Strategies and Tools for the Exploitation of Massively Parallel Computer Systems

Emyr Wyn Evans

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the Degree of Doctor of Philosophy.

9th September, 2000

Parallel Processing Research Group
School of Computing and Mathematical Sciences
University of Greenwich
London UK

Acknowledgements.

There are several people who I wish to thank for their help during the time that it has taken to accomplish this work and for the writing of this thesis.

My supervisors, Professor Mark Cross and Doctor Stephen Johnson, for their support and guidance, and especially their overwhelming patience.

My colleagues Constantinos Ierotheou, Peter Leggett, Kevin McManus, Chris Walshaw, Jackie Rodrigues, Chris Bailey and Peter Chow who have assisted me in varying degrees for the completion of this thesis.

Finally, to my wife Elisabeth and my parents for supporting me on this long journey.

Abstract.

The aim of this thesis is to develop software and strategies for the exploitation of parallel computer hardware, in particular distributed memory systems, and embedding these strategies within a parallelisation tool to allow the automatic generation of these strategies.

The parallelisation of four structured mesh codes using the Computer Aided Parallelisation Tools provided a good initial parallelisation of the codes. However, investigation revealed that simple optimisation of the communications within these codes provided an even better improvement in performance. The dominant factor within the communications was the data transfer time with communication start-up latencies also significant. This was significant throughout the codes but especially in sections of pipelined code where there were large amounts of communication present.

This thesis describes the development and testing of the methods used to increase the performance of these communications by overlapping them with unrelated calculation. This method of overlapping the communications was applied to the exchange of data communications as well as the pipelined communications.

The successful application by hand provided the motivation for these methods to be incorporated and automatically generated within the Computer Aided Parallelisation Tools. These methods were integrated within these tools as an additional stage of the parallelisation. This required a generic algorithm that made use of many of the symbolic algebra tests and symbolic variable manipulation routines within the tools.

The automatic generation of overlapped communications was applied to the four codes previously parallelised as well as a further three codes, one of which was a real world Computational Fluid Dynamics code.

The methods to apply automatic generation of overlapped communications to unstructured mesh codes were also discussed. These methods are similar to those applied to the structured mesh codes and their automation is viewed to be of a similar fashion.

Contents

List of Figures

Chapter 1

1 Introduction.

1.1 Why Parallel Processing?

The need for parallel processing is born from the fact that computer users always require their programs to perform computation at a much faster rate. There are many large scale codes available for Computational Fluid Dynamics, Computational Mechanics, etc that require a large amount of processing power. These codes often take hours, even days to run and a greater amount of power is therefore required to allow these codes to run in a fraction of the time.

To meet this demand for greater processing power supercomputers were developed with vector or pipeline processors. These processors instead of operating on a single variable at a time, allowed a vector of data to be processed simultaneously [1]. This required the code author to optimise the code in order to exploit valid vector operations and ensure that the correct vector operands were loaded from memory [2]. This led to the development of vectorising compilers which automatically optimised the code for vector parallelism [3].

This in turn led to the development of supercomputers that consisted of an array of processors (ranging from 1000 to 16,000) which could process data in parallel. All the processors operated on the same instruction set issued by a central processing unit on its own data set. These array structured machines (e.g. the Illiac-IV, ICL DAP, Thinking Machines CM2) are known as Single Instruction, Multiple Data (SIMD) [4] machines. These machines increased the performance of codes significantly so long as the problem was structured in nature and did not consist of any serial operations [5].

The advent of the Transputer processor [6] in the early 1980's allowed manufacturers to design relatively inexpensive parallel machines. These processors could execute their own instructions on their own data set. These machines are referred to as Multiple Instruction, Multiple Data (MIMD) [4]. There are two distinct variants of the MIMD class : Shared Memory (SM) which has a common (shared) memory space and Distributed Memory (DM) where each processor has its own private memory [7]. Both of these sub classes of the MIMD have their disadvantages.

In the case of the SM-MIMD class, memory contention causes bottlenecks when executing serial loops and all the processors have to access the shared memory via the same data bus. There is also the need for synchronisation points within the parallel code. This incurs an overhead and can also create idle time while processors wait for other processors to complete their tasks.

In the case of the DM-MIMD class there are several causes of bottlenecks. These consist of too many communications in the parallel code, which may also communicate large volumes of data or communicate data to all other processors as opposed to their nearest neighbours only. There are also the possibilities of idle time and of duplicated calculation that will reduce the efficiency of the parallel code but will remove the requirement for communication.

