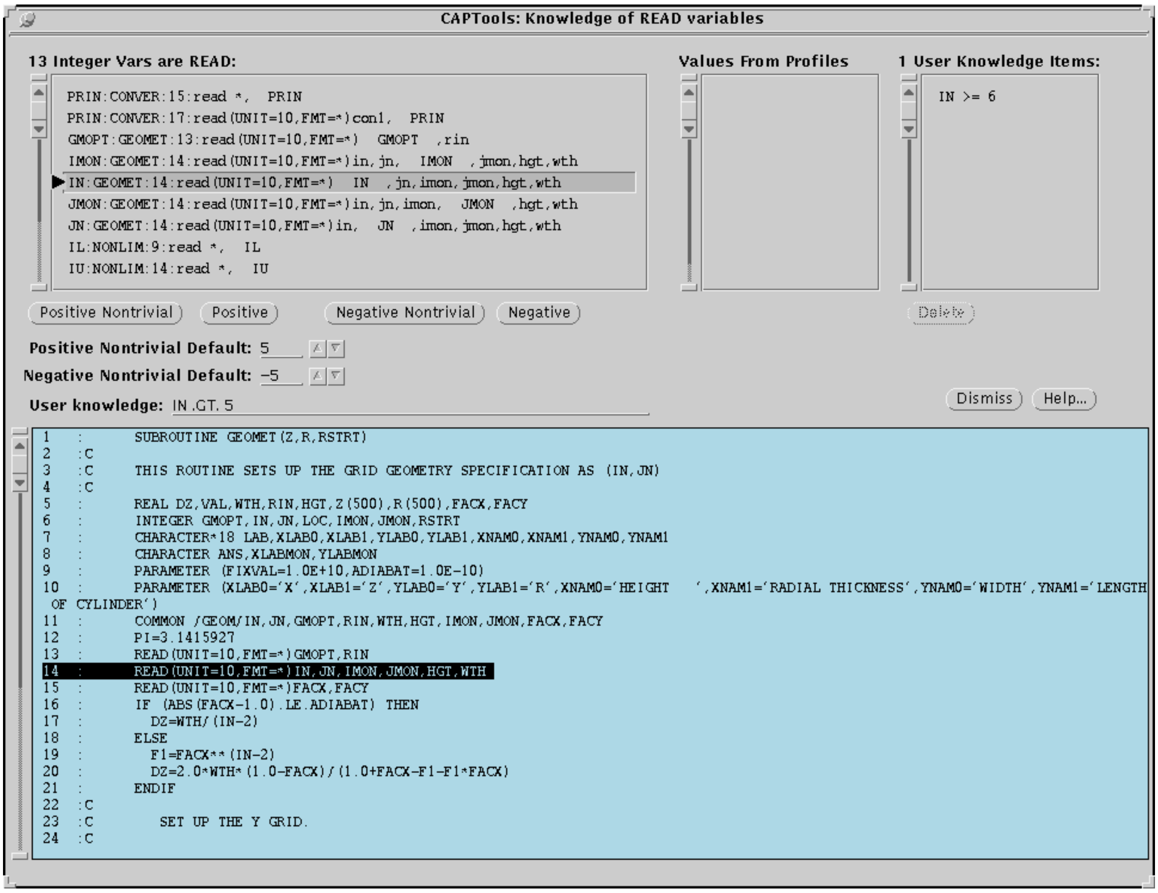


Figure 5 An example of the Read Knowledge window.

The information that variables IN and JN are positive integers on definition in the statement that assigns them their initial value (i.e. the READ statement) is added to the knowledge base.

Another way of entering knowledge is to use the **User Knowledge** window, again from the **Edit** menu of the main base window. The **User Knowledge** window allows the user to select a routine and then to provide information about variables in the form of inequalities. The syntax used to define the inequalities is based on the FORTRAN syntax, e.g. JN+IN.GT.KN. This window is used to specify more complex information involving several variables and/or information about variables that are not assigned in READ statements. If several statements are identified in the routine that assign a value to the variable specified in the added information, these are presented to the user in a pop-up scrolling list for the user to select the appropriate defining statement for the variable.

Using the information entered in either the **READ Knowledge** or **User Knowledge** window CAPTools will carry out a more accurate dependence analysis in the next stage.

7.3 Selecting The Analyse Option

Once a program has been loaded and you have possibly entered some knowledge about the code (see previous section), the next stage is to perform a dependence analysis to create the dependence graph that the following parallelisation stages rely. Press the **Analyse** button of the base window to pop up the Analyse window. This window offers you the choice of various levels of analysis.

There is a quick choice selection of **Basic**, **Intermediate** or **Full** which will set a combination of the individual options below it. The complexity of your program in terms of data flow determines which options you should select. As a rule of thumb choosing **Intermediate** from the quick choice is a good start. The final option of **Logic** while very important to pruning dependencies can slow down your analysis to several hours. A brief description of each individual option follows:

- **Scalar Equality** performs symbolic variable equality tests.
- **Exact** performs tests to set the exact dependence attribute to dependencies.
- **Interprocedural** performs full interprocedural analysis, if not set, any assignment/usage in a referenced routine to an array is conservatively assumed to reference the entire array.
- **Disproof** use information in the knowledge base to attempt to prove inequalities in dependence tests false.
- **Knowledge** use the inference engine, i.e. complex knowledge built up by the analysis for the program being analysed control structures along with that entered by the user.
- **Logic** use the logical substitution of symbolic variables through the dependence graph. Use of this option allows the tracing of variable definitions through IF statements.

There is the option of which Dependence test to perform :

Banerjee.

Omega.

There is a Limit Analysis Time that limits the maximum time allowable for analysis based in hours or minutes.

7.4 Using The Knowledge Base Window

Once an initial analysis has been completed the user should check the Knowledge Base window to see whether they can help CAPTools by answering any questions it may have wanted to know the answer to. To do this select the menu option **Knowledge Base** from the **Edit** menu of the main base window to pop up this window (Figure 6).

As it performed the dependence analysis, CAPTools may have made conservative assumptions about the existence of some dependencies. To eliminate any dependence it must prove it definitely does not exist. For each dependence a list of boolean expressions are built up, all of which must be proved false. Many of these expressions involve values of variables about which CAPTools has no information and therefore cannot eliminate the dependence. The Knowledge Base window shows these questions in the Questions list. Questions are ordered by frequency of occurrence. By selecting an expression from the list you can inspect the variables referenced in the question to try and decide whether the expression is definitely always true. If the user can provide definite information about the expression this is given by selecting the **True** button. Every time the user identifies a true question this information is added to the user knowledge list below the question list.

Once the user has answered as many questions as possible the user will need to instruct CAPTools

to perform an incremental analysis by returning to the **Analyse** window and selecting the **Analyse** button again, perhaps selecting a more powerful analysis than previously.

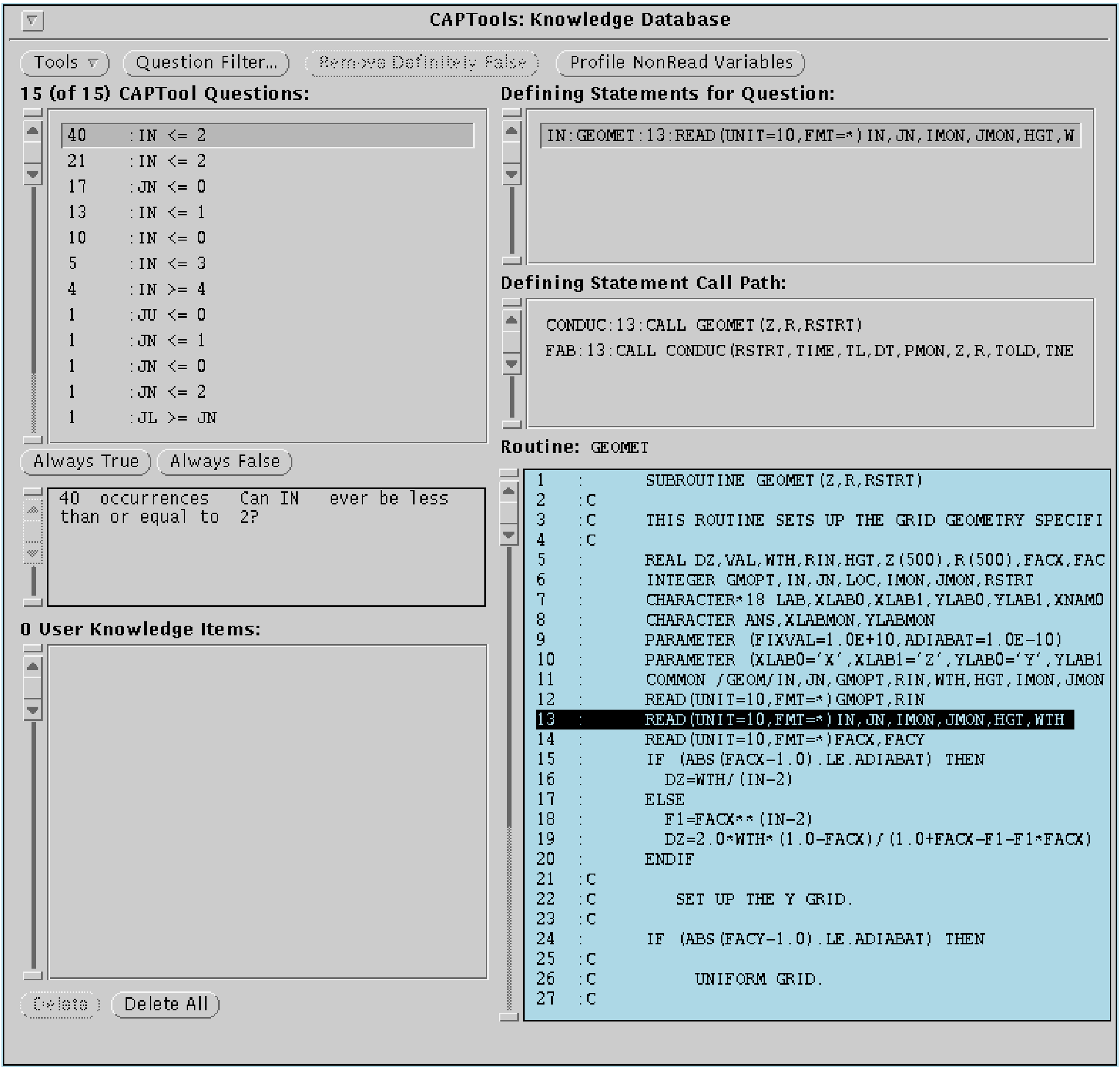


Figure 6 An example of the Knowledge Base window.

7.5 Saving The Database

The current version of CAPTools allows progression though the stages of parallelisation in a forward manner only. For instance, once you have accepted a partitioning of the data you cannot return to the pre-partitioning stage directly. For this reason you should at the end of each code generation stage save the dependence graph and associated information to a database file. You can do this using the **Save Database** menu item under the **File** menu of the base window. Doing this will mean the user can return to a previous state by restarting CAPTools and reloading the database file using the **Load Database** option of the **File** menu. The user should also save the database after analysing so there is no need to repeat the full analysis again.

7.6 The Dependence Graph Viewer

The dependence graph window provides the user with a mechanism for viewing the dependence graph produced from the dependence analysis. Using this window the user can investigate which loops are serial, inquire why there are serial and perhaps provide CAPTools with enough information to prove that a serial loop is in fact parallel. To access the DepGraph window select the **Dep Graph** item from the **View** menu in the main CAPtools window. The Dependence Graph Browser window is now displayed. On the left hand side of the window is a textpane which displays the text of a particular routine. On the right is a graph window that will show the corresponding dependence graph for the selected source code. In the graph each ellipse represents a statement in the text pane. Each arc represents a dependence between two statements and the direction of the arc (shown by an arrow head) identifies the source and sink of the dependence. From the dependence graph in Figure 7 the arc between statements 15 and 16 defines a true, infinity level data dependence of array A. In general, this is indicated by the array name and the dependence level and/or type below it, in this case A and infinity. Other examples include:

- The anti dependence between source statement 15 and sink statement 17 (array B, type - and level infinity)
- The control dependence between source statement 18 and sink statement 19 (type c and level infinity)
- The level 1 (outer loop) dependence between source and sink statement 25 (array C and level 1)
- The level 2 (inner loop) dependence between source and sink statement 26 (array E and level 2)

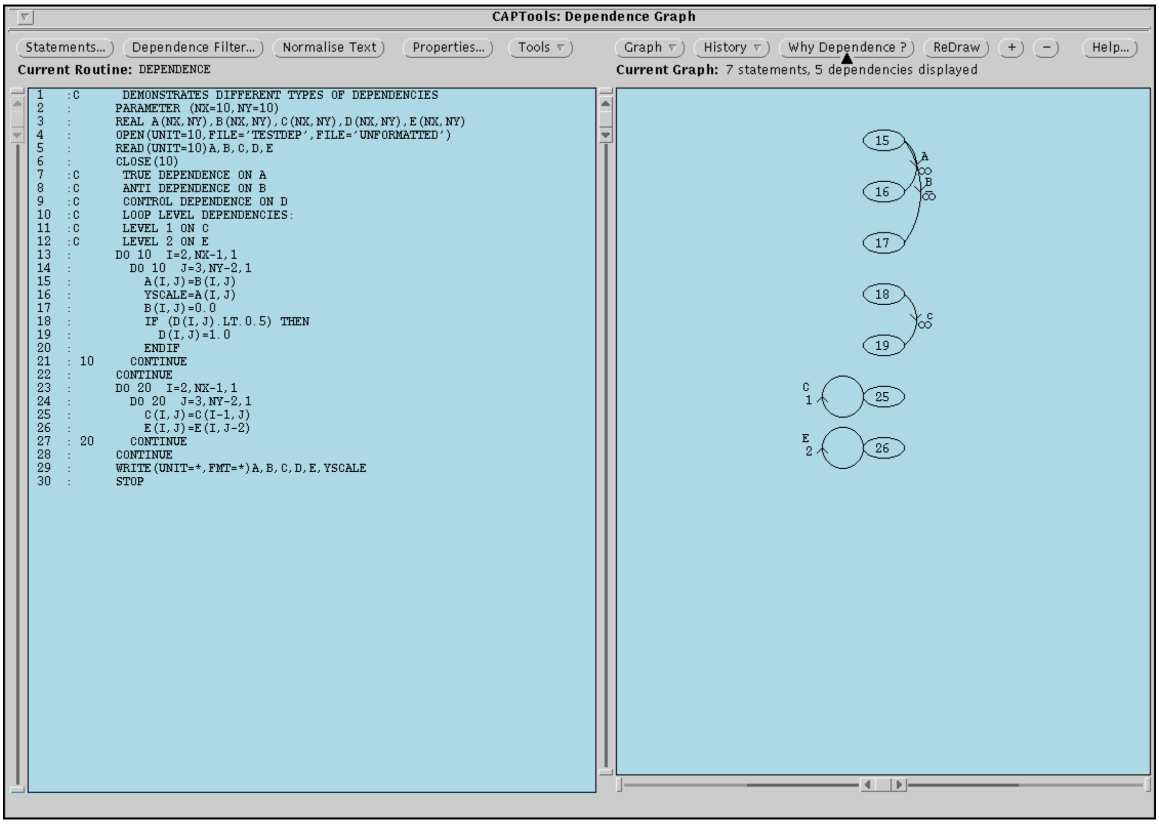


Figure 7 An example of the different dependencies.

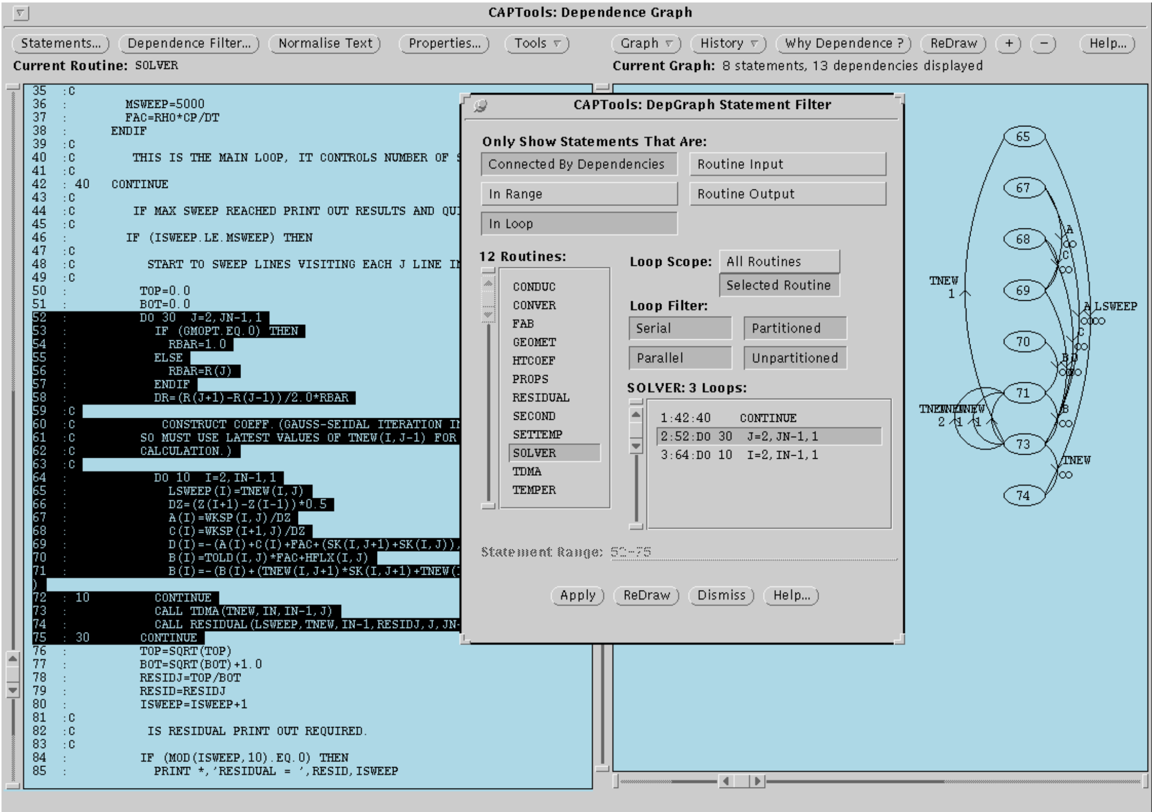


Figure 8 An example of the Dependence Graph window and Statement Filter.

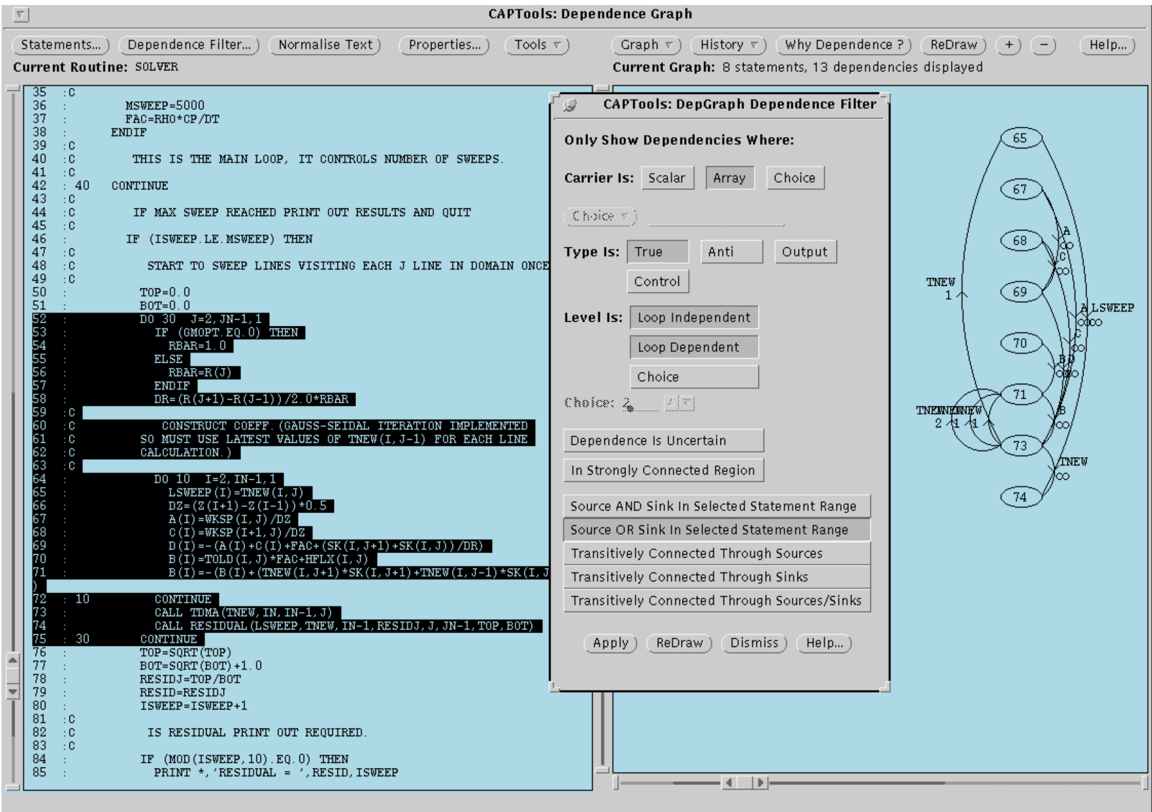


Figure 9 An example of the Dependence Graph window and Dependence Filter.

Many routine dependence graphs are so complex that user inspection is impossible, so for this reason and clarity there are a number of filters the user can apply before displaying a graph. To view the dependence graph for a particular routine the user must pop up the Statements window by selecting the **Statements** button (Figure 8). In this window the user selects the routine of interest in the Routine list. However, before selecting the **ReDraw** button the user selects the type of dependencies to be viewed. Selecting the Dependencies Filter button will pop up the Dependencies **Filter** window (Figure 9).

The user can select from this window the Carrier, Type and Level of dependence to be viewed in the graph window. At this stage the user can select the **ReDraw** button in either the Statement Filter window, Dependence Filter window or main DepGraph window and this will draw the dependence graph.

Most use of the Dependence graph window will be in the investigation of why loops are serial. Selecting the in Loop option in the statement filter window will enable the Loop list. Select **All Routines** and the **Serial** filter item. All the serial loops in the code can now be viewed in the Loop list. Select a particular loop in the list and in the Dependence Filter window select carrier **Array**, type **True** and Level according to the level of the loop the user is interested in. Select the **ReDraw** button. Only those dependencies serialising the loop will now be displayed. Some of these may be uncertain dependencies. The elimination of uncertain dependencies that serialise loops can be of enormous value. Select the Dependence **Is Uncertai**n Item in the Dependence Filter window and ReDraw the window to see only uncertain dependencies. Uncertain dependencies are those CAPTools cannot definitely prove are true or false. These dependencies may have a set of questions associated with them that can be answered to eliminate the dependence from the graph. To view and answer the question CAPTools may have about uncertain dependencies select the **Why Dependence?** button to pop up the Why Dependence window (Figure 10).

7.7 The Why Dependence Window

The Why Dependence window provides the user with all the information about a particular dependence in the code. This window can be also be used to remove dependencies either explicitly or by answering questions (if it is an uncertain dependence). Selecting a dependence arc in the graph window will show the information relating to that dependence in the Why dependence window. By selecting a particular index for the carrier variable in this window the user can view any questions CAPTools needs answered to eliminate the dependence from the graph. Selecting the **Question is Definitely True** button tells CAPTools that this dependence can be eliminated from the graph.

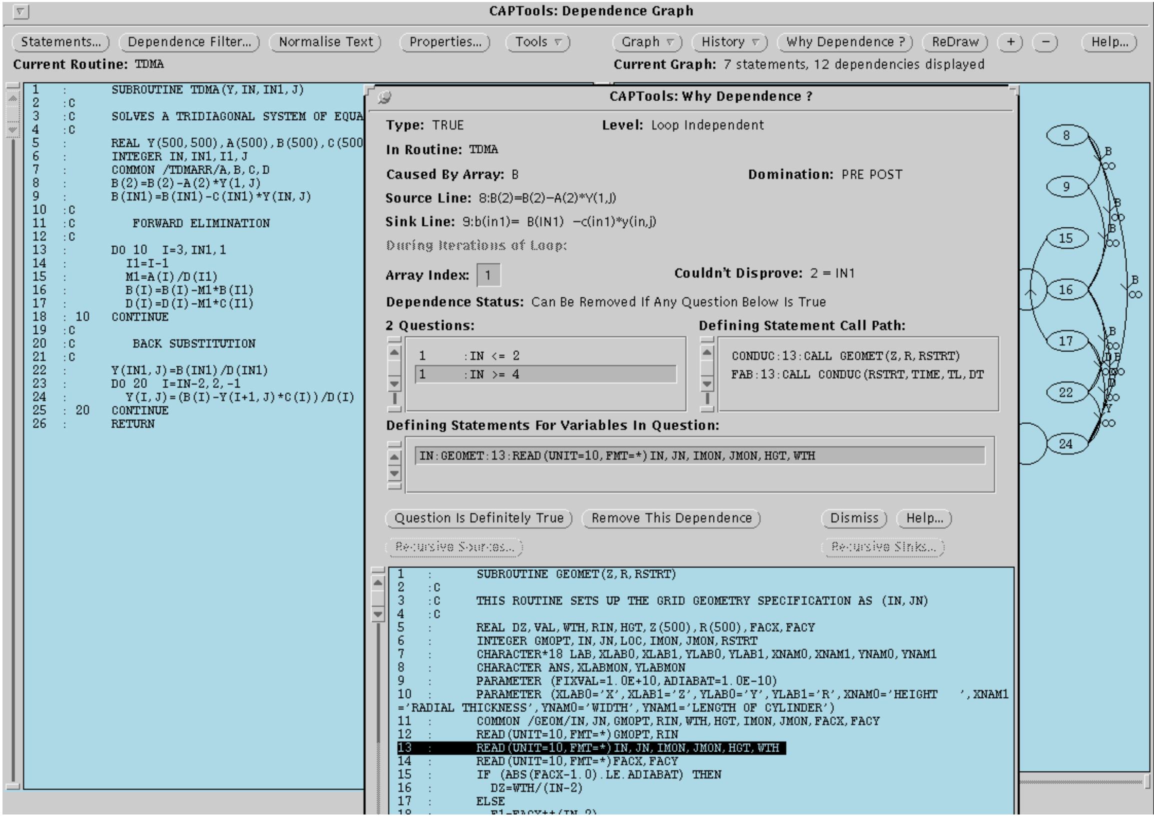


Figure 10 An example of the Dependence Graph window and Why Dependence window.

Other features available in the Dependence graph include a history mechanism which stores the previous graph viewed and the last graph viewed for a particular routine. Access to the history list is obtained by selecting the **History** button in the DepGraph main window. Another facility is the ability to print a PostScript version of a particular graph. Access to this function is via the **Graph** menu in the main DepGraph window.

7.8 Partitioning Of Data Using The Partitioner Window

Once satisfied with the dependence graph constructed in the dependence analysis, the user can move to the first stage of parallel code generation. The first stage is to calculate a data partition for the arrays of the code being parallelised (see concepts). Selecting the **Partitioner** button from the main base window accesses the **Partitioner** window. The two main lists in the **Partitioner** window are for unpartitioned arrays and partitioned arrays in each routine. Obviously at this stage the user can only see unpartitioned arrays.

7.8.1 Choosing The Initial Partition

The partitioning algorithm is automatic but requires an array index to start from. For structured mesh CM codes the user should decide which dimension of the mesh to partition across the pipeline of processors and then select a suitable array and index associated with this dimension. e.g. assume there is a 3D mesh of temperature, pressure and velocity variables stored in T(I,J,K), P(I,J,K) and V(I,J,K). For a partition in the second dimension (i.e. J) the user could choose either T,P or V and use index 2 as your initial partition. The partitioning algorithm will then inherit that partition to other arrays and interprocedurally to all other routines in the code.

There are usually many candidates for the initial partition array and routine. A basic rule for their selection would be an array that is widely used throughout the code and a routine which is central to the code being parallelised, for example, a linear equation solver in a typical CFD code.

On selection of a routine such as the main solver routine of the code, the unpartitioned array list will be set up with the arrays used in this routine. The user selects an entry in this list which sets up the variable field. The index field must then be setup with the required index of the selected array. It may be the case that the required array component is a subset of an array index, such as when a 3D problem is stored in a 1D array. To set such a partition, the modulus and divisor expressions must be set up. Consider the following example :-

Modulus expression : NI*NJDivisor expression : NI

When applied to a 1D array the above modulus and divisor expressions specify a partition equivalent to a partition of a 3D array in index 2 declared as :-

```
DIMENSION A(NI,NJ,NK)
```

The other settings in this window are explained in detail in the reference section.

Pressing the **Generate Partition** button starts the automatic partitioning algorithm. This algorithm works by examining each usage or assignment of partitioned arrays starting with the initial partition array and building a list of for and against reasons why each array may or may not be partitioned.

Some reasons for partitioning might be:

1. A direct assignment of one array element to another using the same loop iteration counter. e.g.

```
DO I = 1, N
  B(I) = A(I)
END DO
```

If A was the initial partitioned array, one reason to partition B is the direct assignment from A using the same linear expression of I.

2. An indirect assignment implied by data flow dependencies from another line. e.g.

```
DO I = 1, N
  X = A(I)
  B(I) = X
END DO
```

Here B(I) is assigned from X. Using the dependence graph it is possible to trace the assignment of X to A(I).

3. An indirect assignment in another routine. CAPTools can trace assignments to an array through routine boundaries and dimensional changes. e.g. array A may be assigned values in one routine, declared as a 2 dimensional array, with values being used in another routine and A declared as a 1-dimensional array.

One reason for not partitioning an array is if there is a linear relationship with unpartitioned components of a partitioned array. Consider array A partitioned in index 2 then

```
DO I = 2 , NI-1
  DO K = 2 , NK-1
    B(K,I) = 0.0
    DO J = 2 , NJ-1
      B(K,I) = B(I,K) + A(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Array B index 1 inherits a reason not to partition from array A index 3 of array A and B index 2 inherits a reason not to partition from A index 1.

The assignment range of each partitioned array in the partitioned direction is calculated from all relevant routines. Calculation of the maximum and minimum assignments are important as they are used later in the computation of the partition range variables to ensure that the used range is divided between the processors as opposed to the declared range. Often, even when using powerful symbolic testing algorithms in CAPTools a choice of ranges may exist, in this case CAPTools will pop up an Assignment Range Question window to ask you to identify the actual assignment range prior to completion of the automatic partitioning stage. The Assignment Range window will present to the user a series of symbolic expressions, requiring the selection of either the maximum or minimum values depending on the question posed. This selection can involve selecting several alternate expressions where CAPTools will generate a MAX or MIN function as appropriate when calculating the partition range variables. The user selects their choice from the list and presses the **Apply** button in each case. If the user does not know the maximum expression it is still possible to press the **Apply** button to skip to the next question or inspect the variables in the window to determine the maximum/minimum using other facilities within CAPTools.

7.8.2 Inspection Of Generated Partition

Once any questions have been answered and the partitioning has been completed the user can inspect the results using the **Partitioner** window. It is important at this stage to be satisfied that the partition produced is a good one, it may take several attempts to generate the best partition. The ideal choice is the one that will minimise the amount of communication generated in the final stages of code generation. But of course to determine this the user will have to get to the communication generation stage to find this out. Key points to remember for are:-

Reasons why some arrays have not been partitioned, it may be that an array has not been partitioned even though many reasons to partition were identified because some reasons not to partition were also detected. You can examine these by pressing the **Why UnPartitioned ?...** button below the Unpartitioned array list. Several conclusions can be reached from this process ;, accept the array as unpartitioned, force the array to be partitioned by selecting it as an initial partition to be incrementally added to the partition so far or perhaps copy a routine when, for example, a routine is called several times with differing orientations of data (see tutorial for the ARC3D code).

Unpartitioned arrays that have no reasons for or against partitioning. Selecting such an array as an initial partition to be incrementally added can lead to a more complete partition.

A small number of partition ranges should be generated with many arrays sharing the same ranges. Sensible assignment ranges indicate a successful partition, any that aren't should possibly be deleted along with the partitions on all arrays using that range. The **Browse Partition Ranges...** window is used to inspect the partition ranges and which variables are using them.

It is possible to delete the partition for a particular variable, do a second partitioning using a different initial partition or change the partition range for a particular array.

When you are satisfied with the partition you accept it by selecting the **Accept All Partitions** button. At this point you should save the database so you can return to this point quickly from the next stage if needed by restarting CAPTools and reloading the database (see previous section on saving database.)

7.9 Execution Control Mask Generation

After the generation of a partition the next stage of parallel code generation is to generate masks for each statement that assign partitioned data and then to optimise the mask placement. Masks are generated using the Code Generator window selected from the main base window by pressing the **Generator** button. Once the Code Generator window is displayed press the **Masks** button to start the mask generating algorithm.

The Masking algorithm works by generating execution control masks for:-

1. Statements that assign partitioned data.
2. Statements that assign values that are used in masked statements.
3. Statements that use data assigned by masked statements.
4. Statements that use partitioned data.

This is an iterative algorithm and repeating the above steps CAPTools will generate more transitive masks from the initial set. At this point the identification of distributed loops is possible for those loops who have the same mask on every statement inside the loop where that mask is a function of the loop variable. As a result, in the final code the masks will be removed and moved to the loop limits, i.e. the loop will now only iterate over the partitioned data stored on each processor.

Some routines that have the same masks on every statement can possibly be masked at their calling point with the removal of masks from within the routine. A routine that contains no I/O statements or masks can be masked dependent on dependence relationships in the calling routine(s). This enables, for example, intrinsic function usages to be masked.

7.9.1 Interrogation Of Generated Execution Control Masks

Once the mask generation is complete, the user can inspect the masks using the **Mask Browser** window selected from the base window sub-menu under the **View** button. This window allows you to inspect all masked and unmasked statements in the code and determine whether the automatic masking algorithm has done a satisfactory job.

Any masked statement can be selected to inspect to see why it was generated. Selection causes the textfield to indicate which of the four reasons for partitioning caused the selected mask and the related statements are listed for masks inherited due to dependence relationships in the algorithm.

Masks can also be deleted, added and edited as required using buttons in the **Mask Browser** window, for more details the user is referred to the reference section.

It is important to note that any unmasked statements will execute every instance of that statement on every processor. It is therefore an indication of a poor execution mask phase if statements that access partitioned data are not masked. The real test for a successful masking phase is only possible after the communication generation phase where poor masks will cause excessive numbers of communications to be generated.

Once the user is satisfied with the masking results it is advisable to save the database. This allows the user to return quickly to the stage at any time to adjust the masks as indicated after communication generation.

7.10 Communication Generation

The execution masks calculated in the previous stage enforce parallel execution control on every statement. Using these masks the usage of data on a processor which did not assign it can be identified. Obviously, a communication is necessary from the processor assigning the data to the processor using the data for the correct values to be used and this forms the basis of the communication generation in this stage. Communications are calculated, placed, merged and generated to satisfy the usage of data in masked and unmasked statements.

The user can start communication generation by pressing the **Calc Comms** button in the Code Generator window. A Communication may be generated for several reasons.

7.10.1 Commutative Operation Communication

Firstly commutative operations are identified such as summations and maxima calculations etc. The communication generated is of the form

```
CALL CAP_COMMUTATIVE(VALUE,TYPE,CAP_FUNCTION)
```

Where the value is on input the local value on this processor of the variable and the global value on exit. The CAP_FUNCTION is a function from the CAPTools library performing the appropriate operation for the type of data under consideration, i.e. CAP_RMAX for a real maxima calculation, CAP_IADD for an integer summation. To ensure correct operation it is essential that every processor attains the identical global value when roundoff is considered since calculation of, for example, convergence residuals with slightly different values on different processors could lead some to perform an extra information whilst others exit the iterative loop.

7.10.2 Partitioned Data Communications

The basic rule for communication requirement is data used on a processor that is potentially assigned on another processor. For partitioned arrays, since the partition definition determines the location of an up to date value of each array element in the processor topology, the partition is essential to communication requirement calculation. The execution control mask of the statement using the partition array determines which processor needs the value where an unmasked statement indicates that every processor needs the value. The location of the value is calculated through a comparison between the partitioned component of the partitioned array index expression in the using statement with the partition definition for that array.

7.10.3 Normalised Communications

Consider the following example where array A is partitioned in index 2 :-

```
DO K = 2, NK-1
  DO J = 2, NJ-1
    DO I = 2, NI-1
      A(I,J,K) = (2*A(I,J,K)+A(I,J+1,K)+A(I,J-1,K)) / 4.0
    ENDDO
  ENDDO
ENDDO
```

The execution control mask for the assignment statement is :-

```
IF (J.GE.CAP_LA.AND.J.LE.CAP_HA)
```

Where CAP_LA and CAP_HA are partition range variables for array A. This mask forms the requirement location for the data used in the statements right hand side.

Consider the first usage of array A. The location of the data (i.e. ownership) is determined using the partitioned component of this array usage (J in this case) and the partition definition (CAP_LA to CAP_HA). Since the ownership expression can be expressed as a function of the requirement expression the expressions are "normalised" where the actual values that must be communicated are represented by the CAPTools variable CAP_ICOUNT where in this case CAP_ICOUNT = J. The construction of the communication requirement control expression creates two sets, one for data on a previous processor (i.e. a processor with a lower id number in the direction specified as CAP_LEFT) and another for data on a later processors (i.e. a processor with a higher id number in the direction specified as CAP_RIGHT) :-

The CAP_LEFT direction communication requirement control is :-

CAP_ICOUNT < CAP_LA and CAP_ICOUNT ≥ CAP_LA and CAP_ICOUNT ≤ CAP_HA

The CAP_RIGHT direction communication requirement control is :-

CAP_ICOUNT > CAP_HA and CAP_ICOUNT ≥ CAP_LA and CAP_ICOUNT ≤ CAP_HA

where the first inequality in each set is the "not owner" control and the last two inequalities are the "requirement" controls. In this case, simplification of the communication controls proves that no communication is required. The additional information that CAPTools uses to prove this is that every processor has at least one unit of the partitioned dimension which implies that CAP_HA ≥ CAP_LA.

For the second usage of array A, the construction of the communication requirement control proceeds as with the first usage but with a different partitioned index expression J+1 and the CAPTools variable CAP_ICOUNT = J+1. The controls are :-

The CAP_LEFT direction communication requirement control is :-

CAP_ICOUNT < CAP_LA and CAP_ICOUNT-1 ≥ CAP_LA and CAP_ICOUNT-1 ≤ CAP_HA

The CAP_RIGHT direction communication requirement control is :-

CAP_ICOUNT > CAP_HA and CAP_ICOUNT-1 ≥ CAP_LA and CAP_ICOUNT-1 ≤ CAP_HA

Simplification proves that the CAP_LEFT control set requires no communication whilst the CAP_RIGHT communication reduces to :-

CAP_ICOUNT > CAP_HA and CAP_ICOUNT-1 ≤ CAP_HA

Which only has a single value solution of CAP_ICOUNT = CAP_HA+1thus the generated communication (with the **Bulk** communication option selected in the **Generator** window) will be:-

CALL CAP_EXCHANGE(A(2,CAP_HA+1,K),A(2,CAP_LA,K),NI-2,2,CAP_RIGHT) surrounded by a K loop copy. The first call parameter is the receive start location, the second is the send start location, the third is the number of elements of data to communicate, the fourth is the communication data type and the last is the RECEIVE direction. The communication becomes :-

```
CALL CAP_BEXCHANGE(A(2,CAP_HA+1,2),A(2,CAP_LA,2),NI-2,DIM,NK-2,2,CAP_RIGHT)
```

If **Gather/Scatter** communications is selected where the stride parameter DIM is taken from the declaration of A and NK-2 is the number of strides required. Note that no extra loops surrounding the communication are required in this case.

The placement of communication will be outside that loop nest with communication migration only blocked by an assignment to the communicated data in previous code. The detection of communication migration barriers uses the dependence graph to determine barrier statements.

The third usage of array A sets the CAPTools variable CAP_ICOUNT equal to J-1 due the partitioned index of A. In this case, the CAP_RIGHT communication control requires no communication whilst the CAP_LEFT communication request control simplifies to :-

CAP_ICOUNT < CAP_LA and CAP_ICOUNT+1 ≥ CAP_LA

which again requires a single value to be received of CAP_LA-1.For this communication the barrier to migration is a loop carried dependence of array A carried by the J loop (level 2) which causes a pipeline communication to be generated :-

```
DO K = 2 , NK-2
  CALL CAP_RECEIVE(A(2,CAP_LA-1,K),NI-2,2,CAP_LEFT)
  DO J = 2 , NJ-2
    DO I = 2 , NI-2
      A(I,J,K) = (2*A(I,J,K)+A(I,J+1,K)+A(I,J-1,K))/2.0
    ENDDO
  ENDDO
  CALL CAP_SEND(A(2,CAP_HA,K),NI-2,2,CAP_RIGHT)
ENDDO
```

Other circumstances can occur where an additional communication request control equality is added specifying a definite value of the partition index that is required, restricting the volume of data to be communicated.

7.10.4 Unnormalised Communications.

Normalisation of the communication request controls is not always possible, for example when no relationship between ownership and requirement controls exists. Two possible controls and related communications can result.

Consider the following code after execution mask calculation :-

```
IF (X.GE.CAP_LA.AND.X.LE.CAP_HA) A(1,X,1) = A(1,Y,1)
```

Where X and Y are constant. The communication request controls cannot be normalised and neither CAP_LEFT nor CAP_RIGHT controls can be proven false or simplified. This will result in two communication sets being generated, one to receive from processors to CAP_LEFT and another to receive from processors to the CAP_RIGHT where both communications are tightly masked to only operate on a subset of the full processor network :-

For receiving data from the CAP_RIGHT :-

```
 IF (X.LT.CAP_LA.AND.Y.GE.CAP_LA.AND.Y.LE.CAP_HA)
+  CALL CAP_SEND(A(1,Y,1),1,2,CAP_LEFT)
 IF (X.LE.CAP_HA.AND.Y.GT.CAP_HA)
+  CALL CAP_RECEIVE(A(1,Y,1),1,2,CAP_RIGHT)
 IF (X.LT.CAP_LA.AND.Y.GT.CAP_HA)
+  CALL CAP_SEND(A(1,Y,1),1,2,CAP_LEFT)
```

In the first mask, the first inequality can be interpreted as "data needed to the left (CAP_LEFT)" and the next two inequalities are the ownership masks. In the second mask, the first inequality can be interpreted as "data need on this processor or one to the left (CAP_LEFT)" and the other inequality as "data from a processor to the right (CAP_RIGHT)". In the third mask, the first inequality can be interpreted as "data required to the to the left (CAP_LEFT)" and the other inequality as "data from a processor to the right (CAP_RIGHT)".

The communication for a receive from the left (CAP_LEFT) is of a similar form with three statements performing the communication.

The final case is when normalisation is not possible and both ownership and requirement masks are variables covering a range of values (for example functions of different loop variables). In this case, the requirement controls are removed leaving only the non-ownership control. This causes communications to be generated that broadcast the value to every processor.

7.10.5 Unpartitioned Data Communications.

For unpartitioned arrays and scalars, a comparison of the masks and index expressions on both the using statement and the assigning statement as determined by a true dependence relationship from assignment to usage. The process consists of communication request control construction and normalisation along with communication generation. Consider the following example :-

```
      DO K = 1 , NK
        DO I = 1 , NI
s1:       IF (I+W.GE.CAP_LA.AND.I+W.LE.CAP_HA) B(I+X,K) = ....
        ENDDO
      ENDDO
      DO K = 1 , NI
        DO J = 4 , NJ-5
          DO I = 1 , NI
s2:         IF (J+Y.GE.CAP_LA.AND.J+Y.LE.CAP_HA)
     +        A(I,J+Y,K) = A(I,J+Y,K) + B(J+Z,K)
          ENDDO
        ENDDO
      ENDDO
```

Where array A is partitioned in index 2 and array B is unpartitioned (possibly due to its use elsewhere in a different orientation) with W, X, Y and Z having constant values. The true dependence between the assignment of B in s1 and its usage in statement s2 may cause the usage to have a communication request for array B generated. Normalisation is possible due to the linear relationship in both s1 and s2 between the mask and an index of array B, where the same index is involved in both statements (index 1). In this case, normalisation sets CAP_ICOUNT = J+Z and the communication request controls are :-

For communication request from the CAP_LEFT :-

CAP_ICOUNT+W-X < CAP_LA and CAP_ICOUNT+Y-Z ≥ CAP_LA and CAP_ICOUNT+Y-Z ≤ CAP_HA

For communication request to the CAP_RIGHT :-

CAP_ICOUNT+W-X > CAP_HA and CAP_ICOUNT+Y-Z ≥ CAP_LA and CAP_ICOUNT+Y-Z ≤ CAP_HA

One or both of the above requests may be proven to require no communication and simplification may be possible. The actual values to be communicated in the index of B used to provide normalisation (i.e. index 1) are the actual values of CAP_ICOUNT that solve the appropriate communication controls.

As with partitioned data, normalisation is often not possible and the same operation as for partitioned data is performed. A difference is that the location of up to data values of unpartitioned arrays is determined by execution control masks and not be the data partition. As a result, sometimes the location of data is only known in, for example, the loop nest containing the assignment. In such cases, the communication is generated as a broadcast of the assigned values to every processor with the communication place at an appropriate place within the loops surrounding the assigning statement.

The generation of communications proceeds in the same way as for partitioned data.

7.10.6 Interrogation Of Generated Communications

Using the Communications Browser window accessed from the **View** menu of the base window the user can investigate the communications generated in this stage. Using this Browser the user can determine if the dependence graph, partition definition, execution control masking and communication calculation performed has led to a minimised amount of communication.

Communications can be selected and the statements that generated requests for the data received in that communication listed and viewed in their code sections.

The Why Communication window, accessed from the **Why...** button, displays information relating to why the communication request was made. The communication request controls as described in the previous section are shown along with all other relevant information. For partitioned arrays each statement that generate a communication request for this communication can be selected and the relevant code and execution mask of that statement displayed. For unpartitioned data, both using statements and assigning statements are displayed where each pair generated a communication request. Selecting either list in this case also set the appropriate statement in the other list and also displays the key dependence that led to the communication request. That dependence may be between the assigning and using statements displayed but can in some cases be in a caller routine where the communication is required for data assigned in one call path and used in another call path. A hot link can be used to display the Why Dependence window, set up with the details for the key dependence.

The reasons for the communication placement are also identified (i.e. the barrier to the movement of the communication upward in the code). The reason can be the assignment of the communicated data, a carrier loop of a loop carried dependence of the communicated data causing a pipeline (as mentioned earlier), the head of a loop where a loop carried dependence exists but pipelining was not appropriate or assignment of a variable used in the indices (or execution control masks) of the generated communication.

With almost all current message passing DMS, communication startup latency is a very significant factor. Communications nested in loops can form a significant inhibitor to efficient execution in parallel since many startup latencies are incurred. If the barrier that forced the communication to remain inside a loop involves only index or mask data being assigned it may be possible to list a few alternate values for the barrier variable(s) which CAPTools will then substitute into the communication information creating several requests which can then pass through the barrier and, hopefully, out of the loop.

CAPTools merges communications as far as possible, however, merging is only performed when every memory location referred to in the merged data area exists in one of the original data areas. There could be a case where two communications refer to adjacent or overlapping communications but merging is not possible i.e.

