Generation of Efficient Parallel Code Using Commutative Communications and Iteration Space Alignment.

S.P.Johnson, C.S.Ierotheou, E.W.Evans and M.Cross
Parallel Processing Research Group
University of Greenwich
London SE18 6PF UK

Technical Report PPRG-98-012

Abstract.

The parallelisation of real world application codes needs a wide range of techniques to tackle the variety of software with which they will be presented. This requires specific attention to commonly occurring situations where basic code generation techniques fail to produce efficient parallel code. In particular, the avoidance of serialised loops is crucial if effective parallelism is to be achieved. The recognition of commonly used coding practices and the development of transformations to generate efficient parallel code in such situations is therefore crucial for the parallelisation of real application codes. In this work, a number of the most commonly occurring cases are addressed, with the resultant techniques embedded within the Computer Aided Parallelisation Tools (CAPTools) environment [1].

1 Introduction.

The parallelisation of application codes for distributed memory systems requires efficient parallel code to be generated for all sections of parallel code. Any poor parallelisation can suffer from such severe slowdown that it can result in overall slowdown for an application code, even if the remainder of the code is efficiently parallelised. The overriding requirement of the generated parallel code is that it produces correct results, forcing conservative assumptions to be used whenever uncertainty exists. The exploitation of parallelism is, therefore, secondary to correctness. As a result, it is fairly common for some code sections to be generated that involve very frequent communications or serial execution (and often both) to ensure correctness. Serialisation obviously can have dramatic effects on parallel performance, particularly as processor numbers increase and the frequent communications can incur such drastic startup latency costs that they can easily increase the runtime several fold.

The automated generation of parallel code has been the focus of significant effort for many years. The popular approaches, such as High Performance Fortran (HPF) [7], require directives to be placed within source code to indicate loop parallelism, the nature of variables within such loops (e.g. *shared* or *private*), interprocedural side effects and the details of array decomposition. Even in shared memory parallelisation, data placement can prove vital for high parallel efficiency. In these parallelisations, the user is required to ensure that source code conforms to the requirements of the compiler, requiring a high level of expertise. The parallelisation of large

application codes is therefore a non-trivial task with significant code re-authoring often required.

The Computer Aided Parallelisation Tools (CAPTools) [1,2,3,4,5] are an interactive set of parallelisation tools aimed at parallelising both structured (regular) [3] and unstructured (irregular) [5] mesh based application codes. The basic role of the user is the selection of an array in a routine in the application code on which to base the array decomposition. A comprehensive data partition, with inheritance to other arrays in all routines of the code, is often produced from a single user selection. The remainder of the parallelisation process is performed automatically, although user inspection and interaction is enabled and encouraged. The processes involved in the parallelisation of an application code involve the accurate, symbolic, interprocedural, value based dependence analysis [2], followed by the array partition determination, execution control mask addition and communication insertion [3]. The algorithms used by CAPTools remove the need for user directives to be placed in the application code, and since they have been developed to parallelise complex application codes, the need for code re-authoring is almost entirely removed.

The data partition, execution control masks and communications are calculated and generated based upon symbolic variables held on each processor in the Single Program, Multiple Data (SPMD) parallel code that represent the range of the relate arrays that is owned, and therefore assigned, by that processor. These partition range variables are assigned values at runtime based upon the range of the related arrays actually accessed in the execution of the application code, along with the requested processor topology, allowing the same parallel code to be used for a range of mesh sizes and processor topologies. Each communication is migrated from the statement that issued the request for non-owned data, to pass out of surrounding loops and through routine boundaries to minimise the frequency of its instigation (to reduce startup latency overheads) and to allow merger with other communications.

One of the crucial factors in a CAPTools parallelisation is the use of interaction with the code paralleliser to obtain often vital information that is not explicitly stated in the source code of the application. Such information typically refers to minimum bounds on the dimensions of the problem mesh for any execution of the code (e.g. the number of cells in one dimension must be positive) and is expressed in terms familiar to the code paralleliser. To allow effective user interaction throughout the parallelisation process, it is essential to avoid drastically altering the source code so that it may still be recognisable to the code paralleliser. This contrasts with many other parallelisation systems, such as High Performance Fortran (HPF) compilers, since they do not exploit any user interaction based upon the generated code. As a result, the application of transformations that are often automatically applied in such systems may not be desirable in interactive parallelisations since the appearance of the code can be significantly altered. Instead, they are applied under user direction with a previewing facility to allow the user to ensure recognition of the transformed code prior to the transformation acceptance.

2 CAPTools Parallel Code Generation.

The communication generation algorithms used in CAPTools take information from the partition definition, execution control masks and the dependence graph to determine the need

and optimal location of communications. The data partition definition defines how arrays are decomposed across the processor topology in terms of array index components. The execution control masks are conditional expressions that determine whether a processor will execute a particular instance of the controlled statement. The requirement of a communication for a partitioned array usage is then determined by a comparison between the partitioned component of the usage and the execution control mask. For arrays that are not partitioned and scalar variables, communication requests are detected by comparing the masking and array index details of the usage of the variable with the masking and array index details of the related assignments, as identified using the dependence graph. These communication requests are then migrate upwards in the control flow graph to pass out of loops and through routine boundaries. This migration is impeded by dependencies of the usage issuing the request and the related execution control mask. Loop carried dependencies (i.e. between iterations of a surrounding loop) blocks migration at the loop head (e.g. a **DO** loop statement), keeping the communication within the loop.

The existing code generation techniques used in CAPTools [3] detect some classes of serialised loops, where global values are calculated by the simple combination of local processor values. These calculations relate to, for example, summations, products, maxima and minima calculations, all referred to as reduction operations in MPI. Such cases can be detected in a code and transformed into efficient parallel code as shown in Figure 1. In this example, a summation of the contents of array **A** is performed. In the parallel SPMD version, this code is transformed by partitioning the array **A** in index 2 and index 3, to produce a 2 dimensional decomposition. The loop limits of the **K** and **J** loops are adjusted to only process one partitioned segment of array **A** on each processor. The variables **CAP1_L** and **CAP1_H** represent the range of array **A** owned on the executing processor in the third index and **CAP2_L** and **CAP2_H** represent the owned range of index 2 [3]. The **MAX** and **MIN** functions used in the loop limits ensure that the original loop limits are honoured in the parallel execution.