Several European manufacturers used the Transputer in the late 1980's and early 1990's in the creation of modest inexpensive parallel machines [8, 9]. Examples of these are the Transtech Paramid [81] and the Parsys SN9500 [82] which are mentioned in greater detail in Section 4.3. These manufacturers as well as many others have since adapted other better performing processors to build even more powerful parallel machines.

There are many variants of parallel machines now available for users [8, 9]. Examples of SM-MIMD machines that are commonly available today are the DEC AlphaServer Clusters and SGI Origin 2000. Examples of DM-MIMD machines are the IBM SP2, Cray T3D and Cray T3E.

1.2 Problems of Creating Parallel Codes.

The hardware for parallel processing is obviously widely available. The main difficulty is providing parallel codes for this hardware. This may be achieved by writing a parallel code in a new language or by adapting existing sequential code to run on these machines. To convert a large serial code to be parallel may take many man months [10] or years to achieve. This method of parallelisation should eventually provide the most efficient form of parallel code but is however very tedious and is open to error. There is currently a range of options available to automate the process of parallelism: optimising and vectoriser compilers; shared memory parallelising compilers; distributed memory parallelising compilers; High Performance Fortran (HPF); or parallelisation tools.

Many of today's compilers may optimise and/or vectorise a serial code to exploit parallelism within loops using code transformations such as scalar expansion and loop splitting. These compilers are fast and provide a reasonable improvement to the code

Automatic compilers were designed to automatically parallelise the code. In the case of the shared memory system this has provided some satisfactory results for some limited cases [11]. This, however, was only achieved once the user had inserted compiler directives into the source code. The main problem is that the user (from previous experience) expects compilers to be quick. This leads to the compilers making many conservative assumptions often presuming there is a dependence if it cannot prove otherwise. This often leads to a particular code section being serial. These compilers concentrate on only a small section of code, such as a loop, to try and obtain parallelism and are always intra-procedural, i.e. they do not take a global view of the code, concentrating parallelism within a procedure. They will also typically only parallelise one loop within a given nest of loops.

In the past two decades a number of research programmes have pursued the concept of parallelising compilers for distributed memory, with the more recent projects focusing on HPF. These include Paraphrase at the University of Illinois [12], the KAP paralleliser [13], Parallel Fortran Converter (PFC) [14] and FORTRAN-D [15] at Rice University, SUIF [16] at Stanford, VIENNA-FORTRAN [17, 18] at the University of Vienna and PARADIGM [19] developed by the University of Illinois.

There are at present a number of groups who are attempting to develop parallelising compilers and parallelisation tools. Parallelisation tools are a compromise solution to the desire for parallel compilers that generally produce poor parallel efficiencies by comparison with manual parallelisations that produce the most efficient parallel code. Most of these tools make use of compiler technology to convert serial code to parallel code, for a particular parallel machine. All tools must make conservative assumptions when generating parallel code and therefore much potential parallelism might be omitted to ensure correct code.

A method, currently much promoted, is the use of High Performance Fortran (HPF) language [20]. This requires the programmer to posses a significant amount of expertise, and even then the amount of effort required can be substantial. It is also restrictive in that most dusty deck Fortran codes will require considerable amounts of re-engineering and rewriting before the code is actually suitable for HPF. Once the source code has been converted to HPF the performance of the code in parallel are, for certain test cases, not very good. [21].

There are also at present a small number of parallelisation tools available or being developed. These are Forge 90 [22] developed by Applied Parallel Research, Vienna Fortran Compilation System [23] developed by the University of Vienna, D System [24] developed by Rice University, PARADIGM [25] developed by the University of Illinois and Computer Aided Parallelisation Tools (CAPTools) [26, 27, 28, 29] developed at the University of Greenwich.

Forge 90 [22] developed by Applied Parallel Research is an integrated collection of interactive tools to enable the parallelisation of Fortran. The tools generate fully scalable Fortran 77 Single Program Multiple Data (SPMD) program with support for many different message passing libraries such as IBM's MPL, PVM, Express and Linda. It will also allow standard Fortran 90 and HPF directives to be used to control the parallelisation of the program.

The D System was developed by Kennedy et al at Rice University and grew out of ParaScope [30]. It consists of a suite of tools developed to aid in the development of programs in Fortran D [30]. Fortran D is an extension to existing Fortran 77 or Fortran 90 compilers. It was primarily designed to create a machine independent set of extensions to aid in the distribution of data onto parallel machines. Fortran D compilers have been developed for several parallel machines including the Intel Paragon and Thinking Machines CM-5. High Performance Fortran (HPF) is an extension to Fortran 90 and was inspired by the original work on Fortran D. It provides support for data parallel programs and for the control of data distribution.