```
      CALL CAP_EXCHANGE(A(1,2,CAP_LA-1),A(1,2,CAP_HA),NI-5,2,CAP_LEFT)
      CALL CAP_EXCHANGE(A(1,3,CAP_LA-1),A(1,3,CAP_HA),NI,2,CAP_LEFT)
```

If merged, the location A(NI,2,CAP_LA) would be communicated along with several others which are not in either communication, therefore CAPTools would conservatively decide not to merge them. The user can easily merge the communications using the **Edit..**. and **Delete...** buttons.

The definitive test of a parallel code is, of course, to run it on a particular parallel machine so that the precise characteristic of that machines calculation performance, communication speed and communication startup latency have an influence. A profile of the parallel code using a machine dependent profiler can be used to identify inefficient code sections due to either lack of parallelism or communications. This can then guide the use CAPTools to determine why parallelism is lacking or why the influential communications have been generated and why they have been placed where they are. This could lead through many stages of the parallelisation process with adjustment and repeat of the stages as required. The inefficient code will often be due to the algorithm used in the serial code. In the present version of CAPTools, algorithm replacement cannot be performed within the environment, thus alteration of the serial code can be made and the parallelisation repeated or the alteration could be made in the generated parallel code.

7.11 Overlapping Calculation And Communication

As introduced in section 2.2.2, many parallel systems enable communication to be performed at the same time as calculation. The automatic generation of parallel code exploiting this capability is available, generating the calls to the asynchronous communications in the CAPTools library and the required synchronisation points to ensure correct code is generated. There are four techniques automatically employed to transform synchronous parallel code sections to become overlapped :-

Simple overlapping - this merely indicates that the communication call itself is adjusted to its asynchronous counterpart and that one or more synchronisation points are generated later in the code before the communicated data is used to ensure correct operation. The overlapping occurs between the communication and unrelated calculations performed after the communication initialisation and before the synchronisation point. The following example shows the communication of the array A being overlapped with the code updating array B since this code does not use the communicated data.

```
 CALL CAP_AEXCHANGE(A(CAP_L-1),A(CAP_H),1,2,CAP_LEFT,
+                   CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
 DO I=1,1000
   B(I)=B(I)+C(I)+D(I)
 ENDDO
 CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
```

2) Partial loop overlapping - here a communication before a loop is asynchronised with a synchronisation point placed within the loop controlled by a conditional expression so that synchronisation is only performed during a "late" iteration of that loop. The communication is thus overlapped with calculations during "early" iterations of that loop. The following example shows the communication overlapped with all but the last iteration of the loop on each processor, where the synchronisation of the communication is performed at the start of the last iteration since the communicated value is then required.

```
 CALL CAP_AEXCHANGE(A(CAP_H+1),A(CAP_L),1,2,CAP_RIGHT,
+                   CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
 DO I=MAX(2,CAP_L),MIN(NI-1,CAP_H)
   IF (I.GE.CAP_H) THEN
     CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
   ENDIF
   B(I)=B(I)+A(I+1)
 ENDDO
```

3) Partial loop overlapping using loop unrolling - here the data communicated before a loop is required in an "early" iteration of that loop. The "early" iterations that require the communicated data are "unrolled" from the loop into a copy of the loop immediately after the original. The synchronisation point is generated between the original loop and the copied loop ensuring the data is received before the "early" iterations that use that data are executed. The overlapping occurs with the "later" iterations of the original loop. The following example shows the first iteration of the loop on each processor removed and placed in a copy of the loop immediately following the original loop with a synchronisation point placed between the two loops.

```
 CALL CAP_AEXCHANGE(A(CAP_L-1),A(CAP_H),1,2,CAP_LEFT,
+                   CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
 DO I=MAX(2,CAP_L)+1,MIN(NI-1,CAP_H)
   B(I)=B(I)+A(I-1)
 ENDDO
 CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SYNC_SEND,CAP_SYNC_RECEIVE)
 DO I=MAX(2,CAP_L),MIN(NI-1,CAP_H,MAX(2,CAP_L))
   B(I)=B(I)+A(I-1)
 ENDDO
```

Pipeline overlapping - the communications in a pipeline (see section 3.1) are overlapped so that during each iteration of the pipelined loop, the data required for the next iteration is being received. The execution diagram for a basic synchronous pipeline is shown in Figure 11 with the associated overlapping pipeline execution diagram in Figure 12. As can be seen, the execution time for the overlapped pipeline is considerably less than that of the synchronous pipeline.

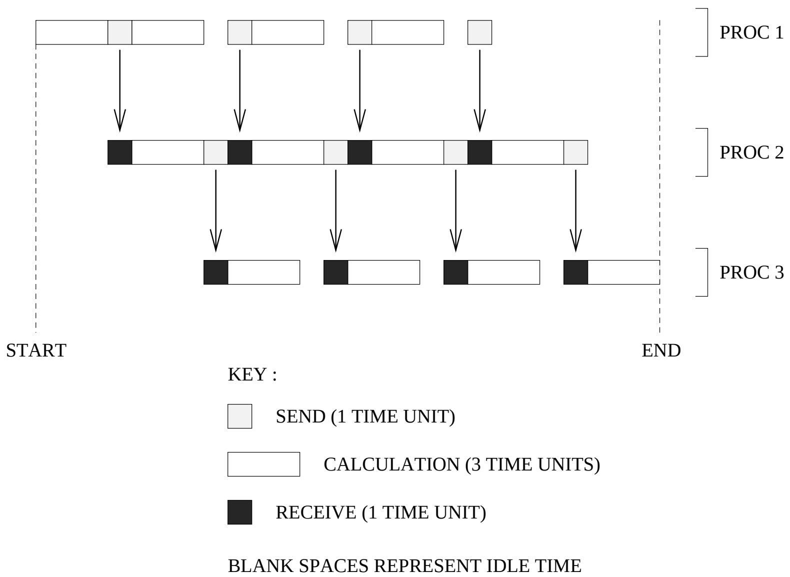


Figure 11 Execution of a synchronous pipeline.

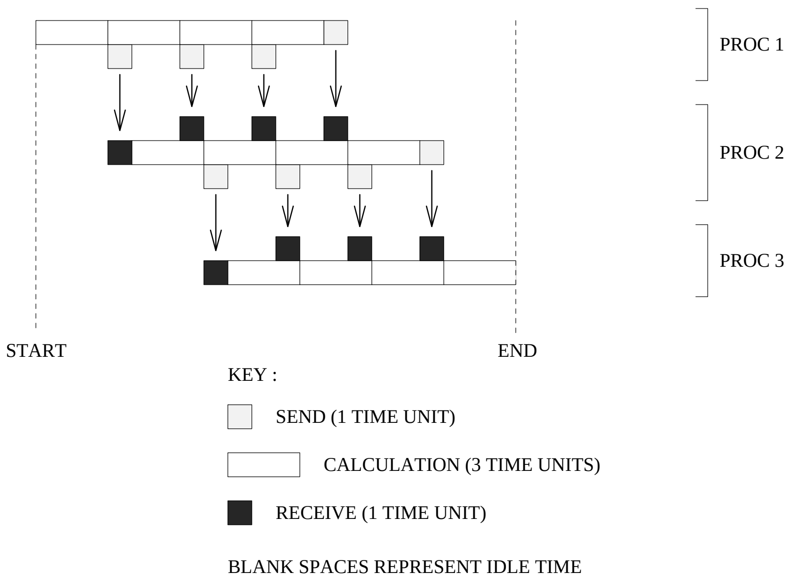


Figure 12 Execution of an overlapped pipeline.

The following shows a simple example of a synchronous pipeline :-

```
DO I=2,NI-1
  DO J=2,NJ-1
    CALL CAP_RECEIVE(A(I,J,CAP_L-1),1,2,CAP_LEFT)
    DO K=MAX(2,CAP_L),MIN(NK-1,CAP_H)
      A(I,J,K)=A(I,J,K)+A(I,J,K-1)
    ENDDO
    CALL CAP_SEND(A(I,J,CAP_H),1,2,CAP_RIGHT)
  ENDDO
ENDDO
```

When transformed to overlap communications with calculation, the above code becomes :-

```
CAP_SYNC_SEND=0
CALL CAP_ARECEIVE(A(1,1,CAP_L-1),1,2,CAP_LEFT,CAP_SYNC_RECEIVE)
DO I=2,NI-1
  DO J=2,NJ-1
    CALL CAP_SYNC_RECEIVE(CAP_LEFT,CAP_SYNC_RECEIVE)
    IF (J+1.LE.NJ-1) THEN
      CALL CAP_ARECEIVE(A(I,J+1,CAP_L-1),1,2,CAP_LEFT)
    ELSEIF (I+1.LE.NI-1) THEN
      CALL CAP_ARECEIVE(A(I+1,2,CAP_L-1),1,2,CAP_LEFT)
    ENDIF
    DO K=MAX(2,CAP_L),MIN(NK-1,CAP_H)
      A(I,J,K)=A(I,J,K)+A(I,J,K-1)
    ENDDO
    CALL CAP_SYNC_SEND(CAP_RIGHT,CAP_SYNC_SEND)
    CALL CAP_ASEND(A(I,J,CAP_H),1,2,CAP_RIGHT,CAP_SYNC_SEND)
  ENDDO
ENDDO
```

7.12 Reducing The Memory Used In The Parallel Version

In CAPTools the data is partitioned such that each processor has its own set of data. This distribution of data, Figure 1 shows how the array T (with 1000 elements) is distributed over 4 processors.

Processor 1 owns the data 1-250 with an overlap for 251; Processor 2 owns the data 251-500 with an overlap for 250 and 501; Processor 3 owns the data 501-750 with an overlap for 500 and 751; Processor 4 owns the data 751-1000 with an overlap for 750.

For each processor the remainder of the memory space is not used, so for example, for processor number 3 the memory space available between 1 to 499 and from 752 to 1000 elements is not used. The size of the problem is therefore limited by the maximum problem size that may be solved on a single processor.

To allow larger problems to be executed in parallel the memory space allocated must be reduced to cover just the assignment area, any overlap areas and any other data which is needed by each processor, but which they do not own.

Consider the following serial code:

```
DO I=1,NI
   DO J=3,NJ-1
      A(I,J)=A(I,J-1)+A(I,J-2)+A(I,1)
   ENDDO
   DO J=1,NJ
      B(I,J)=A(I,J)-A(I,NJ)
   ENDDO
ENDDO
```

When this code is parallelised using CAPTools and selecting array A index 2 for the initial partitioning and without the memory reduction option, the following parallel code is obtained

```
IF (1.GE.CAP_BLA.AND.1.LE.CAP_BHA)CALL CAP_SEND(A(1,1),NI,2,CAP_RIGHT)
IF (1.LT.CAP_BLA)CALL CAP_RECEIVE(A(1,1),NI,2,CAP_LEFT)
IF (1.LT.CAP_BLA)CALL CAP_SEND(A(1,1),NI,2,CAP_RIGHT)
IF (NJ.GE.CAP_BLA.AND.NJ.LE.CAP_BHA)CALL CAP_SEND(A(1,NJ),NI,2,CAP_LEFT)
IF (NJ.GT.CAP_BHA)CALL CAP_RECEIVE(A(1,NJ),NI,2,CAP_RIGHT)
IF (NJ.GT.CAP_BHA)CALL CAP_SEND(A(1,NJ),NI,2,CAP_LEFT)
DO I=1,NI,1
  DO CAP_ICOUNT=CAP_BLA-1,CAP_BLA-2,-1
    CALL CAP_RECEIVE(A(I,CAP_ICOUNT),1,2,CAP_LEFT)
  ENDDO
  DO J=MAX(3,CAP_BLA),MIN(NJ-1,CAP_BHA),1
    A(I,J)=A(I,J-1)+A(I,J-2)+A(I,1)
  ENDDO
  DO CAP_ICOUNT=CAP_BHA,CAP_BHA-1,-1
    CALL CAP_SEND(A(I,CAP_ICOUNT),1,2,CAP_RIGHT)
  ENDDO
  DO J=MAX(1,CAP_BLA),MIN(NJ,CAP_BHA),1
```

```
        B(I,J)=A(I,J)-A(I,NJ)
      ENDDO
    ENDDO
```

Figure 13 shows the memory allocation for a processor from CAP_LA to CAP_HA with overlap areas either side and other used data at each end. The data area between the lower overlap and the first slab of the array is not used. It is therefore possible to reduce the amount of memory used by moving the position of CAP_L and CAP_H and its overlaps to be the next data set after the first slab. This is seen in the memory reduced code as an offset or shift from the original position of CAP_L i.e. OFFSET=-CAP_L. However, it is also necessary to move the data for slab N. This is placed immediately after the first slab and before the lower overlap. This results in a modification to the original offset so the new offset is defined as OFFSET=-CAP_L+5. All references to this partitioned array must be adjusted by this offset. The only exceptions to this are where the other slabs (in this case slabs 1 and N) are explicitly used. In such cases, for the processor which contains that slab in its assignment area, the OFFSET adjustment is made. All other processors use the location of the slab in the memory reduced memory map, in this case slab 1 for original slab 1 and slab 2 for original slab N.

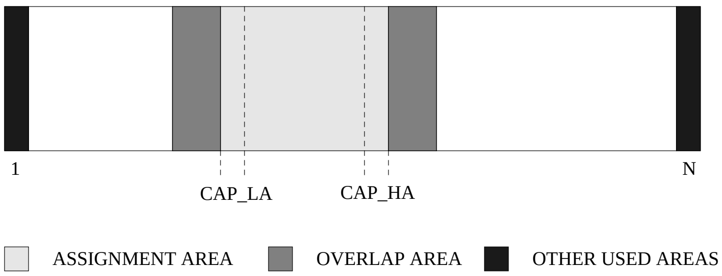


Figure 13 Memory allocation for a partitioned array without memory reduction.

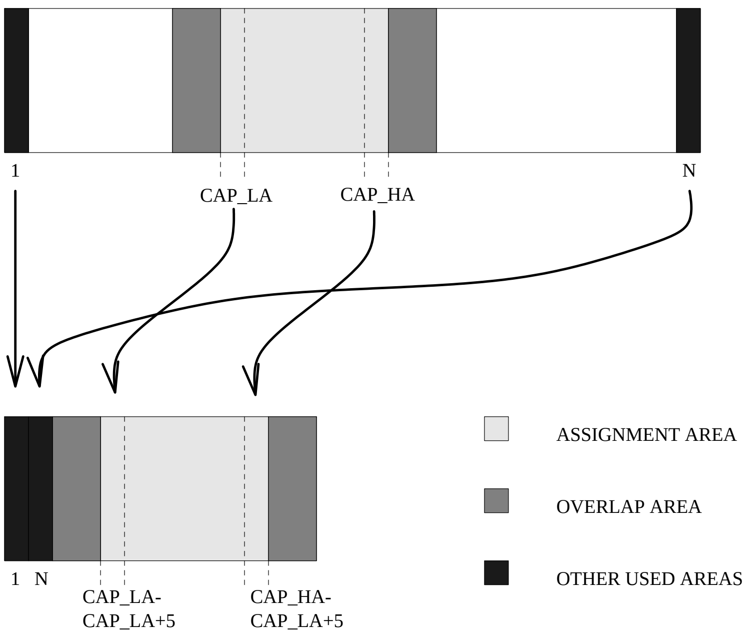


Figure 14 Effect of using memory reduction optimisation on parallel code.

Therefore, when the reduced memory option is applied, CAPTools generates the following code

```
      IF ((1.GE.CAP_BLA).AND.(1.LE.CAP_BHA)) THEN
        CAP_1OFFSET1=1-(CAP_BLA-5)
      ELSE
        CAP_1OFFSET1=1
      ENDIF
      IF ((NJ.GE.CAP_BLA).AND.(NJ.LE.CAP_BHA)) THEN
        CAP_1OFFSET2=NJ-(CAP_BLA-5)
      ELSE
        CAP_1OFFSET2=2
      ENDIF
      IF (1.GE.CAP_BLA.AND.1.LE.CAP_BHA)CALL CAP_SEND(A(1,CAP_1OFFSET1),NI,2,
     +CAP_RIGHT)
      IF (1.LT.CAP_BLA)CALL CAP_RECEIVE(A(1,CAP_1OFFSET1),NI,2,CAP_LEFT)
      IF (1.LT.CAP_BLA)CALL CAP_SEND(A(1,CAP_1OFFSET1),NI,2,CAP_RIGHT)
      IF (NJ.GE.CAP_BLA.AND.NJ.LE.CAP_BHA)CALL CAP_SEND(A(1,CAP_1OFFSET2),NI,2,
     +CAP_LEFT)
      IF (NJ.GT.CAP_BHA)CALL CAP_RECEIVE(A(1,CAP_1OFFSET2),NI,2,CAP_RIGHT)
      IF (NJ.GT.CAP_BHA)CALL CAP_SEND(A(1,CAP_1OFFSET2),NI,2,CAP_LEFT)
      DO I=1,NI,1
        DO CAP_ICOUNT=CAP_BLA-1,CAP_BLA-2,-1
          CALL CAP_RECEIVE(A(I,CAP_ICOUNT-(CAP_BLA-5)),1,2,CAP_LEFT)
        ENDDO
        DO J=MAX(3,CAP_BLA)-(CAP_BLA-5),MIN(NJ-1,CAP_BHA)-(CAP_BLA-5)
          A(I,J)=A(I,J-1)+A(I,J-2)+A(I,CAP_1OFFSET1)
        ENDDO
        DO CAP_ICOUNT=CAP_BHA,CAP_BHA-1,-1
          CALL CAP_SEND(A(I,CAP_ICOUNT-(CAP_BLA-5)),1,2,CAP_RIGHT)
        ENDDO
        DO J=MAX(1,CAP_BLA)-(CAP_BLA-5),MIN(NJ,CAP_BHA)-(CAP_BLA-5),1
          B(I,J)=A(I,J)-A(I,CAP_1OFFSET2)
        ENDDO
      ENDDO
```

Visually, the reduced memory representation of the A array will look as shown in Figure 14. This reduced memory version will now allow larger size problems to be executed on parallel systems.

7.13 Multi-Dimensional Partitioning.

Earlier versions of CAPTools partitioned codes in a single dimension only (i.e. split into mesh lines in 2D or mesh slabs in 3D). CAPTools may now partition second and third dimensions of the mesh, one at a time, transforming the code from the previous passes. This multi-dimensional partitioning has been incorporated into the existing CAPTools environment where, after communication generation for one dimension, partitioning of a subsequent dimension may then be commenced.

The multi-dimensional partitioning is performed within CAPTools one dimension at a time for several reasons. Firstly, it simplifies the user interaction during the partition, masking and communication stages. During partitioning, some arrays will only be partitioned in certain dimensions, making user inspection complicated in a multi-dimensional context. The generation of multi-dimensional masks for statements would also be very complicated since each statement may be controlled by varying numbers of masks. For communications, there will be considerably more communications for the user to investigate. The 'diagonal' communications are also simpler to generate a dimension at a time. Finally, since CAPTools was developed originally for a 1-dimensional partition, it is a natural progression to use the methods for the first dimension for partitioning further dimensions.

The generation of multi-dimensional partitioned parallel CFD codes will allow them to be executed on 2D and 3D grids of processors allowing a larger number of processors to be exploited with minimal data volumes being involved in communications. The phases involved in the further partitioning of a code consist of prescribing a new partition, the generation of execution control masks, the generation of communications, the adjustment of communications from previous partitions and the final generation of the multi-dimensional partitioned parallel code.

For the secondary partitions the partition ranges are defined by the variables CAP<x>_L<array_name> and CAP<x>_H<array_name> where the <x> is an integer representing the order of partition.

7.13.1 'Diagonal' Communications In A Multi-Partitioned Code.

The generated multi-stage communication schedule allows data on 'diagonal' to be moved in two stages without the need for additional communications between diagonal neighbour processors.

In Figure 16 to calculate the value of **B(250,251)** for the assignment on processor 5, using the 1-point stencil for the code in Figure 15 requires the value of **A(251,250)** (dark shaded box) which is calculated on processor 2. This would seem to necessitate a 'diagonal' communication between these two processors. This is avoided by performing two orthogonal communications in strict order. First, the communication from processor 2 into the left overlap area of processor 6, as required for the first partition, is performed, updating the overlap area of processor 6. This is followed by the communication from processor 6 into the upper overlap area of processor 5, where this second communication involves the range CAP1_L-1 to CAP1_H-1 due to the index **I-1**. This second communication therefore sends some data in the left overlap that has just been received from processor 2. From the figure it can be seen that these two communications have copied the value of **A(251,250)** from processor 2 to processor 5 without the need for a 'diagonal' communication.

```
DO J = MAX(2,CAP2_L), MIN(999,CAP2_H)
  DO I = MAX(2,CAP1_L), MIN(999,CAP1_H)
    B(J,I) = A(J+1,I-1)
  ENDDO
ENDDO
```

Figure 15 Fortran code for a 1-point stencil calculation.

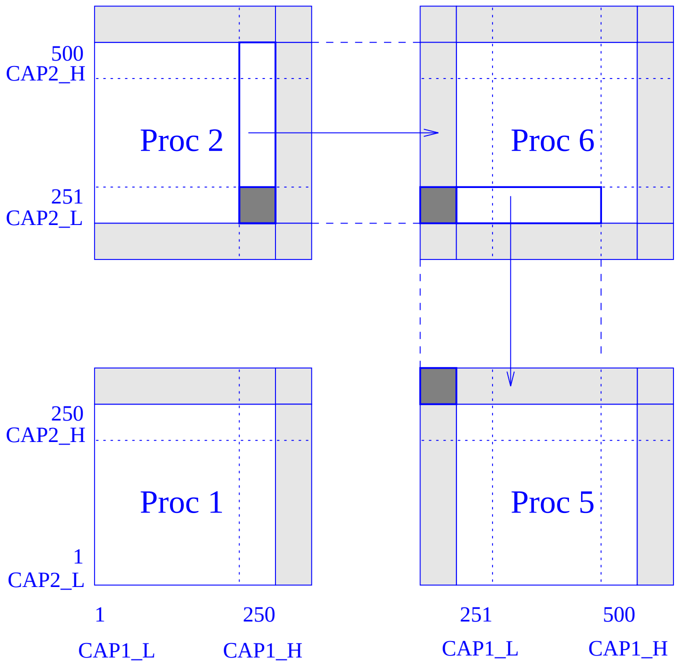


Figure 16 'Diagonal' Communication of data for 1-point stencil

7.14 Unstructured Mesh Parallelisation.

7.14.1 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 17 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 18, 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.

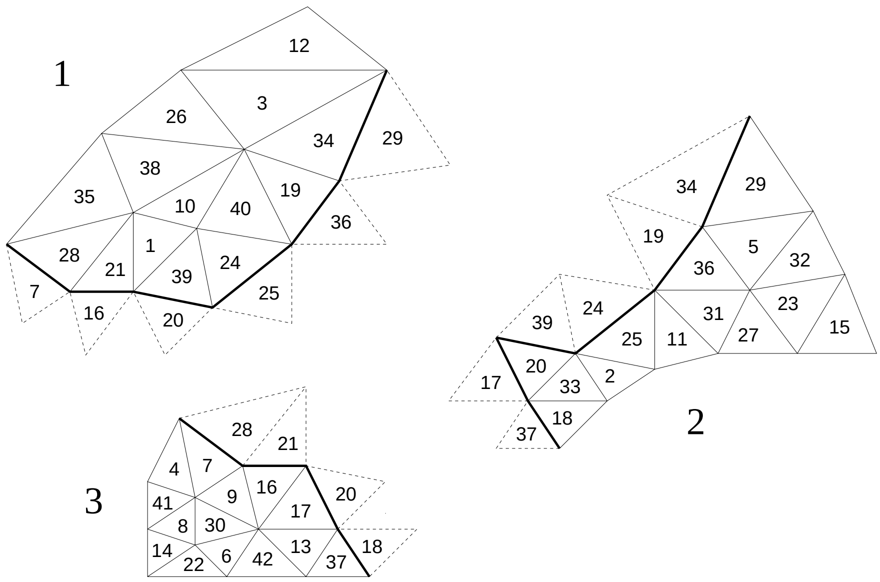


Figure 17 Data partition of elements and nodes in a simple mesh onto two processors.

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 17) 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 17, as few as four communications may be required:-

1. Overlap elements required for element computations

2. Overlap elements required for nodal computations
3. Overlap nodes required for element computations
4. 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 18 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 18 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 18 is used in the examples in section 7.14.2.

7.14.2 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 19 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 18. The code example in Figure 19 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 19 Simple code section using data structures from Figure 18.

7.14.3 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. When no such obstacle exists, a geometric decomposition of the unstructured mesh usually enables an effective parallelisation. 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. 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 18): **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. In Figure 19, for example, arrays **NTEMP**, **OTEMP**, **NUM_NEIGHBOURS** and **ELETOP** all use the processor ownership array for elements (**P_elem**).

7.14.4 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 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 20 for the example of Figure 19.

The transformation of masks onto loop limits requires the mesh to be renumbered so that owned entities are packed together in all related arrays so that the loop only iterates over the owned set. This allows the element loop in Figure 20 (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.

```
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 20 Block execution control mask in example of Figure 19.

7.14.5 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 20, 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. 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. 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).

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 7.14.1. 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:-

```
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 20 is shown in Figure 21.

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

Figure 21 Inspector loop for the communication of **OTEMP** in S_1 from Figure 20.

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. 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.

7.14.6 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. The process of merging compares all inspector loops throughout the code with the tests performed in the three stages listed below:

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.

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.

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.

7.14.7 Communication Generation Example

To demonstrate the complexity of operation required for even simple real codes, Figure 22 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 22 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 due to the assignment of the index array **WORK** in the call to routine **SETUP**. Allowing loops not common to the requesting statement enables the assignment to **WORK** to be included in the inspector loop and thus not be a barrier to communication migration. 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 22. Figure 23 shows the inspector loop generated.

```
IEND = 0
DO I = 1,NETYPE
   ISTART = IEND + 1
   IEND = IEND + NTYPE(I)
   DO IE = ISTART , IEND
        NPTS = GPTIEL(I)
        CAP_WORK(1) = ETOP(NPTS,IE)
        DO K = 2,NPTS
              CAP_WORK(K) = ETOP(K-1,IE)
        ENDDO
        DO K = 1,NPTS
              CALL CAP_CONNECT (IE,CAP_WORK(K))
        ENDDO
   ENDDO
ENDDO
CALL CAP_OVERLAP(0,0,ID1)
```

Figure 23 Inspector loop from Figure 22

```
         DO IT=1,NTIME
S1         CALL CAP_SWAPOVER(U,ID1)
S2           CALL CAP_SWAPOVER(V,ID1)
S3           CALL CAP_SWAPOVER(W,ID1)
S4           CALL CAP_SWAPOVER(P,ID1)
S5           CALL CAP_SWAPOVER(T,ID1)
              .  .  .
S6           CALL FLOW(U, V, W,  P, T,...)
         ENDDO
```

Figure 24 Usage of communication set calculated from inspector loop after merger.

The calls to **CAP_CONNECT** creates a graph of edge pairs. 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 (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 24 shows a further extract from generated parallel code that follows the code section in Figure 23, 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 23. The call to **CAP_OVERLAP** generates the communication set from the inspector loop 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.

7.15 Final Code Generation

The final stage in generating parallel code in CAPTools is to select the **Final Code** button in the Code Generator window. This button pops up a File Save window. If the source code was all in one file the user selects a new or existing file to save to and clicks on the Save button. CAPTools generates the final parallel code to this file. If a series of files was read in at the loading stage the user is offered 3 choices. The code can be saved to the original filenames (perhaps in another directory). It can be saved to original filenames pre-pended with a prefix. Finally the entire code can be saved to a single file.

7.16 Compiling And Running The Parallel Code

To compile and run the parallel code generated by CAPTools use the CAPLib package. This is available from the CAPTools web site. Several scripts are provided with the CAPLib package to help the user to compile and execute the parallel code he has generated using CAPTools. A more detailed description of CAPLib is provided by the CAPLib User Manual.

7.16.1 Capmake

The script *capmake* allows the user to compile an executable, linking in the correct CAPTools library. e.g.

```
capmake -p pvm3 -m sun fabpar
```

will look for a file fabpar.f and compile and link this for running under pvm3 on a Sun workstation. The –p argument specifies the underlying parallel environment. The –m argument specifies the machine type.

or

```
capmake -p i860toolset -m paramid fabpar
```

will look for a file fabpar.f and compile and link this for running under i860toolset for the Transtech Paramid.

or

capmake –p shmem –m t3e –unstruct femesh

will look for a file femesh.f and compile and link this for running under Cray SHMEM for the Cray T3E.

To see a full list of options type *capmake*. Usually when CAPLib is installed the user runs a *usecaplib* script, supplied with CAPLib, from their .cshrc or .login file. This sets up default values for the parallel environment and machine type. Therefore the –p and –m arguments are only neccessary in compiling or running for a non-default environment.

7.16.2 Caprun

The script *caprun* allows the user to run a previously generated executable under a particular parallel/machine enviroment. e.g.