Using the communication generation algorithms discussed earlier, the loop carried dependencies of scalar variable **SUM** will cause communication requests to be issued. Since loop carried dependencies relate the assignment to the usage, the processor owning the assigned value is defined by a different instance of the associated loop (either **K** or **J**) to the usage, and may therefore be on a different processor. The loop carried dependence also causes the communication to be kept inside the carrier loop, although pipeline communications would be generated immediately surrounding that loop, if possible.

This code generation can be avoided if the nature of the operation concerned (i.e. the summation) can be detected. Such detection implies that the precise ordering of the operation does not affect the final result (apart from round-off errors). This allows a more efficient code generation to be used.

On exiting these loops, each processor holds the summed value for their owned portion of array **A**. The call to the CAPLib library [6] routine **CAP_COMMUTATIVE** takes as input the processors local contribution and returns the global value by internally summing all local contributions and performing the required communications. The parameter **CAP_ADD** is a function in the CAPLib library that performs the simple operation required to calculate the

global value from the local contributions.

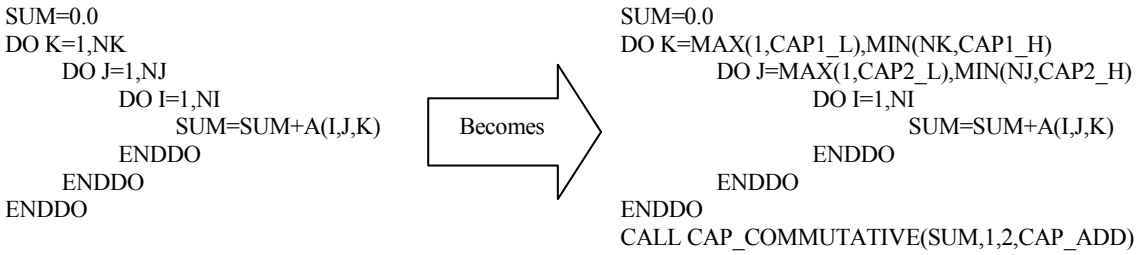

Figure 1 Code generation for a simple summation operation.

With this code generation, the loop carried dependencies of variable **SUM** are, ineffect, satisfied. The dependencies are therefore marked to avoid them causing communication requests in subsequent communication generation.

Many shared memory parallelising compilers allow directives in the serial code to indicate the existence of reduction operations. The High Performance Fortran (HPF) language [7] has similar directives to aid the compiler in exploiting these simple cases. There, however, a wide range of similar cases, such as some of those in this paper, that contain complications and therefore cannot be handled by this simple transformation alone.

In this work, commonly occurring cases where serialised loops would be generated involving frequent communications as found in real application codes, are investigated to identify techniques that result in efficient parallel code. These techniques must also be automatable so that they can be embedded within the code generation algorithms of parallelisation tools. These cases involve more complex reduction style operations, searches, error checking, algorithmic use of I/O conditions and other cases where the serial operation that would otherwise be employed can be avoided. The examples used in the following sections are all taken from real application codes.

3 Complex Commutative Operations.

The detection and code generation for commutative (reduction) operations employed by previous versions of CAPTools, although capable of handling many cases, could not cope with some commonly occurring situations. The test for a commutative operation basically involves a loose pattern match with the set of cases implemented in the code generation algorithm. Any code that does not fit into those cases is therefore generated as a serial loop, so maximisation of the range of recognised cases is essential.

3.1 Multi-Statement and Absolute Value Commutative Operations.

Simple examples of commutative operations involve multiple statements and absolute values being used in the calculation as shown in Figure 2. The first example in Figure 2 involves four statements contributing to the variable **SUM** where each statement adheres to the additive

requirement in this summation operation so that this set of statements form the commutative operation. The CAPLib functions used in the calls to **CAP_COMMUTATIVE** in the absolute value example in Figure 2 take account of the precise nature of the related comparison used in the maxima calculations. The variable **MAXI** stores the value of the element of array **A** with the highest absolute value, whilst the variable **PMAXI** stores the absolute value of the highest entry in array **A**. The functions **CAP_AANMAX** and **CAP_ANAMAX** relate to the use of absolute values in the components of the maximum calculation. The 'A' or 'N' characters in the function name indicate if the absolute value is used for the key components of the commutative operation. The first character relates to the variable representing the maximum in the conditional (i.e. **MAXI** in statement S_1), the second to whether the absolute value is used for the new value in the conditional (i.e. **A(I,J,K)** in S_1) and the third to whether the absolute value is used in the right hand side of the maxima assignment (i.e. **A(I,J,K)** in S_2). These functions in the CAPLib library then precisely follow the operation of the commutative in the original code, therefore ensuring the same result is produced.

```
SUM=0.0
DO K=MAX(1,CAP1_L),MIN(NK,CAP1_H)
      DO J=MAX(1,CAP2_L),MIN(NJ,CAP2_H)
            DO I=1,NI
                  SUM=SUM+A(I,J,K)
                  SUM=B(I,J,K)+SUM
                  IF (C(I,J,K).GT.0) THEN
                        SUM=SUM-C(I,J,K)
                  ENDIF
            ENDDO
            SUM=SUM+D(J,K)
      ENDDO
ENDDO
CALL CAP_COMMUTATIVE(SUM,1,2,CAP_ADD)
```

```
MAXI=A(1,1,1)
PMAXI=0
DO K=MAX(1,CAP1_L),MIN(NK,CAP1_H)
      DO J=MAX(1,CAP2_L),MIN(NJ,CAP2_H)
            DO I=1,NI
S1              IF (ABS(A(I,J,K)).GT.ABS(MAXI)) THEN
S2                    MAXI=A(I,J,K)
                  ENDIF
                  IF (ABS(A(I,J,K)).GT.PMAXI) THEN
                        PMAXI=ABS(A(I,J,K))
                  ENDIF
            ENDDO
      ENDDO
ENDDO
CALL    CAP_COMMUTATIVE(MAXI,1,2,CAP_AANMAX)
CALL CAP_COMMUTATIVE(PMAXI,1,2,CAP_ANAMAX)
```