Vienna Fortran [23] was developed by Zima et al at the University of Vienna. It was developed from the original SUPRENUM [31] project. It is a machine independent language extension to Fortran 77, allowing the user to write programs for DMS using global addresses. Vienna Fortran is now part of the Vienna Fortran Compilation System (VFCS) which provides source to source conversion of Vienna Fortran or Fortran 77 code to explicit parallel Message Passing Fortran for use on Intel iPSC/860, Intel Paragon, and machines that support Parmacs.

PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) [25] developed at the University of Illinois provides an automated means of parallelising and optimising serial programs for efficient use on a distributed memory system. PARADIGM allows automatic data distribution, communication optimisation and the exploitation of both functional and data parallelism. It has been used on several DMS machines such as the Intel Paragon, the Thinking Machines CM-5 and the IBM SP-1.

Computer Aided Parallelisation Tools (CAPTools) [26, 27, 28, 29] is a toolkit developed at the University of Greenwich to automate most of the process of parallelising scalar Fortran 77

codes. The aim of CAPTools is to obtain code that is as efficient as manually parallelised code by using a combination of parallel compiler technology and as much user interaction as is necessary. The time and effort required by a user to create such a parallel code should be minimal, but the resulting code should be as efficient as possible.

The final parallel code generated by the first version of CAPTools adheres to the Single Program Multiple Data (SPMD) model. In the SPMD model each processor executes the same code but on a subset of the program data. The parallel code produced will be as similar as possible to the original serial code allowing the parallel code to be easily optimised and maintained by the user and easily portable to any Distributed Memory System. The parallel code will differ from the serial code in that it will now contain communication calls and also execution control masks to ensure that each processor will only operate on its own data subset. The communications generated are high level generic communication calls which map onto low level communications of either machine specific communications or communication libraries (Section 1.6).

Chapter 2 discusses the Computer Aided Parallelisation Tools in more detail.

1.3 Requirements of Parallel Processing.

There are a number of objectives that must be achieved by a satisfactory parallelisation strategy:

1. Minimise the changes to the original algorithms :

   The parallel code should produce exactly the same results as serial. Identical results provide the user with confidence that the parallel code is correct.

2. Recognisable code :

   The parallel code should also be recognisable and therefore easily maintainable and/or optimised by the original serial code author.

3. Maximise the invisibility of the parallel execution :

   The user should not notice any difference between running the parallel and serial code except for an increase in speed and possibly the size of the problem that can be solved.

4. Maximise parallel efficiency :

   Ensure that the parallel code produces significant increase in speed up in relation to the serial code. This ensures that the parallel machine is being used efficiently.

5. Efficient use of all available memory:

   Ensure that the problem size is proportional to the total local memory size available from every processor.

Different members of the Parallel Processing community will place varying amounts of importance to these objectives. The user of a parallel code will be concerned with objectives 3, 4 and 5: the code looks the same during execution; will reduce the computation time; and allow bigger problem sizes to be executed. The application code author will primarily only be interested in objectives 2 - the minimum amount of change to the code - but due to the needs of the end user must also pay attention to the other objectives. An in-house code developer wishing to parallelise their codes will place an emphasis on all the above five objectives. Developers of parallelisation tools on the other hand must bear in mind the needs of the users, authors and in-house developers and must concentrate on all of these objectives.

1.4 The Use of Meshes in Computational Mechanics Codes.

This work focuses upon certain important classes of application code and their related SPMD parallelisation strategies. Computational Mechanics is a diverse area that includes the modelling of fluid dynamics, structural mechanics and electromagnetics. The application of a system of equations to a problem leads to the concepts of a grid or mesh. There are two distinct types of meshes predominantly used to discretise a problem in Computational Mechanics : Structured and Unstructured Mesh. Examples of these meshes may be seen in Figure 1.1.