```
caprun -p pvm3 fabpar
```

or

```
caprun -p i860toolset fabpar
```

will attempt to run the executable fabpar under pvm3. To see a full list of options type *caprun*.

7.17 Debugging

With the current early version of CAPTools there may be small deficiencies in the parallel code that require debugging as well as errors introduced in optimisation by the user. The technique that has proved most useful in the development of CAPTools in debugging the generated parallel code follow in two stages. Firstly the parallel code is run on one processor in the debugger and compared with the original serial code run on the same machine. In some cases, slight differences in round off have been identified. To aid debugging it may be desirable to append the appropriate brackets to the parallel version so that identical values are produced to allow simple comparison with the original serial code. Once the one processor version is correct, comparison between the one processor parallel run and a multi-processor parallel run using debuggers can then be used, monitoring values during the execution to identify any discrepancies. Commutative operations do not preserve the precise round-off errors from the one processor version, particularly in summations, since each processor calculates its local contribution before the global value is attained. This may complicate the comparison and it may be desirable to temporarily replace the commutative operations with pipelines.

CAPTools Tutorials

8 Introduction

The tutorials described below are a series of worked examples, increasing in complexity, that give the user a basic understanding of how to use CAPTools to parallelise an application code.

8.1 Starting CAPTools

To start CAPTools, assuming it has been installed correctly, type

captool

The main CAPTools window will appear. If there are any warning messages about fonts these can be safely ignored providing the windows are readable.

8.2 Loading Code and Databases

Use the **File** menu of the main CAPTools window to load source code, either directly from a Fortran source file, or if code is split across several files from a *.list* file containing the pathnames to each source file, one per line. Databases from previous CAPTool sessions can be loaded using the **Load Database…** menu item.

8.3 Saving DataBases

We strongly suggest that you save a database at each major point during the CAPTools parallelisation process. For example, after an initial analysis, after partitioning and after each stage of code generation (masking, communication calculation). This will enable you to quickly start from a previous stage if needed and can save many hours when parallelising large codes. A database naming convention is suggested in the Save Database window based on the name of the source code filename and the stage at which you are saving the database. For example, after analysing a code called foo with full options, this would produce the suggested name of *foo_full.dbs*. Databases should be saved using a *.dbs* file extension, otherwise they will not appear in the directory list when loading back into CAPTools.

8.4 Moving In Context Between Windows.

Many of the lists, textfields and every textpane in CAPTools has a context sensitive menu accessed using the right-hand-button whilst over an object. Using these menus will allow you to move from one CAPTools window to another in context. For example, you can select a line in a CommsBrowser window textpane and using the right-hand-button and the context-sensitive menu move to the Variable Definition Browser window. The appropriate routine will be selected and the line chosen in the CommsBrowser window will be highlighted and ready for investigation. This can make your analysis of the parallelisation at each stage much faster.

9 TUTORIAL 1: Parallelisation Of A Simple 1d Jacobi Program

After completing this first simple tutorial the user will have learnt the basics of the parallelisation process using CAPTools. During this session the user is taken through the steps involved in parallelising a very simple one-dimensional Jacobi program. This example is also used in the introduction to the CAPTools parallelisation model described in section "CONCEPTS - What is the SPMD model?". It is assumed that the user is familiar with Open windows and the Sun Unix environment.

Start CAPTools by typing captool in a cmdtool window. The main CAPTools window will appear on the screen. From this main window the user can access the full functionality of CAPTools.

Select the **Load f77 source...** menu item from the **File** menu of the main CAPTools window. A file selection window will appear. Using this window the user can load FORTRAN source code into CAPTools (Figure 25). For this first tutorial load the file jacobi.f from the directory $CAPHOME/examples/jacobi. Do this by typing $CAPHOME/examples/jacobi in the Name textfield and hitting return to cd to the examples directory. Now double clicking on jacobi.f entry in the scrolling list or click once and select the **Load** button. CAPTools now loads in the FORTRAN code from the file and pops up the Analyser window that is the next stage in the CAPTools parallelisation process.

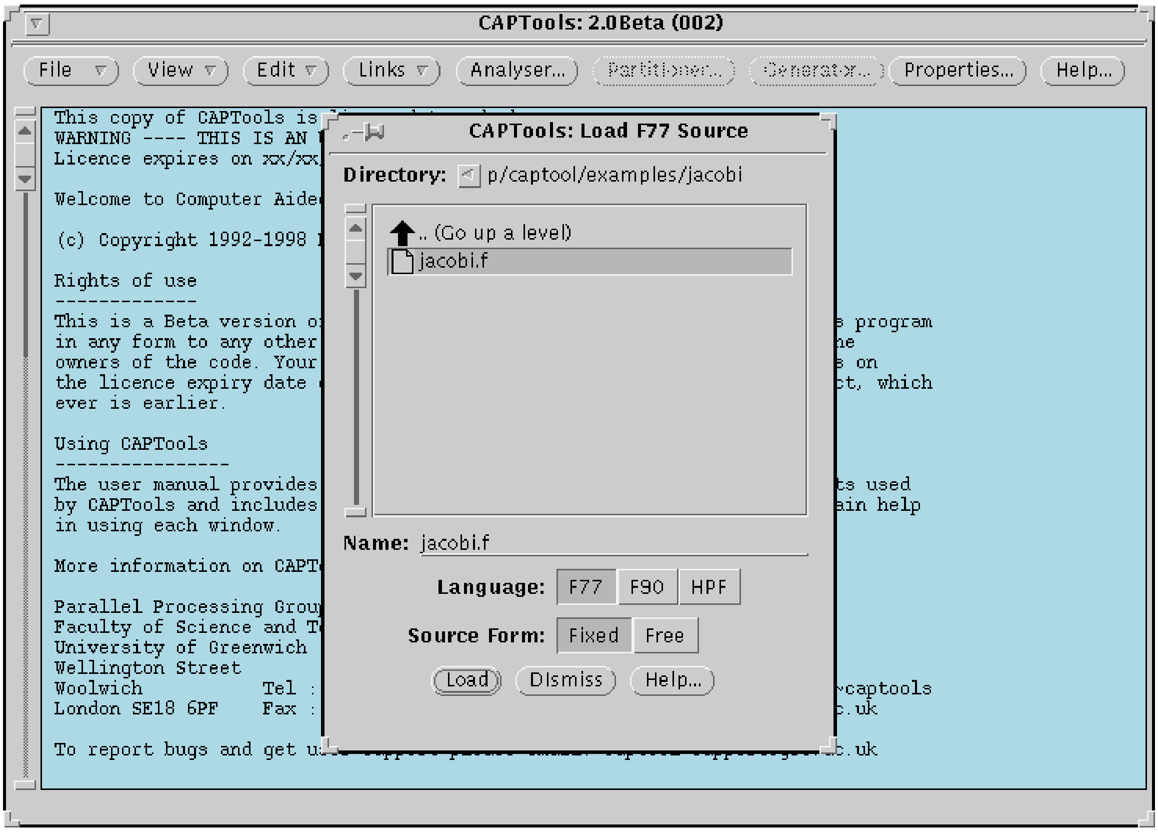


Figure 25 Loading the Jacobi code into CAPTools.

9.1 Analysing

The Analyser window is used to start the dependence analysis that CAPTools uses to produce parallel code. This window offers various options to control the power of the analysis that CAPTools performs. These are explained in full in the section headed "USING CAPTools - Selecting the Analyse option", for now select all options by clicking on **Full** followed by the **Analyse** button (Figure 26). CAPTools will now proceed to build up a dependence graph of the Jacobi program. For a simple code like Jacobi this will take seconds, larger programs take longer. A busy clock cursor is displayed to indicate that CAPTools is working to show that whilst the analysis is taking place no interaction with CAPTools is possible.

Once the analysis is complete the next stage is to perform a partitioning of the code data using the Partitioner window. This can be popped up from the main CAPTools window by selecting the **Partitioner...** button.

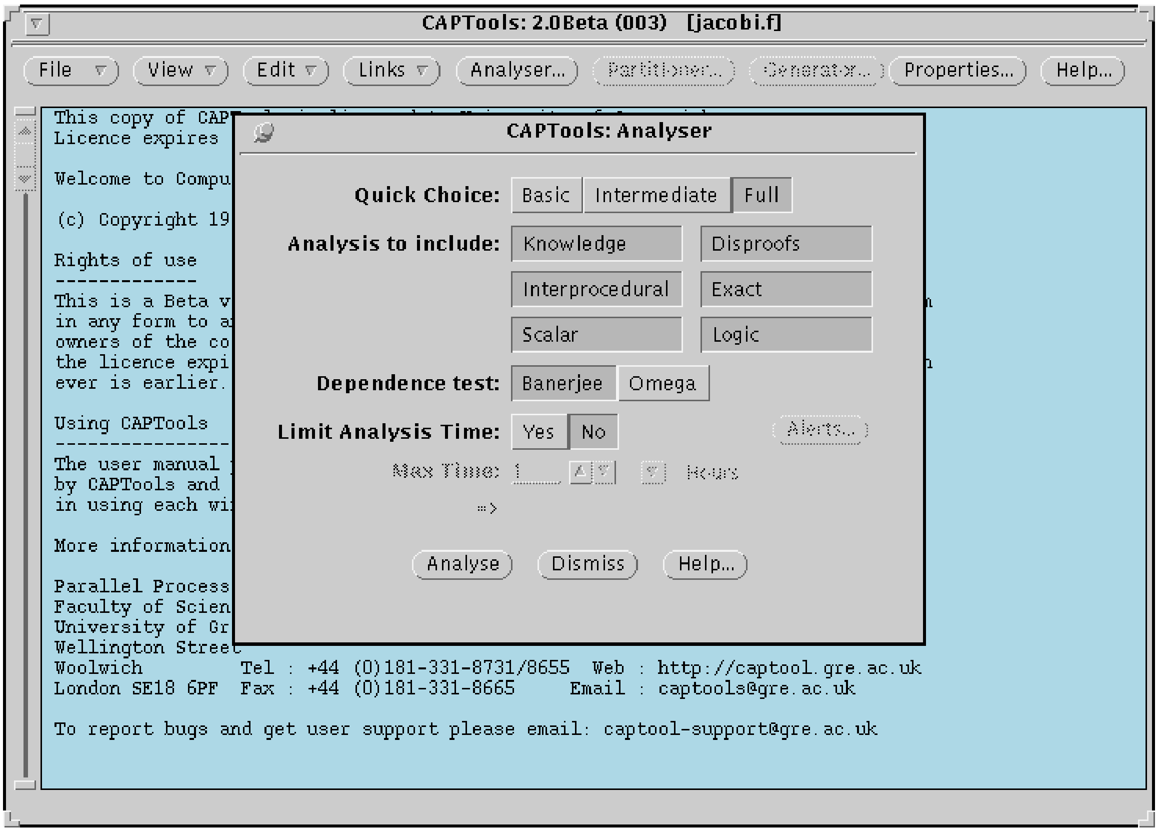


Figure 26 Performing a full analysis on the Jacobi code.

9.2 Partitioning

The Partitioner window is used to decide which arrays to partition and distribute across processors. CAPTools will use this partition definition in the next stage of the parallelisation process in generating parallel code to satisfy the distribution of arrays that the user selects here. The partitioning stage is semi-automatic. The user provides CAPTools with an initial starting array. From this CAPTools will attempt to calculate which other arrays in the code it should partition.

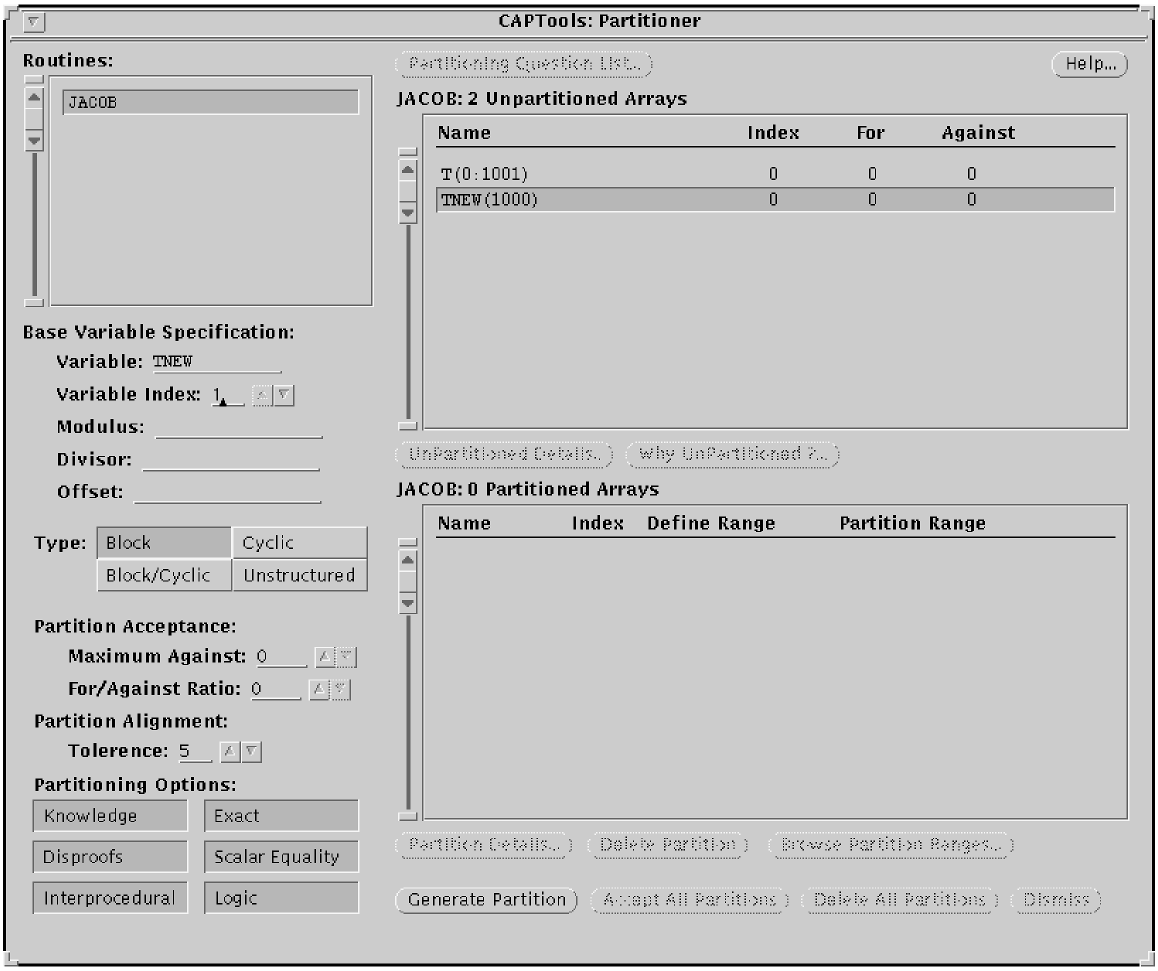


Figure 27 The user selection of the partitioned information in the partitioner.

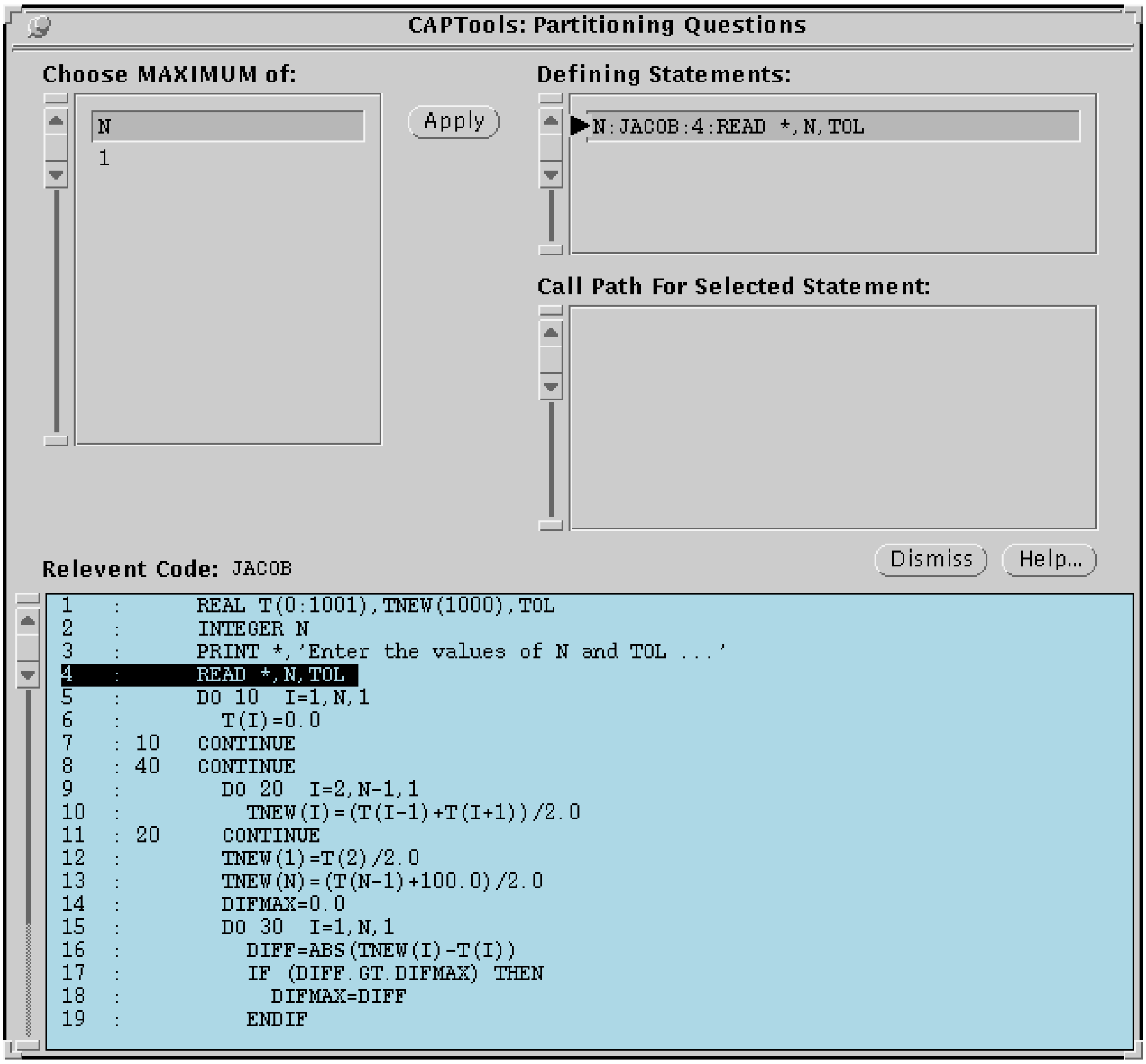


Figure 28 Answering questions about maximum selection in the partitioning questions window.

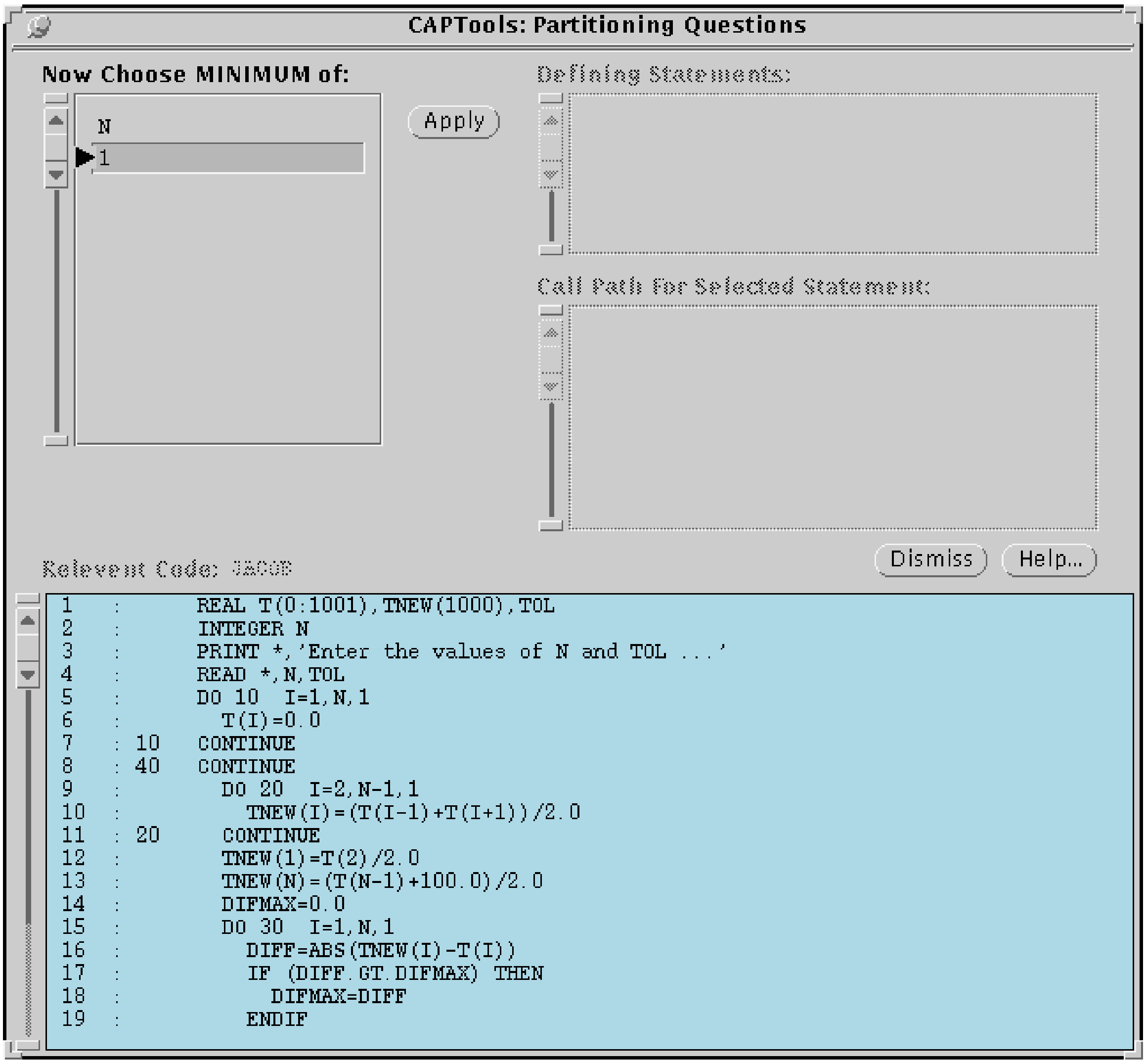


Figure 29 Answering questions about minimum selection in the partitioning questions window.

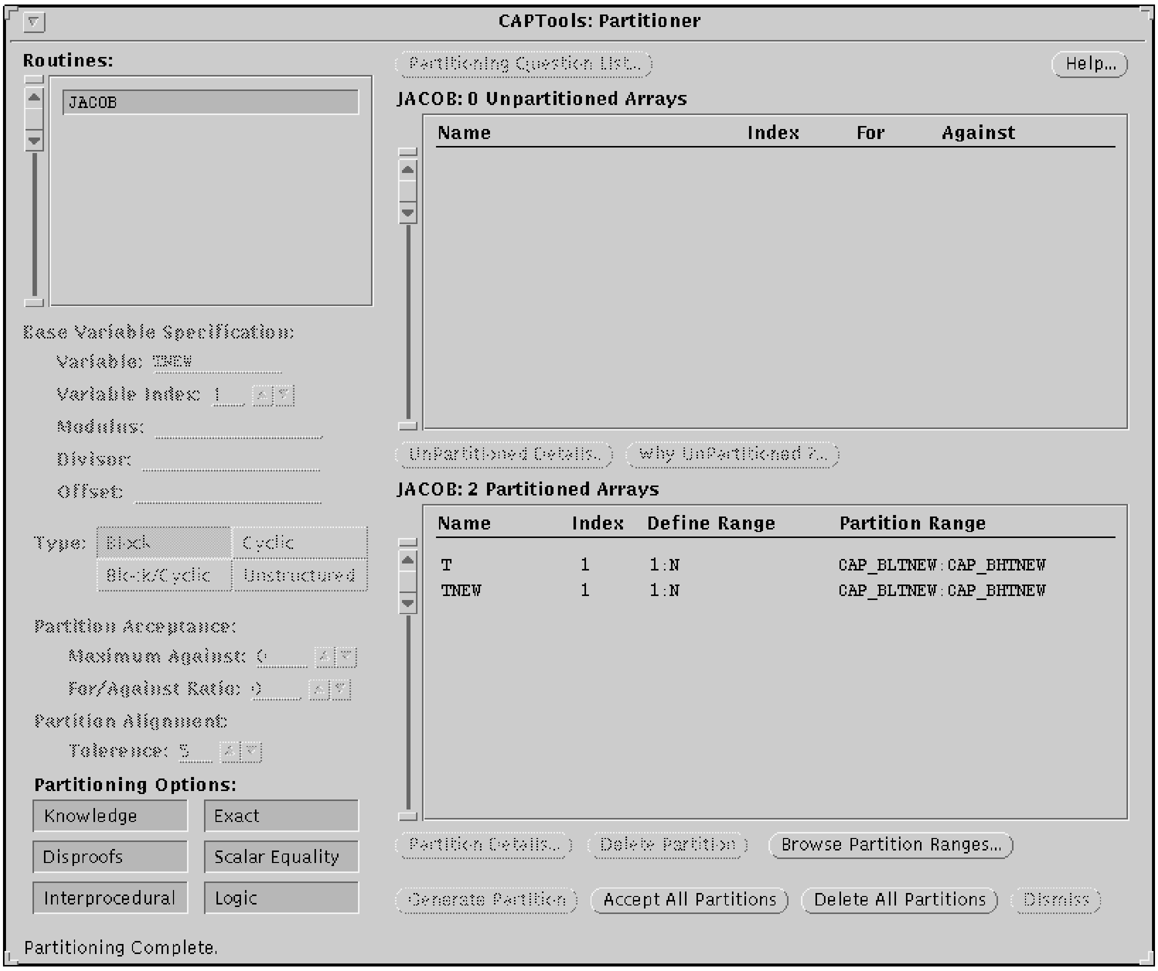


Figure 30 The completed view of the partitioner window.

Select the JACOB main routine from the Routine list and then select the array TNEW from the Unpartitioned Array list. Enter 1 in the index textfield. Now select the **Generate Partition** button to begin the partitioning process (Figure 27). CAPTools now proceeds to calculate a partitioning strategy and will display the results once it has finished. Again, for this simple code the process will take seconds. Partitioning is discussed further in the section headed "USING CAPTools - Partitioning of data using the Partitioner option", basically CAPTools partitions arrays in the code in such a way as to try and avoid the generation of communication statements between processors at the code generation state later.

During the partitioning process CAPTools may need to ask the user some questions. This is the case with the Jacobi code. CAPTools has no information on the value of N (the size of the grid solved by this example program) since it is read in at run time. CAPTools will now pop up a **Partitioning Questions** window and display a series of expressions in a list. In this example the questions asked are:- Which is the maximum of N and 1? Select N in the list and select the **Apply** button (Figure 28); Which is the minimum of N and 1? Select 1 in the list and select the **Apply** button (Figure 29). Once the user has done this CAPTools continues.

After partitioning has completed, each array in the code is now either partitioned or unpartitioned (Figure 30). The user can look at the Partitioned Array list to see which arrays have been partitioned. The list shows that both arrays in the jacobi example have been partitioned (as one would expect). The user can select the **Accept All Partitions** button to confirm that this partition is satisfactory. Once this button is selected there is no returning back to do a re-partition so the user should be satisfied with the partitioning before selecting this button! The user can begin to generate the parallel code.

9.3 Code Generation

Select the **Generator..** button from the main CAPTools window and this will display the Code Generator window. First select the **Masks** button (Figure 31). CAPTools will now calculate execution control masks on statements in the code wherever possible. A mask is merely a FORTRAN IF construct placed at the beginning of a statement to limit its execution to a particular processor. Masking provides parallelism by allowing each processor to execute a different set of statements at the same time, for example, performing different iterations of a distributed loop. See section "USING CAPTools - Execution control mask generation" for a description of the masking process. Now select the **Calc Comms** button followed by the **Generate Comms** button and the **Generate & Save Final Code** button (Figure 32). These last 3 stages complete parallel code generation by placing subroutine calls (to the CAPTools communication library) in the code to satisfy the masking produced in the previous stage. The final stage will pop up a Save File window for the user to enter a filename to save their code into. The user should type in jacpar.f and click on the **Save** button.

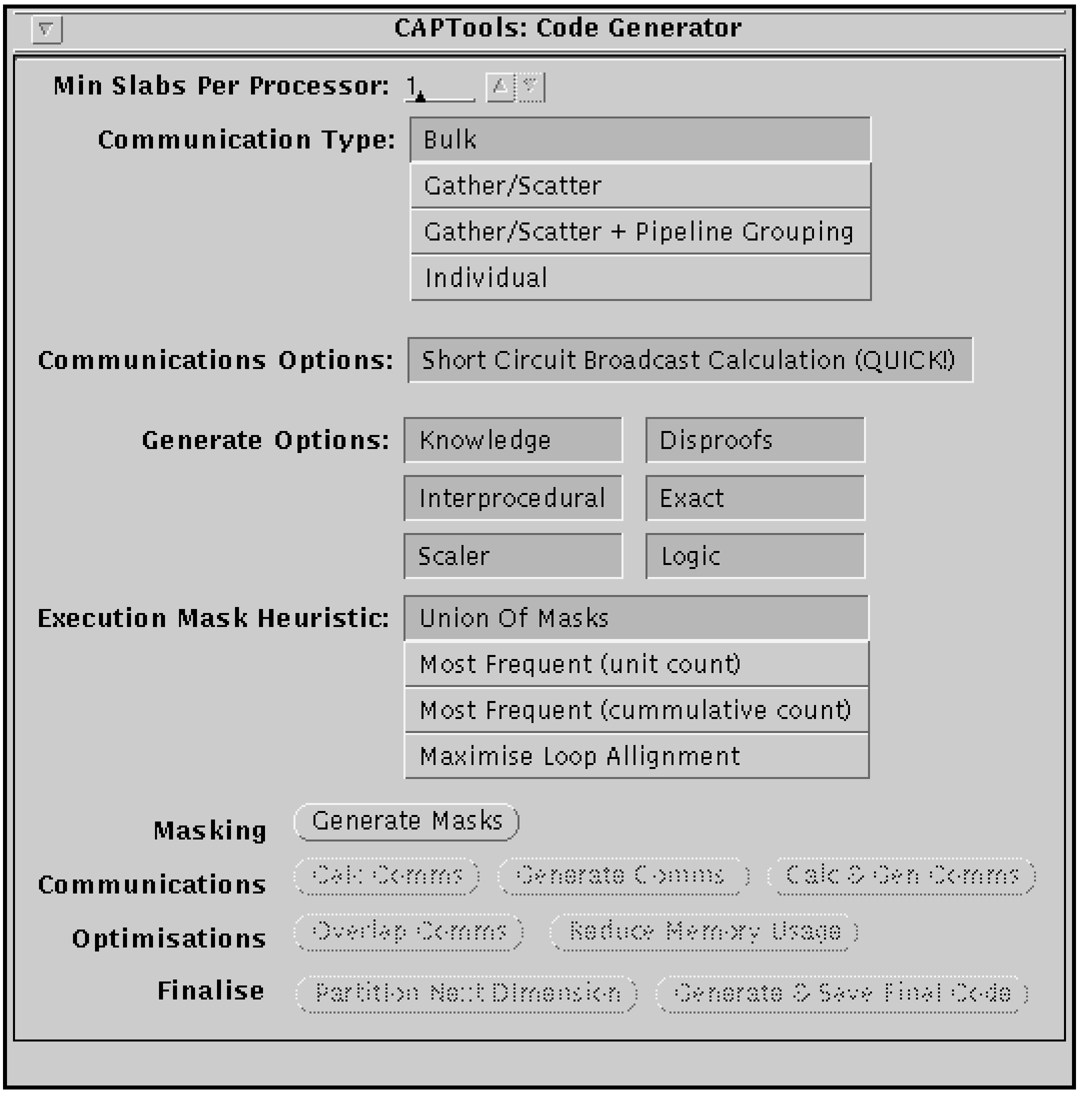


Figure 31 The generation of execution control masks from the code generator window.

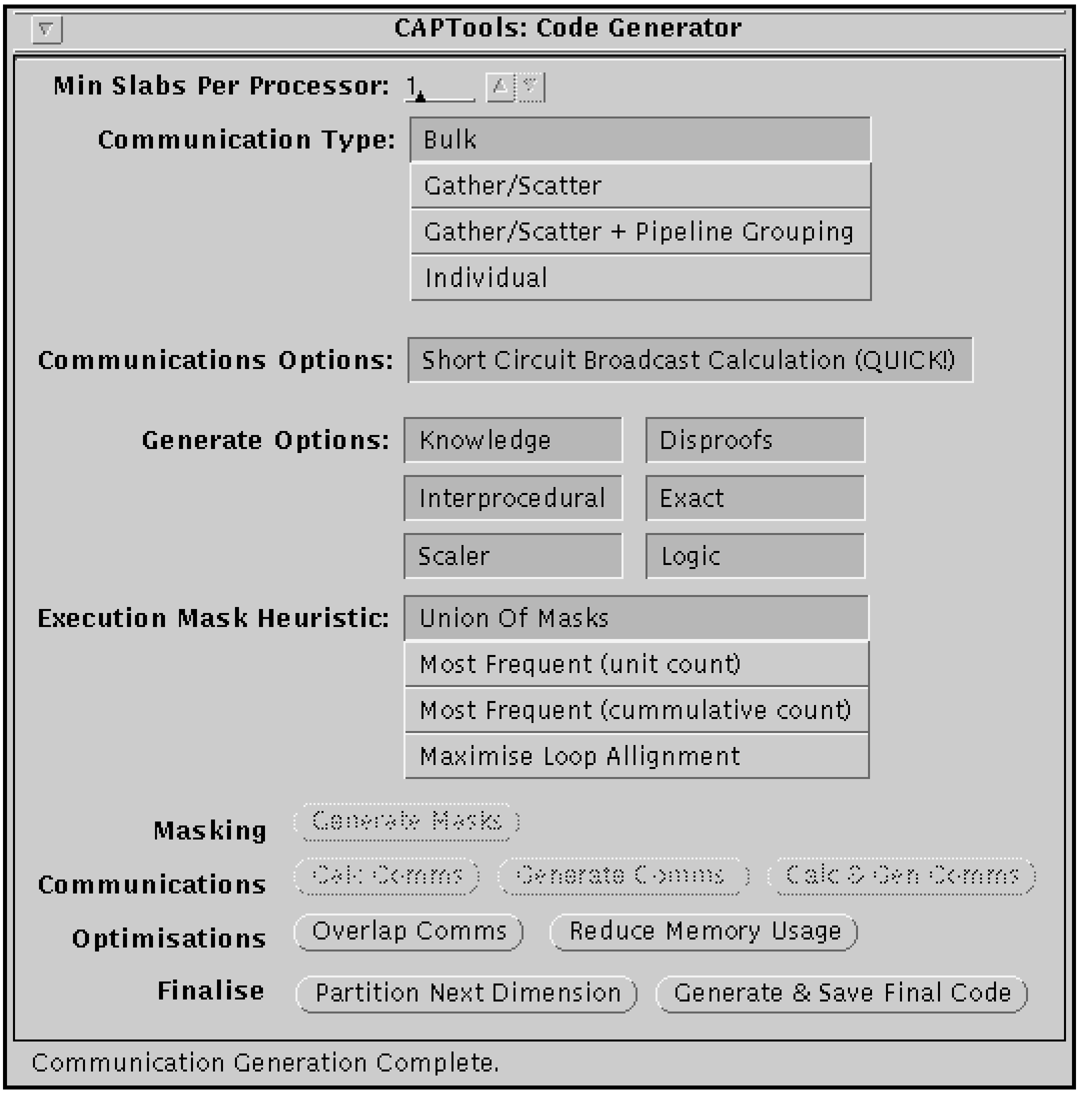


Figure 32 The generation of parallel code from the code generator window.

The communications generated may be viewed from the Comms Browser window. This can be displayed by selecting the **Communications...** menu item from the **View** menu of the main CAPTools window (Figure 33).

Select **All Routines** in this window and the user can see a list of all the communications generated for this code. Note the CAP_EXCHANGE calls to communicate overlap areas and the CAP_COMMUTATIVE call to do the global residual summation as discussed in the section headed "CAPTools communication primitives". Also note that the communications statements used to ensure all I/O is performed correctly are NOT included in the list of communications statements.

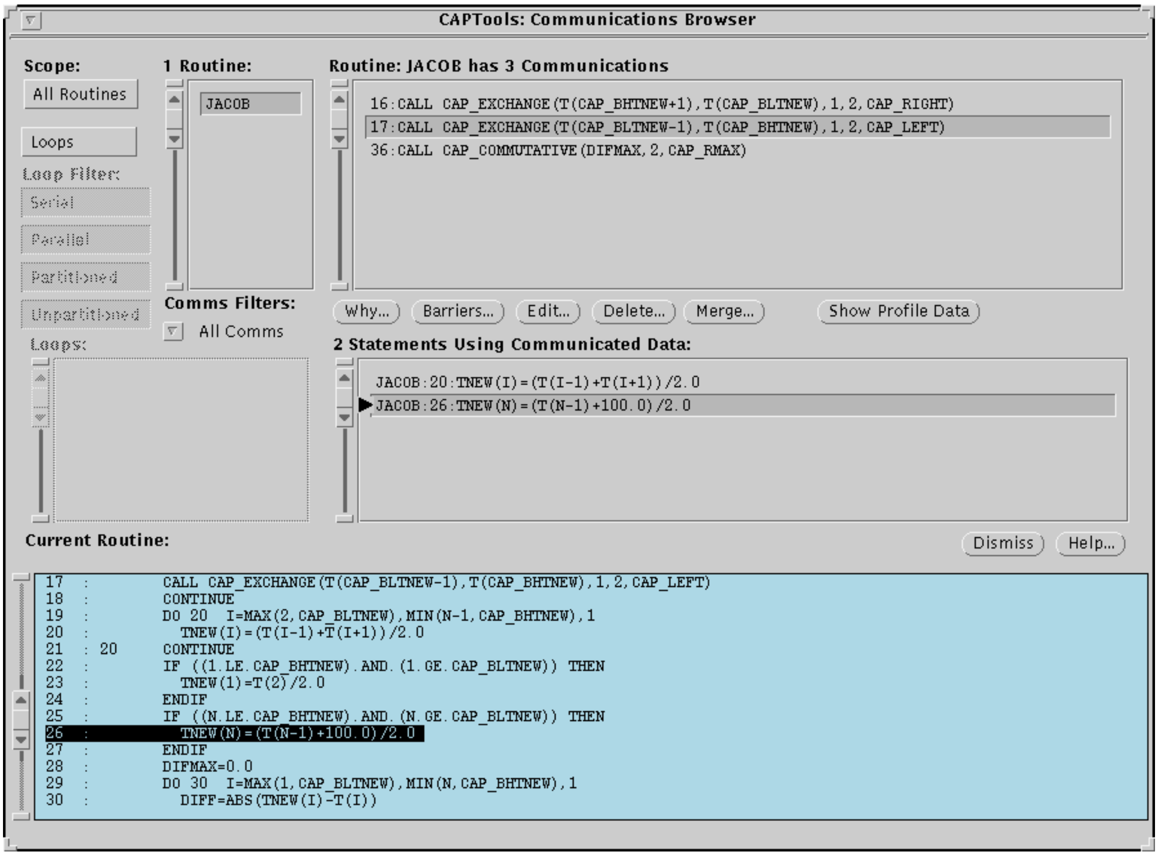


Figure 33 The communications browser window showing the communications generated for the Jacobi code.

9.4 Execution Of The Parallel Code

The code can be compiled and executed as outlined in the section headed "USING CAPTools - Compiling and running the parallel code".

10 TUTORIAL 2: Parallelisation Of The Fab Code.

This tutorial covers a more involved parallelisation than the first tutorial. The FAB code is a code written to solve two dimensional heat diffusion and conduction problems on a structured grid. It consists of about 700 lines of FORTRAN and around 15 subroutines. For the purposes of this tutorial it is assumed that the user is not familiar with the FAB code but have performed a serial profile of the code to identify the key areas where the computational effort takes place.

10.1 Loading The FAB Code

The user follows the same procedure as in the first tutorial. The source code is loaded into CAPTools using the **Load f77 Source** menu item in the **File** menu of the main CAPTools window. The FAB code is in the file fabser.f located in the directory $CAPHOME/examples/fab. Once loaded, the Analyser window appears. For now, the user should ignore this window because before starting the analysis the user needs to provide CAPTools with some information on variables assigned in READ statements. Select the **READ Knowledge** menu item from the **Edit** menu and the READ Knowledge window is displayed.

10.2 Using The READ Knowledge Window

The READ Knowledge window allows the user to enter information on some of the variables in the source code that are assigned by READ statements. CAPTools cannot infer any information on the range of these variables and this can prove very important in dependence analysis by allowing the elimination of possibly false dependencies. The READ Statements list displays all the READ statements in the code. Select the entry for IN and followed by the **Positive Nontrivial** button. The user should repeat this procedure for JN. The variables IN and JN are the dimensions of the problem solved by FAB. By informing CAPTools that the values taken by IN, JN are greater than 5 CAPTools will be able to improve its analysis of the code. Ultimately, this results in a better parallel code. Note also that the information the user has entered goes into the User Knowledge list. We now dismiss this window and move attention to the Analyser window.

10.3 Analysing FAB

The options to the Analyser are discussed in the section headed "USING CAPTools - Selecting the Analyse option". Select the **Full** option from the Analyser window and then deselect the **Logic** option. This is done to allow demonstration of some features of the Partitioner window in the next stage. As a general rule of thumb, the user should probably turn the Logic option off for large programs since the Analysis can take a long time if this option is on. For many codes the user will probably have to re-analyse after answering some questions from CAPTools or has entered user knowledge about the code. The user can always turn on the Logic option in the re-analysis of the code. The user then selects the **Analysis** button, the analysis will takes a few minutes.

10.4 Saving The Database

When using CAPTools to parallelise a code the user will usually have to cycle around the parallelisation stages to achieve the best parallelisation, so it is best to save the analysis at this stage in a database. If the user needs to repeat the next stages of partitioning and code generation this can be done quickly by reading in the saved database. The saving of the database is done by selecting the **Save Database** menu item from the File menu of the main CAPTools window.

10.5 Using The Loop Browser

Once the Analysis is complete the user needs to investigate which loops of the program are parallel and which are serial. A serial profile of the FAB code shows that 95% of the computational effort takes place within the

SOLVER routine. Obviously the loops in this routine are the ones to focus on. Select the **Loops** menu item from the **View** menu of the main CAPTools window. The Loop Browser window is displayed. Select the **One Routine Scope** option and then select the SOLVER routine in the routine list and select **Serial** and **Parallel** in the Loop Filter setting. Now select the **Update Loop List** button and the loops in the SOLVER routine will be displayed. Two loop headers are displayed in the Loop List. Select the DO 30 J=1,JN list entry to highlight this loop in the textpane in the bottom half of the Loop Browser window. Examination of the loops in the textpane reveals that these are iterating over the mesh of the problem with the DO 10 I=1,IN loop within the DO 30 J=1,JN loop (Figure 34).

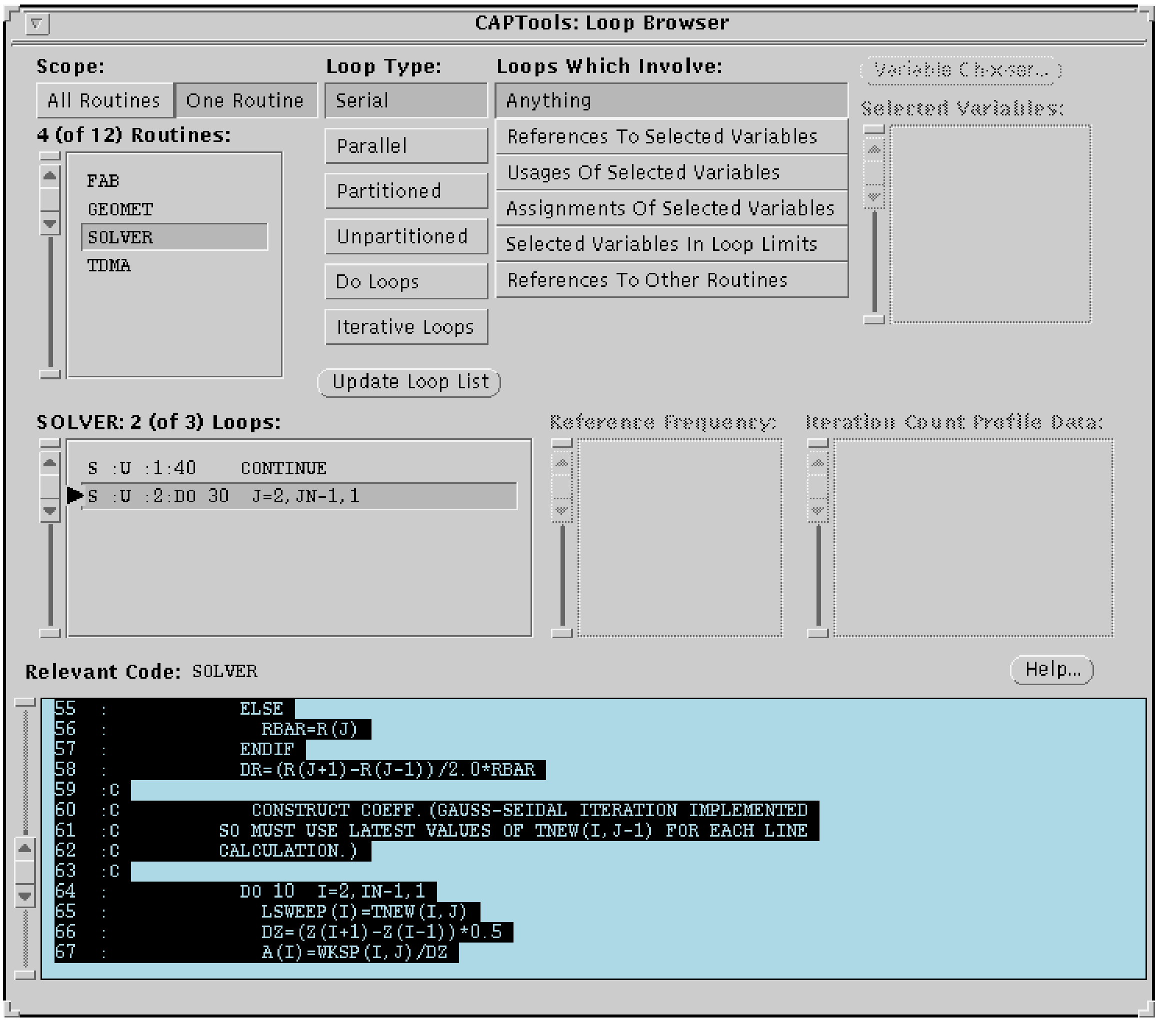


Figure 34 The loop browser window showing the serial loop in routine SOLVER.

Notice that each header in the Loop List has a number pre-pended to it. This is the level of nesting that the loop has in the routine. i.e. the DO 10 loop has a nesting level of 3 and is within the DO 30 loop which has a nesting level of 2. The outer most loop in the SOLVER routine (level 1) is an iterative one and is not displayed in the Loop List but can easily be identified by scrolling the textpane.

Select the **Parallel** setting in the loop filter off and again select on the **Update Loop List** button. The user can now see that the DO 30 loop is a serial loop. A successful parallelisation of FAB on a pipeline of processors will need to distribute the DO 30 J=1,JN loop over the processor topology. Distribution of a serial loop results in the loop executing in a serial pipeline. To achieve a successful parallelisation we need this loop to operate in parallel. We can use the Dependence Graph viewer to investigate why the loop is parallel. Make sure the DO 30 J=1,JN loop is selected in the Loop list and click and hold the right hand mouse button over the list. A

'HotLink' menu appears. Hot links provide quick in context access to other CAPTools windows from lists and textpanes, see section headed "Getting started on CAPTools - List Menus and Textpane Menus" for a more detailed description of HotLinks. Select the **Dep Graph** menu item from the HotLink menu and the Dependence Graph Window will appear with dependencies displayed for the DO 30 J=1,JN loop.

10.6 Using The Dependence Graph Viewer.

The Dependence Graph Browser provides the user with a graphical display of dependencies found during CAPTools analysis, see section headed "USING CAPTools - The Dependence Graph Browser" for more information. The user can look at the dependencies that are serialising this loop. These are data flow, loop carried dependencies between iterations of the DO 30 J=1,JN loop. Selecting the **Dependence Filter** button pops up the Dependence Filter window. Select only **Array** dependencies, **True**(data flow) dependencies and select level 2 dependencies (between iterations of this level 2 loop). Now select the **ReDraw** button. Notice that there is now only one dependence displayed in the graph window. So, there is in fact only one dependence causing this loop to be serial. This is between lines 71 and 73 of the loop and is caused by the assignment to the array TNEW. The user has to decide if it is safe to inform CAPTools to ignore the dependence during the following parallelisation. For more information on this dependence now click on the **Why Dependence?** button to pop up the Why dependence window.

10.7 Using The Why Dependence Window

The Why Dependence window provides the user with all the information on a particular dependence. Select the dependence arc in the graph window using the left hand mouse button. Look at the source and sink lines shown in the window (Figure 35). The source of the dependence is the call to the TDMA routine inside the J loop. The sink is the B(I)=... line, from this information the user can deduce that this dependence is caused by the calculation of TNEW(I,J-1) in one iteration and its usage in the next. If this dependence was deleted from the database, CAPTools would no longer know that the calculation of TNEW(I,J-1) is dependent on a previous iteration and would therefore proceed with code generation assuming the loop was parallel. Obviously deletion of dependencies from the database is very dangerous as we are effectively changing the solution algorithm in the code. However, in this case it is a standard way of achieving a satisfactory parallelisation. The Why Dependence window provides a mechanism for dependence deletion. Selecting the **Remove Dependence** button and a warning message is displayed. Select the **Continue** button and the dependence is deleted. We can now proceed to the first stage of parallel code generation, the partitioning of the code. Click on the **Partitioner** button from the main CAPTools window or use one of the lists on the screen to access the Partitioner window through a HotLink menu.

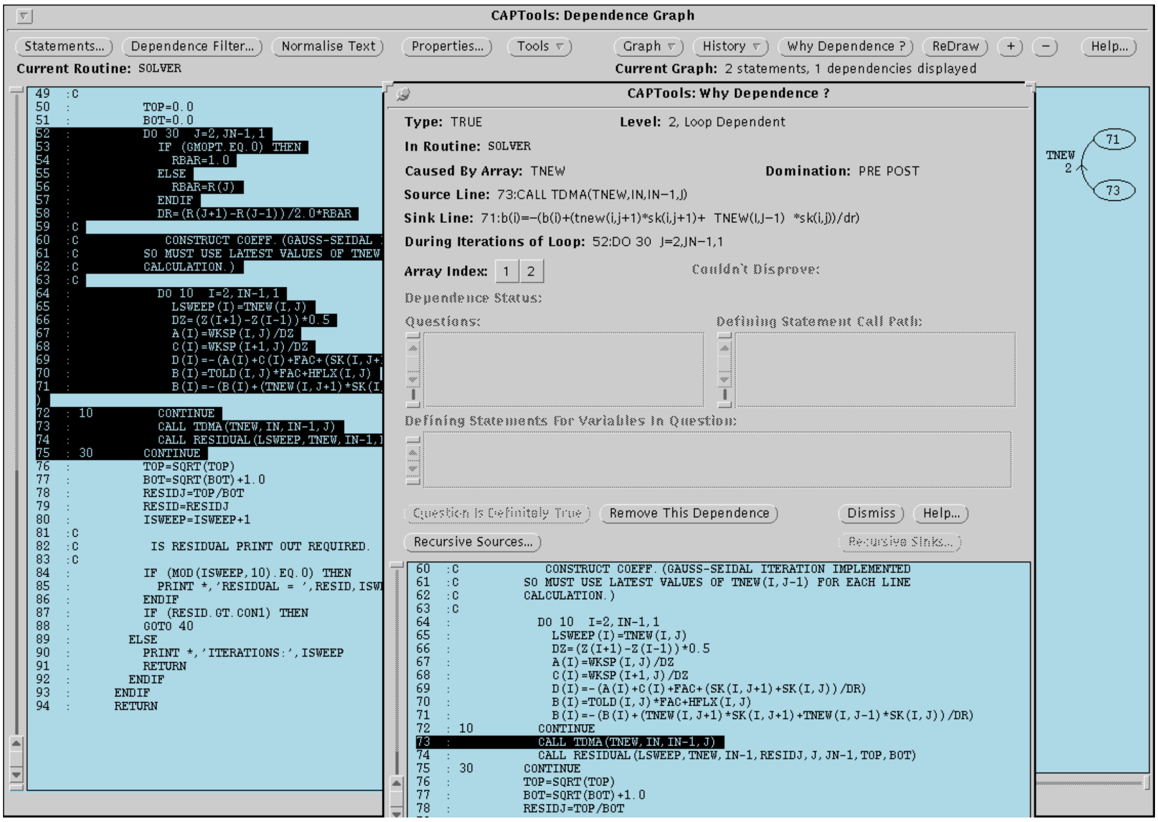


Figure 35 The Why Dependence graph window showing the serial loop in routine SOLVER.

10.8 Using The Partitioner Window

The Partitioner window provides the user with a mechanism for partitioning or distributing appropriate arrays in the code across processors. The process of partitioning is semi-automatic. The user provides a starting array and index, (optionally set a few parameters) and CAPTools will suggest a partitioning of those arrays that would best be partitioned along with the starting array to produce what it thinks would be the best parallel code. For instance, our choice of starting point will be the array TNEW in index 2 since this is the index of TNEW that is updated during iterations of the SOLVER routine DO 30 J=1,JN loop. A partition of TNEW in index 2 across processors would result in the distribution of the DO 30 loop. An obvious candidate for partitioning also is TOLD in index 2 since this holds previous values of TNEW. If TOLD was not partitioned and held on one processor, an awful lot of communication would have to take place at the end of each outer iteration. Therefore, CAPTools will automatically partition TOLD when TNEW is partitioned. So, the user should select the SOLVER routine in the Routine list. The Unpartitioned Arrays list is updated to show all arrays in the SOLVER routine. Select the TNEW entry from this list and the **Partition Generation** setting will enable on the left hand side of the window. Select Variable index 2 and then click on the **Generate** button. CAPTools now commences partitioning of the entire code based on the initial partition the user has just entered. This process takes a few minutes.

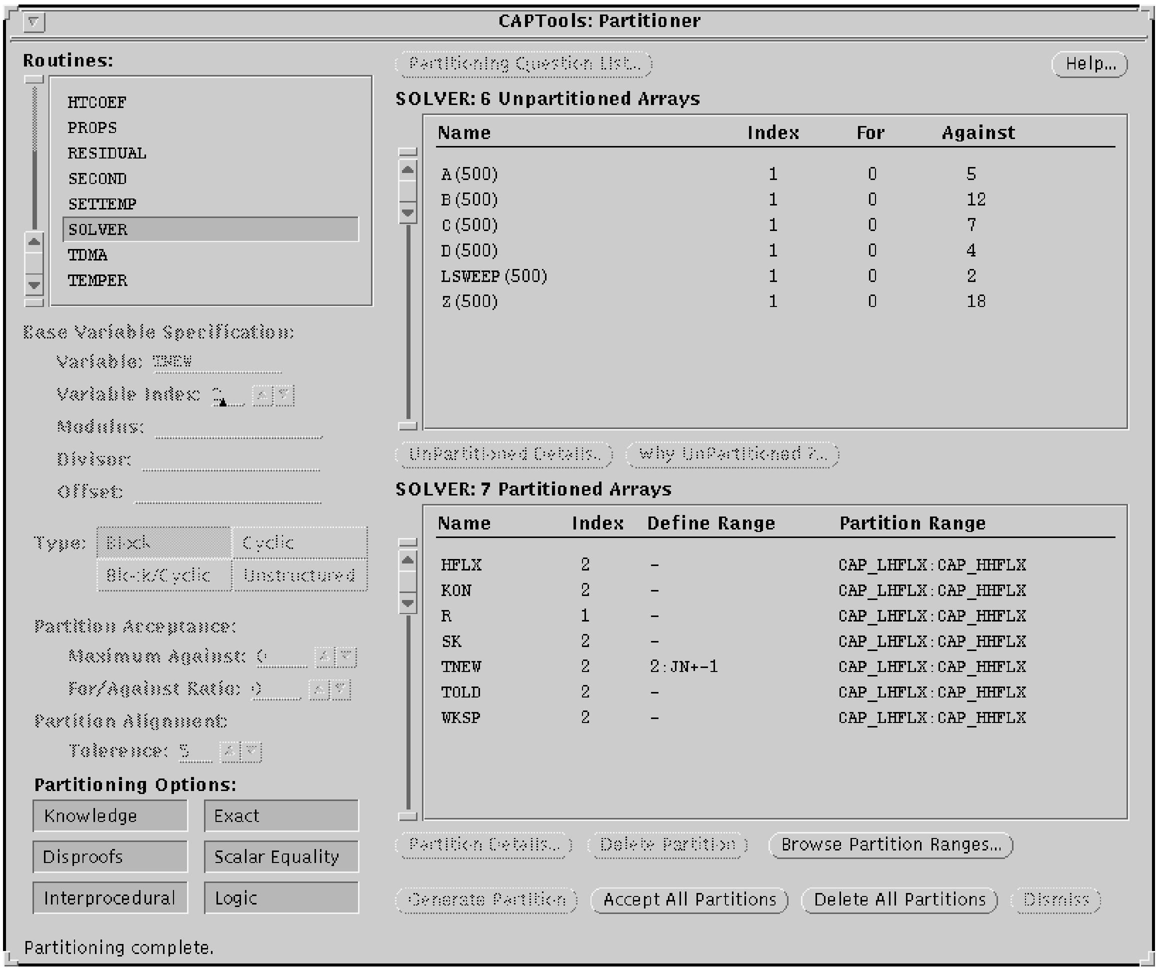


Figure 36 The Partitioner window showing the arrays partitioned in routine SOLVER.

When partitioning is complete the Partitioned Array list shows those arrays that have been partitioned (Figure 36). Those that have not, remain in the Unpartitioned list. Note that TOLD has also been partitioned in index 2 along with several other arrays. By selecting different routines from the Routine list the user can browse the entire code and see which arrays have been partitioned and those that have not. Note the fields For and Against in the Unpartitioned Array List. During the partitioning process CAPTools built up a list of reasons for and against partitioning each array in each index. These are listed now for examination using the **Why UnPartitioned ?...** button and the UnPartitioned Info window. See section headed "USING CAPTools - Partitioning of data using the Partitioner option" for a more detailed description of these features.

At this point the user is free to change the partition CAPTools has generated by deletion of individual array partitions or can partition more of the code by selecting another array for partitioning. The user should normally save the database at this point using the **Save Database** menu item of the **File** menu of the main CAPTools window so that it may be possible to return at any time to this point in the parallelisation process. Finally, to move onto the next stage in parallel code generation the user needs to accept the current partition definitions by selecting the **Accept All Partitions** button.

10.9 Using The Code Generator Window

The next stage in the parallelisation of the FAB code is to generate execution masks. The concept of masking is described in the section headed "USING CAPTools - Execution control mask generation". Select the **Generator** button of the main CAPTools window. The Code Generator window is displayed. This window is described in the section headed "USING CAPTools" and the subsections therein. Selecting the Masks button, CAPTools now proceeds to generate masks on those statements assigning partitioned data directly or indirectly. This process takes very little time for the FAB code. Masking is an automatic process carried out by CAPTools but may require user addition or deletion of masks in some cases. The user should usually save the database now so he can return to this point in the parallelisation process at any time. The FAB code needs no user intervention in the calculation of masks so the user can proceed to select the **Calc Comms** and **Generate Comms** buttons. CAPTools now calculates and generates communication statements based on the masking in the previous stage.

10.10 Saving The Final Code

We have now reached the final stage in parallel code generation. Select the **Generate & Save Final Code** button and the Save Parallel Code window is displayed. Enter fabpar.f as a name for the parallel code and select the **Save** button. CAPTools now generates the final code into the named file.

10.11 Viewing Communications Using The Communications browser window

The user can view the generated communications using the Comms Browser window. This window is displayed using the **View** menu of the main CAPTools window. Select All Routines and all the communications in the code will be displayed. Notice that there are very few communications in the entire code. The interesting communications are those in the SOLVER routine. Select the SOLVER routine in the Routine list. Click on one of the communications to display the code in the textpane. By browsing the code one can see that CAPTools has generated two EXCHANGE communications to update overlap areas just before the DO 30 J=1.JN loop. Note that there are no communications inside the inner I andJ loops. These loops will proceed in parallel without any communication occurring until the next outer iteration begins.

10.12 Execution Of The Parallel Code

Depending on which parallel environment you wish to run under follow the instructions in the section headed "USING CAPTools - Compiling and running the parallel code". The FAB code requires a datafile called input.dat. A selection of problems can be found in the FAB examples directory $CAPHOME/examples/fab, e.g. small.dat, medium.dat and large.dat, for now copy large.dat to the current directory.

Before each run of FAB the user should copy the large.dat file to input.dat. Do this because if the FAB example is stopped before completion the user may find the input file has been overwritten. Once the FAB code is running the CAPTools library will be initialised and prompt for a processor topology or ask how many processors to run on. Once this question is answered FAB now asks

DO YOU WISH TO USE EXISTING PROBLEM SETUP (Y OR N) ?

reply Y to this question and FAB will then read in the data file and begin iterating. Iteration residuals are printed out every 10 timesteps.

11 TUTORIAL 3: Parallelisation Of A 2D CFD Code - TEAMKE1.

The TEAMKE1 code is a two dimensional steady state flow prediction code from the University of Manchester Institute of Science and Technology (UMIST). Laminar and turbulent predictions can be made on a structured grid, and the linear equation solver used is a bi-directional LSOR. The code is approximately 2000 lines of F77 and contains 13 subroutines. The user is provided with a 15x15 grid problem (standard as supplied by UMIST) and also a modified 288x288 grid problem set up by the University of Greenwich.

There are two basic changes made to the standard TEAMKE1 code. The first is the inclusion of a READ statement to allow the applications programmer to select their choice of discretisation scheme, rather than setting the variables NQUV, NQKE and NQT through an assignment statement. This would prevent for example, the QUICK scheme from being removed by the dead code elimination algorithm and instead will be included in the generated parallel code. The second were minor changes to the routine GRID to modify the problem size from 15x15 to 288x288 and to all the other routines containing a reference to the 15x15 problem.

11.1 Control Analysis

The source code called TEAMKE1 is loaded, checked for validity of syntax and subsequently converted to an internal form. A control analysis is then performed.

11.2 Dependence Analysis

The initial dependence analysis is based on an intermediate level, i.e. there is no use of logical substitution of symbolic variables through the dependence analysis. The inclusion of the **Logic** analysis option will add to the analysis time and it is not known at this early stage in the CAPTools parallelisation cycle whether or not it is a necessary aid to the overall analysis.

11.3 Examination Of Unresolved Inequalities

This version of the TEAMKE1 code has a single READ statement, therefore if any additional information is to be supplied by the CAPTools user (referred to from here on as the user) it will need to be presented to the system in a different way.

11.4 Examination Of The Loops In The TEAMKE1 Code

It is possible for the user to examine all the loops in the code using the dependence graph viewer, this is done by selecting the **Selected Loop** button in the **Statements** pop up window. Selecting the **Serial** loop type it can be shown that there are potentially 43 serial loops in the code and by selecting the **parallel** loop type there are 53 potentially parallel loops in the code. At this stage a Unix profile of the TEAMKE1 code reveals a large portion of the time is spent in the routine LISOLV. Examination of the LISOLV routine shows that it has 7 potentially serial loops. Given this information it is worthwhile investigating these loops and to establish their probable impact on the parallelisation of the code. An inspection of the loops reveals the following:-

- All potentially serial loops reside within the serial loop 2000.
- The variable NSW can be traversed to its definition using the variable browser and is set to either 1 or 2 depending on the flow variable being solved.

Hence the loop 2000 is used as an iterative loop for each variable being solved, this determines how many times the computation domain is swept but does not sweep over the domain itself. Therefore, loop 2000 is ignored as a basis on which to define the partitioning of the domain.

Loops 100, 101 and 102 sweep through the domain in a west-east direction where the line i=constant is being solved each time. The loop head of loop 100 ranges from istart (either 1 or 2) to NIM1 - the number of cells solved for in the i-direction of the computation domain. It would appear that loop 100 could be a good starting point for partitioning the domain, however, this loop forms the basis for the LSOR algorithm which executes in a sequential manner. The previous tutorial "TUTORIAL 2: Parallelisation of the FAB code" shows how such a loop can be parallelised and will not be repeated here. A similar approach is taken for loops 1000, 1010 and 1020 which sweep through the domain in a south-north direction with the line J=constant.

To summarise, the sequential algorithm comprises a bi-directional LSOR algorithm - an i-directional sweep followed by a j-directional sweep. The effect of partitioning the domain in either direction will result in the pipeline of one direction and the serialisation of the other. Selecting the j-direction as the one for partitioning then loop 100 is serial and 101 and 102 are pipelined, loop 1000 would be pipelined and loops 1010 and 1020 would be serial.

The pipelined loop 1000 could be modified as in the previous tutorial so that the algorithm is made more explicit and as the previous tutorial has shown, this can enhance the parallel performance of the algorithm.

11.5 Prescribing A Data Partition Strategy

The way in which the user defines a data partition has already been discussed in the previous tutorial. However, it is worth discussing some general points which should be considered when choosing a suitable data partition.

- The parallelism in loops can usually be exploited to good effect if the outermost loop is distributed as a result of the selected partition. This may appear in an obvious way to the user, or it may be necessary to perform loop interchanging at a later stage, this re-arrangement of loops would exploit the parallelism inherent in the loops.
- Examination of all possible/probable loop limits can be very informative in identifying which index of an array has the largest numerical value and could therefore be considered as a suitable candidate.
- The knowledge gathered from the interrogation of the code through the dependence graph is also a vital source of information and can be used by the user to asses the likely effect of a chosen data partition. For example, a code which has many potentially serial loops will need more attention from the user than one with many potentially parallel loops. The dependence graph allows the user to understand why the loops are serial and has the capability to identify what information is needed in order to exploit the parallelism.
- Some specific knowledge about the code could assist the user in deciding which array and which index of that array to select. For example, in a CFD code it is reasonable to choose a velocity component array since it is known from the governing momentum equations the velocity components are related to other variables such as pressure, which is in turn related to temperature (enthalpy) and so on. The choice of index could be based on the discretisation scheme used, knowledge about the way in which the array memory is accessed by the processor etc.

In the case of the TEAMKE1 code, we select from routine CALCV, the j-velocity component array V and index 2. Since the problem grid size is square either index could be used, however, selecting index 2 ensures that the data to be communicated will be stored contiguously in memory. Therefore, the communication can be performed without the need for gather and scatter operations.

After pressing the **Generate Partition** button, CAPTools will calculate a comprehensive data partition (Figure 37). The user can then use the main partitioner window to browse any reasons why a partition was not set to a particular variable, with the possibility of adding or deleting variables from the data partition or even re-partitioning. In the case of the TEAMKE1 code, the data partition is satisfactory and the user then selects the **Accept All Partitions** button.

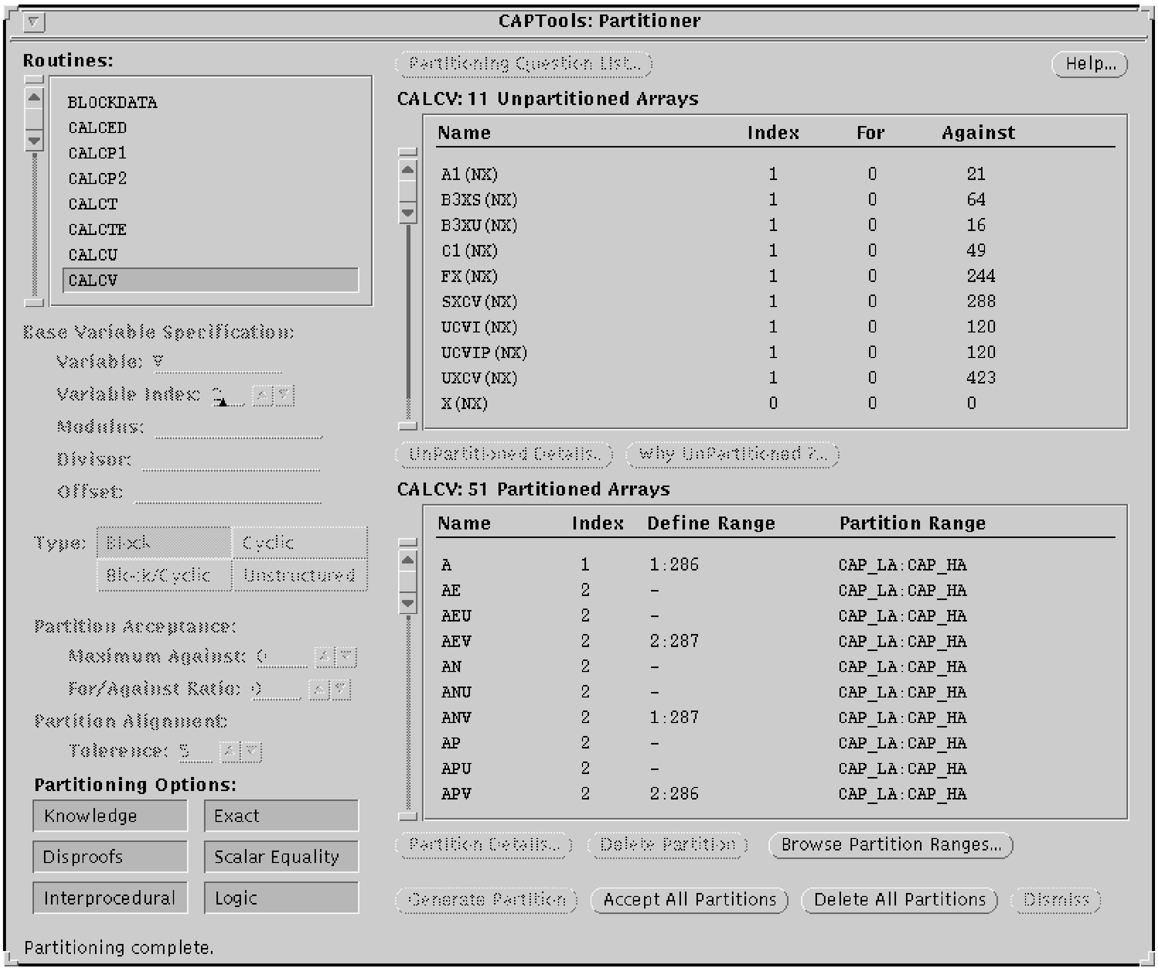


Figure 37 The Partitioner window showing the arrays partitioned in routine CALCV.

11.6 Code Generation

It has already been stated in the previous tutorials that the code generation is made up of a series of stages. This process allows the user to interrogate and refine each process.

The user selects the **Generator** button from the main CAPTools window, the resulting Code Generator window defines the minimum number of lines/slabs per processor, the default is set to 1. However, because the QUICK scheme is a second order differencing scheme and therefore each processor will require two overlap slabs. The user could set the communications type to bulk since the data to be communicated is stored in a contiguous fashion. The user then selects the "generate masks" button to create the control execution masks.

The mask window shows the effect of adding control execution masks to each statement to ensure correct processor execution of the parallel code. The user can examine the following:-

- Why a series of masks were not inherited. For example, why masks still reside within a do loop.
- The statements which provide data for the control of the masks.

Browsing the masks can be particularly useful when coupled with the generated and placed communication statements. The user can calculate and generate communication statements in either two steps using the **Calc Comms** and **Generate Comms** buttons, or alternatively, in a single step using the **Calc & Gen Comms** button.

The user can get a good insight into why a communication was generated, what the mask on the statement is and how often it will execute according to the profiler (not yet implemented). An example where some interesting communication statements have been generated can be seen by selecting routine CALCU, and highlighting the communication statement