Figure 2 Multiple statement commutative and absolute value commutative code fragments

3.2 Commutative Operations with Subsidiary Values.

There are occasions when not only is a maximum or minimum value required, but also, for example, the actual location within an array of that extreme value. The original commutative operation detection algorithm would not permit a transformation when intermediate values of the commutative variable were used within the loops concerned. In the example in Figure 3, the variables **JT**, **KT** and **LT** are assigned only when **SIGABC** is greater than **SIGMAX**. These assignments therefore require intermediate values of the global **SIGMAX** variable, making the correct assignments to **JT**, **KT** and **LT** impossible in parallel execution with the calculation of local maximums on every processor.

```
        JT=1
        KT=1
        LT=1
        SIGMAX=0.
L1      DO 10  L=2,LM,1
              DO 10  K=KLOW,KUP,1
                    DO 10  J=2,JM,1
                          SIGA = ...
                          SIGB = ...
                          SIGC = ...
                          SIGABC = AMAX1(SIGA,SIGB,SIGC)
S1                        IF (SIGABC-SIGMAX.LT.0) THEN
                                GOTO 10
S2                        ELSEIF (SIGABC-SIGMAX.GT.0) THEN
                                GOTO 38
                          ELSE
                                GOTO 10
                          ENDIF
38                        JT=J
                          KT=K
                          LT=L
                          SIGMAX=SIGABC
10      CONTINUE
```

Figure 3 Code example with subsidiary variables dependent on maximum value.

Variables **JT**, **KT**, and **LT** are referred to as subsidiary variables since they represent the location of the identified maximum within the three dimensional mesh that the calculations are based upon and are not directly involved in the calculation of the maximum value. Although their computation is controlled by the control dependencies on the **IF** statement S_1 and the **ELSEIF** statement at S_2, the controlled operations assign to variables, killing all previous assignments of those variables. No use of intermediate values of variables **JT, KT** and **LT** is made, allowing a post-loop calculation of those variables a possibility. As a result, it is legal to use local calculations on each processor in a parallel version, with the local values of **SIGMAX**, **JT**, **KT** and **LT** being used to calculate the global values after the loops have been completed. Figure 4 shows the parallel version of the code in Figure 3 where the call to CAPLib routine **CAP_COMMUPARENT** is used to identify the global maximum from the local maximums on each processor, and the three calls to **CAP_COMMUCHILD** evaluate the global values of **JT**, **KT** and **LT**.

```
L1      DO 10  L=MAX(2,CAP_L),MIN(LM,CAP_H),1
              . . . .
              . . . .
10      CONTINUE
        CALL CAP_COMMUPARENT(SIGMAX,2,CAP_RMAX,.TRUE.)
        CALL CAP_COMMUCHILD(JT,1,1)
        CALL CAP_COMMUCHILD(KT,1,1)
        CALL CAP_COMMUCHILD(LT,1,1)
```

Figure 4 Loop partition and communications for parallel version of the example from Figure 3.

The **CAP_COMMUPARENT** routine performs the same function as the **CAP_COMMUTATIVE** routine discussed earlier except that it internally stores the processor

that provided the global value. The **CAP_COMMUCHILD** routines use this internally stored processor identification number to broadcast the correct local values to every processor. In order to obtain identical results to the original serial code, it is vital to correctly handle repeated values in the maximum comparisons. The final parameter in the call to **CAP_COMMUPARENT** in Figure 4 indicates, in the event of two processors having identical local maximums, which processor's local values of the subsidiary values should be used. Two checks are required to determine if the lower or higher numbered processor should provide the global value. Firstly, the direction of iterations across the processor topology must be determined. In the example of Figure 4, the partitioned loop L_1 iterates by increasing the values of counter variable **L**, with the partition of the loop is based on positive **L**. Secondly, the determination of whether a repeated value will cause a repeated assignment to the subsidiary variables must be made. In Figure 4, the **.GT.** comparison in statement S_2 shows that if a second identical maximum value is encountered (i.e. a value of **SIGABC** that is the same as the current **SIGMAX**) then the conditional will produce a false result and the assignments will not be performed. This indicates that the first of the identical values encountered in the loop provides the location in the serial code. Since the iteration of the partitioned loop is in a positive direction in relation to the processor topology, the local values on the lower numbered processors should therefore be used for the global value, setting the **.TRUE.** parameter as an input to routine **CAP_COMMUPARENT**.

4 Efficient Parallelisation of Searches and Error Exits.

In general, the parallel version of a code should not perform iterations of loops that are not performed in serial. There are, however, special instances where efficient parallelisation requires extra parallel operations. Common occurrences of such cases are in the implementation of linear searches and when algorithmic errors are detected within loops and the relevant loops are exited.

Often within a code there are loops which contain exits, either in the form of **GOTO** statements or, in the case of Fortran 90 for example, where the **EXIT** statement is used. In a serial code, this poses no problem to correct and efficient operation. In a parallel, however, code of this type can present difficulties. Consider the serial code in Figure 5a.

```
      DO I=1,N

         IF (A(I).EQ.X) THEN
            GOTO 10
         ENDIF
      ENDDO
10    CONTINUE
       .
```

```
      DO I=1,N
            CALL CAP_BROADCAST(A(I),..)
            IF (A(I).EQ.X) THEN
                  GOTO 10
            ENDIF
      ENDDO
10    CONTINUE
      .
```

Figure 5a) Serial search code. b) Crude Parallel Code.