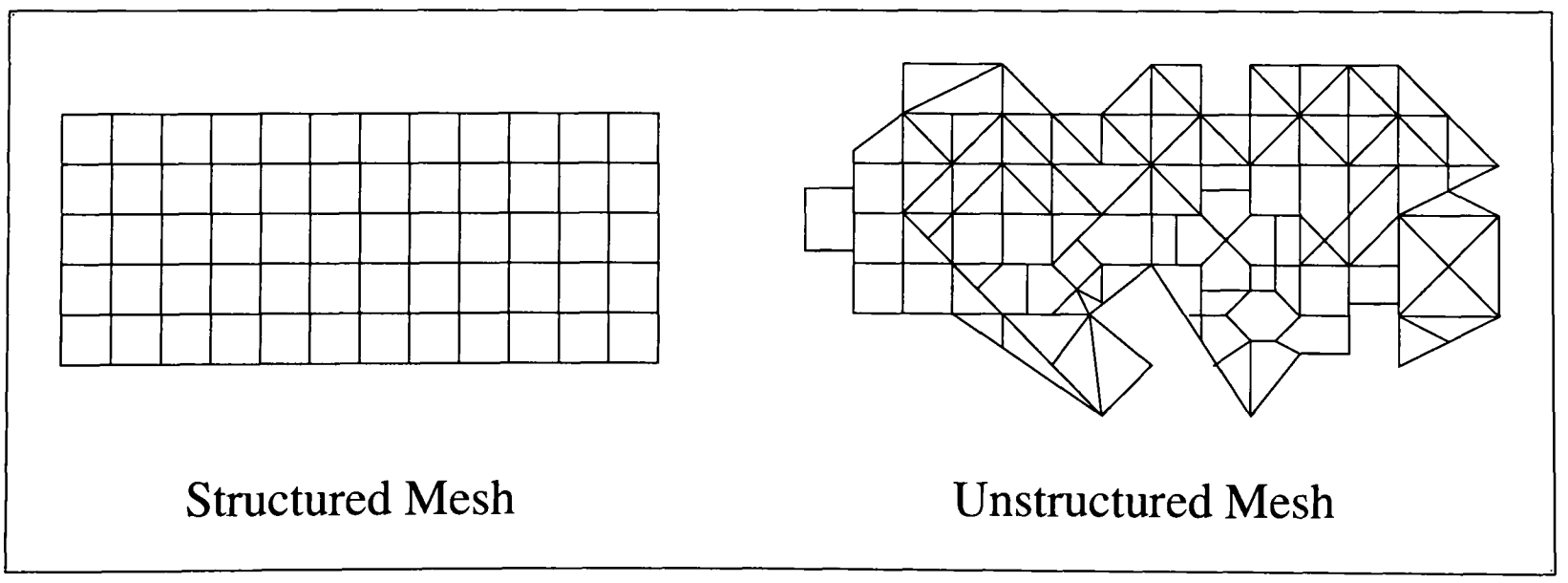


Figure 1.1 : Simple structured and unstructured meshes.

In a structured mesh code the mesh is regularly structured and is well suited for control volume (cells) and finite difference problems. The advantage of using such a mesh is that the topology of the mesh is stored implicitly allowing simple addition or subtraction to calculate its neighbouring cells. Figure 1.2 shows that to calculate the value of A(I,J,K) require the values from both its immediate neighbours in both the J and K dimension. The main disadvantage of using a structured mesh is that only regular shaped geometries may be solved.

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

Figure 1.2 : A typical code section from a structured mesh code.

Unstructured or irregular meshes are used for the solving of finite element or control volume problems or any other inter-related entities. In an unstructured mesh the topology is explicit with the relationships between elements and nodes of the mesh explicitly stored, e.g. ELETOP(ELEMENT, NODE) will contain the relationship between the nodes and elements. Figure 1.3 shows that to calculate the value of A for a particular element requires the topological information for B from the element and nodal information of A.

```
DO ELEMENT = 1, NELEMENT
    DO NODE = 1, NNODE(ELEMENT)
        A(ELEMENT)=B(ELETOP(ELEMENT,NODE))
    ENDDO
ENDDO
```

Figure 1.3 : A typical code section from an unstructured mesh code.

The use of unstructured mesh allows more complex geometries to be solved but unfortunately they will not be as efficient as structured meshes since they require indirect address accesses of arrays in their calculations, i.e. in the use of ELETOP.

1.5 Parallelisation Strategies.

There are three predominant methods available to parallelise a code. These are Task Farming, Algorithmic Decomposition and Domain Decomposition. These may also be used together to form a hybrid.

Task Farming [32] (or task-scheduling) attempts to ensure that all processors are always kept busy with computation. This method requires one process acting as the master processor distributing tasks for each of the other slave processors. The master will also collate the results from these slave processors. The master also attempts to ensure that every slave processor is kept busy at all times thus avoiding idle time. Good parallel efficiencies can therefore be obtained as long as the time required for each task is significantly more than the communication time of the task between the master and slave. There should also be a significant number of tasks in comparison with the number of processors. This method of parallelisation is only suitable for problems where there is no other communication required with any other slave, i.e. each slave task must be independent of any other calculation on another slave task. It has proven well suited for radiation field calculations [33, 34] and Monte Carlo techniques [35].