```
CALL SEND(ASU(I,J),1,2,CAP_LEFT)
```

Examination of the control mask can be obtained by using the right button of the mouse and selecting the **Masks Browser** option, this accesses the mask window through a hot link. The mask shows that the communication SEND will only take place when the J index is equal to the first line of the partitioned data (J=CAP_LOW). The corresponding communication RECEIVE for this data occurs when J=CAP_HIGH+1i.e. in the overlap area of the left neighbouring processor (Figure 38). Note that the communication is not a pipelining or serialising communication since it does not form a recurrence (detected as a strongly connected region with the dependence graph).

```
ASU(I,J) = ...
ANU(I,J-1) = ASU(I,J)
```

i.e. the first iteration on each processor assigns ASU which is needed on the last iteration on the previous processor. Communication must be performed after assignment in the first iteration and before the usage in the last iteration. The communication will not enforce idle time on any processor, as would be the case with a recurrence. This communication, which is also found in other CALC routines will have very significant effects on the parallel code and the parallel performance. For each call to communicate this data there is only one data item, this will incur a large startup latency penalty on some parallel systems (e.g. workstation clusters using the Ethernet). Thus, a single data item is sent NINJ times for each call to the CALC routines.

Some communications can be handled more effectively by performing a loop interchange of its I and J loops. For example, this will have the effect of allowing the communication to move between the outer J and inner I 100 loops and the message lengths have increased from1 to NI-2. The loop interchange transformation will be in a later version of CAPTools.

Other communication statements placed in routine CALCU which communicate scalars such as GAMN and DVDXN show how important it is to view the generated statements and to understand the reason(s) for their placement. For example, the assignment and usages of the scalar DVDXN hold the key to the communication statements that have been generated. The following section of code shows this,

```
     DO I=2,NIM1,1
 s1: communication for DVDXN
       DO J=2,NJ,1
 s2:   communication for DVDXN
 s3:   pipeline for DVDXN,GAMN
       ...
       ...
       IF (J.NE.NJ) THEN
         DVDXN=RV(J)*(V(I+1,J)-V(I,J))/UXCV(I)
       ENDIF
       DVDXS=RV(J-1)*(V(I+1,J-1)-V(I,J-1))/UXCV(I)
       ...
       ...
       SU(I,J)=(DUDXE*VIS(I+1,J)-DUDXW*VIS(I,J))/UXCV(I)
     &   +(DVDXN*GAMN-DVDXS*GAMS)/RSYCV(J)
     &   +(P(I,J)-P(I+1,J))/UXCV(I)
```

Clearly, for iterations of the J loop the scalar DVDXN is computed for $2 \leq J \leq NJ-1$ and is used in the range $2 \leq J \leq NJ$. Therefore it is possible to assign DVDXN at J=NJ-1 and to use this value at J=NJ(which could be computed by a different processor), this is the reason for the communication statement at s2. However, if we look at the algorithm we can see that DVDXS is DVDXN in the previous iteration of J (similarly for GAMS and GAMN) so that when J=NJ the expression DVDXN*GAMN-DVDXS*GAMS must always equal zero.

For iterations of the I loop we can see that s1 has been generated because it may be possible for DVDXN set at the end of one I iteration to be used at the start of the next I loop. However, this is not the case since DVDXN is assigned on the first iteration of the J loop with the mask IF(J.NE.NJ). Therefore, we can remove the communication statement at s1. Later versions of CAPTools will not set such dependencies.

As part of the coefficient generation process in the CALC routines (typified by DO 100 loop in CALCU) the source term array SUU is set up as follows:-

```
      NSUP=IFIX(SIGN(1.0,CW))          SIGN       +1.0  -1.0
      ...                              IFIX         +1    -1
      ...                              NSUP         +1    -1
      ...
  111 KYP=J-NSUP                       KYP         J-1   J+1
s1:   LYP=KYP-(1+NSUP)/2               (1+NSUP)/2   1      0
      ...                              LYP         J-2   J+1
      ...
      ...
s2:   SUU(I-1,J)=SUU(I-1,J)-TEMPM*U(I,LYP)
```

The migration of the communication of array U is blocked by the assignment of array index LYP. Here the index LYP is dependent on the scalar NSUP, from this CAPTools can determine that NSUP can only ever be set to-1 or +1. As a consequence, CAPTools can determine that LYP can only ever be set to J-2 or J+1. Therefore, the communications are traversed and merged outside the nested loops, allowing bulk communications and possible merging with other communications.

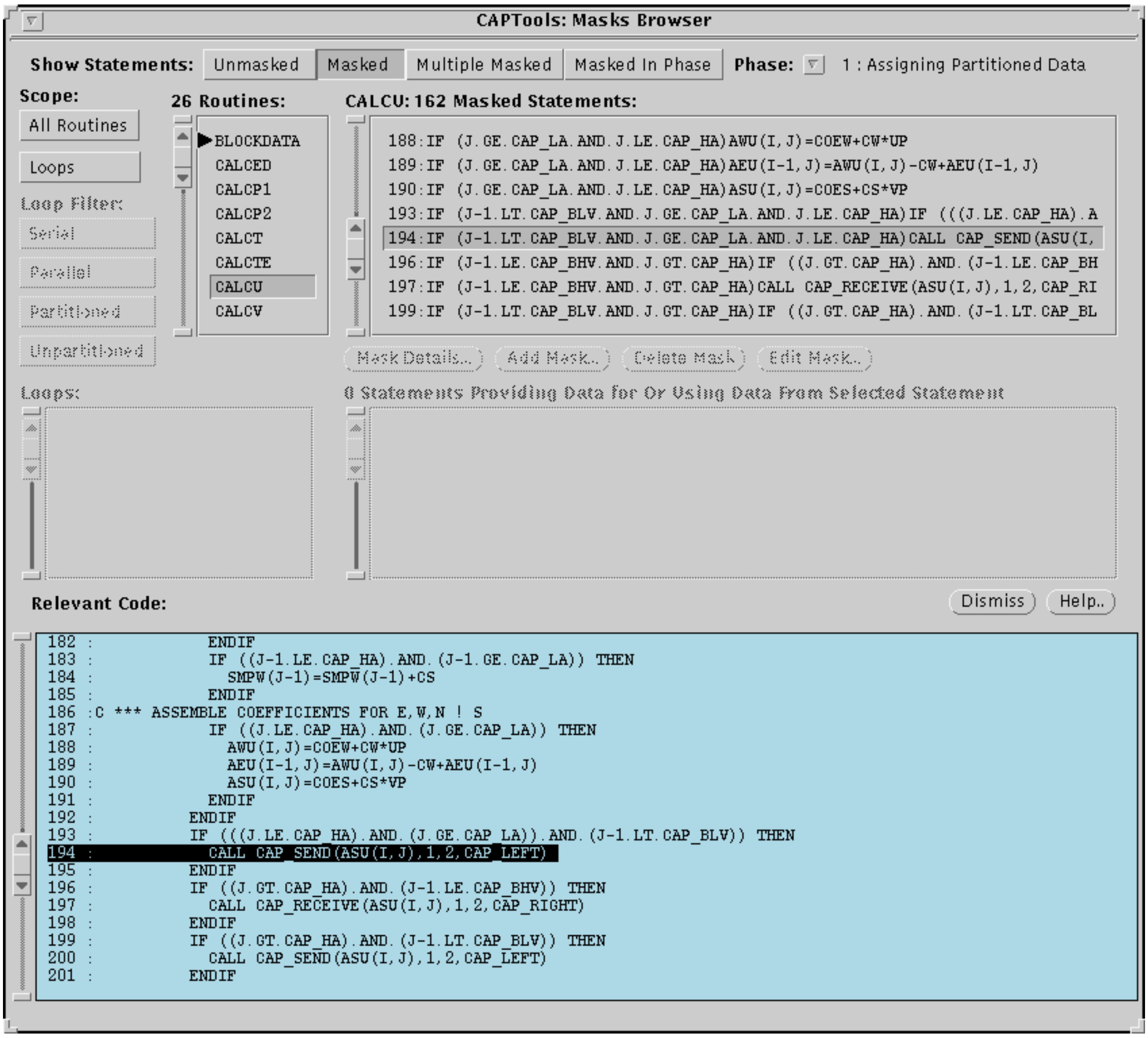


Figure 38 The Mask window showing the execution control masks for routine CALCU.

11.7 Execution Of The Parallel Code

Depending on which parallel environment you wish to run under follow the instructions in the section headed "USING CAPTools - Compiling and running the parallel code". The code runs without the need for input data files. Once the code is running the CAPTools library will be initialised and prompt for a processor topology or ask how many processors to run on. Once this question is answered the user is asked:

Type 0 or 1 to choose basic upwind or Quick scheme for velocities, turbulence and temperature

Reply 1 0 0 to this question and the code then proceeds with the execution.

12 TUTORIAL 4: Parallelisation Of A NAS-PAR benchmark code – APPLU.

The APPLU code is a code from the NASA Parallel (NAS-PAR) benchmark suite. The NASPAR benchmark codes were developed by the NASA Ames Research Center to evaluate the performance of parallel supercomputers. The APPLU code is the lower-diagonal (LU) CFD application benchmark. However, it does not perform a LU factorisation but instead implements a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system. The code is approximately 3300 lines of Fortran 77 code and consists of 17 subroutines.

12.1 Control Analysis

The source code is first loaded into CAPTools. The analysis should first of all be carried out using only the **Disproof**, **Exact** and **Scalar** analysis options. This will produce a quick analysis of the code and enable the browsing of the Requested knowledge window obtained through the **Edit** menu. In doing this it is possible to establish if there is any information requested by CAPTools which could be supplied by the user to give an improved analysis. Looking at the questions posed by CAPTools it could be seen that there was an extremely large number of questions relating to the variables NX, NY and NZ. Selecting these variables in the question list showed that the values of NX, NY and NZ are defined in a READ statement. The READ statement reads the data from the input file inputlu.data on unit 5. Therefore, CAPTools cannot determine the value or the sign of these variables. From looking at the input file it can be seen that the variables NX, NY and NZ define the grid size of the problem and are therefore compelled to be positive variables. The other variables in the question list of the Requested Knowledge window have far lower occurrences and are not considered important at this stage.

The code is now re-analysed with all options and provided with the user knowledge for NX, NY and NZ. The read knowledge information is provided before the analysis, via the Read Knowledge window with the variables NX, NY and NZ being positive non-trivial. The analysis is then performed using all six analysis options.

12.2 Using The Loop Browser

After the analysis has completed it is necessary to inspect the dependency graph. The most important thing to examine is which loops are serial within the code. It will also give an immediate indication of how much of the code it is possible to parallelise. In this code there are 135 potentially parallel loops to 33 potentially serial loops thus indicating that a good parallelisation could be obtained. It must also be stressed here that it might also be possible to generate parallelism out of some of the serial loops by means of pipelining the data and commutative operation detection, e.g. summations.

12.3 Prescribing A Data Partition Strategy

From browsing the code it is suggested that one of the best variables to base the partition upon would be array A in the main routine APPLU. This variable was chosen because it is used or assigned in four other routines, and could thus transfer its partition information to other variables in those routines. These other variables could in turn transfer their partition to other variables etc. The variable A was also chosen since it was a 5-dimensional array. The higher the number of dimensions in the partitioned array then the more for and against reasons for partitioning are generated and the better the partition.

Looking at the size of each of the dimensions of the array A it can be seen that the first two array dimensions are set to 5. These therefore are not the best dimensions to base the partition since the number of slabs on each processor would be so small and the maximum number of processors with a balanced workload (of 1 slab each) would be five processors. Looking at the last three dimensions of this variable the sizes are set to ISIZ1, ISIZ2 and ISIZ3 respectively. These are adjustable and are presently set in the APPLU code to 12. It can therefore be seen that any one of these dimensions may be used as a basis for the partition and would ensure that each processor would have a sufficient number of slabs. However, it would be more advantageous to select the final index as the communication would allow communication of contiguous array data and thus allowing bulk communication of data as opposed to smaller size data communications with numerous start up latencies.

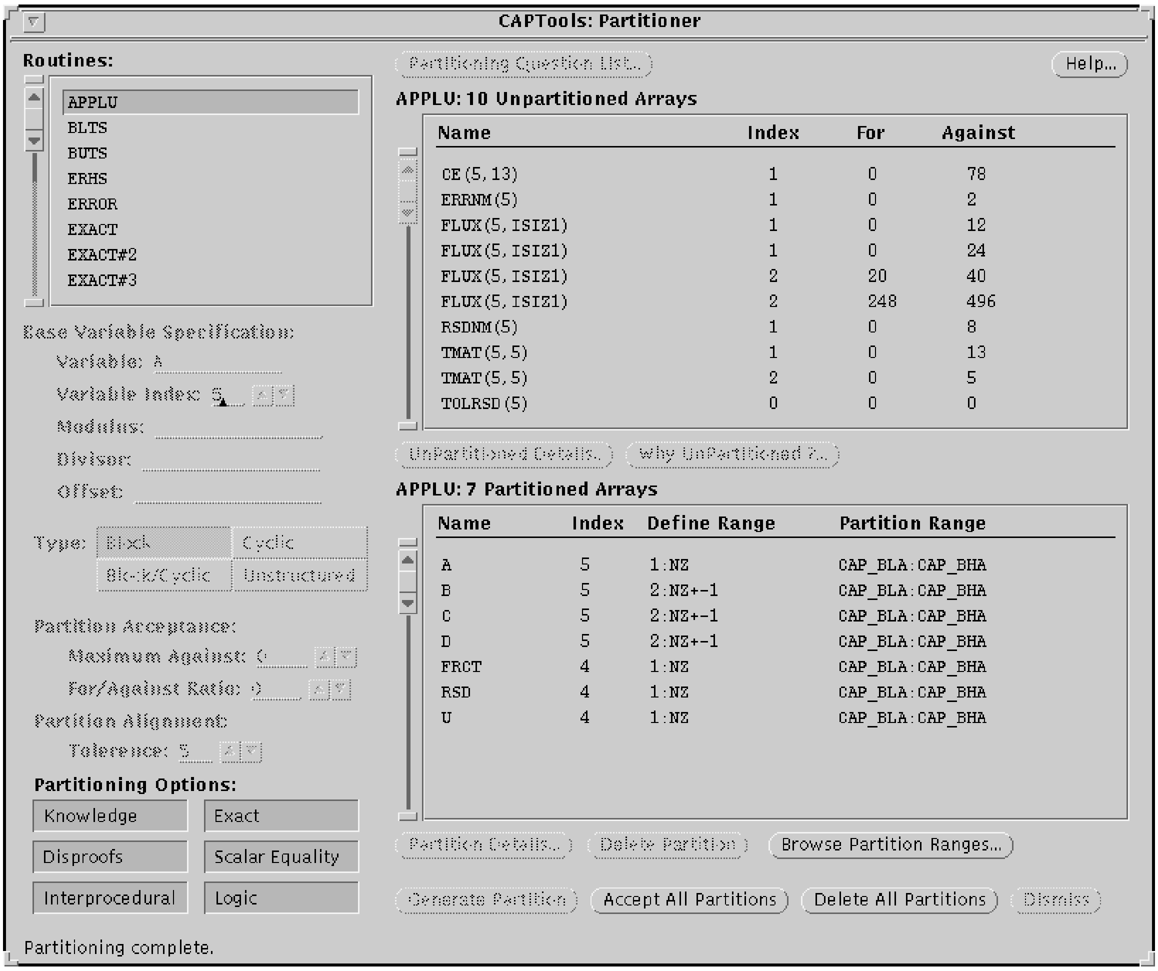


Figure 39 The Partitioner window showing the arrays partitioned in routine APPLU.

After the partition is completed the partitioned variables may be viewed in the Partitioner window (Figure 39). The main objective is to inspect the defining range of the partitioned variables to ensure that there are no anomalies. It is essential to ensure that a reasonable partition has been generated as this directly affects the next stages of the mask, communication and code generation. A definition range of 1 to NZ or 2 to NZ-1 covers all the partitioned variables. If the partition strategy is found to be acceptable then the user acknowledges this by selecting the **Accept All Partitions** button. It might also be advisable at this stage to save the database, this would allow the user to return to the partitioner stage in the case that the results obtained are not satisfactory.

12.4 Code Generation

It is now possible to proceed to the code generator stage by selecting the **Generator...** button. The minimum number of slabs should be set to 2, since it is a second order problem requiring data for calculation of 2 slabs from the processor data boundary, i.e. A(I)=A(I+2). If there was only one slab per processor then it would require communicated data from the next two processors. However, if each processor has at least 2 slabs then it only requires data communication from the next processor. The communication type may be set to **Bulk**

Communication or for **Gather/Scatter** Communications. For this tutorial the communications generated will be bulk Communications. However, there will also be further reference to gather/scatter communications. All six options are selected as for the analysis stage. The execution mask heuristic selected is the union of masks.

When the masks have been generated it is recommended to examine the masks using the Mask Browser window via the **View** menu. This will involve ensuring that the execution masks generated are efficient as this will directly affect the communications generated in the next stage. All execution control mask statements should also be examined to ensure a sensible mask has been defined and to determine why the mask has been generated for that statement. It is imperative to inspect the statements which have no execution masks generated. Since statements with no execution control masks are executed on all processors it is necessary to ensure that the statements do not need to use large amounts of data from another processor, in particular partitioned arrays, as this will necessitate additional communication. The reasoning for a generated mask is given in the **Set In Phase** textline when a statement is selected from the list an explanation for the mask will be displayed along with any related statements.

Browsing the communications in routine SSOR it can be seen that there are 4 CAP_EXCHANGE communications generated. These CAP_EXCHANGE communications all lie within loops which means that each of the CAP_EXCHANGE communications is called more than once. Considering the first CAP_EXCHANGE within this routine, the code is as follows:

```
      DO CAP_J =2,NY-1,1
        CALL CAP_EXCHANGE(U(1,2,CAP_J,CAP_BLA-1),
     +                    U(1,2,CAP_J,CAP_BHA),NX*10-20,2,CAP_LEFT)
      ENDDO
```

The communication involves exchanging 5 sets of double precision data from 2 to NX-1 (a total of NX*10-20 words) for each CAP_J from 2 toNY-1, i.e. a total of NY-2 calls to the CAP_EXCHANGE communication. Figure 40 shows each of the data blocks to be communicated from 2 toNY-1 as visualised on a Cartesian grid and also how the data blocks would be stored in memory.

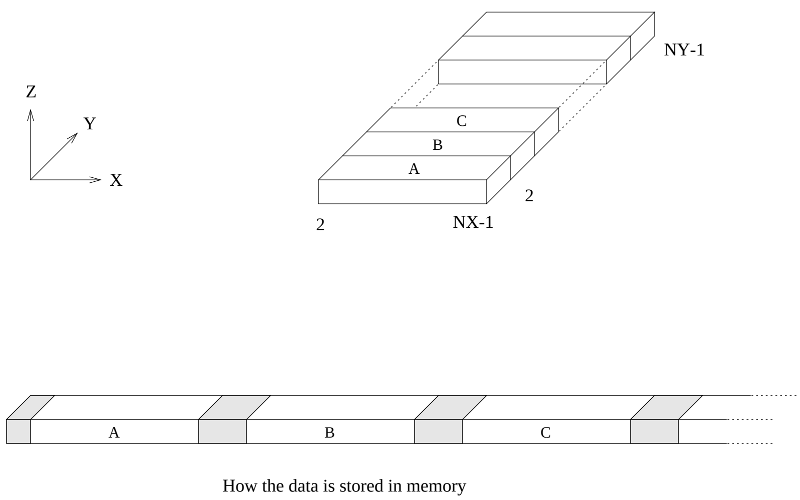


Figure 40 Data storage of arrays

These data blocks, e.g. A, B and C, are stored in memory as shown in the second half of the diagram. Looking at the memory storage it can be seen that either sides of the data blocks A, B and C there are shaded regions of the memory that are not communicated. These regions are not communicated since they are not required in the calculation that requires the communication. CAPTools has thus conservatively determined the minimum amount of data to be communicated. However, due to the discontinuous nature of the data, the data has to be communicated in more than one communication call, i.e. a loop surrounding the CAP_EXCHANGE communications. It might be advantageous to communicate all this data, incorporating the data that was not required, i.e. the shaded area, in one communication. This would thus save on the start up latency incurred for several communication calls. This however would be ideal in the case where the non-required data regions are very small, i.e. a few words between each required memory section. If the required memory sections are very far apart in memory then it could be possible that the amount of data being communicated may far exceed that required thus extending the execution time of the code. The variable NX is entered at run time. If the value of NX was known to be equal or close to the value of the allocated memory space (in this case ISIZ1) it would therefore be advantageous to convert the CAP_EXCHANGE communication to communicate the continuous memory space as follows:-

```
      CALL CAP_EXCHANGE(U(1,1,2,CAP_BLA-1),
     +                  U(1,1,2,CAP_BHA),2*(ISIZ1*5)*(ISIZ2-2),2,CAP_LEFT)
```

However, if the value of NX was not known or was known to be much less than that of the allocated memory space then this would not be advisable. It would be best to preserve the original conservative communication generated by CAPTools.

Another option that might be taken is to buffer all the data for each iteration into a buffer array and then execute a single CAP_EXCHANGE communication. This could have been possible by selecting the generation of **Gather/Scatter** communications instead of **Bulk** communications in the **Generator** window. The main disadvantage of this is that the amount of data to be buffered might be so large that the time taken to buffer the data might far outweigh the time for the start up latencies. This problem is hardware dependent on the start up latencies of the communications and on the data copy time of buffering the data.

Browsing the communications in routines BUTS and BLTS it can be seen that each routine generates two communications. The communications consist of a CAP_SEND and a CAP_RECEIVE which operate as a pipeline communication. This will involve the first processor executing its calculation before communicating the calculated data to the next processor for its own computation. The code generated for the pipeline in subroutine BLTS will look as follows:

```
      DO J=2,NY-1,1
        CALL CAP_RECEIVE(V(1,2,J,CAP_BLA-1),2*(NX*5-10),2,CAP_LEFT)
      ENDDO
      DO K=MAX(2,CAP_BLA),MIN(NZ-1,CAP_BHA),1
        DO J=2,NY-1,1
          DO I=2,NX-1,1
C           ...
C           Other calculation.
C           ...
          ENDDO
        ENDDO
      ENDDO
      DO J=2,NY-1,1
        CALL CAP_SEND(V(1,2,J,CAP_BHA),2*(NX*5-10),2,CAP_RIGHT)
      ENDDO
```

The pipelines generated involve processor 1 doing all of its calculation (while all other processors are idle) before communicating its data onto the next processor. The second processor will then receive the data and carry out its calculation (while all other processors are idle) before passing the data onto the next processor, etc. The main disadvantage is that during the whole execution time of the pipeline only one processor is actually doing any work while all the other processors are being idle. It is advantageous to ensure that each processor is employed as much as possible during these pipelines. In order to accomplish this it is necessary to either conduct a loop interchange to ensure that each processor has a balanced amount of work to perform or to simply re-partition the code selecting the fourth index of the array A as the partition base.

The user may interchange the J and K loops within CAPTools using the **Loop Interchange** transformation and reanalyse the code. Alternatively perform a manual loop interchange of loops J and K and alter the required communications in the pipelines of the subroutines BUTS and BLTS that have been generated by the code generation of CAPTools. Either way the pipeline generated in BLTS will now look as follows:

```
       DO J=2,NY-1,1
         CALL CAP_RECEIVE(V(1,2,J,CAP_BLA-1),2*(NX*5-10),2,CAP_LEFT)
         DO K=MAX(2,CAP_BLA),MIN(NZ-1,CAP_BHA),1
          DO I=2,NX-1,1
C           ...
```

```
C           Other calculation.
C           ...
          ENDDO
        ENDDO
        CALL CAP_SEND(V(1,2,J,CAP_BHA),2*(NX*5-10),2,CAP_RIGHT)
      ENDDO
```

Looking at the pipeline generated it can now be seen that the data is communicated in smaller sections of data, i.e. lines of data. This allows the second processor in the pipeline to begin processing the received data before the first processor has completed all its calculation. The second processor may then communicate its data to the next processor and perform more calculation. Therefore, all processors are busy and this avoids excessive idle time on each processor. However, there will still be some idle time on some processors due to the start up and shut down of the pipeline. This pipeline start up and shut down may be decreased by communicating points of data. This may be achieved by administering another loop interchange of the loops I and K or by partitioning on the third index of array A. The communications in routine MAXNORM consist of a total of eight sets of communications. After the calculation of the MAX norm of vector V the value of VNM and the coordinates IMAX, JMAX and KMAX are communicated to its neighbouring processors. However, prior to the calculation VNM is communicated once again. The reason for this is due to the original calculation of the communication of VNM to be a pipeline communication. There is also a requirement to also communicate the values of IMAX, JMAX and KMAX to all other processors at the same time. If there was a pipeline generated for the communication of VNM and also the communication of IMAX, JMAX and KMAX to all other processors, then a deadlock would occur. CAPTools therefore has converted the pipeline communication to be a non-pipeline communication. This problem will be remedied by the creation of a special commutative operation within the CAPTools library routines in a later version.

Browsing the communications in routine PINTGR it can be seen that a summation to calculate the surface integral is expected. This summation consists of three summations in the I, J and K direction. In the K direction (the direction in which the data is partitioned) all the required data is communicated to the last processor and the summation calculated on that processor before the summation is communicated to all the other processors. In the other two directions a part summation has to be executed on all processors (due to the partition) before a commutative call to sum all the partial summations together. These three summations are then used for the calculation of the surface integral. CAPTools has successfully calculated an efficient parallelisation strategy for this routine.

The final code may then be generated by selecting the **Final Code** button in the Code Generator window.

12.5 Execution Of The Parallel Code

Depending on which parallel environment you wish to run under follow the instructions in the section headed "USING CAPTools - Compiling and running the parallel code". Once the code is running the CAPTools library will be initialised and prompt for a processor topology or ask how many processors to run on. It is necessary for the execution of this code to ensure that the inputlu.data file is available and that the size of NX, NY and NZ in the data file match those set in the code.

13 TUTORIAL 5: Parallelisation Of A PERFECT benchmark code - ARC3D.

ARC3D is taken from the Perfect Club Benchmark suite of codes. It consists of 25 routines with approximately 3600 lines of code.

13.1 Dependence Analysis Of ARC3D

As with the previous tutorials the start point is to volunteer any information to CAPTools before commencing an initial analysis. If little or no information can be added then an intermediate level analysis should be used to quickly perform an analysis and generate a list of questions. From the Knowledge base question list, it is obvious that a large number of questions relate to the variables JMAX, KMAX and LMAX as read in routine INITIA. Browsing of the code shows that these variables are central to the code, being the loop limits for most of the loops in the code. It becomes obvious that they are, in fact, the dimensions of the problem domain and can therefore only have positive, non-trivial values. This information can be added in the Read Knowledge window (via the **Edit** menu) using the non-trivial default value i.e. JMAX.GT.5, KMAX.GT.5 and LMAX.GT.5.

A subsequent analysis produces a more accurate dependence graph with fewer dependencies set with aid of the extra information. A detailed inspection of the dependence graph leads to the routines VPENTA and VPENTA3 where recurrences are employed. Dependencies are set between code sections processing both the start and end of the recurrences which in reality cannot exist (although admittedly during our parallelisation of this code, this was only realised after code generation when these dependencies caused communications to be generated). For the algorithms to work correctly in the VPENTA routines, JMAX, KMAX and LMAX must be greater than six as can be determined using the dependence graph viewer. First select the VPENTA routine, **Array**, **True**, **Loop Independent** dependencies, then pop up the why dependence window (by selecting the **Why Dependence?** button) and selecting the offending dependence from the graph with the left hand mouse button. Selecting index 1 in the window presents the questions identified in testing for the existence of that dependence, which in this case is JU $\geq$ 8 (where JU is either JMAX-1, KMAX-1or LMAX-1 as identified in the Variable Browser window). This question is, of course, listed in the question list in the **Knowledge Base** window, however, due to the small number of occurrences of this question is was not at the top of the list and probably not addressed, or certainty of the truth of the expression could not previously be guaranteed by the user, and was therefore not answered before. The knowledge added to the knowledge base is thus JMAX.GT.7, KMAX.GT.7 and LMAX.GT.7.

Inspection of the dependence graph produced using this information appears satisfactory and therefore, the database is saved so the parallelisation can be returned to this stage if found necessary during the remainder of the parallelisation.

13.2 Transformations Prior To Partitioning

Without experience of either using the code or a previous attempted parallelisation of this code, the next stage would be partitioning (in the third dimension), followed by masking and communication generation. During these stages many indications are given that the parallelisation is not proceeding satisfactorily.

In final communication generation, a large volume of data communication is found in routine STEPF3D that calls both VPENTA and VPENTA3. The frequency of occurrence and volume of data involved in these communications indicate that parallel performance will be poor, therefore instigating detailed investigation as to why these communications have been generated. All the significant communications are found to be requested by either VPENTA or VPENTA3 where the using statements as identified in the Why Communication window have no execution control masks and must therefore execute on every processor.

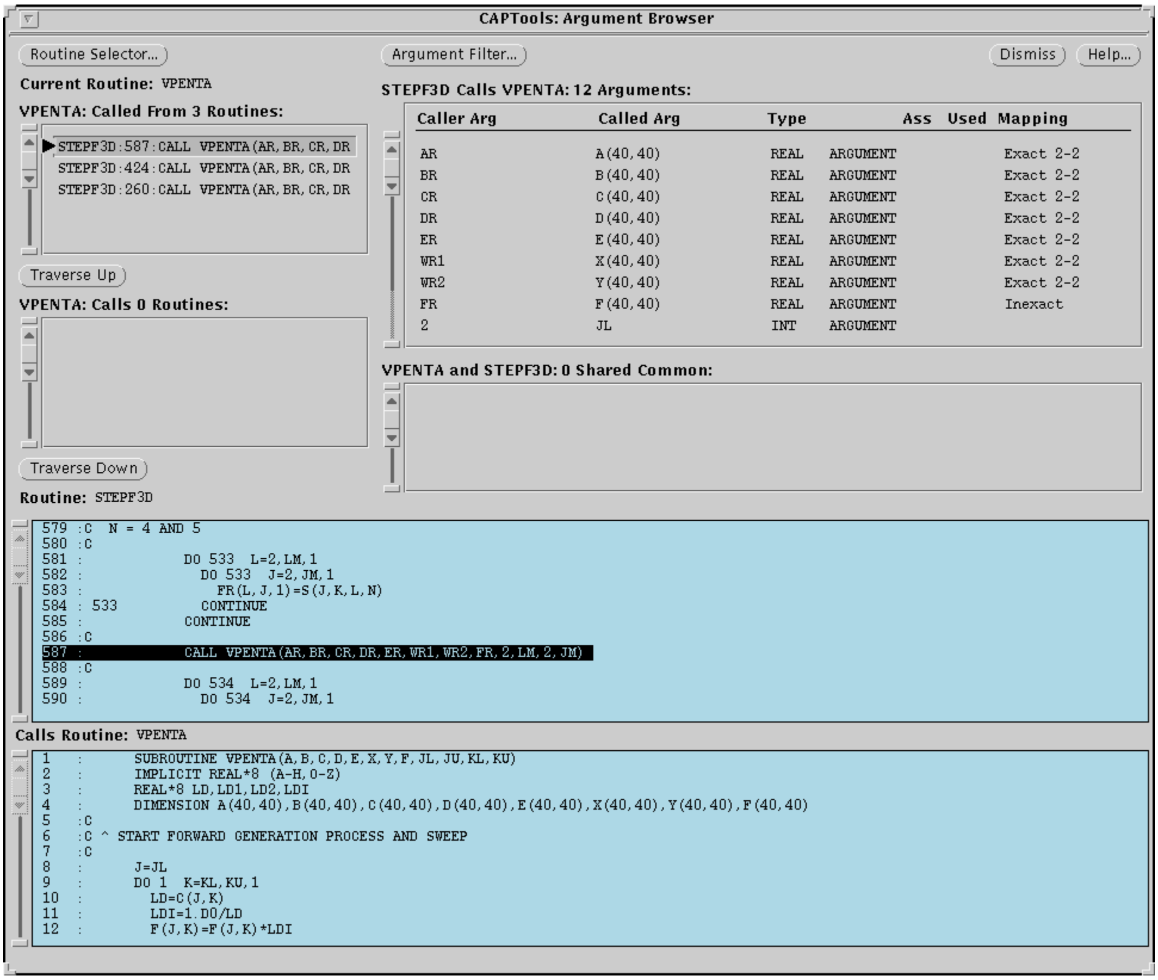


Figure 41 Using the Argument Browser to find all calling routines for routine VPENTA

This leads the investigation into the **Mask Browser** window (using the hot link from the Why Communication window using statement list setting up the **Mask Browser** ready for inspection). The statements in VPENTA and VPENTA3 are related to masked statements in STEPF3D that are masked in different ways, some using an index of the array that is passed into the VPENTA routines and some using variables not related to routine VPENTA. The **Partitioner** window is then used to find that the VPENTA routines contain no partitioned data and the arrays in them have many reasons to partition and many against, in fact most have around twice as many reasons not to partition. Inspection of the code in the Call Graph Browser and/or Argument Browser, Figure 41, (or elsewhere) reveals that each VPENTA routine is called three times in STEPF3D, each call using workspace which has been set up with different orientations of the problem data. The first call operates in J-K orientation, the second call in the K-J orientation and the final call in the L-J orientation. Since the first two calls do not involve data in the partitioned L-direction a mask on the calls in STEPF3D is sufficient. The third call processes data in the L-direction and thus for this call masks are desirable within the VPENTA routine.

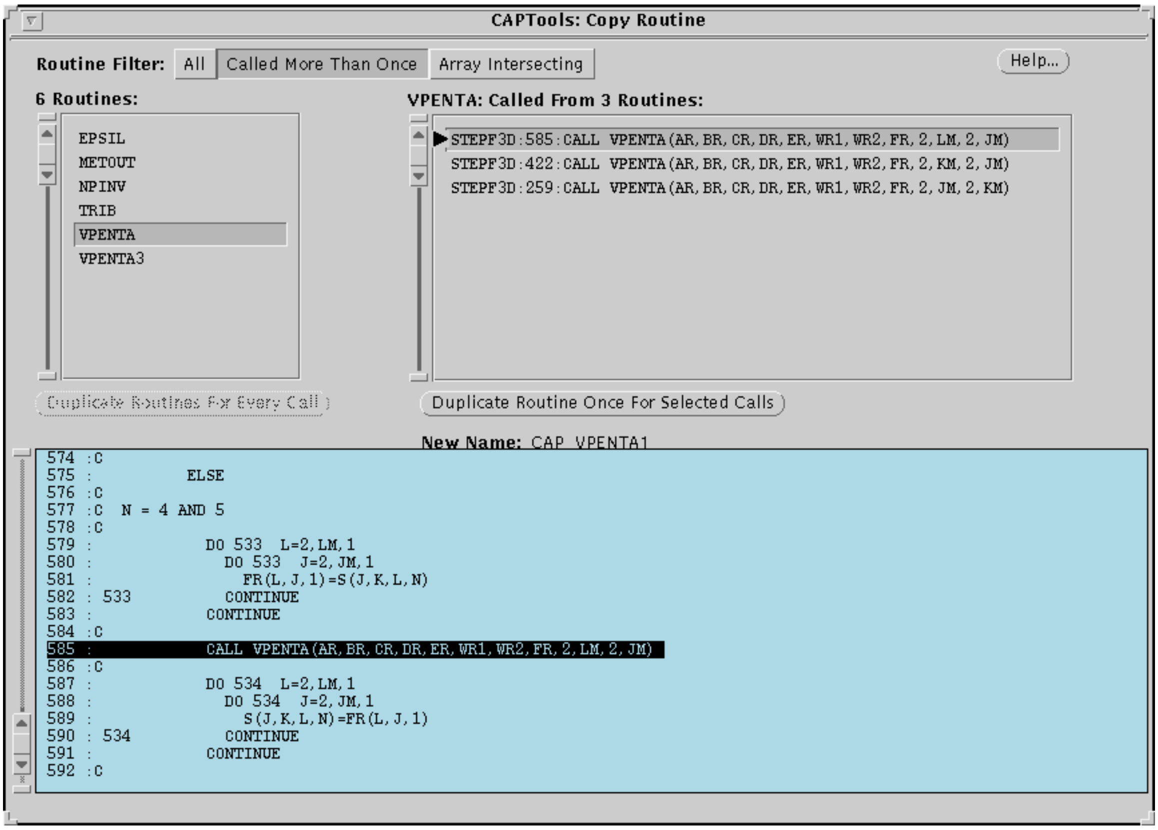


Figure 42 Using the copy routine window to replicate routines VPENTA and VPENTA3

The solution is to use the **Copy Routine** transformation window accessed from the **Transformations** button in the CAPTools **Edit** menu (Figure 42). In this window select the routine VPENTA which will set up a list of the calls to VPENTA. All three are listed and the call using data in the L-dimension is selected and the **Duplicate Routine Once for Selected Calls** button pressed which makes a copy of the VPENTA routine (by default named CAP_VPENTA1) and changing the selected calls to reference the new routine. This is repeated for VPENTA3 to create routine CAP_VPENTA31. Now the partitioning phase can be re-commenced avoiding the problems detected earlier.

13.3 Partitioning The Arrays In ARC3D

Selecting the initial partition array in what appear to be main solver routines VPENTA or VPENTA3, as would often be recommended does not lead to a satisfactory partition, with very few partitioned arrays. This is because the partition implications inherited by other arrays in routine STEPF3D would be both reasons to and not to partition those arrays, the differing implications occurring through the three different calls to each routine using data in differing orientations. The selection of an array in the copied routine CAP_VPENTA1 would be satisfactory, however an array that is more commonly used throughout the code was finally selected. The chosen array was array S in STEPF3D where index 3 (the L dimension) was chosen to provide contiguous data in communications.

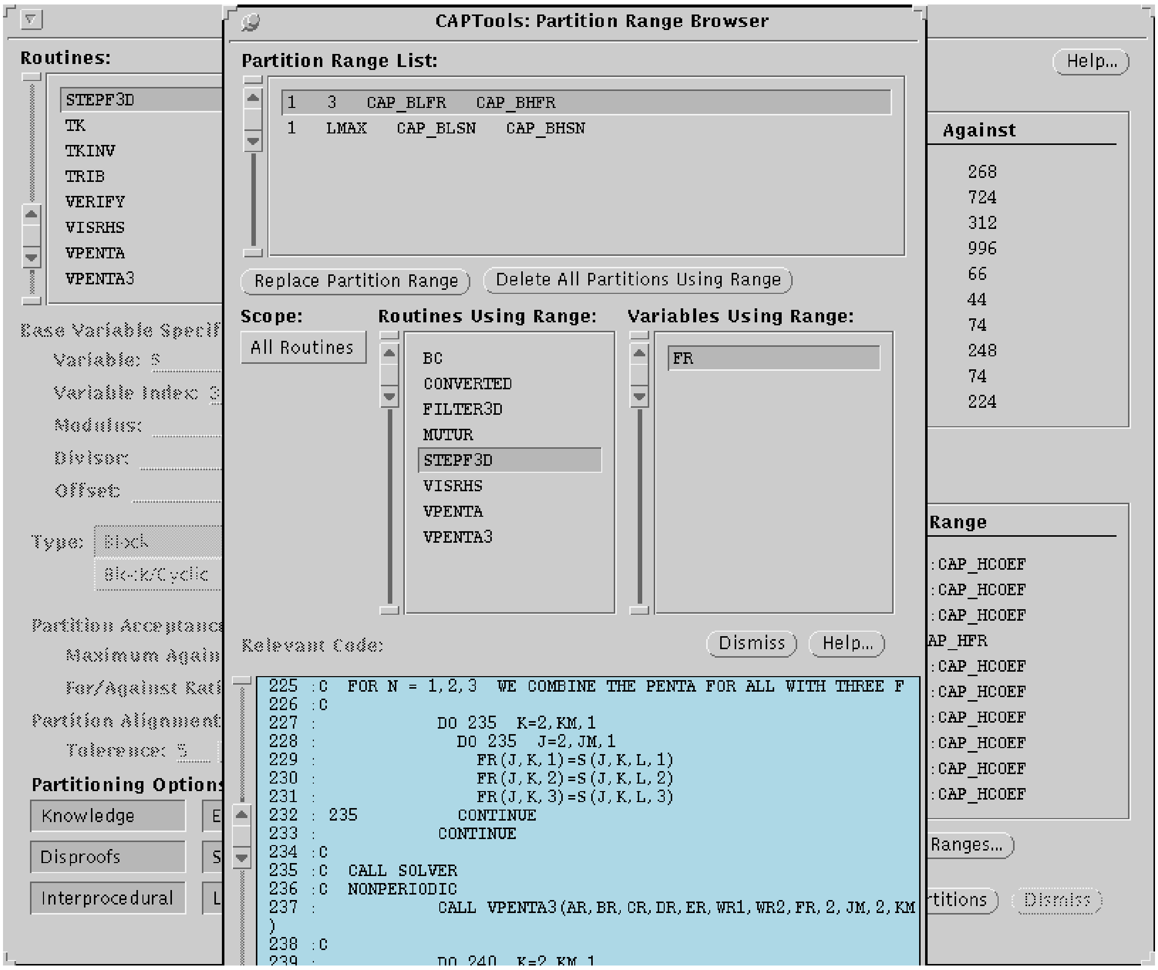


Figure 43 Partitioner window showing the different partition ranges for the ARC3D code

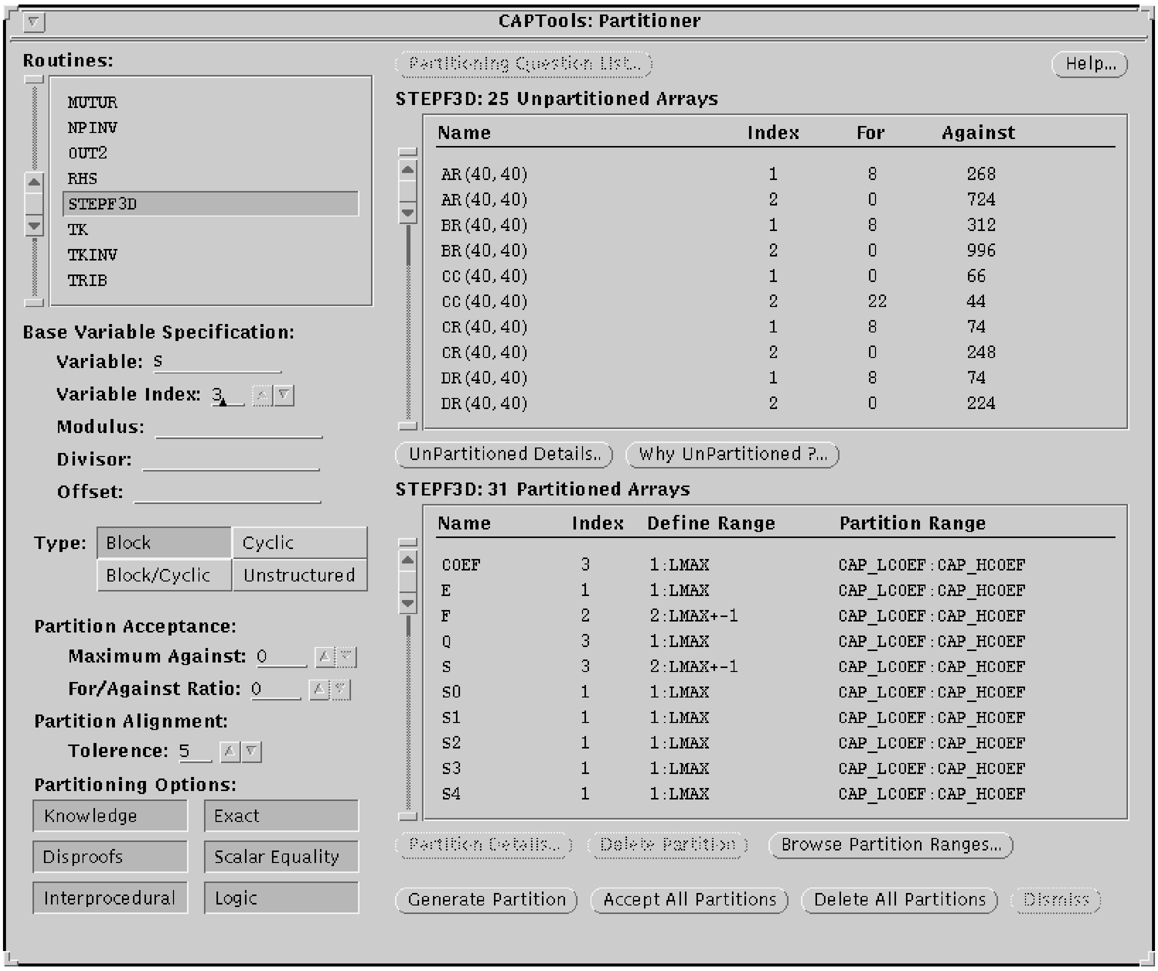


Figure 44 Partitioner window showing the final partitioned arrays in the ARC3D code

After partitioning is complete, the **Browse Partition Ranges...** window is used to identify two partition ranges used throughout the code (Figure 43). The first has an assignment range of 1:LMAX as would be expected with an L dimension partition. The second assignment range is 1:3, which would clearly not be desirable to distribute across the processor topology. This range is therefore selected and the **Delete All Partitions Using Range** button is used to delete this range and the partition definition for all arrays throughout the code based on this partition range. An inspection of the remaining partition indicates it is sensible and complete, although, obviously, the subsequent parallelisation stages will confirm this. The partition (Figure 44) is then accepted and the database again saved with the partition definition information included (although not overwriting any previous database!).

13.4 Execution Control Mask Generation

After execution control masks have been generated, the resultant masks are browsed in the Mask Browser window. Checking unmasked statements throughout the code indicate that a comprehensive set of masks have been generated. In the copied VPENTA routines, many statements have multiple masks where the statement will execute if any of those masks determine execution should occur. These multiple masks have been generated to minimise the communication requirements from those statements to the statements that use those values by assigning the required arrays potentially on several processors, wherever that data is needed. This only occurs when the extra computation is not significant, in particular when single values are calculated and would otherwise have to be communicated.

13.5 Generation Of Data Communications

Of the generated communications, four groups appear to require further investigation. Firstly, in STEPF3D, a set of communications are generated that are requested from the copied VPENTA routines. All these communications are, however, tightly masked and would therefore not be executed in most circumstances. These communications are only needed if a processor boundary (i.e. partition range variables) exists in either the first five or last five "slabs". For example if the high range variable (i.e. CAP_BHQ) on processor 1 is 3 the communication will be used between processors 1 and 2. Two of the other sets of communication that required further investigation, in CAP_VPENTA1 and CAP_VPENTA31, are tightly masked in the same way and therefore do not represent an inherent barrier to efficient parallel execution.

The final set of communications is in routine MUTUR where the communications have been generated to ensure correct code and are not, on inspection, optimal. In this routine values are calculated for every cell in the J-K plane, using information from the partitioned L dimension. The problem with the parallelisation occurs since any processor can potentially provide the value for any J-K location. The communication generation algorithm places the communications of these values within the assigning L-loop since the location of the correct values is not known outside that loop. This could be manually improved by treating the computation as a commutative operation and letting all processors calculate their local values for the J-K arrays and combine those values after the loops with careful selection of which processor's values to use as the global value.

13.6 Execution Of The Parallel Code

Depending on which parallel environment you wish to run under follow the instructions in the section headed "USING CAPTools - Compiling and running the parallel code". The code runs without the need for input data files. Once the code is running the CAPTools library will be initialised and prompt for a processor topology or ask how many processors to run on. It is necessary for the execution of this code to ensure that the sri5 data file is available.

14 TUTORIAL 6: A Two-Dimensional Parallelisation Of A NAS-PAR benchmark code - APPLU.

In Tutorial 4 the APPLU code was partitioned in the one dimension only. For this tutorial a further secondary partition is applied.

For this parallelisation of the APPLU code, in the first dimension, a slightly different partition strategy can be used. The user should select the array A index 4 as the base partition variable for the first dimension.avoiding, the loop interchange discussed in Tutorial 4. The pipeline generated in routine BLTS for the first dimension using Gather/Scatter communications is as follows :