The serial code (Figure 5a) simply searches in the loop with **I** running from **1** to **N** until an entry **A(I)** is found that has the same value as variable **X**. The code then exits the loop where, typically, the value of **I** is required when a match has been found. A crude parallelisation of this code when array **A** is partitioned (Figure 5b), based purely on the data dependencies existing

within the loop, would be forced to place a call to broadcast the values of A(I) to all processors and execute the loop in serial (where **CAP_BROADCAST** is the relevant communication call in CAPLib).

This loop may, however, operate in parallel by allowing each processor to perform a search of the locally owned section of array **A** and then combining the local results of the search on each processor to obtain the correct global result, as obtained by the serial code. This will require many processors to perform iterations of the **DO** loop which are not executed in the original serial code since the loop exit would previously have been taken. An examination of the code in Figure 5a, however, reveals that the code does not alter any data during the iterations of the loop, so the extra iterations performed in the parallel version will not corrupt any required data. The parallel version of this code is shown in Figure 6 where, as with the subsidiary variable commutative operations discussed in section 3, the **CAP_COMMUPARENT** CAPLib library routine is used. In this case, the call to **CAP_COMMUPARENT** is used to identify the 'first' processor to have taken the exit branch, and therefore provide the correct result as compared to the serial version. As in section 3, the direction of the loop iterations across the processor topology is required to determine whether the lowest or the highest numbered processor that takes an exit branch is the correct one to use. In Figure 6, the last parameter in the call to **CAP_COMMUPARENT** indicates that the lowest numbered processor should be considered.

```
      CAP_JUMP=0
      DO I= MAX(1,CAP_L), MIN(N,CAP_H), 1
            IF (A(I).EQ.X) THEN
                  CAP_JUMP=1
                  GOTO 20
            ENDIF
      ENDDO
20    CONTINUE
      CALL CAP_COMMUPARENT(CAP_JUMP,1,1,CAP_NONZERO,.TRUE.)
      IF (CAP_JUMP.NE.0) THEN
            CALL CAP_COMMUCHILD(I,1,1)
      ENDIF
      GOTO (10) CAP_JUMP
10    CONTINUE
       .
       .
```

Figure 6 Parallel search code using local iteration space.

The parallel code uses a CAPTools variable **CAP_JUMP** to determine if an exit branch has been taken. This variable is initialised to zero, then, if any processor meets the criteria of **A(I)** equalling **X,** the value of the variable of **CAP_JUMP** is set to one and the loop is exited to the statement labelled with the newly created label 40, exiting the loop. The CAPLib function **CAP_NONZERO** detects if either value compared using the function is non-zero, where, if both are non-zero, the flag input into the call to **CAP_COMMUPARENT** is used to indicate which non-zero value should be kept. If the global value of **CAP_JUMP** is non-zero, there has been a loop exit on at least one processor so the value of **I** from that processor is also broadcast using the **CAP_COMMUCHILD** CAPLib routine. The computed **GOTO** statement then branches to the original statement with the label **10**, as in the serial code, if the exit branch has

been taken. The computed **GOTO** is used to allow multiple loop exits to be handled in the code generation, where any second loop exit would set the **CAP_JUMP** variable to be 2 etc. The value of the global **CAP_JUMP** variable then indicates which exit would have been taken and the jump to the appropriate label is made. A global **CAP_JUMP** value of zero indicates that no processor took any loop exit, therefore, any calls to **CAP_COMMUCHILD** should not be made.

To generate this code automatically from within CAPTools requires the determination of whether this transformation of search or loop exit code is legal. It is important to ensure that no data is changed in an iteration that would not be executed in serial, unless this data is not used outside the scope the loop(s) concerned. This may be investigated by checking for dependencies inside the loop related to the code that would be not originally have executed. Consider the code in Figure 7a.

```
            DO I =1,N
                IF (A(I).EQ.X) GOTO 10
S1              A(I) = A(I) + . . .
            ENDDO
        10  CONTINUE
            DO J=1,N
S2              A(J) = A(J)  + . .
            ENDDO.
```

```
            DO I=1,N
                X = X + B(I)
                IF (A(I).EQ.X) GOTO 10
            ENDDO
        10  PRINT*, I
```

Figure 7 a) Transformation is illegal b) Transformation is legal.

In Figure 7a, if extra loop iterations are performed, some values of array **A** will be changed by statement S_1, destroying the values that would be held in the serial version. Any subsequent use of array **A** may, therefore, involve invalid values, producing incorrect results. A check of all dependencies on statement S_1 must be made to determine if any usage exists after the loop with the transformation being illegal if any are detected. In Figure 7a, the dependence from the assignment to array **A** in statement S_1 to the usage in statement S_2 invalidates the use of the transformation. Figure 7b, however, although changing variable **X** in any extra iterations performed in the parallel version, does not prohibit the use of the transformation since no external usage of **X** exists.

A common example of where a loop exit exists, but the extra iterations are performed in the transformed parallel version do not corrupt vital data, is when the loop exit is caused by an error being detected. These error exit branches typically report the error identified, using an I/O statement, and then terminate the execution of the code, therefore not using any of the data items that may have become invalid. This allows the transformation to the efficient parallel version, maintaining the error checking quality of the serial code and the reporting of any problems to the code user. Multiple exits from a loop are correctly handled using the same technique as used for searches (see Figure 6), with each exit assigning a unique number to the **CAP_JUMP** variable, allowing the correct exit to be taken.

Such cases often occur interprocedurally, for example, when an error is detected in a called routine, forcing the check for the usage of corrupt data to follow the control flow path of the error exit into all caller routines. Consider the example in Figure 8 where an error check is

performed at statement S_3 in routine **SUB** to avoid a division by zero in statement S_4.

```
                PROGRAM  MAIN
                ERROR=.FALSE.
L1              DO ISTEP=1,NSTEP
L2                    DO J=1,NJ
S1                          CALL SUB(A(1,J),B(1,J),ERROR)
S2                          IF (ERROR) GOTO 20
                      ENDDO
                ENDDO
          20    STOP
                END
```

```
                SUBROUTINE SUB(A,B,ERROR)
L3              DO I=1,N
                      . . . .
S3                    IF (ABS(A(I)).LE.1E-20) GOTO 10
S4                    B(I) = 1.0 / A(I)
                ENDDO
                . . . .
                RETURN
S5        10    ERROR = .TRUE.
                RETURN
                END
```

Figure 8 Interprocedural example where error checking transformation is legal.