Algorithmic methods or Functional Decomposition [32] involves dividing the algorithm on to several processors. For example consider a three dimensional Computational Mechanics (CM) code which requires velocities for each dimension to be calculated. Algorithmically this could be achieved by allowing each dimensional velocity to be calculated on one of three different processors. The main disadvantages of such a method is that each processor may have a different amount of work to be done and could cause some of the processors to be idle. They are also not scalable, i.e. if there are three different algorithms that may be run in parallel then only three processors are required. The addition of any further processors will not provide any further increase in the efficiency of the code. Also if the calculation on each processor are not independent of each other then the communication overhead may be high.

Domain Decomposition [32], also known as Geometric Decomposition, involves splitting the data as evenly as possible on to each individual processor. Each processor will then operate on its own allocated subset of data. Since each processor has an even amount of data to operate upon, the idle time will often be very small. There are three distinct methods of mapping a domain of interest here : Block, Cyclic and Graph Based Partitions.

A block mapping [32] represents each processor containing an equal continuous section of the problem domain. Each processor would then be allocated one of these blocks of data. For instance a block mapping of a matrix with 7 rows onto 3 processors would be distributed as in Figure 1.4.

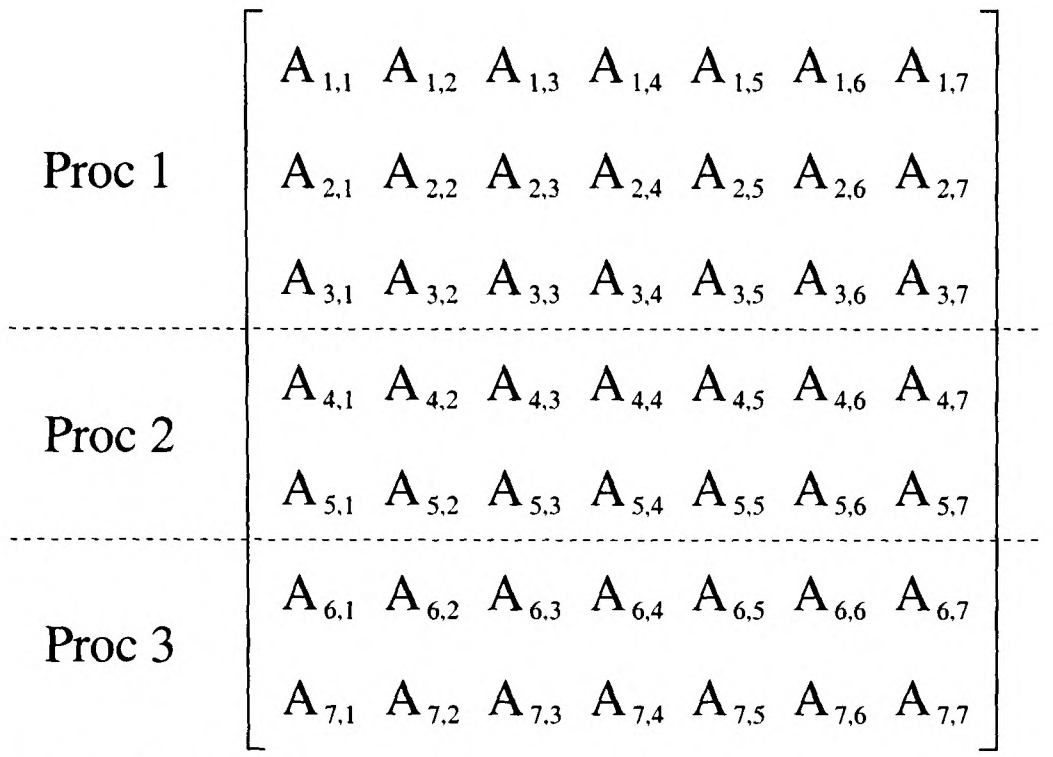


Figure 1.4 : Block Mapping of a Matrix.

Each processor is allocated at least 2 rows each; the first processor was also allocated the extra row. Therefore, each processor has approximately the same number of rows. As the number of rows per block increases then the load balance will also improve.