```
        DO K=2,NZ-1,1
          CALL CAP_RECEIVE(V(1,2,CAP_BLA-1,K),NX*5-10,42,CAP_LEFT)
          DO J=MAX(2,CAP_LD),MIN(NY-1,CAP_HD),1
            DO I=2,NX-1,1
C             ...
C             Other calculation.
C             ...
            ENDDO
          ENDDO
          CALL CAP_SEND(V(1,2,CAP_BHA,K),NX*5-10,42,CAP_RIGHT)
        ENDDO
```

14.1 Partitioning The Next Dimension.

To partition the next dimension requires the re-initialisation of the data structures within CAPTools. This is achieved by selecting the **Partition Next Dimension** button in the Code Generator Window. When this process is completed the Partitioner window will appear automatically. The user may now select the variable to be partitioned for the next dimension. In the previous partition the variable array A index 4 was chosen. For this dimension the next innermost index is chosen, i.e. array A index 3. The masks and communications may then be generated.

14.2 The Generated Code.

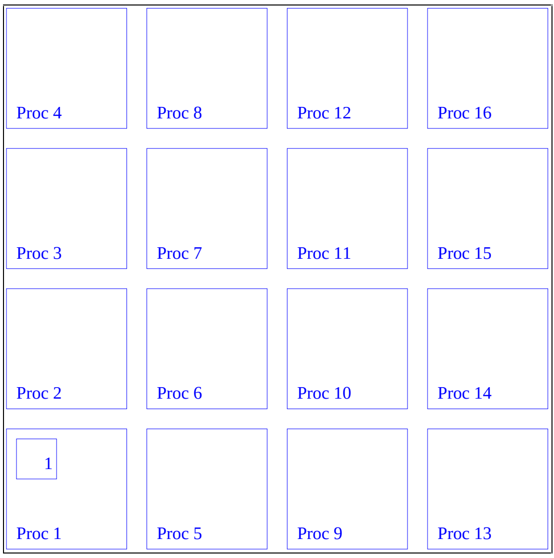
The Gather/Scatter communications generated in the routine BLTS are as follows :

```
        DO K=2,NZ-1,1
          CALL CAP_RECEIVE(V(1,CAP2_LD,CAP_BLA-1,K),-CAP2_LD*5+CAP2_HD*5+5,42,CAP_LEFT)
          DO J=MAX(2,CAP_LD),MIN(NY-1,CAP_HD),1
            CALL CAP_RECEIVE(V(1,CAP2_BLA-1,J,K),5,42,CAP_UP2)
            DO I=MAX(2,CAP2_LD),MIN(NX-1,CAP2_HD),1
C             ...
C             Other calculation.
C             ...
            ENDDO
            CALL CAP_SEND(V(1,CAP2_BHA,J,K),5,42,CAP_DOWN2)
          ENDDO
          CALL CAP_SEND(V(1,CAP2_LD,CAP_BHA,K),-CAP2_LD*5+CAP2_HD*5+5,42,CAP_RIGHT)
        ENDDO
```

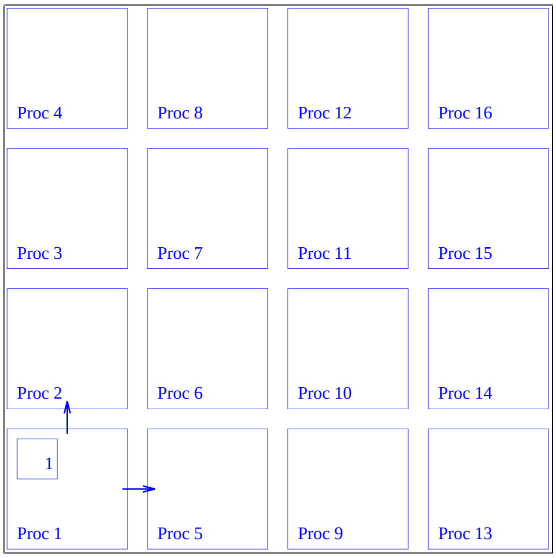
The pipeline communications for the second dimension partition (i.e. CAP_UP2 and CAP_DOWN2) are within the K and J loops. An improved performance may be obtained from the pipeline by moving these communications outside of the J loop to provide the following communications :

```
        DO K = 2,NZ-1
         CALL CAP_RECEIVE(V(CAP2_LD,CAP1_LD-1,K),CAP2_HD-CAP2_LD+1,3,CAP_LEFT)
         CALL CAP_BRECEIVE(V(CAP2_LD-1,CAP1_LD,K),1,LDMX,CAP1_HD-CAP1_LD+1,3,CAP_DOWN)
         DO J = MAX(2,CAP1_LD),MIN(NY-1,CAP1_HD)
               DO I = MAX(2,CAP2_LD),MIN(NX-1,CAP2_HD)
```

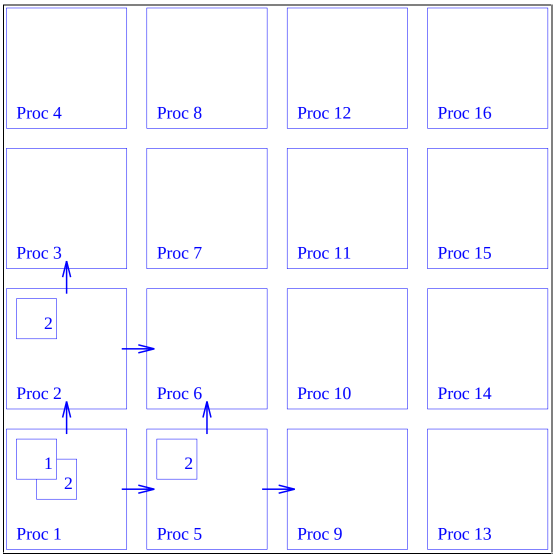
```
C               ...
C               Other calculation.
C               ...
             ENDDO
          ENDDO
          CALL CAP_SEND(V(CAP2_LD,CAP1_HD,K),CAP2_HD-CAP2_HD+1,3,CAP_RIGHT)
          CALL CAP_BSEND(V(CAP2_HD,CAP1_LD,K),1,LDMX,CAP1_HD-CAP1_LD+1,3,CAP_UP)
         ENDDO
```



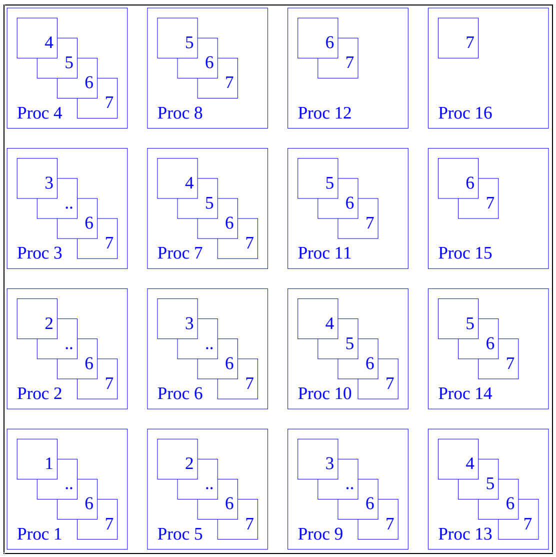
Start of a 2-D Pipeline in the APPLU code.



First communication of a 2-D Pipeline in the APPLU code.



Calculation and communication for the second stage of a 2-D pipeline in APPLU



Calculation for the seventh stage of a 2-D pipeline in APPLU.

Figure 45 Pipeline start-up for a two-dimensional partition for APPLU

This two-dimensional pipeline operates as follows. The first processor (Proc 1 in Figure 45a) will begin the pipeline on the first stage by performing the calculation for when **K** equals 2 (i.e. calculates **V(2,2,2)** to **V(CAP2_HD,CAP1_HD,2)**). All the other processors are idle during this first stage. When it has completed its calculation it may then communicate (Figure 45b) the required data to the processor to its immediate right (Proc 5) and immediately above (Proc 2). All three of these processors may then proceed with the calculation for the second stage (Figure 45c). Processor 1 will calculate for **K** equals 3 while Processors 2 and 5 will calculate for their first iteration i.e. when **K** equals 2. After this calculation, all three processors will send the required data to the processors to their immediate right and above (Figure 45c). This allows processors 1, 2, 5, 3, 6 and 9 to compute during the third stage. By the seventh stage (Figure 45d) all the processors in the grid are able to calculate (i.e. none of the processors are idle). For a 1-D pipe of 16 processors, the pipeline would not be fully operational until the sixteenth stage.

Using a performance evaluation tool such as the Automated Instrumentation and Monitoring System (AIMS) from NASA Ames shows quite clearly the effect of a secondary partition on the pipeline. Figure 46 shows the pipeline for a 1-D partition while Figure 47 shows the pipeline for a 2-D partition. The lines representing the communications have been omitted for clarity. The 12 horizontal bars represent the 12 processors used in these executions. The differing shades of the bars represent the execution of a particular routine in the APPLU code. The gaps between these bars represent the idle time whilst awaiting the completion of a synchronous communication. The AIMS plots for both the 1-D and 2-D parallel versions show 1000 milli-seconds so that the size of the gaps is comparable between the figures. Figure 46 shows clearly a greater amount of idle time at the start-up and shutdown of the pipeline for the 1-D parallel code in comparison with 2-D shown in Figure 47. Figure 47 shows the operation of the 2-D pipeline as described in Figure 45a-d where a 4x3 grid of processors was used, enabling all processors to commence calculation at a far earlier stage than for the 1-D case.

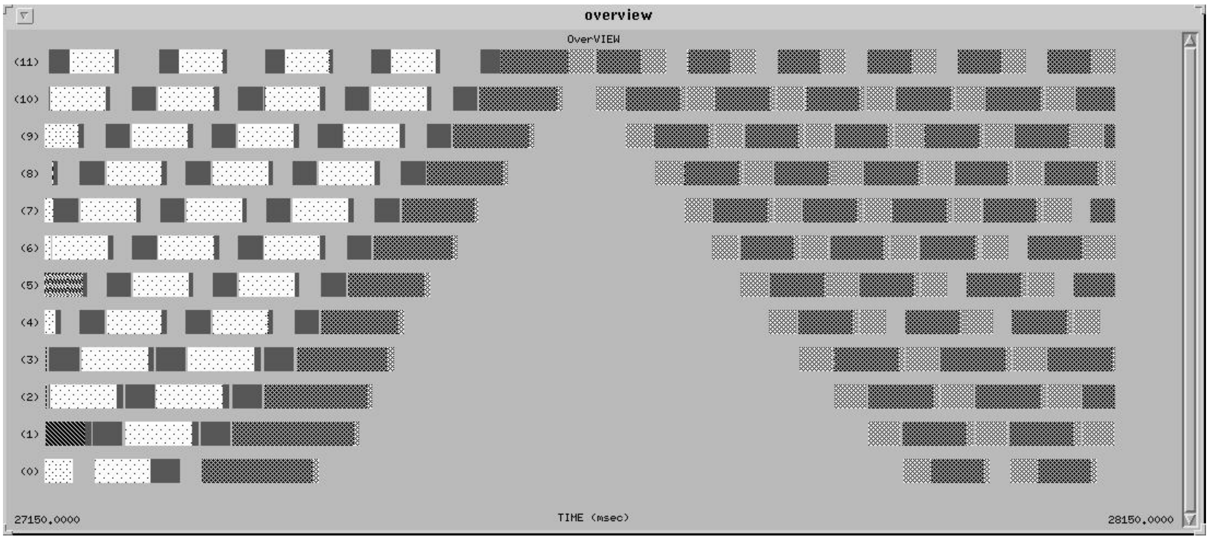


Figure 46 AIMS plot for 1-D partition.

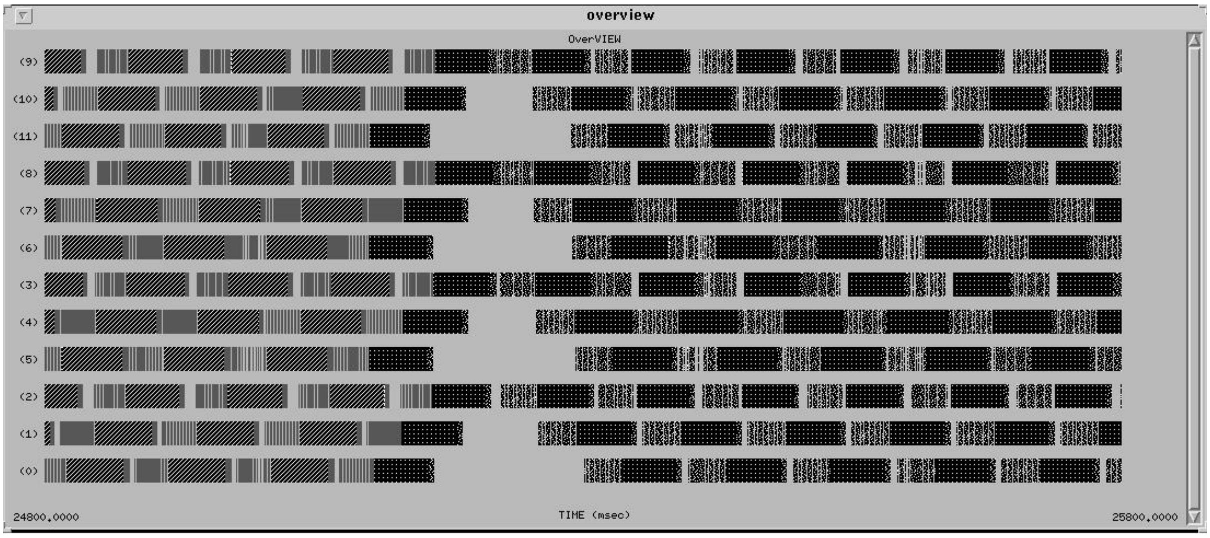


Figure 47 AIMS plot for 2-D partition.

The use of a grid of processors has therefore reduced the startup and shutdown times of the pipeline and thus improved the performance of the code. This leads to a major improvement in performance from the one-dimensional partitioning speed up of 43.7 to the two-dimensional partitioning speed up of 53.8 for 64 processors on the Cray T3D. Moreover, the 2-D partition results indicate a good level of scalability.

14.3 Execution Of The Parallel Code

To compile the parallel code use the capmake command as already described in previous tutorials.

To execute the parallel code over a 2-D process topology the caprun command is

```
caprun –top grid16x8 applu_2d
```

15 TUTORIAL 7 Parallelisation Of A Simple Unstructured Mesh Code FEMESH.

To demonstrate the parallelisation of unstructured mesh codes, the simple code FEMESH is used. The stages within CAPTools are identical to those followed for structured mesh code parallelisations.

The FEMESH code has element (cell) based variables (**ETEMP,XCEN,YCEN**) and node (grid point) based variables (**NTEMP,X,Y,NDISP**). The element to node interconnections are stored in array **MESH**. One of the major challenges in developing generic algorithms for the parallelisation of unstructured mesh codes is the limitless variety of storage techniques for these mesh interconnections. The topology representation in the **FEMESH** example uses a one dimensional list of node numbers stored in the order of the elements to which they are connected. The last node of each element is negated to indicate the end of that element's data.

After loading the source code into CAPTools, some READ KNOWLEDGE can be entered. The variables **NELEM** (number of elements) and **NNODES** (number of nodes) are constrained to be POSITIVE NON-TRIVIAL. Full power dependence analysis can then be performed.

15.1 Partitioning and Masking

As with structured mesh code parallelisations, the next stage is the calculation of the data partition using the PARTITION BROWSER. Selection of routine **FEMESH**, array **ETEMP**, index **1** along with the selection of the UNSTRUCTURED partition type, followed by pressing the GENERATE PARTITION button constructs the partition of the element based arrays (**ETEMP,XCEN** and **YCEN**). A second selection is then required for the node based arrays, routine **FEMESH**, array **NTEMP**, index **1**, again ensuring that the UNSTRUCTURED partition type is selected. Questions are asked, where the maximum of **ABS(MESH(J))** and **NNODES** is **NNODES** (since array **MESH** stores node numbers) and similarly **1** is the minimum of **1** and **ABS(MESH(J))**. This then constructs the nodal partition of arrays **NTEMP**, **NDISP**, **X** and **Y**. As always, the PARTITION RANGE BROWSER should then be checked. In this case there should only be two partition ranges listed, one for the element based arrays and one for the node based arrays.

The next stage is the addition of execution control masks, following the structured mesh process. The masks take the form

```
IF (CAP_P(I).EQ.CAP_PROCNUM) . . .
```

where CAP_P is a partition array, in this case representing either the element or nodal partition where **CAP_P(I)** stores an integer indicating the processor that owns entity **I**.

15.2 Communications and Inspector Loops

Communication calculation and generation can then be completed. The resultant code will contain two inspector loops and calls to a number of **CAPLib** library routines. The inspector loops are used to interpret the data structure used in the code by mimicking an access to that structure. Each inspector loop is initialised by a call to **CAP_CONNECT_INIT** and each connection (i.e. usage of a partitioned entity's data in relation to the execution control mask that determines where that usage will occur in the processor topology) is indicated by a call to **CAP_CONNECT**. Following an inspector loop, calls to one of three **CAPLib** library routines are made. **CAP_JOSTLE** is used to invoke the **JOSTLE** graph partitioner, taking in the connections set up in the inspector loop and returning the partition details for one entity type. Typically, a single call to **CAP_JOSTLE** is required in an application code. **CAP_SUBSIDIARY** is used to create a partition for an entity type based on its relationship with a previously partitioned entity type. It uses the connections from the inspector loop to determine these relationships, along with the partition array of the other entity type to return the partitioned array for the new entity type. Typically, a small number (usually, 0,1 or 2) of calls to **CAP_SUBSIDIARY** are generated, depending on the number of entity types in the application code. **CAP_OVERLAP** is used to construct a communication set that can then be used to update overlap (halo) data during the application code execution. The inputs are the inspector loop connections and the partition arrays related to those connections. An identification number (**CAP_ID1)** is returned where that variable is used to refer to the communication set later in the application code execution. These communication sets are used for communication using the **CAPLib** routine **CAP_SWAPOVER** which takes in the array to be communicated and the communication set identification variable and performs an overlap data update. Two other types of communication are generated for unstructured mesh codes, **CAP_COMMUTATIVE**, as used in structured mesh parallelisations, and **CAP_BROADCAST**. **CAP_BROADCAST** is used for communications that cannot be effectively performed as an overlap update. They can represent a significant overhead in parallel execution and should therefore be investigated in the **COMMUNICATIONS BROWSER** as to why they were required. The inputs to this routine are the array to be communicated and a boolean indicating if the calling processor owns the data item(s) to be communicated.

In the **FEMESH** code, due to the communication migration and the merger of both inspector loops and communications, only six communications are generated that use only two inspector loops. One inspector loop is used to construct the communication set for the overlap nodes used in an element based calculation .The other inspector loop constructs the set of overlap elements used in a node based calculation. Selection of a **CAP_SWAPOVER** call in the **COMMUNICATION BROWSER** highlights the communication in the text window and lists the set of statements that requested (and therefore use) the communicated data. The **WHY COMMUNICATION?** window then displays more information about the communication, including the calls to **CAP_OVERLAP** that immediately follow the inspector loops used in the selected communication.

15.3 Execution Of The Parallel Code

To compile the generated code for an unstructured mesh based parallelisation, the **CAPMAKE** utility can be used where the **-unstruct** option must be included to link in the unstructured mesh components of the **CAPLib**.

```
capmake -unstruct femesh_par
```

To execute the parallel code, the **CAPRUN** utility can be used where the processor topology must be specified as **FULL** (e.g. **full64**) so that communication to all other processors is enabled for each processor to satisfy the requirements of the communication sets.