In Figure 8, three loops contain error exits. If the loop L_2 was partitioned for parallel execution, the error exit at statement S_2 could be processed as in the earlier examples. If, however, the loop L_3 in routine **SUB** was partitioned, the test for use of the invalid values in array **B** that would be assigned in statement S_4 must also consider the control flow and dependencies in the calling routine. The interprocedural dependence analysis used in CAPTools sets dependencies on call statements based upon the usage and assignment of data within those calls. This allows the test for illegal uses of corrupt data in array **B** to check dependencies of the call to routine **SUB** at statement S_1. The value of the variable **ERROR** is also traced interprocedurally, based upon the control flow within routine **SUB,** so that the assignment at statement S_5 is known to have executed and the value of **ERROR** is therefore **.TRUE.**. This allows only dependencies of statement S_1 in the control flow path of the exit branch in statement S_2 to be considered in the legality check, allowing the transformation to be applied.

5 Correct Handling of Input/Output Exceptions in Parallel Execution.

A similar case to that in section 4 occurs with exception handling in I/O statements. The CAPTools generated parallel code currently performs all I/O operations on a master processor only (although the master is treated in the same way as all other processors in the rest of the parallel code). If an error has occurred, or the end of a file is detected during an I/O operation on the master processor, then the other processors must also be informed. Consider the code in Figure 9 where the contents of a file are read until an exception occurs. The variable **CAP_PROCNUM** holds the unique identifying number of this processor, where the master processor has an ID of 1.

```
      IF (CAP_PROCNUM.EQ.1) OPEN(UNIT=IFILE,ERR=10,FILE=FILEFILE,STATUS='OLD',
     &                           FORM='UNFORMATTED')
      I=1
5     IF (CAP_PROCNUM.EQ.1) READ(UNIT=IFILE,END=20,ERR=10) A(I)
      I=I+1
      GOTO 5
20    ... Rest of code ...
      RETURN
10    PRINT*,'Error in file'
      STOP
```

Figure 9 Serial I/O exception dependent code.

In Figure 9, the serial code opens a file and reads data from that file. If there is an error while opening or reading that file then there is an error exit to the statement labelled at 10. In the serial execution this obviously does not pose any problems. However, in parallel if there is an error then only processor 1 will know of this error. It is therefore necessary to inform the other processors. Similarly, the exit of the loop with the label 5 is dependent on the end of file exception causing a jump to label 20. In order for other processors to operate correctly, the end of file detection must be conveyed to all other processors so they can exit the loop in the same iteration. Correct parallel code can be achieved using a similar technique to that used for linear searches, generating parallel code as in Figure 10.

```
      CAP_JUMP=0
      IF (CAP_PROCNUM.EQ.1) OPEN(UNIT=IFILE,ERR=150,FILE=FILEFILE,STATUS='OLD',
     &                                    FORM='UNFORMATTED')
      CAP_JUMP=1
150   CAP_JUMP=CAP_JUMP+1
      CALL CAP_MBROADCAST(CAP_JUMP,1,1)
      GOTO (10) CAP_JUMP
      I=1
      CAP_JUMP=0
5     IF (CAP_PROCNUM.EQ.1) READ(UNIT=IFILE,END= 170,ERR=160) A(I)
      I=I+1
      GOTO 5
      CAP_JUMP=1
170   CAP_JUMP=CAP_JUMP+1
160   CAP_JUMP=CAP_JUMP+1
      CALL CAP_MBROADCAST(CAP_JUMP,1,1)
      GOTO (10,20) CAP_JUMP
20    ... Rest of code ...
      RETURN
10    PRINT*,'Error in file'
      STOP
```

Figure 10 Parallel I/O exception based code.

The value of the variable **CAP_JUMP** is initially set to zero prior to the **OPEN** statement. If there is no error opening the file then the value of **CAP_JUMP** is re-initialised to one immediately after the **OPEN** statement, followed immediately by a statement which increments the value by one. The value will then be 2, which is broadcast to all other processors (using the master broadcast routine **CAP_MBROADCAST** from the CAPLib library). The computed **GOTO** will then not jump to the label and execution will continue from the next statement, which allows the reading of data from the file. If, however, an error was detected opening the file, then the error exit jumps to the statement labelled 150 and the value of **CAP_JUMP** is incremented to be 1, which is then broadcast to all processors. The computed **GOTO** will therefore jump to the statement at label 10, skipping the code section that reads from the file, printing out an error message.

The code generated to satisfy the exception handling in the **READ** statement operates in a similar way to that for the **OPEN**. The values of **CAP_JUMP** at the computed **GOTO** statement may be 1, indicating a jump to label 10 is required since an error was detected, or 2 indicating that an end of file was encountered and the code should continue execution from statement 20. Obviously, the non-exception condition which would produce a value of 3 for

CAP_JUMP cannot be achieved since error or end of file exceptions are the only way that loop formed by the label 5 is exited.

6 Avoiding Serial Loops using Iteration Space Alignment.

The automatic parallelisation strategy used in CAPTools generates efficient parallel code for a wide range of application codes [3,4]. There are, however, many common instances where these techniques alone fail to generate efficient parallel code, particularly when communications are placed within loop nests. Such cases are due to the worst case assumptions taken by the code generation algorithms to ensure correct code is produced. In the following section, a range of such cases are investigated where an efficient solution to this code generation problem is to identify the nature of synchronisation of loop iterations across the processor topology. This allows transformations to be used that enable communications that are performed frequently within loop nests to be performed only once.