```
caprun -top full32 femesh_par < femesh.in
```

Although a fully connected processor topology must be specified, the **JOSTLE** graph partitioner will typically "localise" the communications in the neighbourhood of each communicating processor. This can avoid communication saturation and reduces the risk of communication "hot spots" occurring that can significantly slow the overall rate of communication.

CAPTools Reference

16 CAPTools User Interface Reference

16.1 Terms Used

[Button]	• Buttons are used to activate some kind of function such as popping up a window or updating a list. Buttons which have labels ending with 3 dots will pop up a window.
[Menu Button]	• Menu buttons (denoted by a inverted triangle) display a menu when clicked on with the right hand button of the mouse. The default item in the menu can be selected by clicking with the left hand button.
[Menu]	• Menus offer a choice of options, select a menu item with the left hand button of the mouse.
[Menu item]	• Individual entry in a menu.
[List]	• A scrolling list of information.
[Setting]	• Offer a choice of several items. Click on an item to select it. Settings can be exclusive (only one choice at a time), non-exclusive (multiple choice) or a combination (selection of a choice may deselect others).
[Textfield]	• Textual or numerical information. If the textfield is underlined this denotes that text may be typed into the textfield. Enter return to accept the text into the field.
[Text pane]	• Scrolling window containing textual information. Most of the textpanes in CAPTools display source code.
[Canvas pane]	• Scrolling window containing graphical and textual information.

<u>**Bold & underlined**</u> is used for names of CAPTools windows.

Bold is used to refer to the names of functions within a CAPTools window.

Words in *italics* refer to functions mentioned elsewhere in the manual.

16.2 Generic User Interface Functions

All or many of the CAPTool windows will contain the following panel items:-

Apply [button]

Applies the settings in the current window.

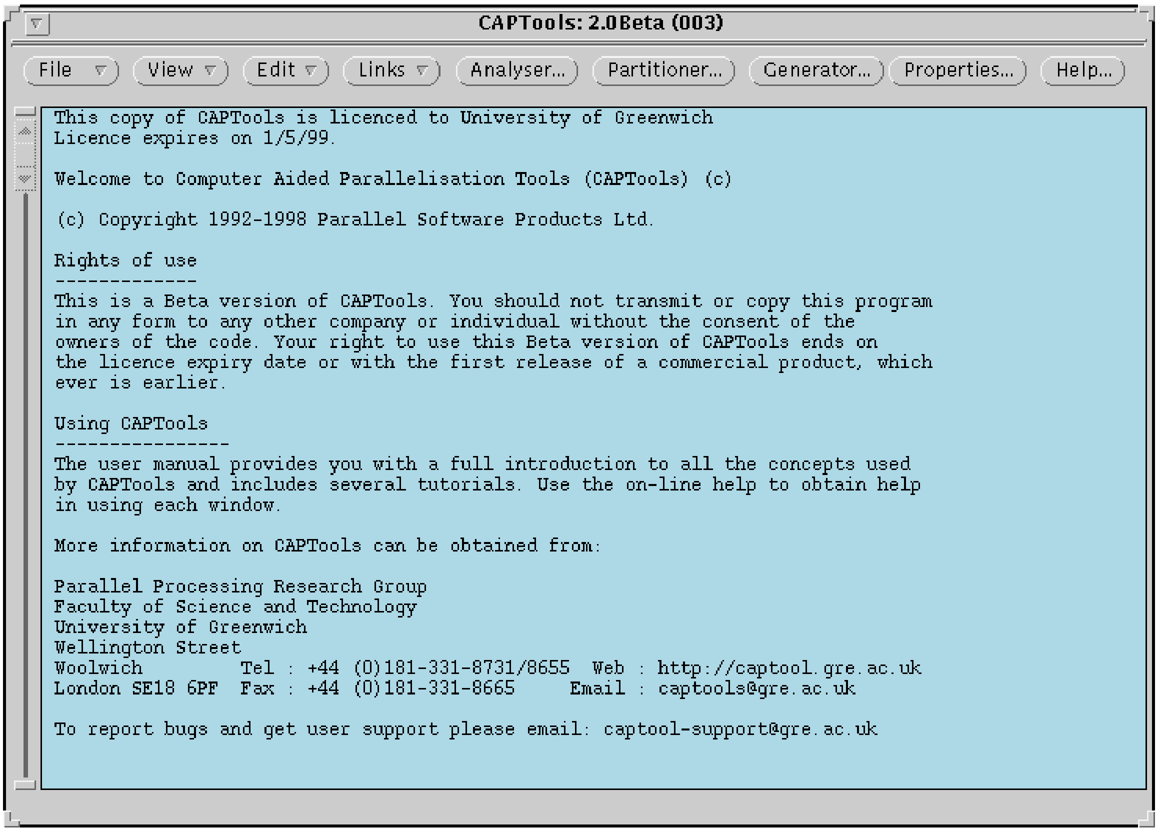
Dismiss [button]

Closes the current window.

Help... [button]

Displays help for the current window.

16.3 CAPTools Base Window



This window provides access to all the other windows of the CAPTools user interface. Access to the main functions such as analysis, partitioning and code generation is by clicking the relevant button. Other browsing and editing windows can be accessed through the Edit and View menus.

Analyser... [button]

Displays the Analyser window. This window allows the user to begin analysis of loaded source code or to do an incremental analysis.

Partitioner... [button]

Displays the Partitioner window. This window is used to partition selected arrays across a processor topology. It is the first stage in the parallelisation process in producing an SPMD parallel code.

Generator... [button]

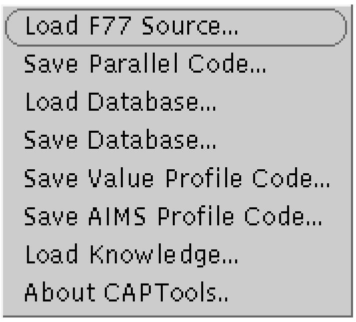
Displays the Generator window. This window is used to generate execution control masks and communication statements once a partitioning has been done.

Help [button]

Displays the online Help window. the help displayed is the same text as in the user manual reference section.

16.3.1 Base Window File Menu

The File menu which provides file i/o functions. The menu options are:-



Load F77 Source...

Load FORTRAN77 code into CAPTools, either from a single file containing the entire source code for parallelisation (.f or .F), or indirectly from a file containing a list of FORTRAN77 source file names (.list). The user can enter a file name either in the Name textfield or by selecting a file name from the list and applying the Load button.

Save Parallel Code...

Save FORTRAN 77 code from within CAPTools. Enter a file name either in the Name textfield or by selecting a file name from the list and applying the Save button.

Load Database...

Load a database file (.dbs) from a previous CAPTools session. The loading of a database returns you to one of the following stages:-   - Dependence analysis   - Partitioning   - Execution Control Masking   providing that the database has first been saved using the Save Database option at the required point in the previous CAPTools session. Enter a file name either at the textline Name or by selecting a file name from the list and applying the Load button. Currently you need to restart CAPTools to load a database.

Save Database...

Save a database file (.dbs) of the current CAPTools session. The volume of information recorded depends on the state of the parallelisation. It is possible to save the database of current work after any of the following stages:-   - Dependence analysis   - Partitioning   - Execution control masking   Enter a file name either in the textfield Name or by selecting a file name from the list and applying the Save button.

Save Value Profile Code…

Save code augmented with calls to store variable values and loop iteration counts that create a database to be read by CAPTools.

Save AIMS Profile Code…

Save Automated Instrumentation Monitoring System (AIMS) profile code that will output a trace file for reading by the AIMS VK tool. .

Load Knowledge...

Load a user knowledge file (.ine) containing information relating to the serial code. The information is represented as FORTRAN inequalities with a 1 preceding routine names and 2 preceding inequalities. The user can enter a file name either at the textline Name or by selecting a file name from the list and applying the Load button.

About CAPTools

Displays the release version number and contact information for user support.

16.3.2 Base Window View Menu

The view menu which provides access to some of the browsing facilities of CAPTools. The menu options are:-

Call Graph...

Displays the CallGraph Browser window. Allows viewing and interrogation the call graph representing the serial code.

Dep Graph...

Displays the Dependence Graph Browser window. Allows viewing and interrogation the dependence graph of a selected routine.

Loops...

Displays the Loops Browser window. Allows viewing and interrogation of information about all loops in the source code including iterative loops.

Args/Commons...

Displays the Argument Browser window. Allows viewing and interrogation of information about the arguments and common blocks present in the source code.

Var Definition...

Displays the Variable Definition Browser window. Allows viewing and interrogation of information about variables in the source code.

Global Vars...

Displays the Global Variable Browser window. Allows viewing and interrogation of information about all the arguments and common blocks for a particular routine or all routines in the code.

Masks...

Displays the Masks Browser window. Allows viewing and modification of the execution control masks generated as part of code generation. This option is only available once masks have been generated from the generator window.

Comms...

Displays the Communications Browser window. Allows viewing and modification of the communications generated as part of code generation. This option is only available once communications have been generated from the generator window.

16.3.3 Main Window Edit Menu

This button displays the file menu which provides access to some of the editing facilities of CAPTools. The menu contains the following options:-

Analyser…

Displays the Analyser window. This is used to define the level of dependence analysis to be performed and then to perform that analysis.

Transformations

This displays the Transformations menu window (see section 16.3.4).

Partitioner...

Displays the Partition window. This window is used to partition selected arrays across a processor topology. It is the first stage in the parallelisation process in producing an SPMD parallel code.

Code Generator…

Displays the Code Generator window. This allows the user to generate parallel code. Firstly through execution control masking, followed by communications generation and finally to parallel code output.

Knowledge Base... [button]

Displays the Requested knowledge window. This window allows the interrogation of the questions CAPTools has generated during dependence analysis. Answering these questions can help CAPTools to eliminate uncertain dependencies.

READ Knowledge...

Displays the Read Knowledge window. This window is used to enter information about variables that are assigned from READ statements prior to dependence analysis. CAPTools cannot calculate and use the range of such variables without this knowledge so this procedure is often vital in achieving a successful parallelisation.

User Knowledge...

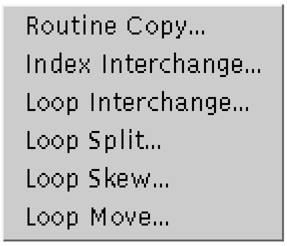
Displays the User Knowledge window. This window provides another way of entering information into CAPTools about variables.

Value Profiler... [button]

Displays the Serial Profiler window. Not available in the release.

16.3.4 Base Window Transformations Menu Button

This button displays the Transformations menu that provides access to some of the transformation facilities of CAPTools. The menu options are:-



Routine Copy...

This pops up the Copy routine window. This creates a copy or copies of a selected routine and renames them with a prefix CAP_ and a suffix number starting from 1.

Index Interchange…

This pops up the Index Interchange window. This swaps indices for selected arrays throughout the code. For example U_VEL(10,300,20) -> U_VEL(10,20,300).

Loop Interchange…

This swaps an inner loop with an outer loop in a nest.

Loop Split…

This splits a loop into two loops.

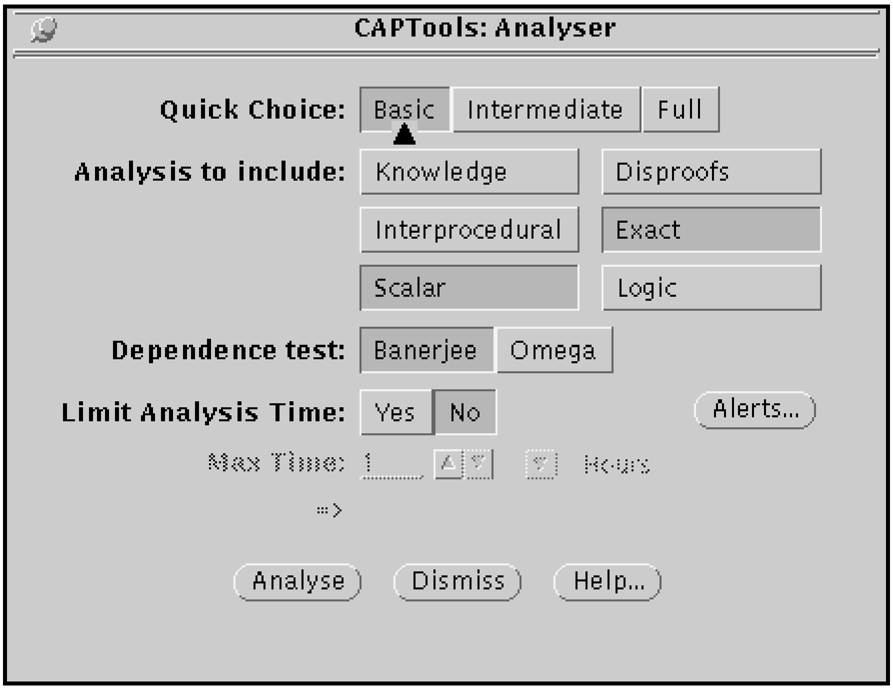
Loop Skew…

This skews the iteration space of a loop nest.

Loop Move…

This moves a loop from one routine into another.

16.4 Analyser Window



Quick Choice [setting]

The level of dependence analysis to be performed on the serial code can be defined simply as:- - Basic - Intermediate - Full By selecting one of these three levels the Analysis to includesetting alters to reflect what the selected level will cover.

Analysis to include [setting]

The user may wish to select their own individual combination of options as part of the analysis. The user can choose from Knowledge, Disproofs, Interprocedural, Exact, Scalar and Logic.

Dependence test

The user may choose which dependence test to perform , either Banerjee. Or Omega.

Alerts…

This pops up the Analyser Alert Browser window after a time limited analysis.

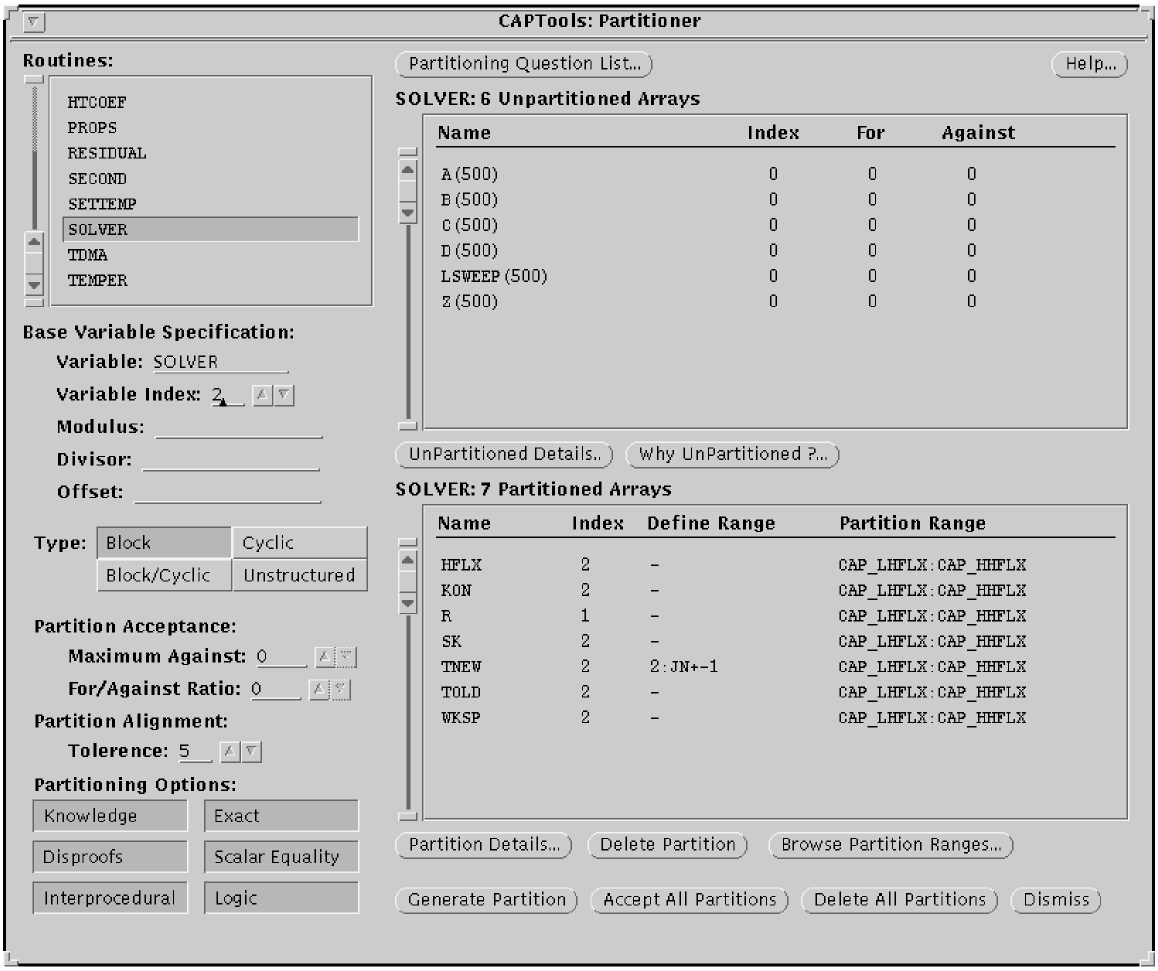
Limit Analysis Time

This limits the maximum time allowable for analysis based in hours or minutes.

Analyse [button]

This will carry out an analysis of the code to the level described by either the Quick choice or Analysis to include sections.

16.5 Partitioner Window



This window is used to partition a number of arrays in the serial code to produce a partition definition for the following parallel code generation.

Routines [list]

This allows the user to choose the routine in which to set the partition variable. When a routine is chosen the Unpartitioned Variables and Partitioned Variables windows are highlighted.

Unpartitioned Arrays [list]

This window allows the user to choose which unpartitioned variable upon which to base the partitioning strategy. The variable chosen for partitioning will then be highlighted in the Base Variable Specification. This will also highlight the Partition Acceptance, Partition Alignment and Partition Options panes as well as the Generate Partition button.

Base Variable Specification textfield

This shows the user the variable they have chosen to base the partitioning strategy upon. The user must then either define the index or dimension of the variable array using variable index or define a subset of the dimension defined by Modulus, Divisor and Offset expressions.

Type [setting]

This allows the user to choose the partition type : Block, Cyclic, Block/Cyclic or Unstructured. The default is set to Block.

Partition Acceptance textfield

This allows the user to specify the criteria for accepting or rejecting partitioned data. The Partitioner may accept a partition if :-

The number of instances of not accepting a partition is less than that specified in the Maximum Against definition, or

The ratio of the number of instances for a partition compared to those instances which do not recommend the partition is greater than that specified in the For/Against Ratio definition.

The default values for both of these are 0.

Partition Alignment textfield

This allows any partitioned variable array to share the same partition range as other arrays with a small difference in their assignment range. The default setting for the Partition Alignment Tolerance is 5. This may be changed by the user.

Partition Options [button]

There are six different options available : Knowledge, Exact, Disproof, Scalar Equality, Interprocedural and Logic. The more of these options selected the more powerful the partitioning generated.

Generate Partition [button]

This will generate a partition when the Base Variable Specification, Partition Acceptance, Partition Alignment and/or Partition Options settings have all been set.

Unpartitioned Details…

This will pop up the Unpartitioned Variable Information window of the variable highlighted in the Unpartitioned Arrays window.

Why Unpartitioned [button]

This will pop up the Why Unpartitioned window. The user must first highlight an Unpartitioned Array before the use of this option.

Partitioned Arrays [list]

This will show (for each selected routine) the partitioned arrays. In choosing one of the arrays in this window allows the user to browse Partition Details, Delete Partition of that partitioned array.

Partition Details... [button]

This will pop up the Partitioned Variable Information window of the variable highlighted in the Partitioned Arrays window.

Delete Partition [button]

This will delete the partition set on the highlighted array in the Partitioned Arrays List.

Browse Partition Ranges... [button]

This will pop up the Partition Range Browser window to allow the user to view the different partition ranges.

Partitioning Question List…[list]

This re-displays the assignment range question window.

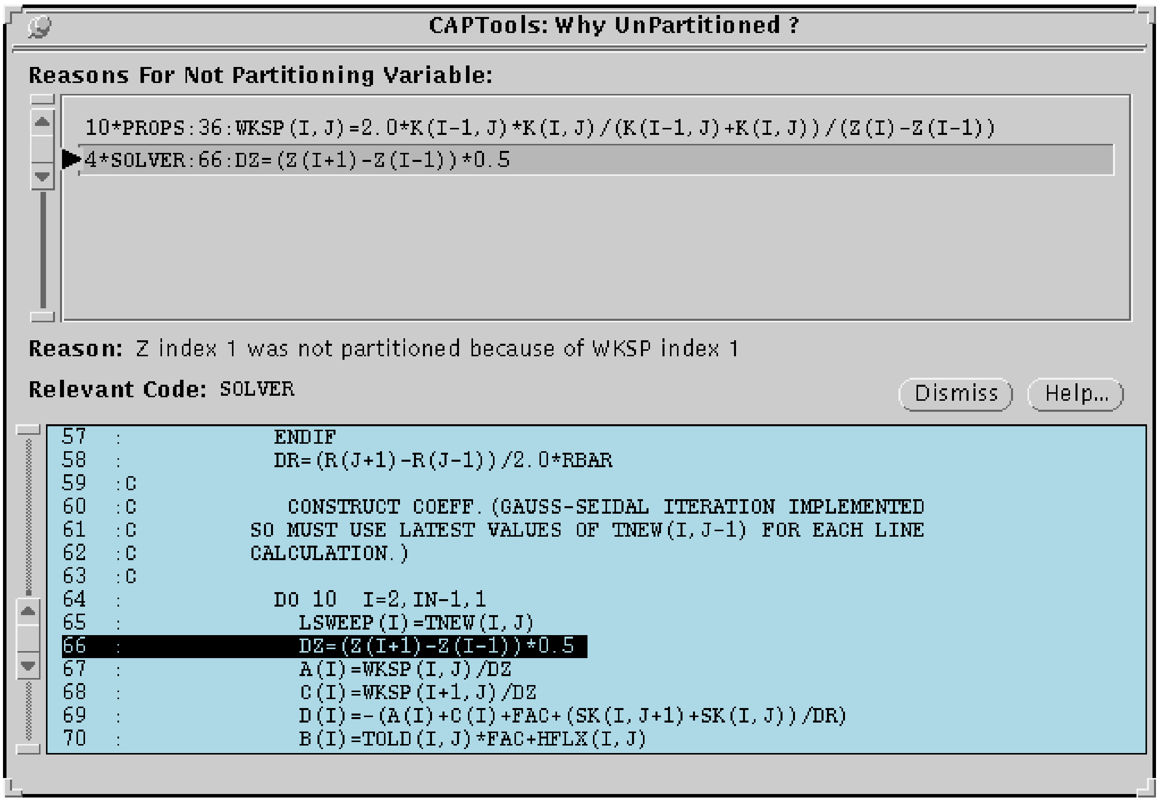
Delete All Partitions [button]

This will delete all previous partitions.

Accept All Partitions [button]

This button will accept the partition generated and any other modifications carried out in the Partitioner window. Once all partitions are accepted no other changes are allowed, unless the database is saved before accepting the partition and then reloaded.

16.5.1 Why Unpartitioned Window



This window shows the reasons why a particular variable has not been partitioned. The reasons are shown in a list. Click on a list entry to see a textual description of the reason. The textpane shows the relevant text.

Reasons For Not Partitioning Variable [list]

A list of statements why the array highlighted in the Unpartitioned Variables window was not partitioned.

(Scrolling List)

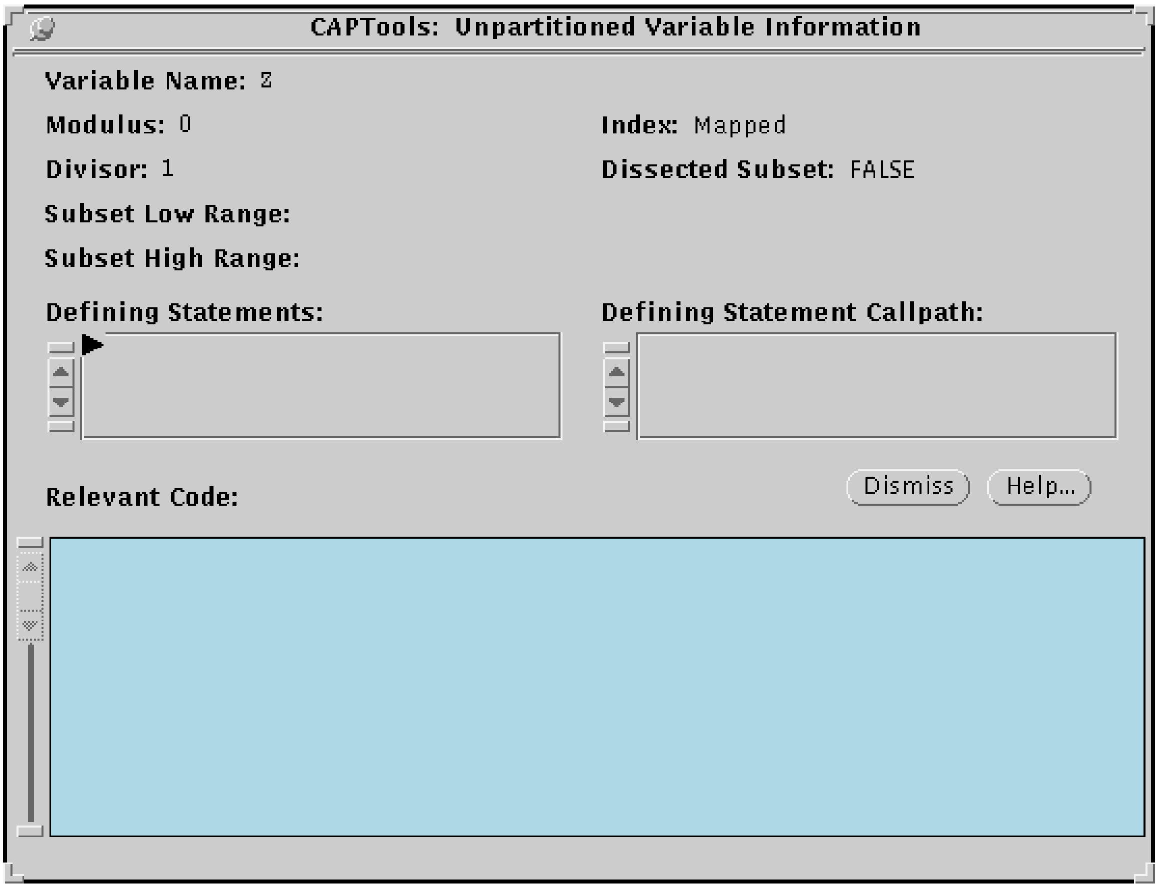
Reason textfield

Explains why the array(s) in the Reasons For Not Partitioning Variable list was not partitioned.

Relevant Code [list]

This will display the code for the routine chosen by the user.

16.5.2 Unpartitioned Variable Information Window



This window shows detailed information about the currently selected variable partition.

Variable Name textfield

The name of the variable highlighted in the Unpartitioned arrays list.

Index textfield

The index or dimension of the variable array that was/was not partitioned.

Modulus textfield

Modulus expression of partition.

Divisor textfield

Divisor expression of partition.

Subset Low Range

Low limit of effect of this partition.

Subset High Range

High limit of effect of this partition.

Defining Statements [list]

This shows a list of possible defining statements for last selected item in this window. Selection of a statement will update the textpane below to show the statement in context.

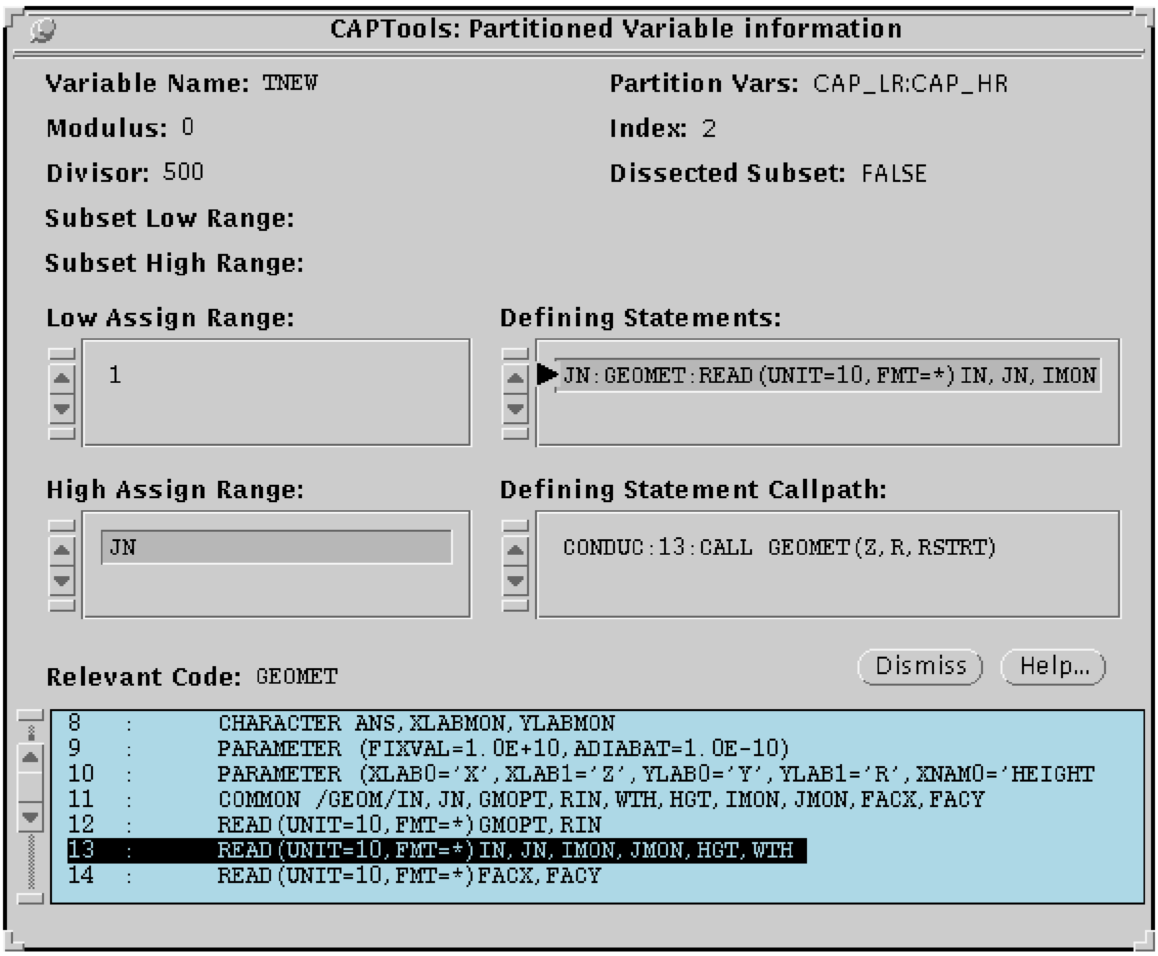
Defining Statements Call Path [list]

This shows the call path relevant to the currently selected defining statement.

Relevant Code [list]

This will display the code for the routine chosen by the user.

16.5.3 Partitioned Variable Information Window



Variable Name textfield

The name of the variable highlighted in the Partitioned arrayslist.

Partition Vars textfield

This informs the user of the names of the low and high range partitioned array.

Index textfield

The index or dimension of the variable array that was partitioned.

Modulus textfield

Modulus expression of partition.

Divisor textfield

Divisor expression of partition.

Subset Low Range

Low limit of effect of this partition.

Subset High Range

High limit of effect of this partition.

Defining Statements [list]

This shows a list of possible defining statements for last selected item in this window.

Defining Statements [list]

This shows the call path relevant to the currently selected defining statement.

Low Assign Range [list]

This informs the user of the Low range of the selected partitioned array as assigned at run time.

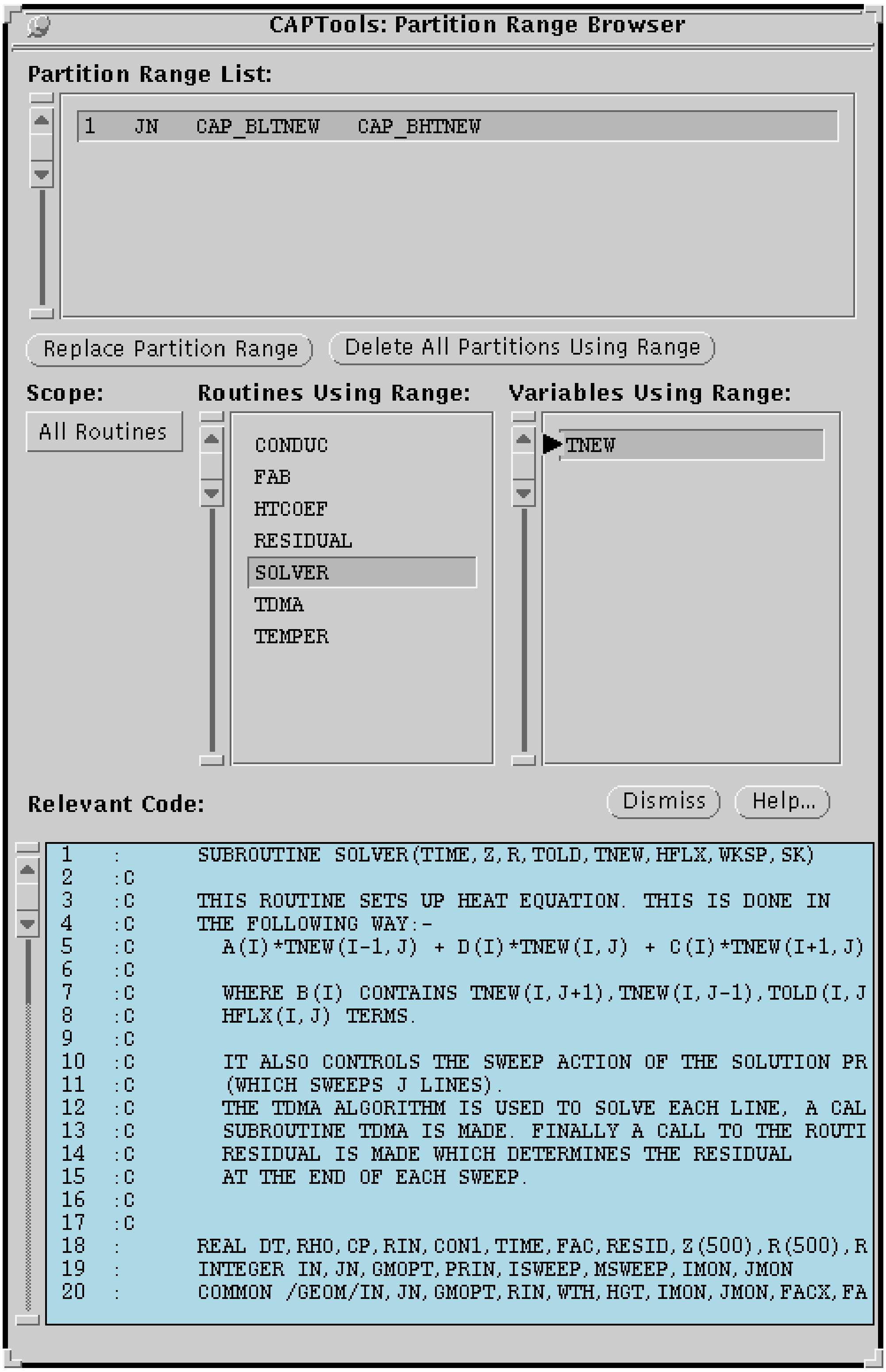
High Assign Range [list]

This informs the user of the Low range of the selected partitioned array as assigned at run time.

Relevant Code [list]

This will display the code for the routine chosen by the user.

16.5.4 Partition Range Browser Window



This window provides a browsing and editing facility of the partition ranges created by the partitioning stage.

Partition Range List [list]

This will show a list of the partition ranges for the partitioned data. Selecting a ange from the list will highlight the Replace Partition Range, Delete All Partitions Using Range buttons, the Scope button and the Routines Using Range list.

Replace Partition Range [button]

This will pop up the Alternative Partitions Ranges window and allow the user to replace the highlighted partition range with another.

Delete All Partitions Using Range [button]

This will delete all partitions that use the highlighted partition range.

Scope [setting]

This allows the user to select All Routines or select a routine from the Routines Using Range list.

Routines Using Range [list]

This allows the user to select a routine from the list of routines that contain arrays that use the selected partition range. This will highlight the Variables Using Range list providing a list of arrays using the selected range.

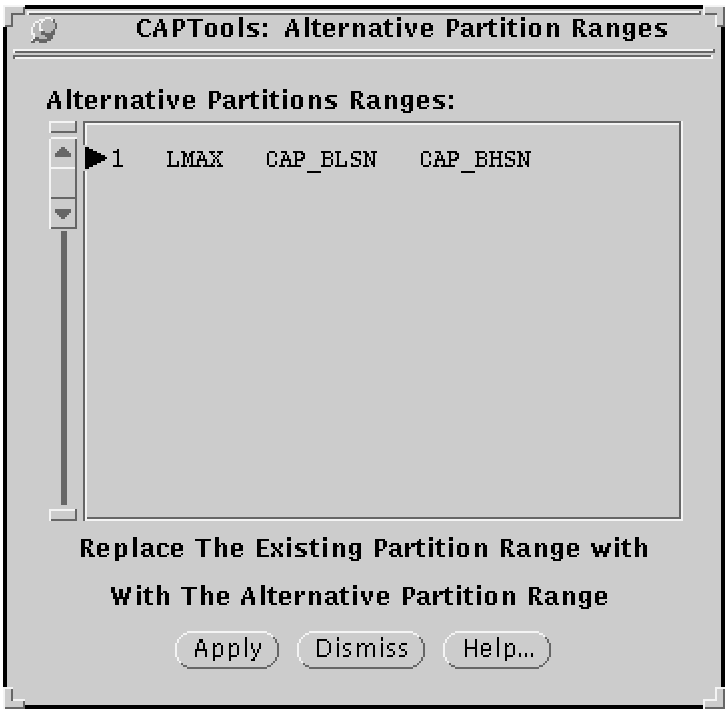
Variables Using Range [list]

Selecting one of the variables in this window will show the code for the routine chosen in the Relevant code text pane.

Relevant Code [list]

This will display the code for the routine chosen by the user.

16.5.5 Replace Partition Range - Alternative Partition Ranges Window



This window shows a list of partition ranges as a choice for replacing the currently selected one in the Partition Range Browser window

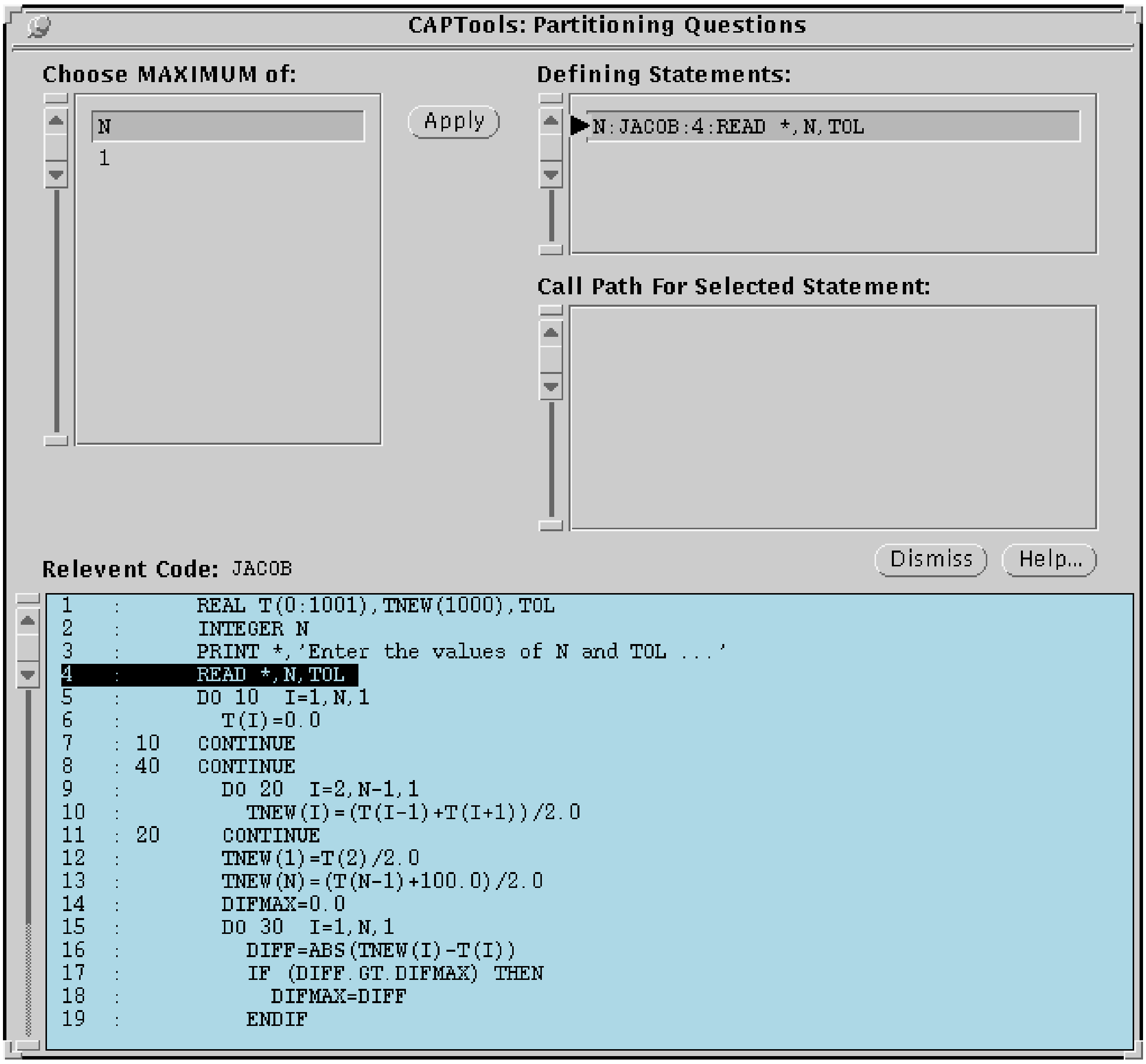
Alternative Partition Range [list]

This informs the user of possible alternative partition ranges which may be applied instead of the existing partition range.

Apply [button]

This will be highlighted when an alternative partition range has been selected. This will apply the new alternative partition range.

16.5.6 Partitioning Questions Window



This window displays a list of questions that must be answered for CAPTools to complete the partitioning. Select those entries in the Choice list that are either the Maximum or the Minimum of the list. The textpane updates to show the code relevant to the last entry chosen in a scrolling list.

Choose MAXIMUM/MINIMUM of [list]

This asks the user to select the maximum/minimum from the scrolling list shown. This is done by highlighting the variable expression and selecting the Apply button.

Defining Statements [list]

Shows the possible defining statement(s) for the highlighted variables in the Choose MAXIMUM/MINIMUM of scrolling list.

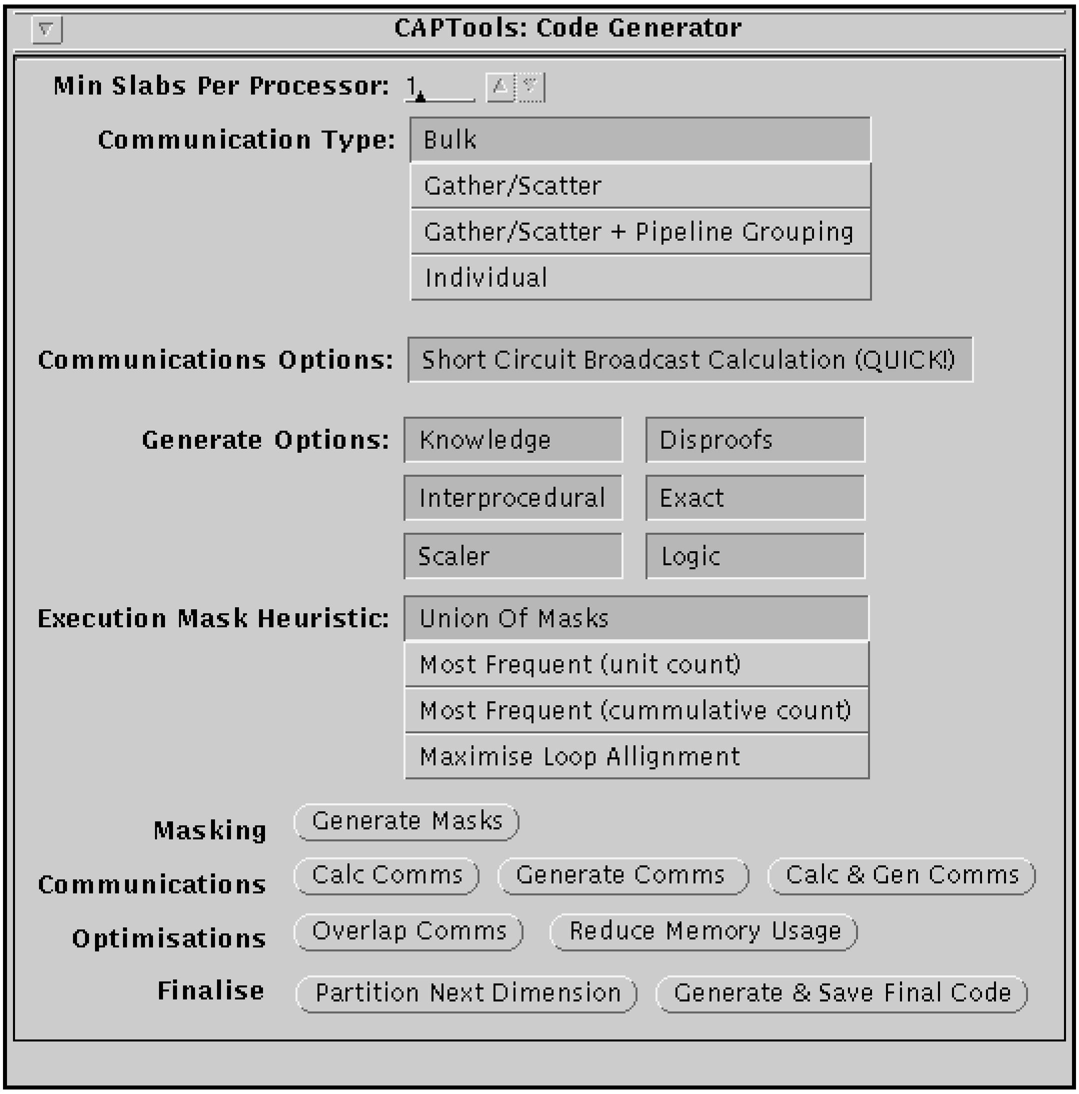
Call path for Selected Statements [list]

Shows the call path for the highlighted variable in the Defining Statements list.

Apply [button]

This accepts all the selected variable in the Choose MAXIMUM/MINIMUM of scrolling list.

16.6 Generator Window



This window allows for the generation of code execution masks, calculation and generation of communication statements and final parallel code.

Minimum Slabs Per Processor textfield

This allows the user to set the minimum number of "slabs" for each processor. For a 1 dimension decomposition of a 3D array slab refers to a 2D plane, for a 1 dimensional decomposition of a 2D array a slab refers to a 1D line of data.

Communication Type [setting]

The user has the choice of four communication types to choose from:

Bulk

This will generate the ordinary synchronous communications exploiting contiguous sections of memory in single messages as described in section 2.2.

Gather/Scatter

This will generate the buffered synchronous communications involving the gathering of data into buffers and the scattering of data from those buffers as described in section 2.2.

Gather/Scatter + Pipeline Grouping

This will generate the buffered synchronous communications as described in section 2.2. Also if there are several pipeline communications for different variables surrounding the same loops, communicating in the same direction then the data from each individual communication is buffered and communicated as a whole. This reduces the excessive number of communication start-up latencies.

Individual

This will perform individual communications (i.e. one array item at a time). Note : this will not generate efficient code and is only for testing purposes.

The default is Bulk communications.

Communications Options [setting]

The user has currently one option which is the default : **Short Circuit Broadcast Calculation (Quick!)** which detects when an assignment statement has had a broadcast of the assigned data placed immediately after it, avoiding the generation of any further communication requests involving that assignment. This will have the minor effect of not listing the entire set of using statements in the **Comms Browser**.