6.1 Independent, Local, Global and Synchronous Iteration Space.

The CAPTools code generation when communications are left inside partitioned loops is based upon synchronous iteration spaces where every processor performs all loop iterations with all processors in the same iteration throughout. This enables simple execution control masking based upon the loop iteration counter with computations being performed in an identical order as in the original serial code (i.e. no parallelism). The communications represent a significant execution time overhead in ensuring data is available as required with most processors idle and awaiting communication for the majority of the loop execution time (as shown in Figure 11). This invariably produces slowdown in the parallel execution. In some cases, loop restructuring transformations, such as loop splitting, can enable such communications to be removed from some loops. These transformations, as discussed in section 1, may dramatically alter the appearance of the generated code and therefore hinder subsequent user interaction and manual optimisation of the parallel code. It is therefore desirable to find alternative, less intrusive techniques whenever possible, even when code restructuring is legal.

The alternative to the synchronous mode of partitioned loop iterations that has previously been used in CAPTools generated code is fully independent iterations, where no communications exist within the loop and each processor performs it's iteration completely independent of all other processors. Here, we consider two further alternative modes of iteration space execution that can be used to significantly reduce communication overheads and often allow parallelism to be exploited.

Local iteration space is where every processor executes its locally masked set of iterations at the same time, with a communication performed at the same point in each processor's local iteration space. This allows parallelism to be exploited between loop iterations and requires a single communication on each processor, which is performed at the same time as those on all other processors.

Global iteration space is where each processor executes its locally masked set of iterations only after all processors 'earlier' in the pipeline have completed their iterations. The communications are implemented as a receive immediately before the loop head and a send immediately on exiting the loop. The direction of the pipeline (i.e. whether the processors to the LEFT or to the RIGHT are earlier than this processor) is determined by examining the execution control masks and indices of the array references involved in the communication. Although no parallelism is exploited by between iterations of this loop, parallelism from outer loops can be exploited in this pipeline execution style [4,8].

Figure 11 shows global and local iteration spaces for three processors with communications indicating the synchronisation points. For the global iteration space, processor 2 can only begin the calculation of its iteration set after receiving a communication from processor 1 after it has completed its iterations. For the local iteration space, all processors execute independently until some point in their local iteration spaces when they synchronise. Clearly, the local iteration space is more efficient than the global iteration space, however, both are far superior to the synchronous iteration space versions of the loops.

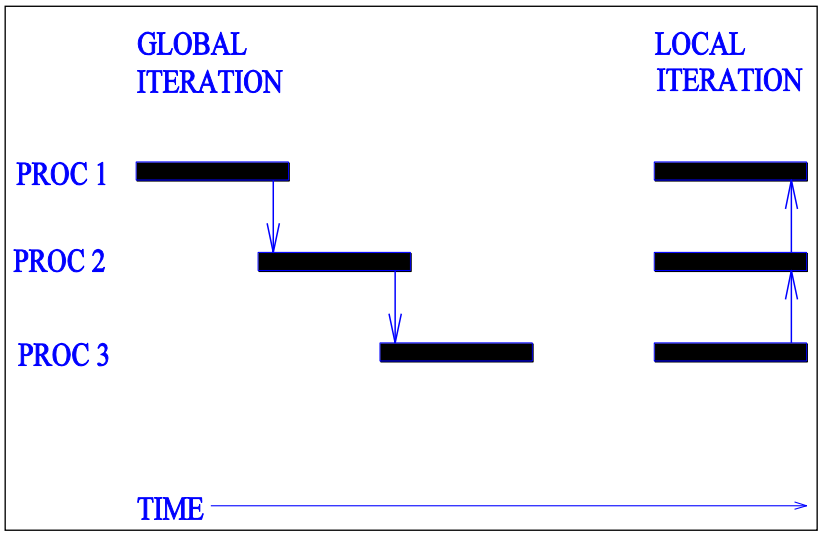

Figure 11 Execution of loops in GLOBAL and LOCAL iteration space modes.

A common cause of synchronous loops in a CAPTools parallelisation is the miss-alignment of execution control masks [9,10] within a loop. The driving force behind the execution control masking algorithm used in CAPTools [3] is the owner computes rule where all assignments to partitioned arrays must be performed only on the owning processor. Related statements involving assignments to scalars and arrays that are not partitioned etc. then inherit masks based upon their relationship with previously masked statements. This can lead to loops containing code sections involving different execution control masks and often multiple masks (where either mask condition being met will cause the execution of the statement). In the following sections, the execution control masking has caused synchronous iteration spaces to be used. The transformation process proceeds by examining all communication requests nested within a loop, checking to see if local or global iteration spaces could be used for each request. Transformation can only be performed if all

communication requests can legally be transformed to the same iteration space, any conflict forces synchronous iterations to be maintained to avoid potential deadlock.

6.2 Transformation into Global Iteration Space.

Consider the example in Figure 12. The array **A** is partitioned across the processor topology in index 2 so the execution control masks on statement S_1 and S_2 are based upon the owner computes rule. A communication request will be issued by the usage of array **A** in S_2 since, for some iterations of the **K** loop, the used value of **A(J,K)** will be on a different processor to where **A(J,K+1)** is owned which is where the computation takes place. The communication request, however, is blocked in its migration due to the loop independent dependence from the assignment in statement S_1 to the communication requesting usage in S_2. This forces the communication request to be placed immediately after statement S_1, leaving the communication inside loops $\mathbf{L_1}$ and $\mathbf{L_2}$, with loop $\mathbf{L_1}$ being forced to execute using synchronous iterations.