Generate Options [setting]

There are six different options available : Knowledge, Exact, Disproof, Scalar Equality, Interprocedural and Logic. In general, the more of these options selected then the more efficient the code generated.

Execution Mask Heuristic [setting]

There are four different options available :-

Union of Masks

This is the recommended option allowing several masks to control a single statement in order to reduce communication overheads.

Most Frequent (unit count)

Forces a single mask to control each statement where that is the most frequently inferred mask in the masking algorithm. This option is not as effective as the default option!

Most Frequent (cumulative count)

Forces a single mask to control each statement where that is the most frequently inferred mask in the masking algorithm taking into account the frequency of inference on the related statements. This option is not as effective as the default option!

Maximise Loop Alignment. (Not Yet Available)

The default is for the Union of Masks and we recommend this is **ALWAYS** used.

Generate Masks [button]

This will generate execution and control masks for the partitioned arrays. The results of the masking may then be viewed in the Masks Browser window.

Calc Comms [button]

This will calculate and place the required communications.

Generate Comms [button]

This will merge and generate the communications statements in the code. The communications may then be viewed in the Communications Browser window.

Calc & Gen Comms [button]

This will combine the work of the Calc Comms and Generate Comms buttons into one operation.

Overlap Comms [button]

This will apply overlapped communications o the parallel code.

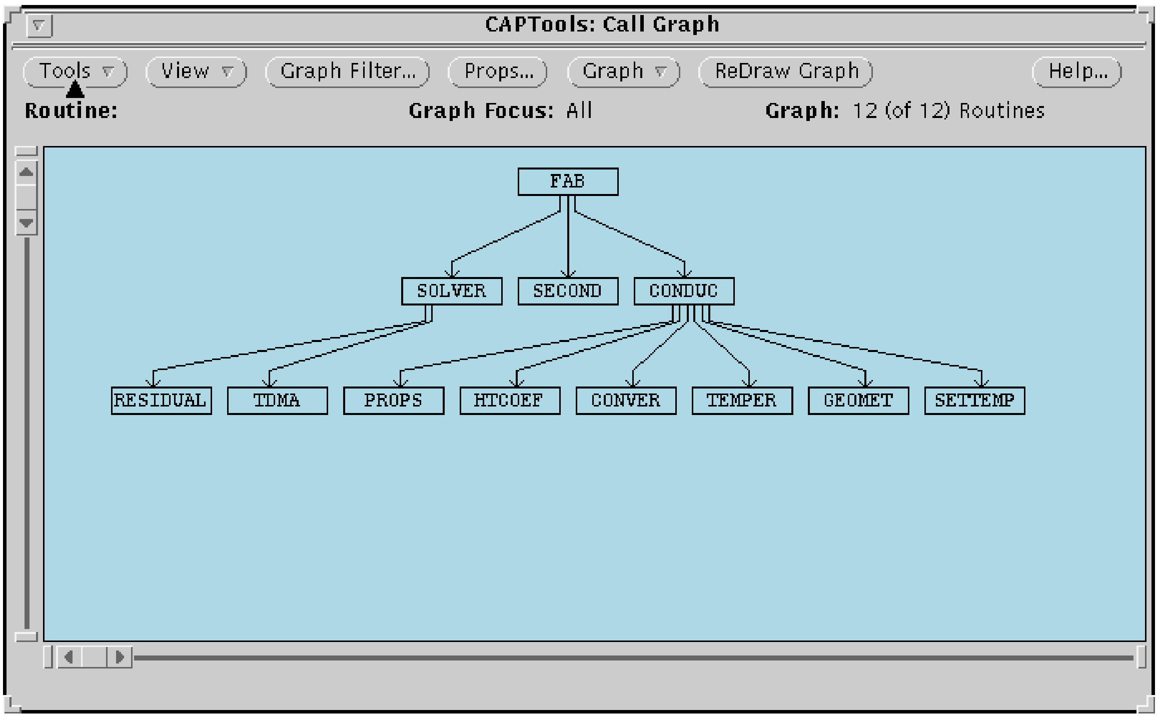
Reduce Memory Usage [button]

This will apply reduce memory usage to the parallel code

Final Code [button]

This button will display a File Chooser window. Once the use selects a file or files to save to the final code is generated out to this file. The code can then be compiled and linked to the CAPTools Communication Library and run on a parallel system.

16.7 Call Graph Browser Window



The Call Graph window allows the viewing and interrogation of the call graph representing the serial code. The graph may be filtered, for example, to display only routines that access certain variable instances or use a particular common block.

Graph [canvas pane]

The graph itself consists of routine boxes and lines connecting the boxes representing a call from one routine by another. The graph is a hierarchical acyclic digraph. A box (routine) is selected with the left mouse button. The right hand button pop up the call graph canvas menu. This menu allows filtering of sections of the call graph depending on which menu item is chosen. The menu also contains additional ‘Hot Links’ to other windows in the CAPTools user interface. The menu options are:- Relatives, Ignore, Calls, Callers, partitioner, Transformations, Call Graph, Dep Graph, Loops, Args/Commons, Var Definition, Global Vars, Masks, Comms. The options specific to this section are :

Relatives

This option shows only the routines that are related to the selected routine. For example, a parent routine is one that calls the selected routine and a child routine is one that is called by the selected routine.

Ignore

This option ignores the selected routine and any other routine calls.

Calls…

This shows a list of all the children of the selected routine. They are displayed as a list in the Routine Calls/Callers window.

Callers…

This shows a list of all the parents of the selected routine. They are displayed as a list in the Routine Calls/Callers window.

Tools [menu button]

This button displays the tools menu which provides access to some of the other windows in the CAPTools user interface. The menu options are:-

Analyser…, Transformations Menu, Partitioner…, Code Generator…, Value Profiler…, Knowledge Base…, READ Knowledge…, User Knowledge…, Call Graph…, Dep Graph..., Loops…, Args/Commons…, Var Definition…, Global Vars…, Masks…, Comms...

View [menu button]

This button displays the view menu that allows filtering sections of the call graph depending on the function chosen. The menu items are :

All This displays all the routines in the code.

Filtered This displays only the routines as defined in the CallGraph Routine Filter window.

Ignore Selected This displays the routines apart form a selected routine and the routines it calls.

Recursive Cycles

Graph Filter … [window]

This displays the CallGraph Routine Filter window.

Props... [window]

This button displays the CallGraph Properties window.

Graph [menu button]

This currently does nothing.

ReDraw Graph [button]

This redisplays the Graph.

Routine textfield

This displays the routine box selected using the left hand mouse button.

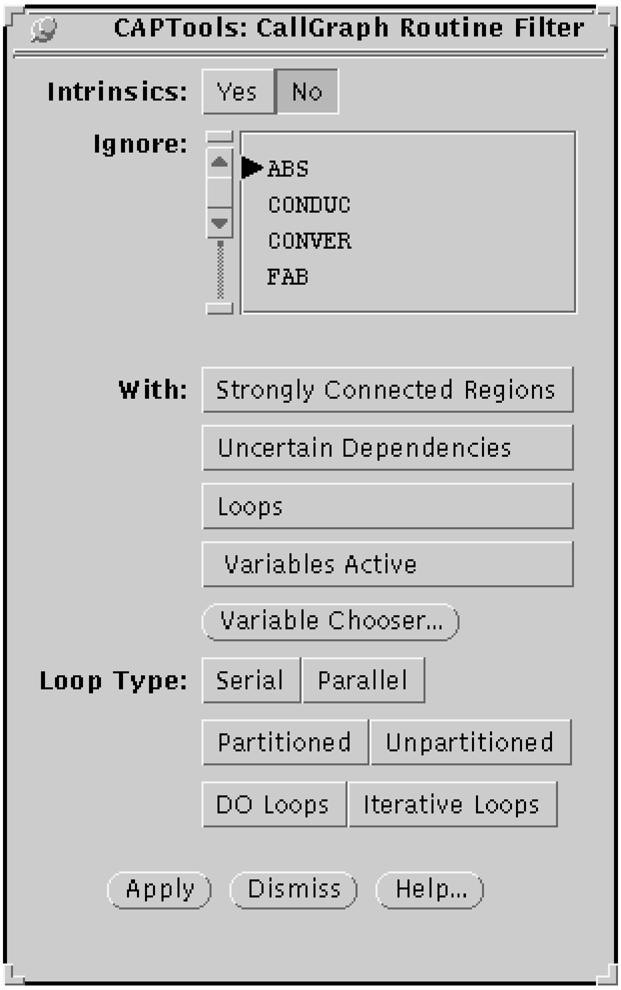
Graph Focus textfield

This informs the user the graph selected from the View menu.

Graph textfield

This displays the number of routine boxes displayed.

16.7.1 Callgraph Routine Filter Window.



This window allows the user to filter the CallGraph to display routines that satisfy the criteria set. The criteria which may be set are :

Intrinsics : [setting]

This allows the user to display intrinsic functions.

Ignore: [list]

The user may select form the list which subroutines to ignore.

With: [setting]

This filters the CallGraph to display routines that contain either of the following :

Strongly Connected Regions, Uncertain Dependencies, Loops, Variables Active.

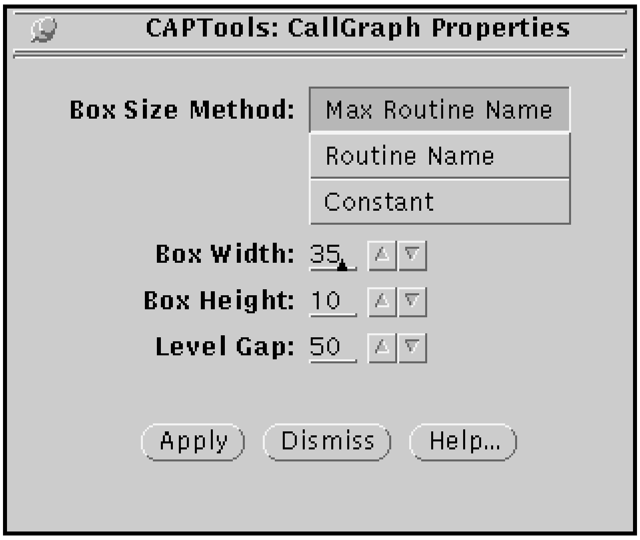
Variable Chooser [button]

This button is highlighted when the Variable Active setting is chosen. This displays the CallGraph Filter Variable Chooser window that allows the selection of variables.

Loop Type: [setting]

This is highlighted when the Loops setting is chosen. The loop types are : Serial, Parallel, Partitioned, Unpartitioned, DO loops and Iterative Loops.

16.7.2 Callgraph Properties Window.



This window allows the user to adjust the properties boxes surrounding the routine names in the CallGraph. These Options are :

Box Size Method : [setting]

This sets the size of the box surrounding the routine name. This can be set to be the same as the Maximum Routine Name, the individual Routine Name or to a Constant size as set by Box Width and Box Height

Box Width:[ ]

This sets the width of the routine surrounding box.

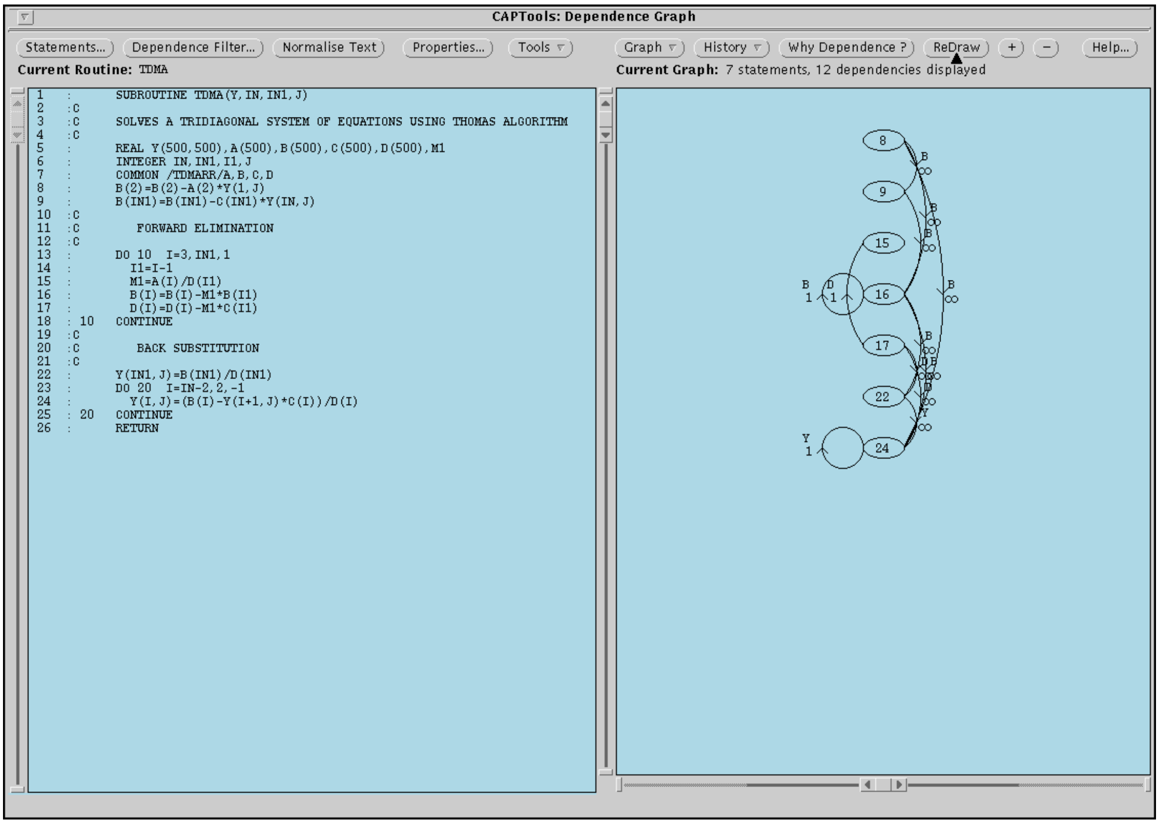
Box Height:[ ]

This sets the height of the routine surrounding box.

Level Gap: [ ]

This sets the gap between the routine box.

16.8 Dependence Graph Browser Window.



The Dependence Graph window allows viewing and interrogation of the dependence graph produced by CAPTools during the dependence analysis it does on FORTRAN source code. The graph is shown for any chosen routine in the right hand canvas pane of the window. The left hand text pane shows the source code. For clarity various filters are provided which allow only particular parts of the graph to be displayed. This window also allows the user to examine and remove particular dependencies via the Why dependence window.

Statements... [window]

Displays the Statements Filter window.

Dependence Filter... [window]

Displays the Dependencies Filter window.

Normalise/UnNormalise Text [button]

This button toggles the display of source code in the text pane between unnormalised (user code) and normalised (CAPTools internal) form.

Properties.. [button]

This button displays the DepGraph Properties window that allows the user to edit display properties for the dependence graph and text presentation.

Tools [menu button]

This button displays the tools menu which provides access to some of the other windows in the CAPTools user interface. The menu options are:-

Analyser…, Transformations Menu, Partitioner…, Code Generator…, Value Profiler…, Knowledge Base…, READ Knowledge…, User Knowledge…, Call Graph…, Dep Graph..., Loops…, Args/Commons…, Var Definition…, Global Vars…, Masks…, Comms...

Graph [menu button]

This button displays the graph menu which has the following menu options: Redraw Graph - redraws the graph. Print Visible Graph - pops up the Print Options window to print the visible graph. Print Complete Graph - pops up the Print Options window to print the complete graph. Zoom In - on the dependence graph. Zoom Out - on the dependence graph. Zoom Reset - returns to the default scale of the dependence graph. Delete All Visible Dependencies – deletes all the dependencies visible in the Dependence Graph.

History [menu button]

Shows previously drawn graphs in the DepGraph window. The first entry in the menu will always be the previous graph drawn. Select this item to redraw the last graph. Other entries in the menu are added whenever a new routine's depgraph is drawn.

Why Dependence? [button]

Displays the Why Dependence? window. This window provides the user with information about any dependence shown in the dependence graph. This is done by selecting an arc using the left button of the mouse once the window is displayed or by placing the mouse over a dependence and using the right hand button to access the depgraph canvas pane menu and then the WhyDependence? menu item.

ReDraw [button]

This will redraw the graph and update the source textpane according the settings in the dependence and statement filter.

+ [button]

Repeated selection of this button zooms in on the dependence graph.

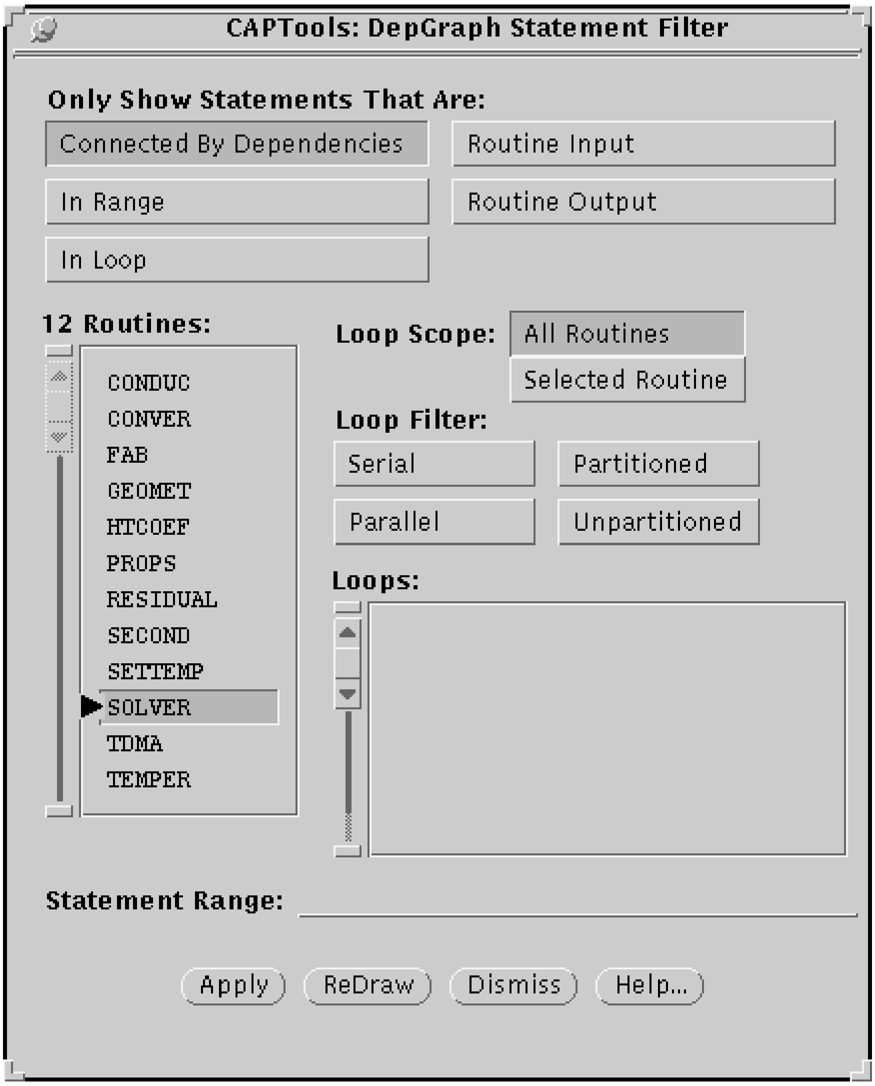
- [button]

Repeated selection of this button zooms out on the dependence graph.

Help... [button]

This provides help for this window.

16.8.1 Depgraph Statement Filter Window



Only Show Statements That Are: [setting]

This gives control over what statements are shown in the dependence graph:-

Connected by Dependencies

Show only statements that are connected by dependencies.

In Range

Show only statements that are in the Statement Range textfield below.

In Loop

Show only statements in the selected loops as selected in the Loop List of this window;

Routine Input

Shows routine input dependencies;

Routine Output

Shows routine output dependencies.

If the In Loop filter is selected then the In Range filter is disabled, and vice versa.

Routines [list]

Select from this list the routine to view in the graph. This is disabled if All Routines are selected in the Loop Scope setting.

Loop Scope [setting]

Filters the display of loops in the Loops list to either a routine selected from the Routines list or all routines in the code.

Loop Filter [setting]

Allows the selection of which loops will be displayed in the Loops list. Choices are of serial, parallel, partitioned or unpartitioned loops for a selected routine or for all routines;

Loops [list]

Displays the loops selected using the Loop Scope and Loop Filter settings. Select a loop from this list to view only dependencies in this loop in the dependence graph.

Statement Range textfield

This item is enabled if the In Range choice was selected in the Statement Filter above. Enter here the range of statements for display in the graph. e.g. '121-123,126-129' or '121,122,123,126,127,129' both of which will display the dependencies for lines number 121 to 123 and 126 to 129 inclusive. Statements can also be selected by:-

Using the left mouse button in the numbered statement ellipse of the dependence graph canvas pane. Using the right mouse button to popup the dependence graph canvas menu and choosing either to add to delete or clear the statement range.

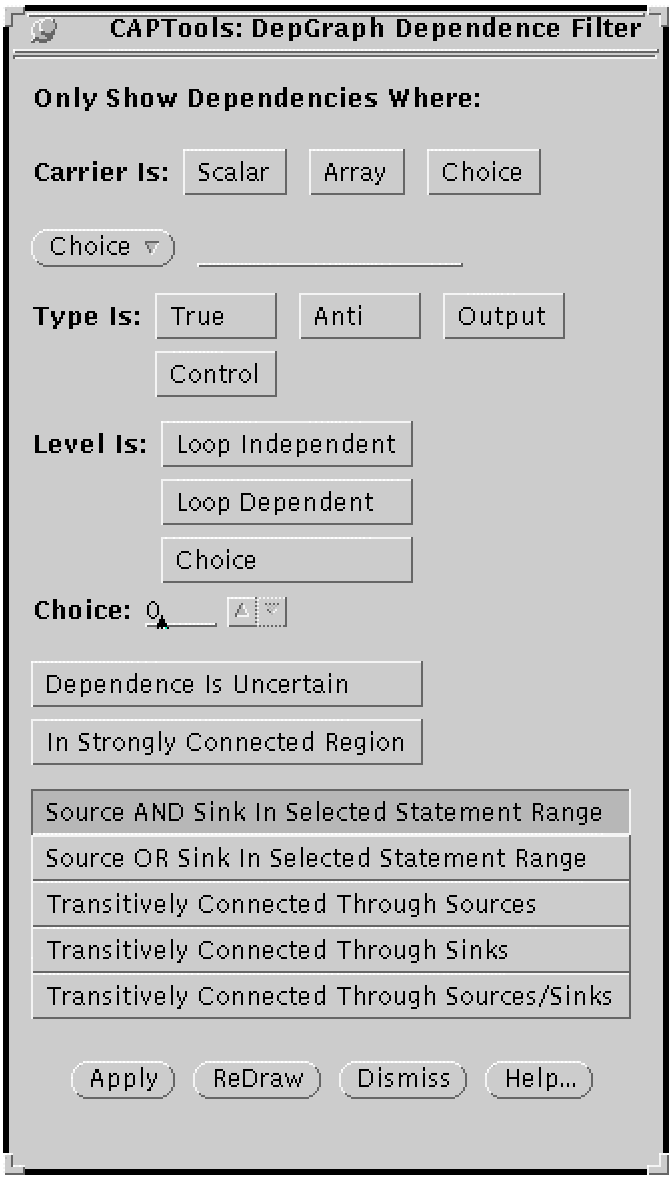
Apply [button]

This accepts all the options selected.

ReDraw [button]

This will redraw the graph and update the source textpane using the options selected in the Statement Filter window and the Dependence Filter window.

16.8.2 Depgraph Dependence Filter Window



This window allows the selection of which dependencies to view in the dependence graph canvas pane.

Carrier Is [setting]

The type of the carrier variables - choice, scalar and/or array. The choice option pops up the Symbol Chooser window allowing the selection of particular variables from a routine.

Type Is [setting]

The type of dependencies to view - true, anti, output and control. For a full explanation refer to the user manual sections on dependence explanations.

Level Is [setting]

Selection of which level of dependencies to show. Either loop dependent i.e. between iterations of a surrounding loop, loop independent in a single iteration of all surrounding loops, or a choice of any dependencies at a particular level.

Dependence Is Uncertain [setting]

Show only dependencies where the dependence was assumed due to unresolved inequalities. These questions can be viewed/answered in the Why Dependence window.

In Strongly Connected Region [setting]

Show only dependencies in strongly connected regions of the dependence graph.

Source AND Sink in Selected Statement Range [setting]

Show any dependencies between statements that are both in the selected sets of statements.

Source OR Sink in Selected Statement Range [setting]

Show any dependencies between statements where either is in the selected set of statements.

Transitively Connected Through Sources [setting]

Show any dependence where the source of the dependence is in the selected set AND add the dependence sink to the selected set to be used to add further dependencies to the viewed graph.

Transitively Connected Through Sinks [setting]

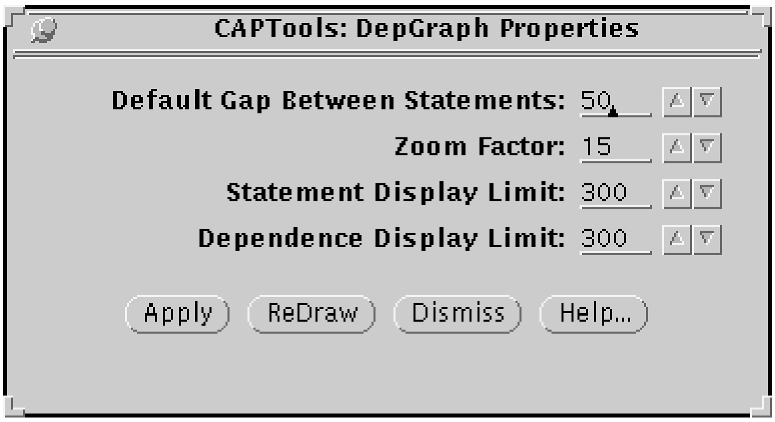
Display any dependence where the sink of the dependence is in the selected set AND add the

dependence source to the selected set to be used to add further dependencies to the viewed graph.

Transitively Connected Through Sources/Sinks [setting]

Display any dependence where either statement is in the selected set AND add the source and sink to the selected set to be used to add further dependencies to the viewed graph.

16.8.3 Depgraph Properties Window.

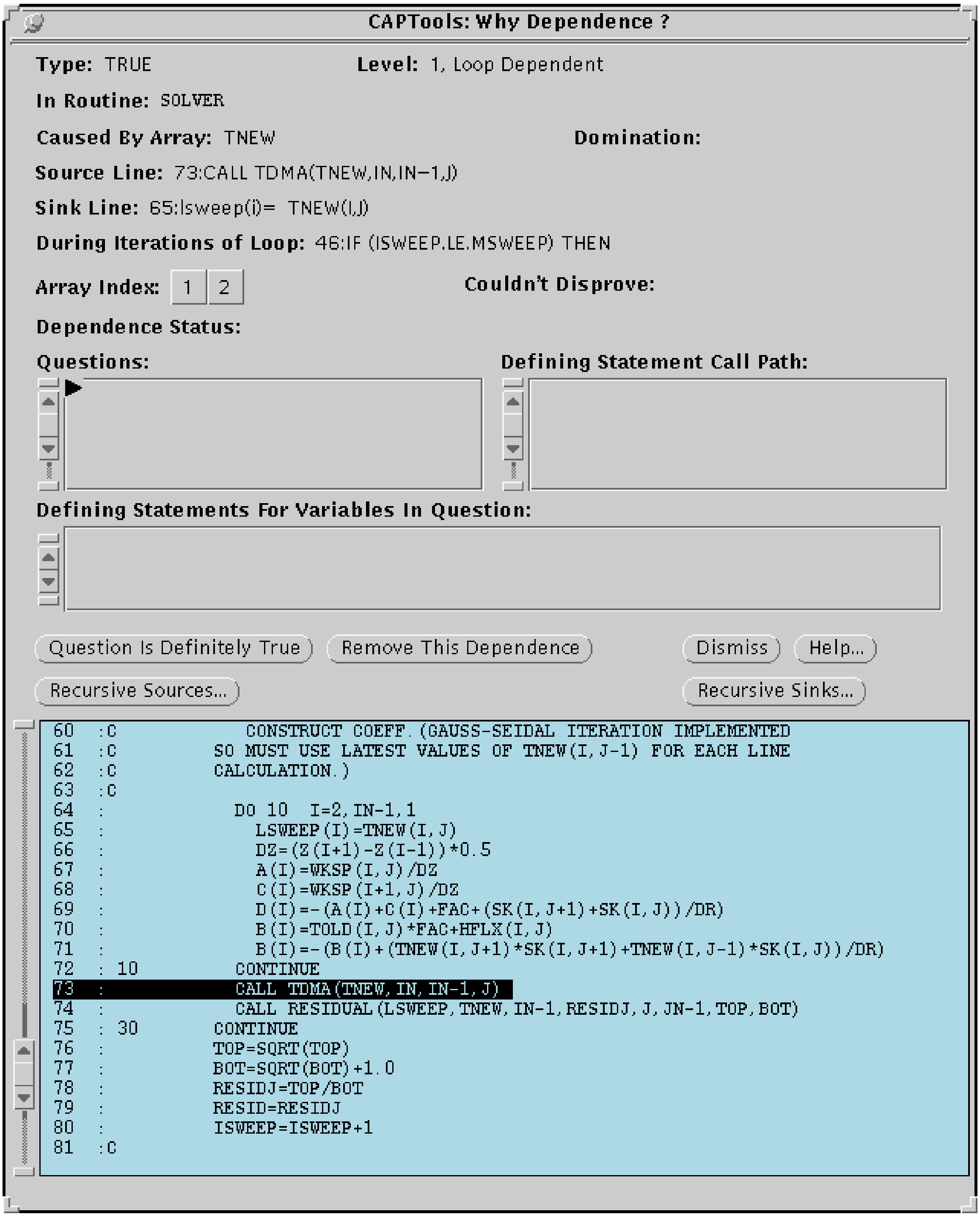


This window allow the user to change properties that effect the dependence graph display. Properties are:-

Default gap between statements in the dependence graph. Zoom increment used when zooming the view of the dependence graph. The maximum number of statements to be viewed in the dependence graph. The maximum number of dependencies to be viewed in the dependence graph.

The Apply button applies this values without redrawing the graph, the ReDraw button redraws the graph using the new values.

16.8.4 Why Dependence ? Window.



This window provides the user with all the information about a particular dependence in the code. It is accessed from the Dependence Graph window or from the Why Communication window in the Communications Browser. This window can also be used to remove dependencies either explicitly or by answering questions (if it is an uncertain dependence).

Type [textfield]

Shows the selected dependence type.

Level [textfield]

Shows the selected dependence level.

In Routine [textfield]

Shows the routine the selected dependence is in.

Caused by Scalar/Array [textfield]

Shows the dependence carrier variable.

Domination [textfield]

States if is a pre dominated and/or a post dominated dependence.

Source Line [textfield]

Show the source line of the dependence.

Sink Line [textfield]

Show the sink line of the dependence.

During Iterations of Loop [textfield]

Array Index [setting]

This setting shows the number of dimensions of an array. Selection of a particular dimension or index will display any questions CAPTools has on this dependence caused by this index. This is only enabled when a dependence is caused by an array.

Couldn’t Disprove [textfield]

Dependence Status [textfield]

This states the status of the dependence, i.e. is it a definite dependence or can it be removed?

Questions [list]

This is a list of questions posed by CAPTools for the selected dependence which were unresolved as a result of dependencies tests in the for the dependence selected. Select a entry to display the defining statements for the variables in the question and to enable the Question Is Definitely True button.

Defining Statement Call Path List [list]

This shows the call path relevant to the variables selected in the Defining Statements for Variables in Question list.

Defining Statements for Variables in Question [list]

Shows a list of possible defining statements for the variables in the Questions list.

Question is definitely true [button]

Allows any of the questions shown in the Questions list to be answered as true. Enabled only when a question is selected.

Remove this dependence [button]

Allows the removal of the selected arc (dependence) from the dependence graph. A warning notice is displayed since this procedure is very dangerous unless one is very sure of the consequences of this action.

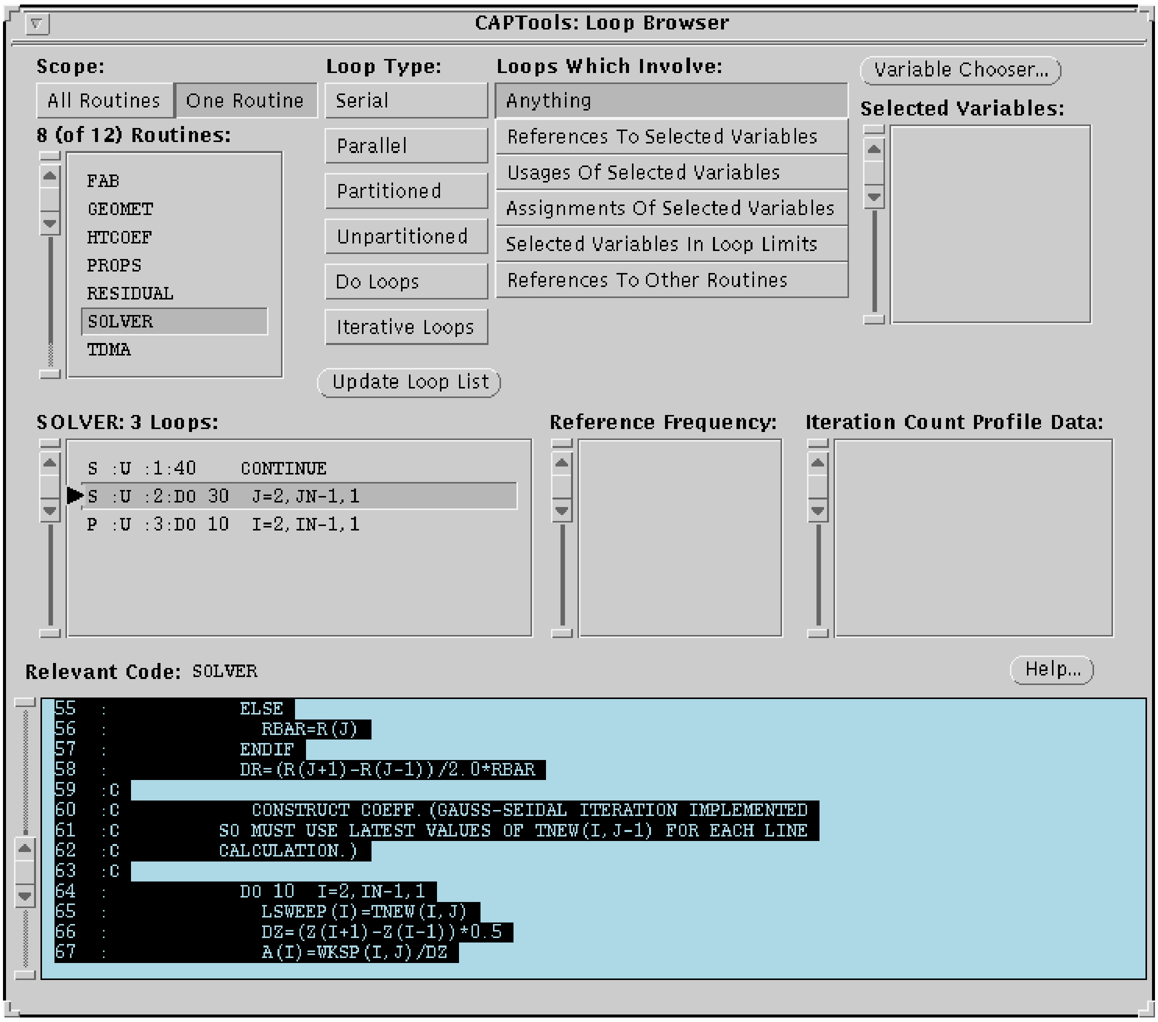
Show recursive sinks [button]

Display window with list of the statements that are potential sources for a dependence whose source is a routine reference.

Show recursive sources [button]

Display window with list of the statements that are potential sinks for a dependence whose sink is a routine reference.

16.9 Loop Browser Window



This window provides loop browsing facilities of all the DO loops and iterative loops in the code being parallelised. Choice of various filters allows the Loops list to be updated to show loops having certain attributes. Selection of a loop in the loop list displays the relevant source code in the textpane below.

Scope [setting]

Select All Routines to process all loops or select One Routine to select a routine from the Routine list.

Routine [list]

Select a routine from this list to process only loops from that routine.

Loop Type [setting]

Filters which loops to view in the Loops list. These can be Serial, Parallel, Partitioned and/or Unpartitioned, Iterative and/or DO loops depending on selection.

Loops [list]

This gives the user a list of the loops that come through the set of filters that have been chosen in Loop Type, and Loops Which Involve. The selection of a loop, highlights that loop, in context of its routine in the textpane below.

Loops Which Involve [setting]

Filters which loops to view in the Loops list. This filter allows the user to look at loops containing :-

Anything, References to Selected Variables, Usages of Selected Variables, Assignments of Selected Variables, Selected Variables in Loop Limits, References to Other Routines

Only one filter at a time can be selected for a loop browse.

Selected Variables [list]

Choose from this list the symbols that are used in the Loops Which Involve filter. The symbols displayed are all those for the current routine.

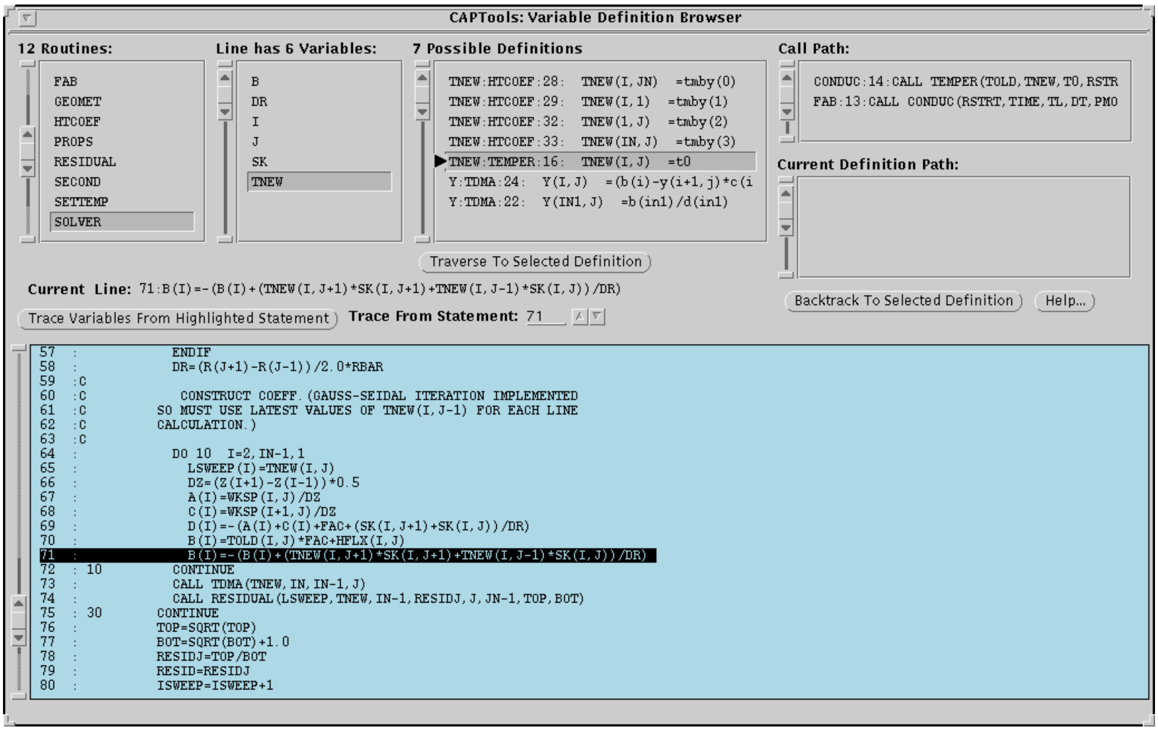
Reference Frequency [list]

This shows the number of times a variable has been referred to. (Option not yet available).

Iteration Count Profile Data [list]

This shows the values that a variable held and the number of times it held the value. (Option not yet available).

16.10 Variable Definition Browser Window



The Variable Definition Browser window allows the tracing of the definition of a variable's values up and down its call path. Select a variable to browse by highlighting a variable or statement line in the textpane or by typing the line number into the Trace from statement textfield.

Routine [list]

This displays a list of all the routines in the code.

Trace Variables From Highlighted Statement [button]

This will set the selected statement to be the first statement highlighted in the textpane.

Trace From Statement textfield

This is the actual line number of the selected statement. It can be either altered by the arrows or typed in directly. The line selected in the Trace From Statement is highlighted in the window textpane, showing it in context with the rest of the selected routine around it.

Variables [list]

This displays all the variables used on the selected statement line.

Possible Definitions [list]

This displays all the possible definitions of the variable selected in the Variable list. The selection of a specific definition must precede the Traverse To Definitions selection. If only one definition is available this is automatically selected.

Traverse To Selected Definition [list]

This traverses the current focus up through the code, to the definition selected in the Possible Definitions list. The Variables and Possible Definitions list is updated to reflect the new current statement.

Current Definition Path [list]

This displays the history of the current definition path. One of the stages must be selected before the Backtrack To Previous Definition, allowing the user to step back to any point along the call path they have chosen.

Backtrack To Previous Definition [list]

This traverses back to the previous definition selected in the Current Definition Path list.

16.11 Argument Browser Window

The argument browser provides a mechanism for tracing up and down the call graph of the code being parallelised examining the argument lists and commons in the called and caller routines.

Routine Selector.. [button]

Displays the Routine Chooser window, the routine selected then becomes the current routine for this window. By default, the main calling routine within the code will be selected as the current routine.

Current Routine [textfield]

Shows the current selected routine.

Global Variables [list]

This list shows the global variables for the selected routine.

Commons [list]

This list shows the common blocks for the selected routine.

Routine Callers [list]

This list shows all the parent routines of the current selected routine. The selection of a caller routine in the list activates the Traverse Up button. This allows the browsing up call paths through the code.

Routine Calls [list]

This list shows all the child routines of the currently selected routine. The selection of called routine in the list activates the Traverse Down facility. This allows the user to browse down call paths through the code.

Traverse Up [list]

This button sets the selected caller to be the current routine and updates the window accordingly.

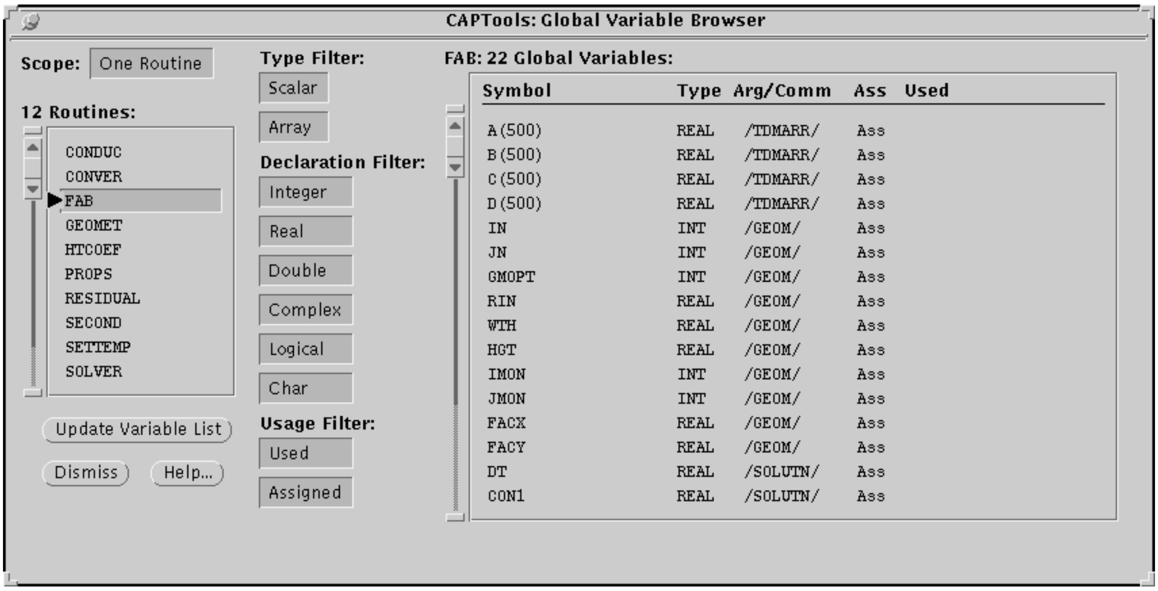
Traverse Down [list]