An examination of this code fragment, however, reveals that global iteration space execution can be legally used. The communication is only required on each processor when **K = CAP_L-1** since this is when non-owned data is used. Although the assignment to **A** in S_1 is in the same iteration of $\mathbf{L_1}$ in the serial code, in the parallel version in Figure 13, the assignment of **A(J,K)** in S_1 will be on a different processor to the usage, although both processors will be performing the same iteration of loop $\mathbf{L_1}$. The assignment and usage are therefore in different instances of the same iteration of $\mathbf{L_1}$ (one on each processor), so the original barrier to migration at S_1 can be ignored. The parallel code can then be generated using global iteration space as shown in Figure 13. The lower limit of loop $\mathbf{L_1}$ is adjusted to perform an extra iteration of the loop (satisfying the contained execution control masks) and communication statements are added to ensure that globally synchronised iterations are performed. Since the communications are outside loop L_2, a single communication of **NJ** data items can be used, greatly reducing the startup latency overhead. The iteration spaces and synchronising communications are shown diagrammatically in Figure 14.

```
L1    DO K=1,NK
L2        DO J=1,NJ
S1            IF (K.GE.CAP_L.AND.K.LE.CAP_H)A(J,K) = . .
              IF (K.EQ.CAP_H) CALL CAP_SEND(A(J,K),1,2,CAP_RIGHT)
              IF (K.EQ.CAP_L-1) CALL CAP_RECEIVE(A(J,K),1,2,CAP_LEFT)
S2            IF (K+1.GE.CAP_L.AND.K+1.LE.CAP_H)A(J,K+1) = A(J,K)
          ENDDO
      ENDDO
```

Figure 12 Execution control masking for loop that can exploit global iteration space.

```
      CALL CAP_RECEIVE(A(1,CAP_L-1),  NJ, 2, CAP_LEFT)
L1    DO K=MAX(1,CAP_L - 1),MIN(NK,CAP_H)
L2        DO J=1,NJ
S1            IF (K.GE.CAP_L.AND.K.LE.CAP_H)A(J,K) = . .
S2            IF (K+1.GE.CAP_L.AND.K+1.LE.CAP_H)A(J,K+1) = A(J,K)
          ENDDO
      ENDDO
      CALL CAP_SEND(A(1,CAP_H), NJ, 2, CAP_LEFT)
```

Figure 13 Code to perform the example in Figure 12 in global iteration space.

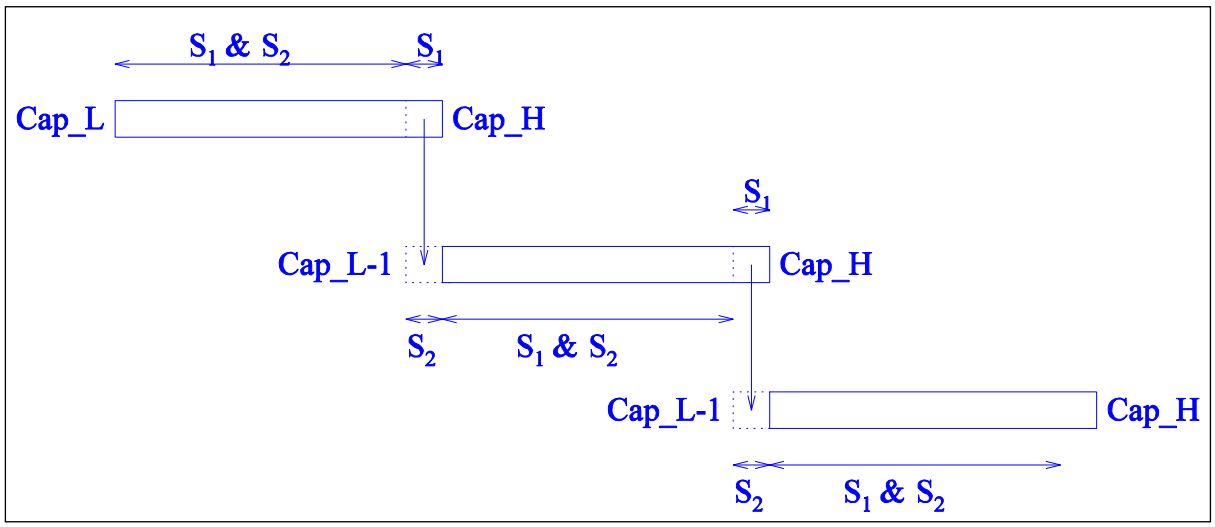

Figure 14 Communications in relation to iterations in global iteration spaces for the code in Figure 13.

6.3 Transformation into Local Iteration Space.

The code section in Figure 15 is subtly different to that in Figure 12. A communication request will still be issued by statement S_2 and consequently be prevented from exiting the surrounding loops by the assignment in statement S_1. The transformation into global iteration space is, however, illegal since when **K = CAP_H+1** each processor will need to use the value **A(J,CAP_H+1)** which would not yet have been calculated on the next processor. In local iteration space, the communication is required for the last owned iteration of loop L_1, when **K = CAP_H+1**, whilst the required value is assigned in iteration **K = CAP_L** on the next processor in the topology. This allows the communication to occur during any iteration between these iterations or at the end of the assigning iteration or the beginning of the using iteration. To allow efficient parallelism, the same point in the local iteration space is selected on every processor so that no idle time will be incurred awaiting the commencement of the communication on any processor. As with global iteration space code, the migration of the communication can precede past the synchronous iteration space barrier at statement S_1 since the assignment and usage, although in the same iteration of the **K** loop, will occur on different processors due to the execution control masks.

```
L1    DO K=1,NK
L2        DO J=1,NJ
S1            IF (K.GE.CAP_L.AND.K.LE.CAP_H)A(J,K) = . .
              IF (K.EQ.CAP_L) CALL CAP_SEND(A(J,K),1,2,CAP_LEFT)
              IF (K.EQ.CAP_H+1) CALL CAP_SEND(A(J,K),1,2,CAP_RIGHT)
S2            IF (K-1.GE.CAP_L.AND.K-1.LE.CAP_H)A(J,K-1) = A(J,K)
          ENDDO
      ENDDO
```

Figure 15 Execution control masking for loop that can exploit global iteration space.

Figure 16 shows the parallel version of the code example in Figure 15 using local iteration space. The communication is still within the **K** loop since both the assignment and the usage of the data being communicated are within that loop. The communication is masked so that it only executes at the start of the iteration of loop L_1 in which the communicated data is used (i.e. **K = CAP_H+1**).

```
L1    DO K=MAX(1,CAP_L ), MIN(NK,CAP_H+1)
          IF (K.EQ.CAP_H+1)CALL CAP_EXCHANGE(A(1,CAP_H+1),A(1,CAP_L),NJ,1,CAP_RIGHT)
L2        DO J=1,NJ
S1              IF (K.GE.CAP_L.AND.K.LE.CAP_H)A(J,K) = . .
S2              IF (K-1.GE.CAP_L.AND.K-1.LE.CAP_H)A(J,K-1) = A(J,K)
          ENDDO
      ENDDO
```

Figure 16 Code to convert the example in Figure 15 in efficient parallel code using local iteration space.

The code in Figure 16 will produce efficient parallel performance, however, it can deadlock if the bounds of the original loop L_1 do not span every processor in the relevant dimension of the processor topology. In such a case, some processors will not enter loop L_1, therefore not correctly synchronising with the exchange communications on neighbouring processors. To avoid this, the communication is broken down into the constituent send and receive of an exchange communication, with execution control masks added to ensure that both processors involved in the communication will perform the relevant iterations of L_1.

The operation of the local iteration space code in Figure 16 is shown diagrammatically in Figure 17 with indication of which statements (S_1 and/or S_2) are executed on each processor. To be legal, therefore avoiding deadlock, all communications within the loop must be transformable into local iteration space.

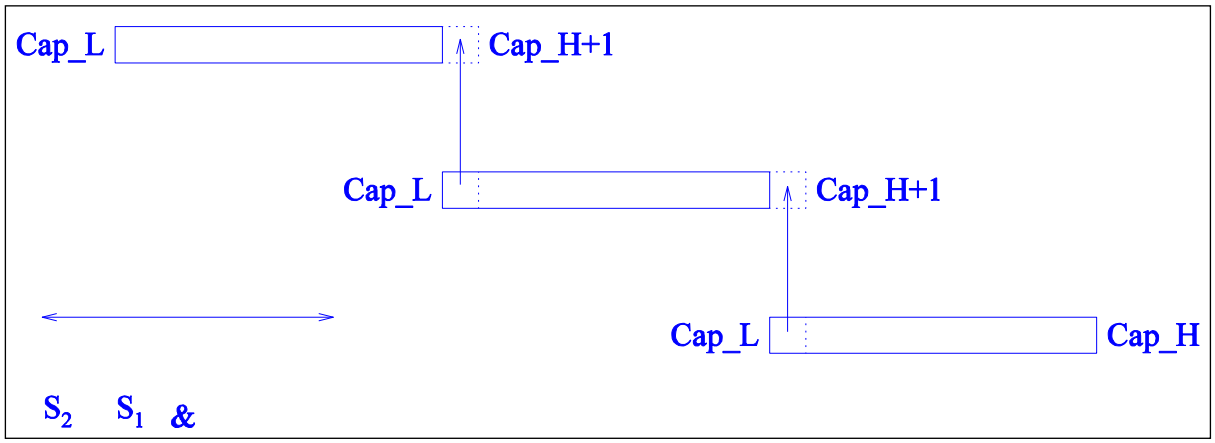

Figure 17 Communications in relation to iterations for local iteration space code for the code in Figure 16.

7 Results.

The previous sections have presented techniques for handling code sections, which would previously have been executed in serial with heavy communication overheads. Using serial execution of these code sections has a very significant effect on the parallel performance of most codes, with slowdown often being exhibited. The results in Table 1 show the number of instances in real application codes where the above transformations were required. For all these codes, slowdown (or deadlock for the I/O exception based loops) is exhibited without these transformations. The codes referred to in Table 1 are:-

- TEAMKE1 CFD code from University of Manchester Institute of Science and Technology (UMIST).
- TEACH CFD code from Imperial College, London
- APPSP NAS benchmark code from NASA Ames.
- ARC3D CFD code from NASA Ames.
- SIMPLE
- UIFS Unstructured mesh multi-physics modelling code from the University of Greenwich.
- BOAST RICEPS benchmark code.

Code	Subsidiary Commutative	Search/Error Checking	I/O Exceptions
ARC3D	3		
SIMPLE	20		
UIFS		1	271
BOAST	16	2	1

Table 1 Occurrences of commutative communication based transformations in real application codes.

Code	Local Iteration Space	Global Iteration Space
TEAMKE1	4	
TEACH		4
APPSP		4

Table 2 Occurrences of iteration space transformations.

8 Conclusion.

A range of techniques have been presented to enable efficient parallel code to be generated for a number of differing classes of code section that formerly presented bottlenecks to parallel performance. Without these techniques, codes typically produce slowdown when executed in

parallel and the manual remedies to these situations require considerable user expertise. The techniques also fulfil the requirement of producing recognisable code, and so maintain the ability of the code paralleliser to optimise and/or maintain the generated parallel version.

9 References.

1 C.S. Ierotheou, S.P Johnson, M. Cross and P.F. Leggett. Computer Aided Parallelisation Tools (CAPTools) - Conceptual overview and performance on the parallelisation of structured mesh codes. Parallel Computing, 22(2),1996.

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

3 S.P. Johnson, C.S. Ierotheou and M. Cross. Automatic parallel code generation for message passing on distributed memory systems. Parallel Computing, 22(2),1996.

4 E.W. Evans, S.P. Johnson, P.F. Leggett and M. Cross. Automatic and Effective Multi-Dimensional Parallelisation of Structured Mesh Based Codes. University Of Greenwich Technical Report PPRG-98-008, 1998.

5 S.P.Johnson, K.McManus, C.S.Ierotheou, P.F.Leggett and M.Cross. Semi-Automatic Parallelisation of Unstructured Mesh Codes by Domain Decompostition. University of Greenwich Technical Report PPRG-98-002, 1998

6 P.F. Leggett. CAPLib, A Portable Communications Library for Automatically Generated Parallel Code. University of Greenwich Technical Report, PPRG-98-013, 1998.

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

8 E.W. Evans, S.P. Johnson, P.F. Leggett and M. Cross. Automatic Code Generation of Overlapped Communications in a Parallelisation Tool. Parallel Computing (23), pp 1493-1523, 1997.

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

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