This button sets the selected call to be the current routine.

Argument Filter [setting]

This is a filter for Arguments and Common blocks, the user can limit what they wish to see updated in the Commons list and the Arguments list.

16.12 Global Variable Browser Window



This window provides a means to view all the arguments and common blocks for a particular routine or all routines in the code possibly after apply a series of filters.

Scope [setting]

This sets the scope for one routine, or all routines in the code. The default is for all routines, the selection of Routine shows the Routine List.

Routines [list]

A list of all the routines in the code, except intrinsic functions, can be selected by the user.

Update Variable List [button]

Clicking on this button updates the Global Variables list.

Type Filter [setting]

The filters available are:- Scalar, Array. In the default all options are active.

Declaration Filter [setting]

The filters available are:- Integer, Real, Double, Complex, Logical, Char. In the default all options are active.

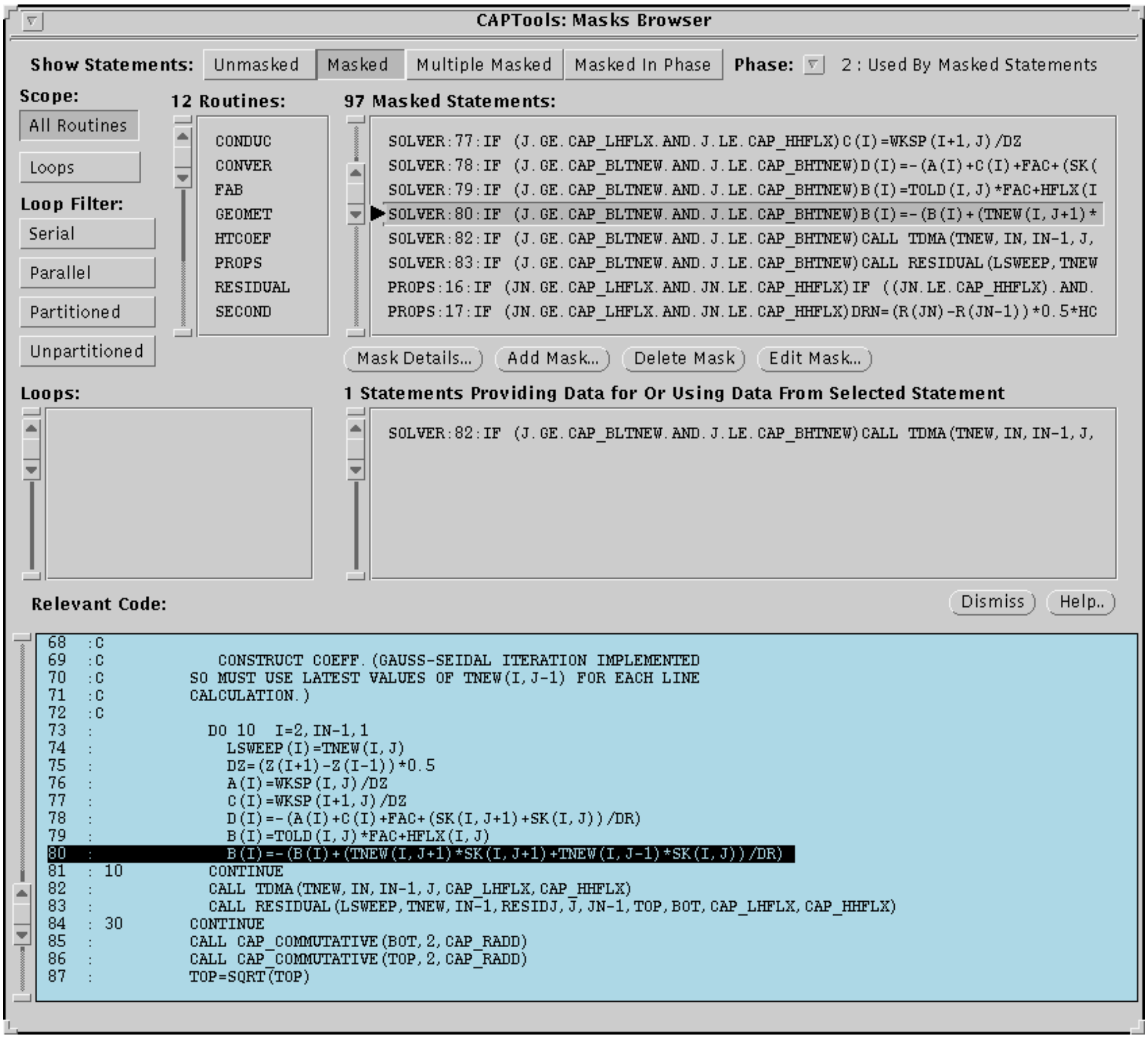
Usage Filter [setting]

The filters available are:- Used, Assigned. In the default all options are active.

Global Variables [list]

This displays information relevant to the options selected.

16.13 Masks Browser Window



This window provides a means of viewing and editing the execution control masks created during the code generation stage in the generator window. The text pane will show the source code for either all routines, routine or selected loops. A entry in the in Masked/Unmasked Statements or Statements Providing Data for or Using Data From Selected Statement list will be highlighted in the textpane when selected.

Scope [setting]

This allows the user to select either All Routines or a routine from the Routines list.

Routines [list]

This allows the user to choose the routine in which to browse the masked/unmasked statements.

Show Statements [setting]

This allows the user to either view a selection of masks filtered using one of the following categories Unmasked, Masked, Multiple Masked or Masked in Phase statements of the chosen routine. These statements are be displayed in the Masked/Unmasked Statements list.

Phase [setting]

This is highlighted when the Masked in Phase option is selected (using the right mouse button). It will show the masked statements for four different possible masking algorithm phases, namely, 1 - Assigning partitioned data 2 - Used by masked statements 3 - Using masked statements 4 - Using partition data

Masked/Unmasked Statements [list]

This displays all the statements for the option chosen in Show Statements for the routine highlighted in Routines.

Loops [setting]

This allows the user to look at masks within loops. When this option is chosen Loop Filter and Loops list is highlighted.

Loop Filter [setting]

This allows the user to filter what loops to view. These can be Serial, Parallel, Partitioned or Unpartitioned.

Loop [list]

This gives the user a list of the loops for that set in the Loop Filter.

Statements Providing Data for Or Using Data From Selected Statement [list]

This will show all statements that either use data or provide data for the statement highlighted in the masks list.

Mask Details ... [button]

This button will be highlighted when a masked statement has been chosen from the Mask Browser window and the Show Statements is set to Masked. It will pop up the Mask Details window.

Add Mask.. [button]

This button will be highlighted when an statement has been chosen from the Masked/Unmasked Statements window. It will pop up the Add Mask window.

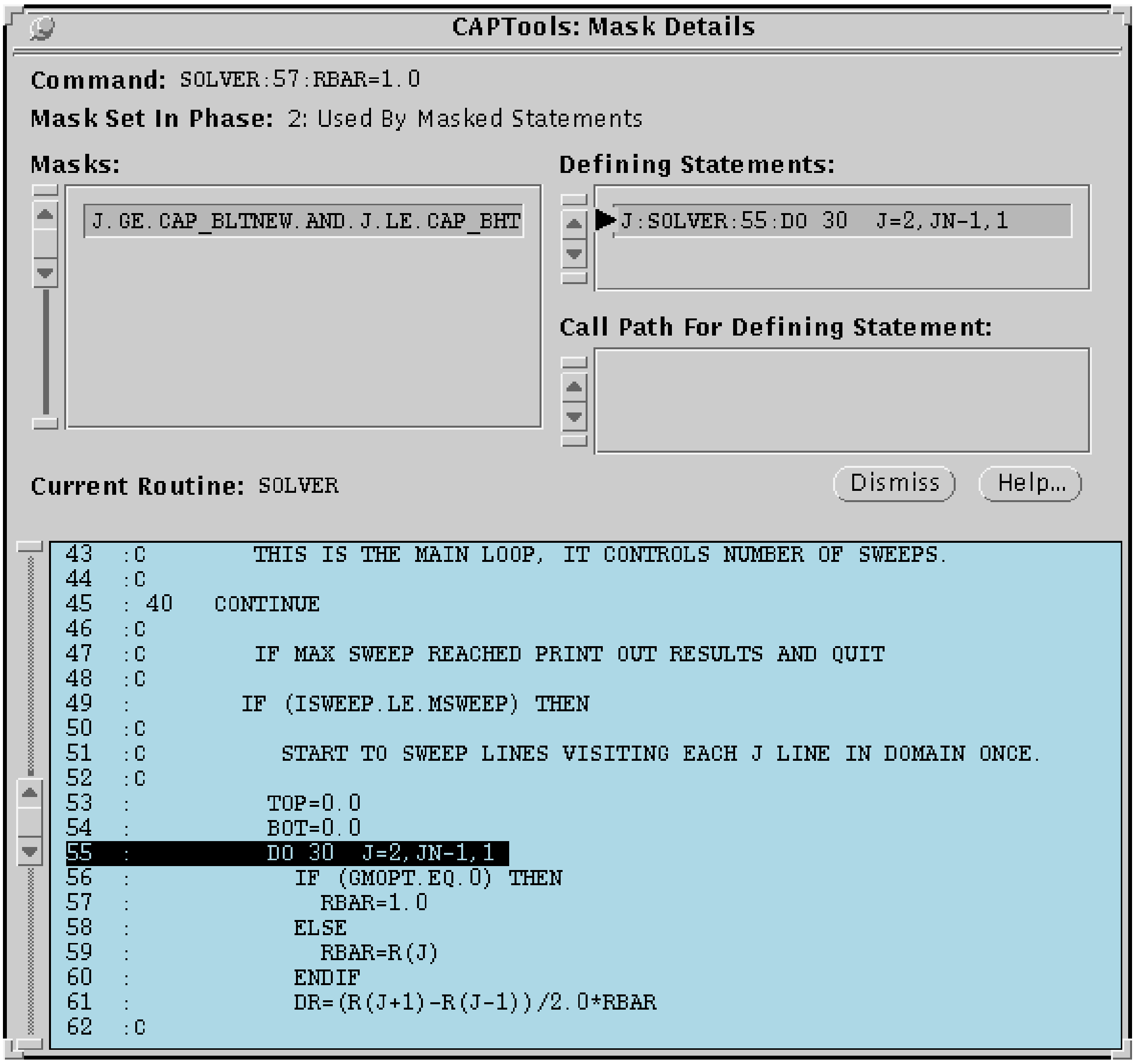
Delete Mask [window]

This will delete a masked statement highlighted in the Masked/Unmasked Statements window.

Edit Mask [button]

(Option not yet available)

16.13.1 Mask Details Window



This window shows details of the current masked statement selected in the main Masks window. The source textpane displays the relevant code for the last selected entry in a list.

Command textfield

This will show the statement that has been masked.

Mask Set In Phase textfield

This informs the user of the reason why a mask was set during the masking algorithm phase.

Masks [list]

This will show the expressions that make up the masks.

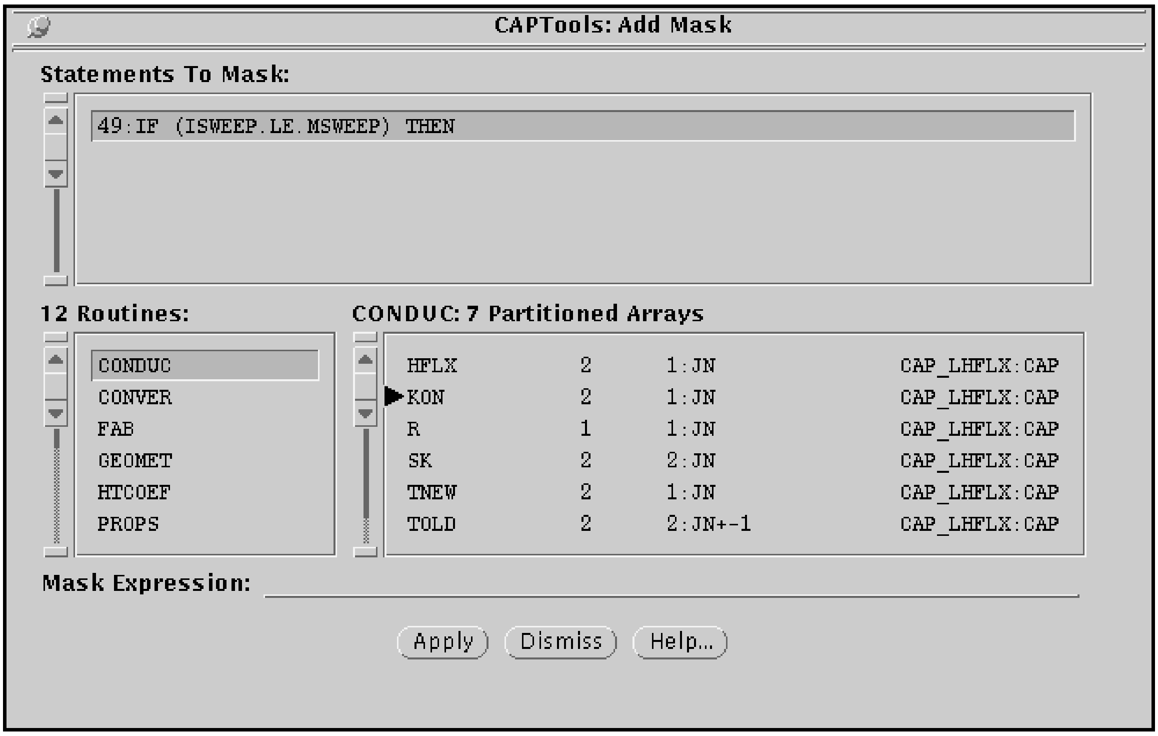
Defining Statements [list]

When a selected mask is highlighted this list shows the defining statement(s) of the variables on which the mask is based.

Call Path for Defining Statement [list]

When highlighting a defining statement the corresponding call path is shown, if one exists.

16.13.2 Add Mask Window



This window provides a way to add a mask to a currently unmasked statement.

Statements to Mask textfield

This will show the statement that is be masked.

Routines [list]

This allows the user to select a routine.

Partitioned Arrays [list]

Lists the partition ranges of arrays in the present selected routine to use in new mask.

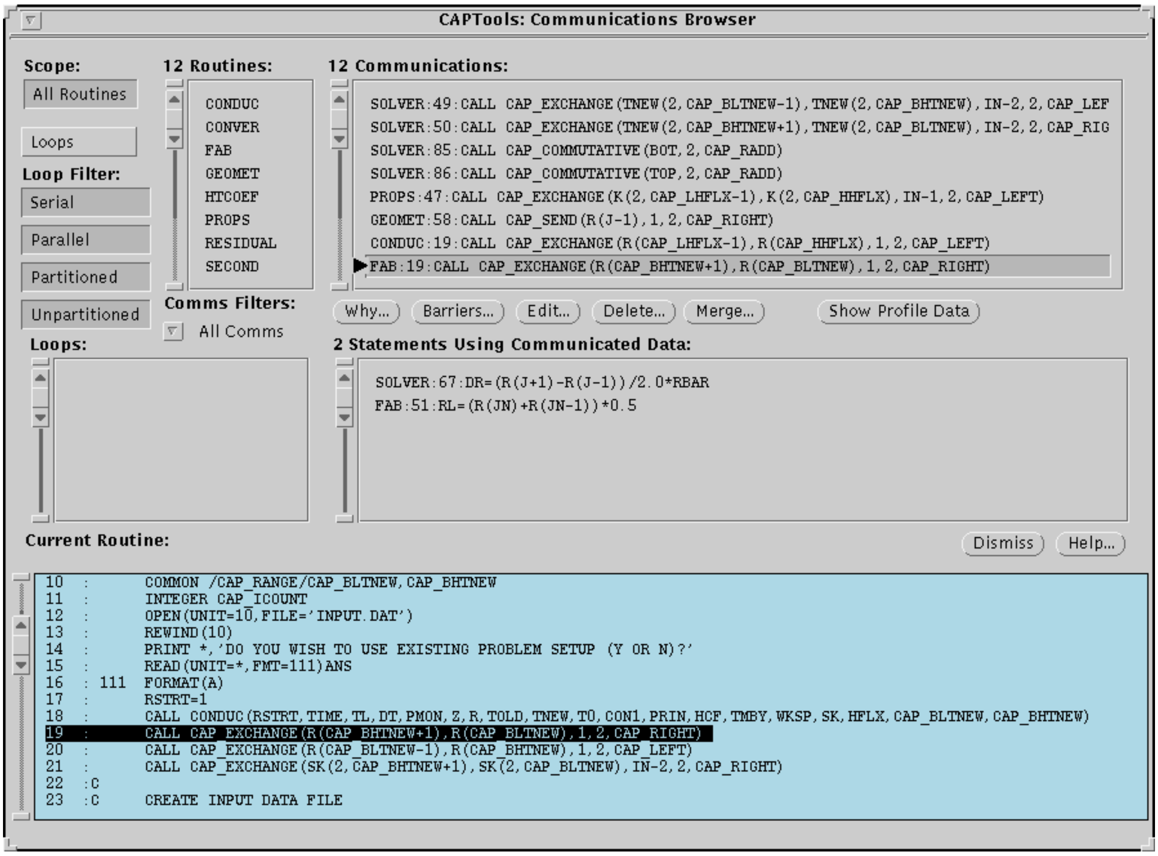
Mask Expression textfield

User must enter an expression to mask the statement with.

Apply [button]

This will be highlighted when a Partitioned Array has been chosen. This will apply the Mask Expression entered and the partition range in the Partitioned Arrays list to the unmasked statement highlighted in the Unmasked Statement.

16.14Communications Browser Window



This window provides a means to view all the communications statements generated during the code generation phase. Use the routine list to select the routine of interest or a particular loop or all routines. The text pane will show the source code for a selection from the Communications list or the Statements using Communicated Values list.

Scope [setting]

This allows the user to select All Routines or select a routine from the Routine list.

Routines [list]

This allows the user to choose the routine in which to browse the communication statements.

Loops [setting]

This allows the user to look at communications within loops. When this option is chosen Which Loops and Loops is highlighted.

Which Loops [setting]

This allows the user to filter what loops to view. These are Serial, Parallel, Partitioned or Unpartitioned.

Loops [setting]

This provides the user a list of the loops from the given loop filter in Which Loops.

Communications [list]

This will list all the communications calculated and generated for the routine or loops chosen. When a communication statement is chosen from the list the Why, Barriers, Edit and Delete buttons and the Statements Using Communicated Values will be highlighted.

Statements Using Communicated Values [list]

This window lists all statements that use values communicated by the communication highlighted in the Communications window.

Why.. [button]

This button will be highlighted when a communication statement has been chosen from the Communications window. It will pop the Why Communication window.

Barriers.. [button]

This button will be highlighted when a communication statement has been chosen from the Communications window. It will pop the Barrier for Selected Communication window.

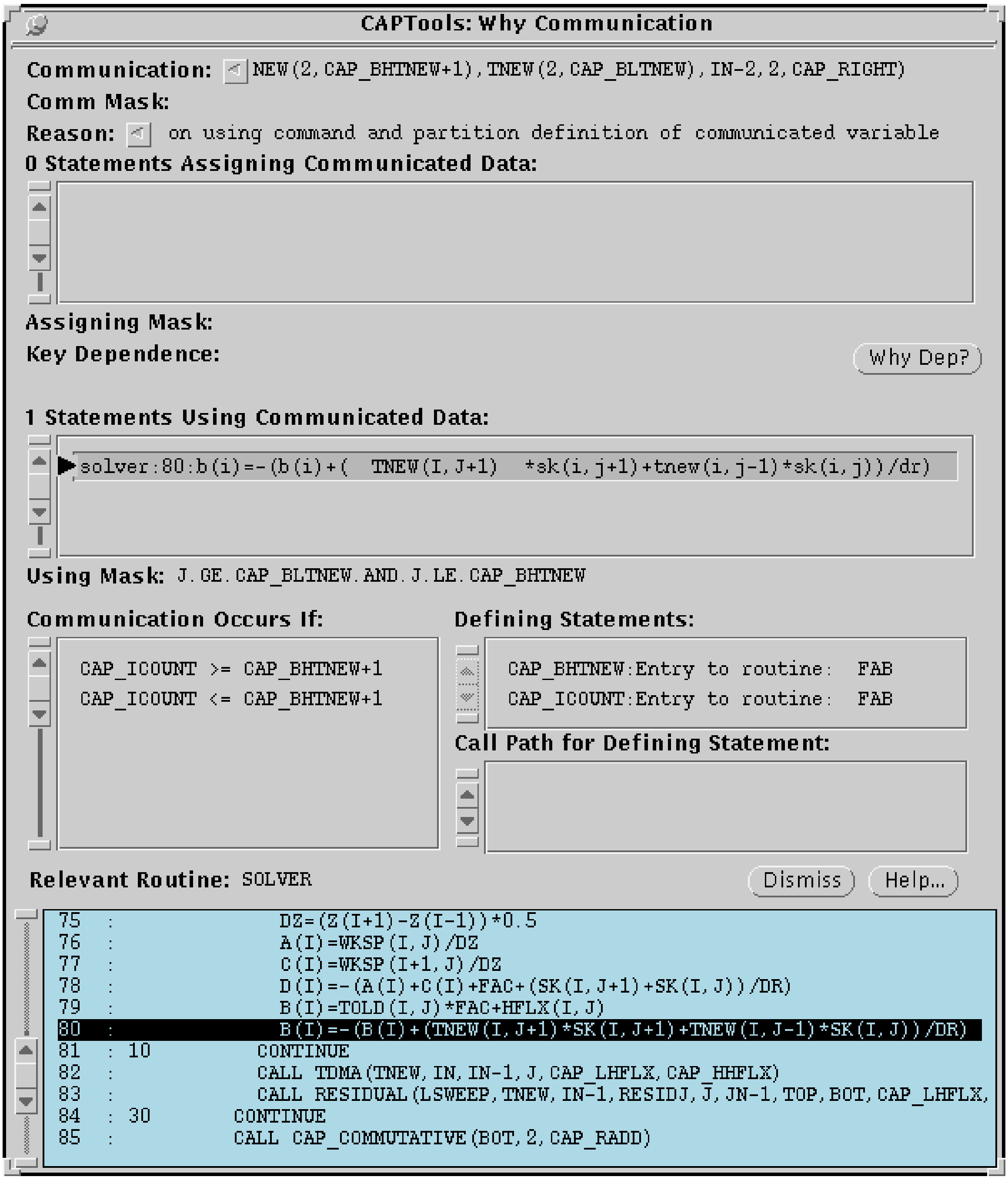
Edit... [window]

This button will be highlighted when a communication statement has been chosen from the Communications window. It will pop up the Edit Communication window.

Delete [button]

This button will be highlighted when a communication statement has been chosen from the Communications window. This will delete the communication statement highlighted in the Communications window.

16.14.1 Why Communications Window



The window shows the reasons a communication has been displayed.

Communication textfield

This shows the communication statement chosen to view.

Comms Mask textfield

This the any execution control mask on the communication.

Reason textfield

This describes the reason for the communication being needed.

Assigning Statements [list]

This will show the statements that assign the values that are being communicated. Each entry is paired with the using statements list and sets that list on selection. This list is only active for unpartitioned data.

Assigning Mask textfield

This shows the any execution control mask on the selected assigning statement.

Using Statements [list]

This will show the statements that use the values that are being communicated. Each entry is paired with the assigning statements list and sets that list on selection.

Key Dependence textfield

This show the dependence that caused the communication

Why Dep... [window]

This displays the Why Dependence window for the key dependence.

Communication Controls [list]

This will show the communication request control expressions for this communication statement chosen.

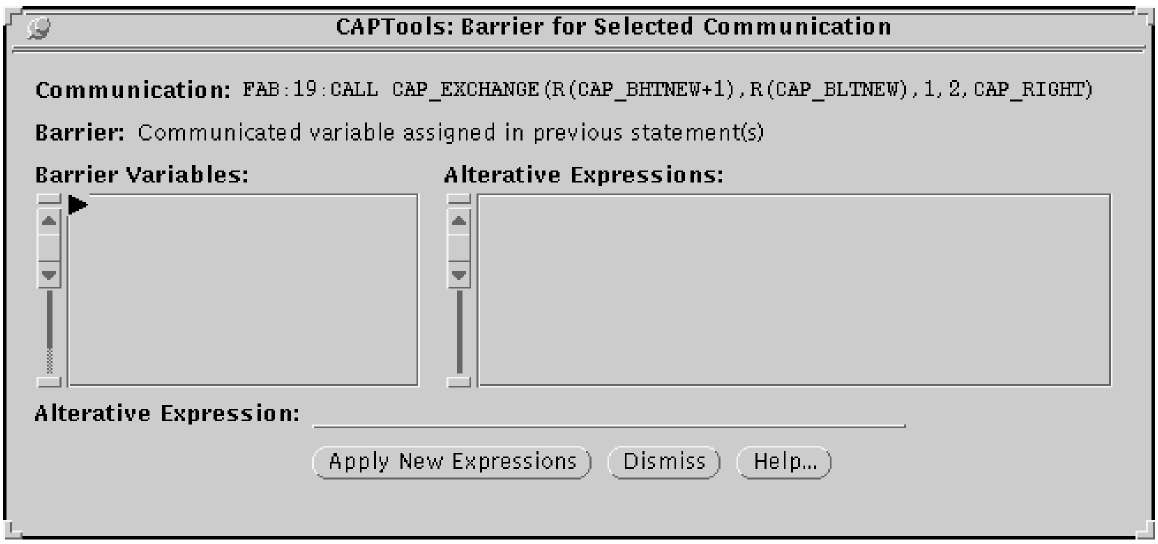
Defining Statements [list]

This lists the variables and defining statements for each variable in the selected communication control.

Call Path [list]

This shows the call path for the selected defining statement.

16.14.2 Barrier For Selected Communication Window



Communication textfield

This will show the communication statement in question.

Barrier textfield

This will give the reason to the barrier which will not allow the communication to be placed further up in the code.

Barrier Variables [list]

This lists the variables that formed the barrier to communication movement, i.e that are assigned in the previous code line. This will only be setup if index expression or mask variables form the barrier.

Alternative Expressions textfield

This allows the user to specify an alternative expression for the selected Barrier Variable.

Alternative Expression textfield

This list the set of alternative values for the variable selected in the Barrier Variables scrolling list.

Apply New Expression [button]

This applies the alternative expressions to the barrier variable, substituting for it and creating a set of communications.

16.15 Requested Knowledge Window

This window provides a way of viewing the questions CAPTools has generated as a product of its dependence analysis. By answering these question the user can help CAPTools in eliminating uncertain dependencies (ones that may be false but could not be proved so). The text pane shows source code relevant to the last selected item from a list in the window.

Question Filter... [button]

This will pop up the Question Filter window. This allows the user to decide which questions and how many of them to view in theRequested Knowledge [window].

CAPTools Questions is [setting]

Allows the user to select the mode of investigation of the unresolved questions, these include:- -If the questions are To Be Examined then by highlighting one of the unresolved questions an expression (in English) is presented in the box immediately below. -If a question is known to be Definitely True then the user can provide this information by selecting the Definitely True filter and then highlighting the question. As a result, the question is removed from this list and is added to the User Knowledge Items list. -The Definitely False filter follows a similar procedure to Definitely True , except that the question appearing in the User Knowledge Items list is opposite to the original question. For example, if J≥3 is Definitely False it is added t the User Knowledge Items list as J<3.

CAPTools Questions [list]

This is a list of unresolved inequalities which are displayed as a result of the filters used in the Question Filter.

User Knowledge Items [list]

This is a list of all user known information defined by the user or derived by CAPTools.

Delete [button]

This will delete a highlighted inequality from the User Knowledge Items list.

Analyse Question Defining Statements [list]

This shows the defining statement(s) from the variables present in the selected inequality.

(Scrolling List)

Analyse Question Statement Call Path [list]

This shows the definition call path relevant to the variables selected in the Analyse Question Defining Statements list.

Remove Definitely False [button]

Allows the user to remove an inequality as being definitely false based on the examination of the inequality.

Profile NonRead Variable [setting]

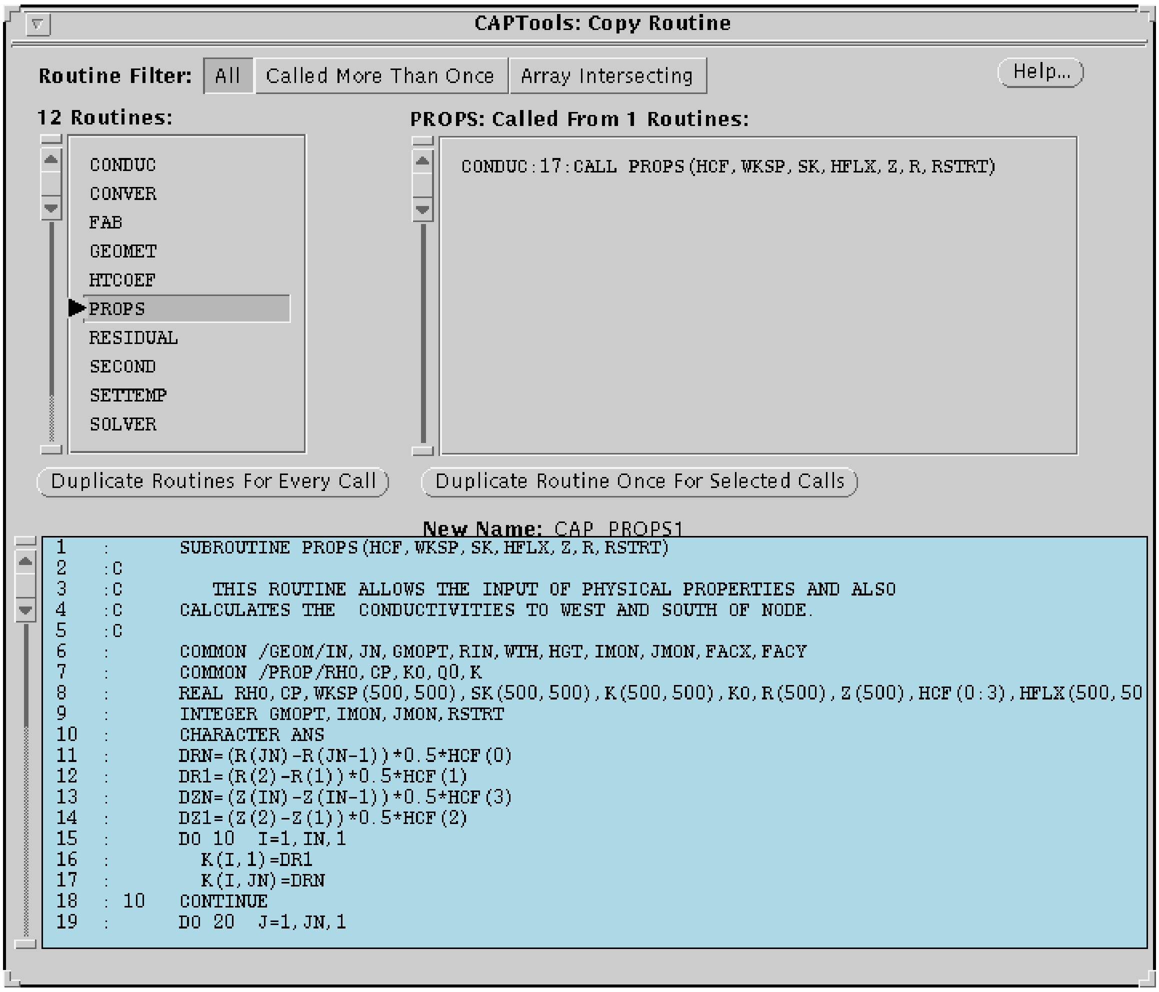
(Option not yet available).

16.15.1 Question Filter Window

This window allows the filtering of questions in the RequestedKnowledgewindow.

Currently under construction.

16.16Transformations - Routine Copy Window



This window provides for the copying of routines. The textpane shows the source code for either the selected routine or the selected call, whichever was selected last.

Routine [list]

Select the routine to be copied from this list

Called from Routine [list]

All calls/function uses to the selected routine are listed. The call(s) that the user wishes to refer to the copied routine should be selected.

New Name [texfield]

This sets the name for the copied routine. The default name is the original name prefixed with "CAP_" and postfixed with a number. The user can alter this text to the desired routine name.

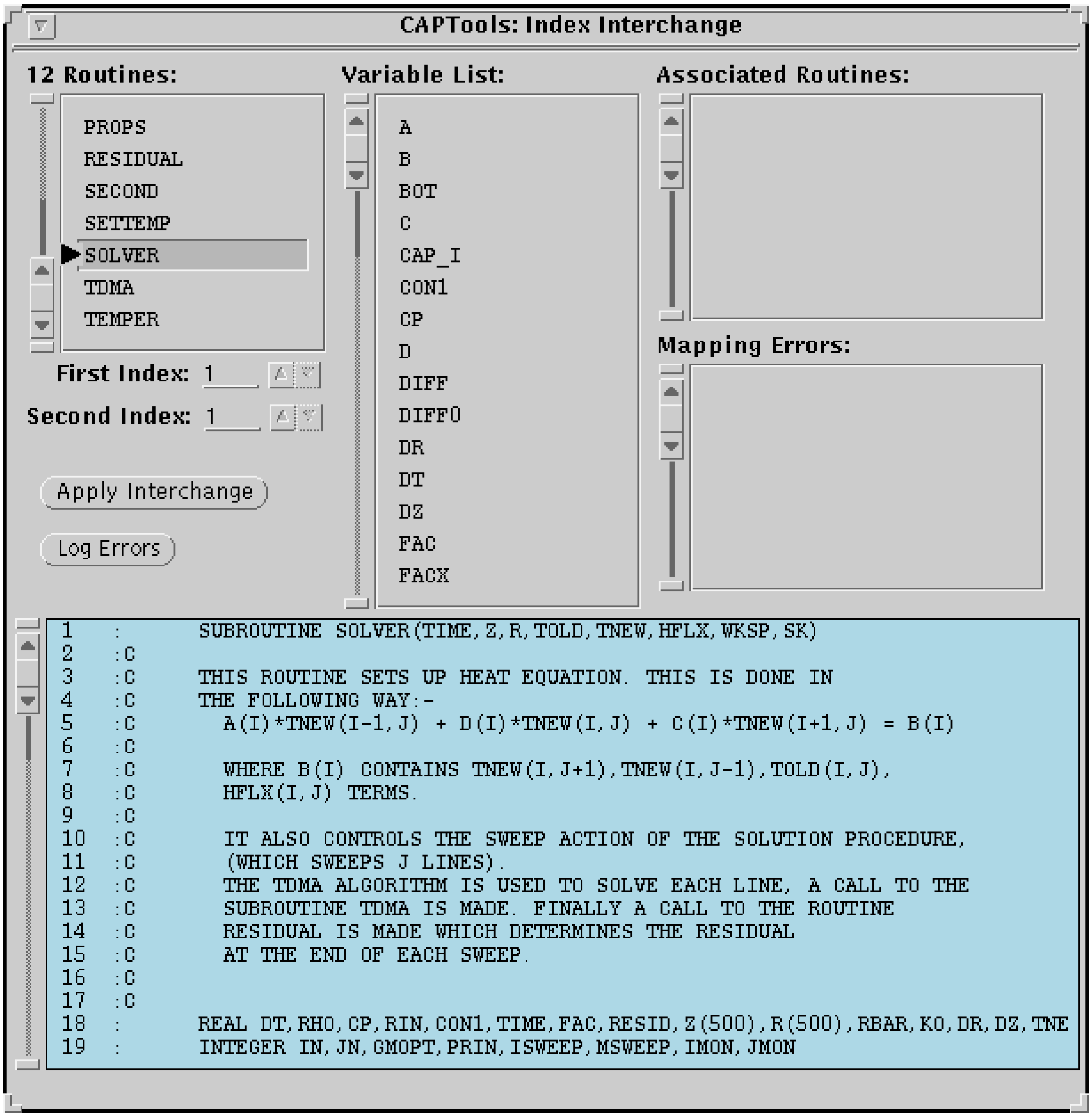
Duplicate Routine Once For Every Call [button]

Copies all the calls to the routine selected in Routines list, and automatically renames the routines, and adjusts the calls selected in the calls scrolling list to reference the new routine.

Duplicate Routine Once For Selected Calls [button]

Copies the selected routine form the Called From Routine list, giving it the new name from the textfield, and adjusts the calls selected in the calls scrolling list to reference the new routine.

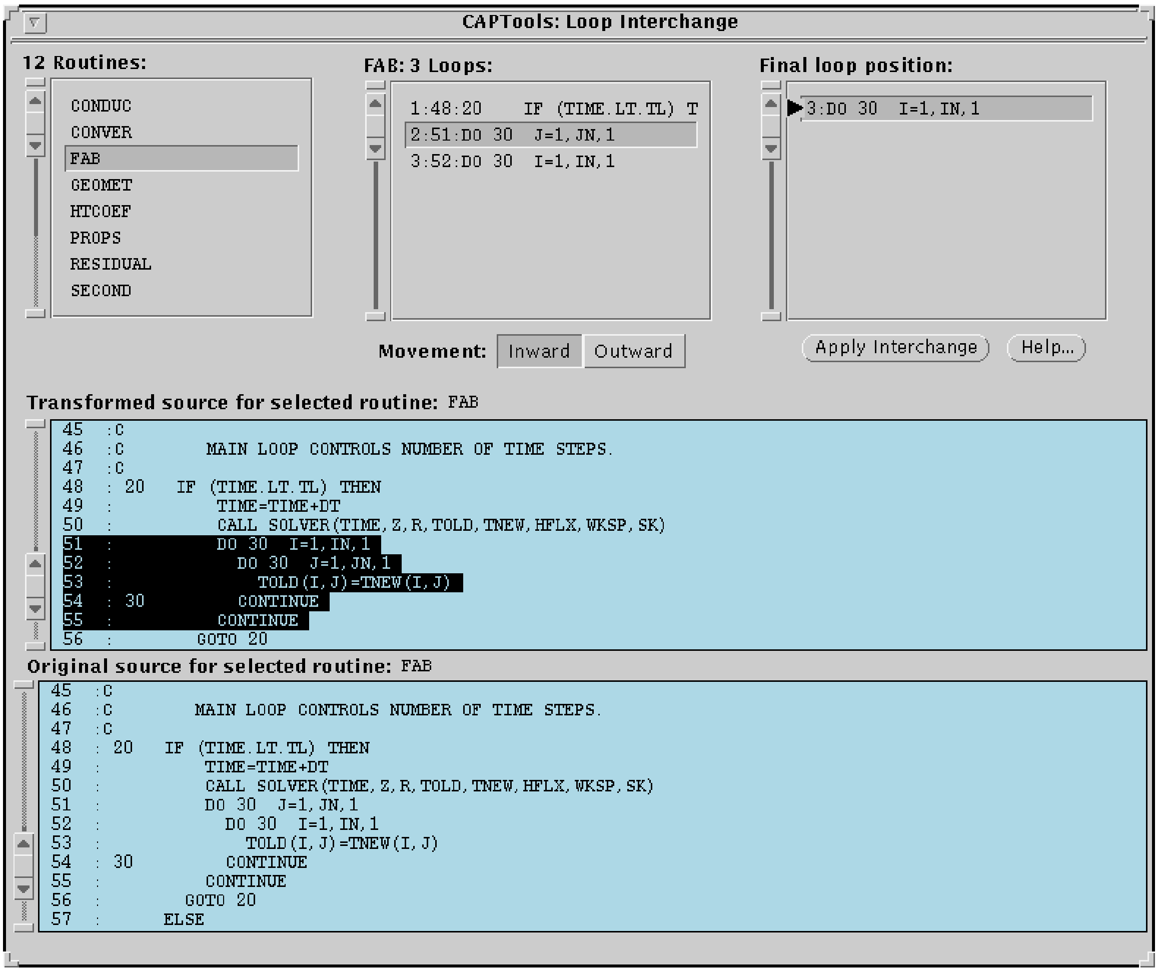
16.17 Transformations - Index Interchange Window



Not available in this release

The Index Interchange... option allows the user to alter indices and, for example, enable contiguous data for communication

16.18 Transformations - Loop Interchange Window



This window provides for the interchange of loops.

Routines [list]

This allows the user to choose the routine in which to apply the loop interchange.

Loops [list]

This allows the user to select the first loop to interchange in the selected routine.

Final Loop Position [list]

This lists loops that can legally be interchanged with the first selected loop.

Movement [setting]

This indicates whether the loop interchange should move the first selected loop **Inward** or **Outward** in the loop nest.

Apply Interchange [button]

This will apply the loop interchange shown in the Transformed source for selected routine window.

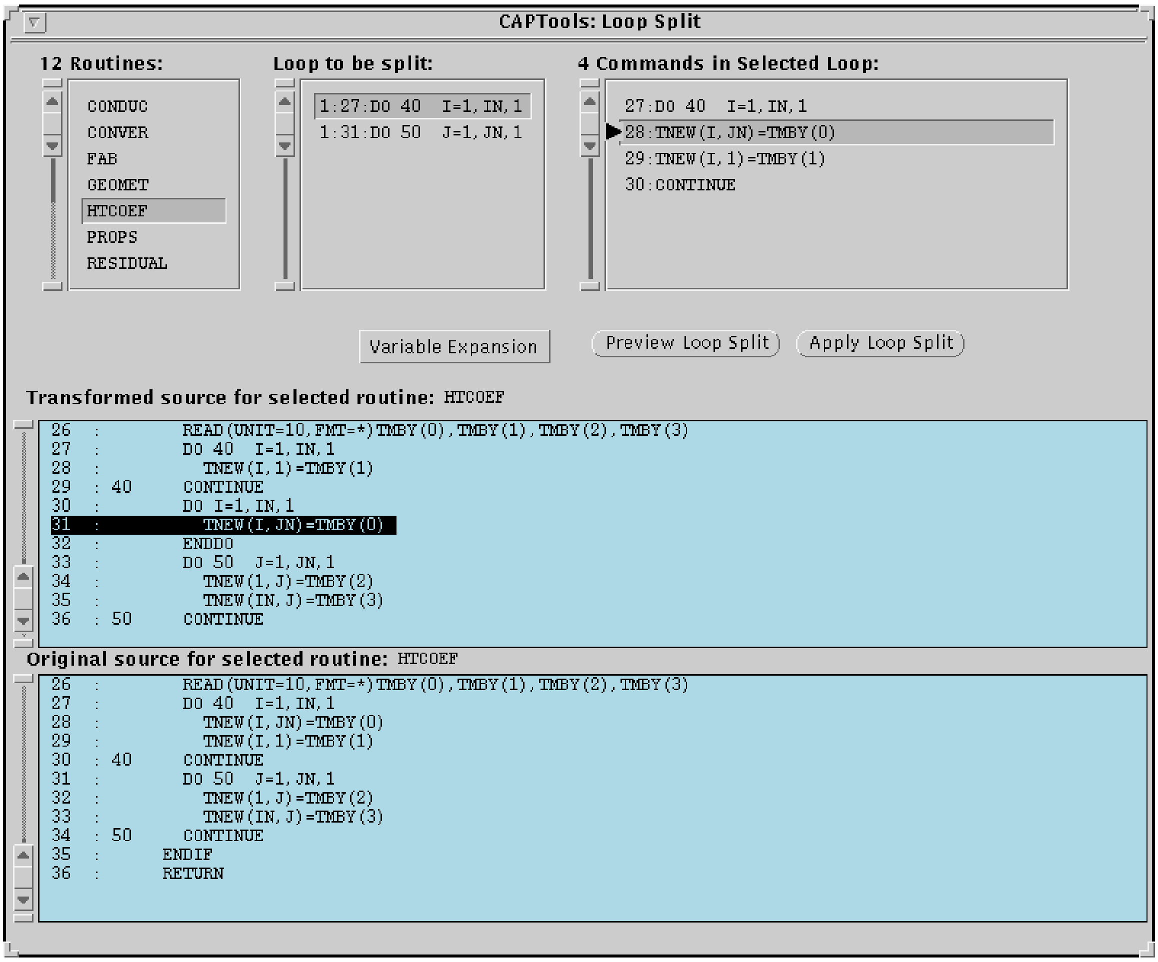
Transformed source for selected routines

Previews the transformation applied to the source code

Original source for selected routine

Shows the original source code before transformation is applied.

16.19 Transformations - Loop Split Window



This window provides for the splitting of loops. This can, for example, transform serialising loop carried dependencies within the original loop into loop independent dependencies between the two loops formed by splitting the original.

Routines [list]

This allows the user to choose the routine in which to apply the loop interchange.

Loop to be split [list]

This allows the user to select the loop to split in the selected routine.

Commands In Selected Loop [list]

This allows selection of the set of statements to be moved from the original loop and placed in the new copy of the loop.

Variable Expansion [setting]

When selected, allows variables to be automatically expanded by an additional index (i.e. scalars become 1D arrays, 2D arrays become 3D arrays) to allow loop carried pseudo dependencies to not prevent splitting.

Preview Loop Split [button]

Allows the transformation effect to be displayed in the **Transformed source** window.

Apply Loop Split [button]

Applies the transformation to the source code.

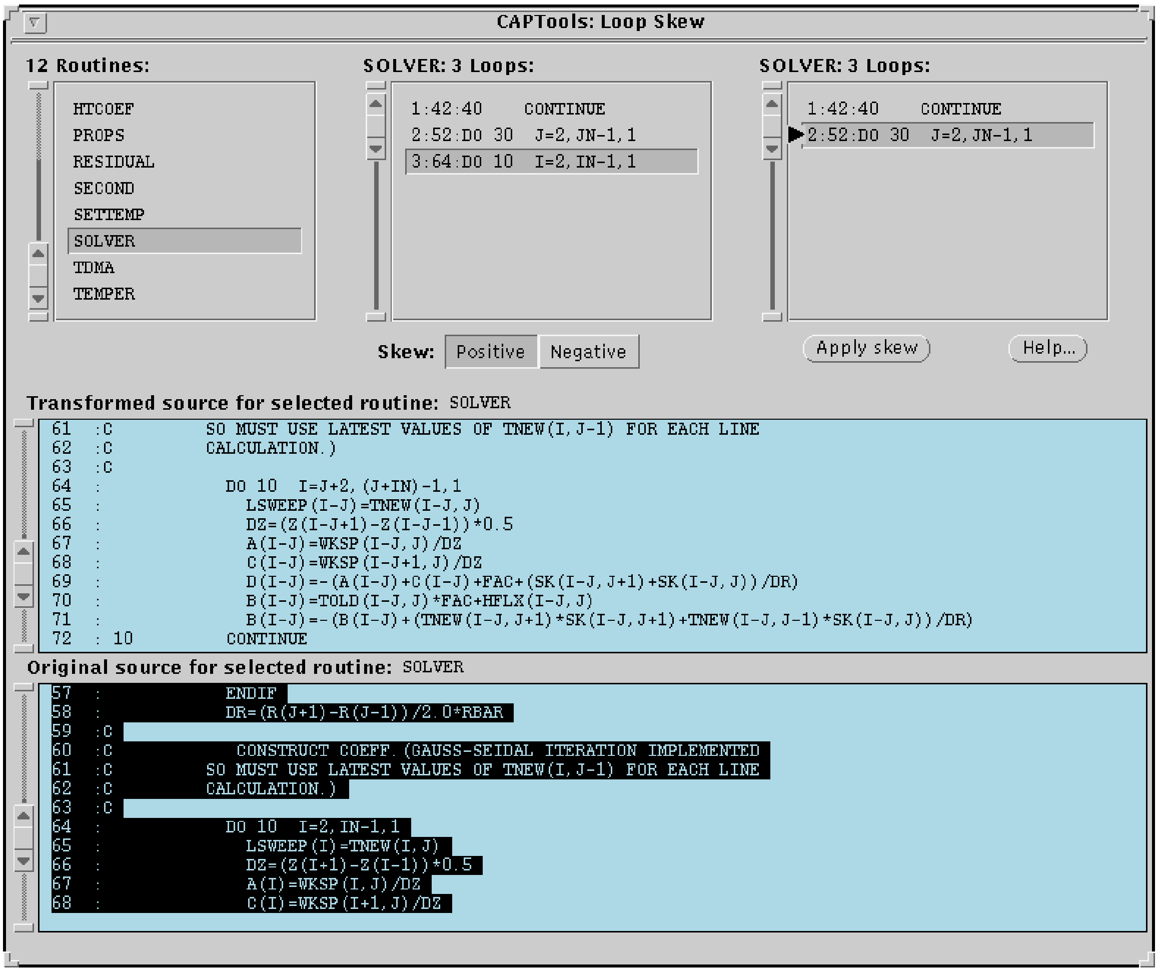
Transformed source [textpane]

Displays the transformed source to allow user inspection prior to applying the transformation to the source code.

Original source [textpane]

Displays the original source for a comparison with the transformed preview source.

16.20 Transformations - Loop Skew Window



This window provides for the skewing of loops to allow one of the transformed loops to execute in parallel.

Routines [list]

This allows the user to choose the routine in which to apply the loop skew.

Loops (left) [list]

This allows the user to select the first loop to skew in the selected routine.

Loops (right) [list]

This lists loops that can legally be skewed with the first selected loop.

Skew [setting]

Indicates whether the skew should be positive or negative.

Apply Skew [button]

Apply the selected loop skew transformation to the source code.

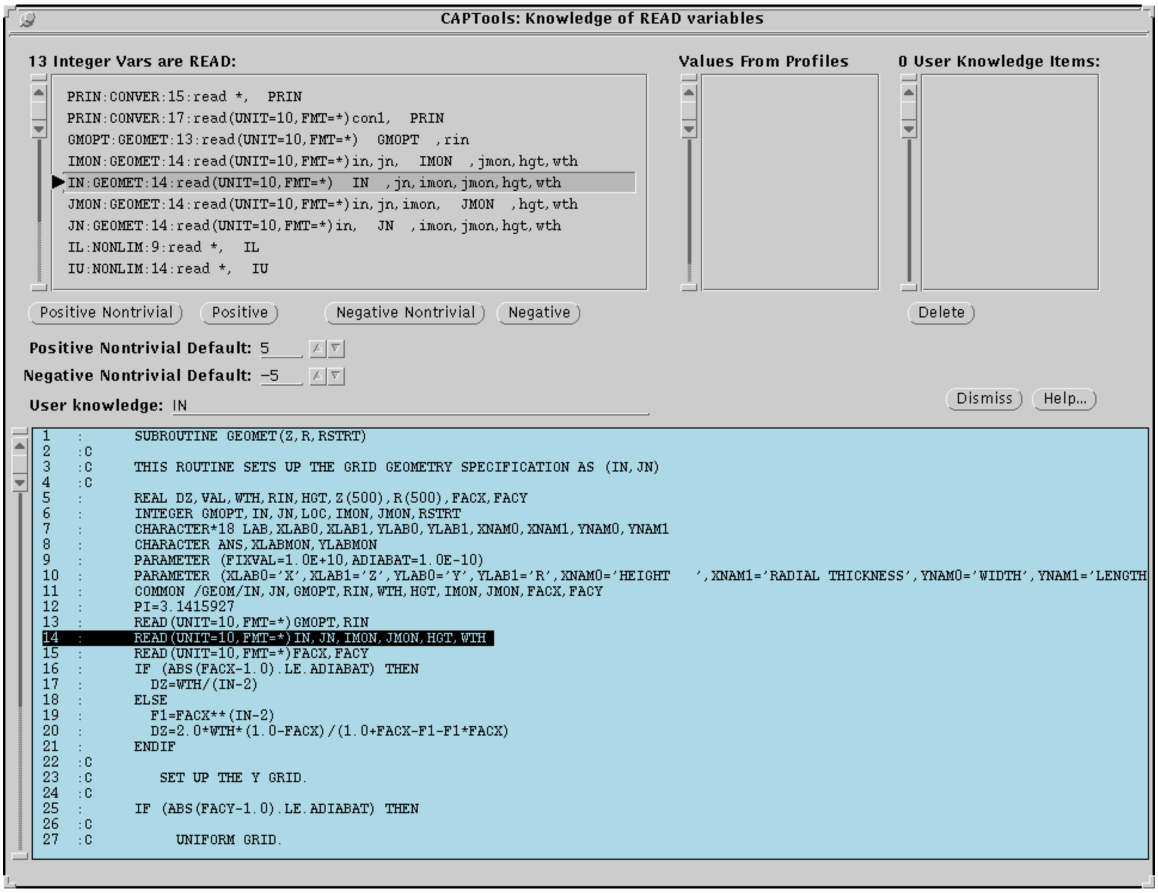
Transformed source [textpane]

Displays the source code that will be generated if the selected skewing is applied.

Original source [textpane]

Displays the original source for a comparison with the transformed preview source.

16.21 READ Knowledge Window



This window allows the user to enter information about variables that are assigned in READ statements in the source code. This information can help in the following dependence analysis. Select a read statement from the list and enter information. This is then entered into the user knowledge base. The textpane shows the relevant statement in context.

READ Statements [list]

This shows all the READ statements in the code being parallelised. Each line relates to a variable in the corresponding READ statement. A selected variable using the left mouse button is highlighted. The statement in the READ statements list is also highlighted in the Textpane, showing the statement in the context of the routine it resides in.

Values From Profiles [list]

(Option not yet available).

Show Profiled Data [button]

(Option not yet available).

Positive [button]

This defines the selected variable as being greater than zero.

Positive Non Trivial [button]

This defines the selected variable as non-trivial as qualified by the inequality shown in the User Knowledge textline. This information is also added to the User Knowledge Items.

Positive Non Trivial Default textfield

This sets the default Positive Non Trivial definition. This is altered using the arrows or can be typed directly.

Negative [button]

This defines the selected variable as being less than zero.

Negative Non Trivial textfield

This defines the selected variable as non-trivial as qualified by the inequality shown in the User Knowledge textfield. This information is also added to the User Knowledge Items.

Negative Non Trivial Default [setting]

This sets the default Negative Non Trivial definition. This is altered using the arrows or can be typed directly.

User Knowledge Items [list]

This shows all the user accumulated knowledge in the form of inequalities. A selected inequality is shown using the left mouse button and is highlighted.

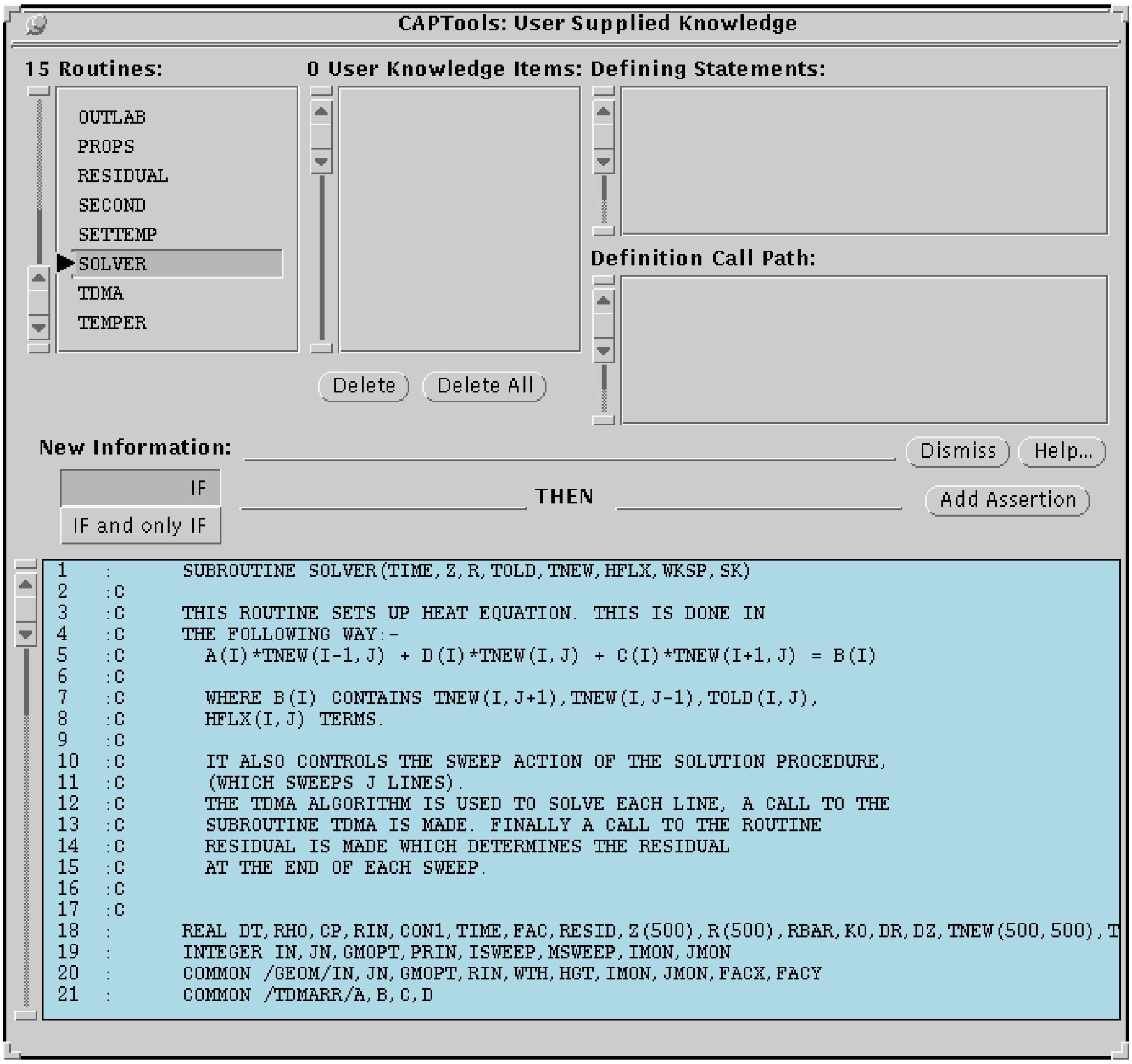
Delete [list]

This allows the user to delete the selected inequality from the User Knowledge Item list.

Information textfield

Displays the selected variable and allows the user to either use the buttons above or type in the new information.

16.22 User Knowledge Window



This window is used to enter knowledge about the code. Select a routine and then enter information about a particular variable. If there is more than one possible definition of the variable then a choice window will appear. Select from this the appropriate defining statement and the information will be added to the user knowledge base.

Routine [list]

This provides a list of all the routines in the code. A selected routine using the left mouse button is highlighted. This shows the source code for the routine in the textpane.

User Knowledge Items [list]

This shows all the accumulated knowledge in the form of inequality expressions. An inequality selected is then highlighted.

Delete [button]

This allows the user to delete the highlighted inequality from theUser Knowledge Items list.

Delete All [window]

This allows the user to delete all the inequalities from the User Knowledge list without the need to select them all.

Defining Statements [list]

This shows the defining statement(s) for the variables present in the selected inequality from the User Knowledge Items list.

Definition Call Path [list]

This shows the definition call path relevant to the variables in theUser Knowledge Items list.

New Information [textfield]

Allows the user to add further knowledge about variables present in the selected routine. This is in the form of an inequality e.g. NI.GT.3, or a set of inequalities e.g. (ELETYP.EQ.3.OR.ELETYP.EQ.5.OR.ELETYP.EQ.9)

IF/IF and only IF [setting]

Determines whether the information added in the **___ THEN___** textfield is to be used in an IF or an IF and only IF condition.

_____ THEN ____ [textfields]

Allows user assertions to be added in the form of **IF** (NK.EQ.2) **THEN** (THREED.EQ..FALSE.) where the user only enters the NK.EQ.2 and THREED.EQ..FALSE. conditions into the appropriate textfield.

Add Assertion [button]

This adds the assertion in the **IF ___ THEN ___** textfields and settings into the knowledge base.

16.23 Symbol Chooser Window

This window shows a list of symbols for a particular routine. Any number of symbols can be selected. The action taken on selecting the apply button will depend on why the window was activated.

Symbol Scrolling List [list]

This allows the user to interrogate variable(s). Multiple selection of variables is allowed. The selected variables appear in the Symbols Textline.

Apply [button]

This accepts all the selected variables.

Reset [button]

This will reset the Symbols Textline and deselect all the items from the Symbols Scrolling List.

16.24 Routine Chooser Window

This window provides a method for choosing a particular routine from a list. The action taken on clicking on the apply button will depend why the window was activated.

Routine Scrolling List [list]

A scrolling list of all the routines in the code for selection. The routine is confirmed by selecting the Apply button.

Apply [button]

This sets the selected routine to be the current routine for the base window.