Cyclic or wrap mapping [32] involves distributing each consecutive row from a matrix to consecutive processors. Consider Gaussian Elimination [36, 37] where only values in a column below the pivot are eliminated to form an upper triangular matrix (Figure 1.5).

Figure 1.5 shows that for the first column to be eliminated the amount of work on each processor is approximately even. However, looking at the sixth column to be eliminated both processors 1 and 2 do not have any elimination to carry out and are therefore idle.

Therefore, when column 1 is being eliminated there is the same amount of work being carried out on each processor. As the algorithm proceeds processor 1 will become idle; then processor 2 will become idle; etc, until only the last processor is doing any work. This is known as Load Imbalance.

To overcome this problem, the matrix is distributed using wrap mapping (Figure 1.6). Wrap mapping involves distributing each consecutive row from the matrix to consecutive processors. For example, processor K, of N processors, would be allocated rows K, K+N, K+2N, K+3N,... Figure 1.6 shows the matrix A after elimination.

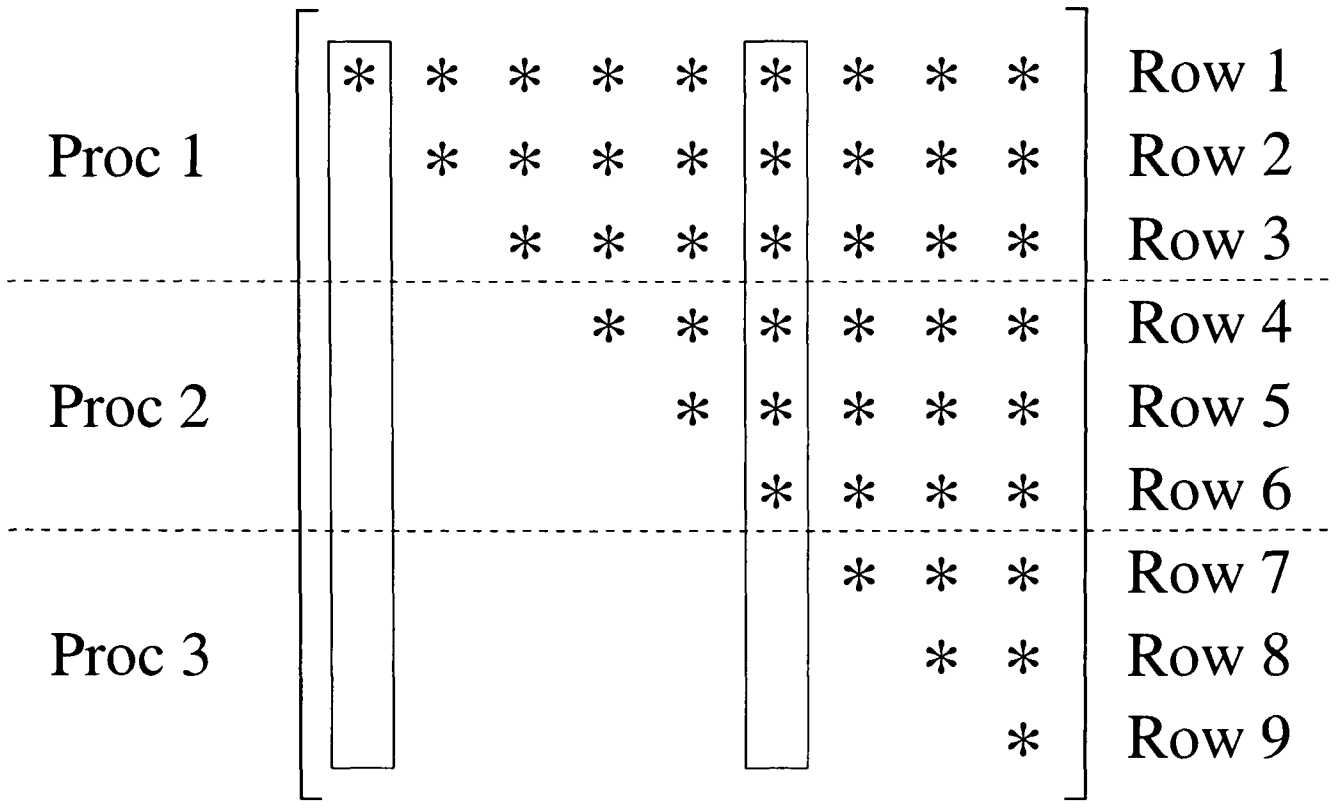


Figure 1.5 : An Upper Triangular Matrix with Block Mapping (N.B. The Symbol * represent elements not processed).

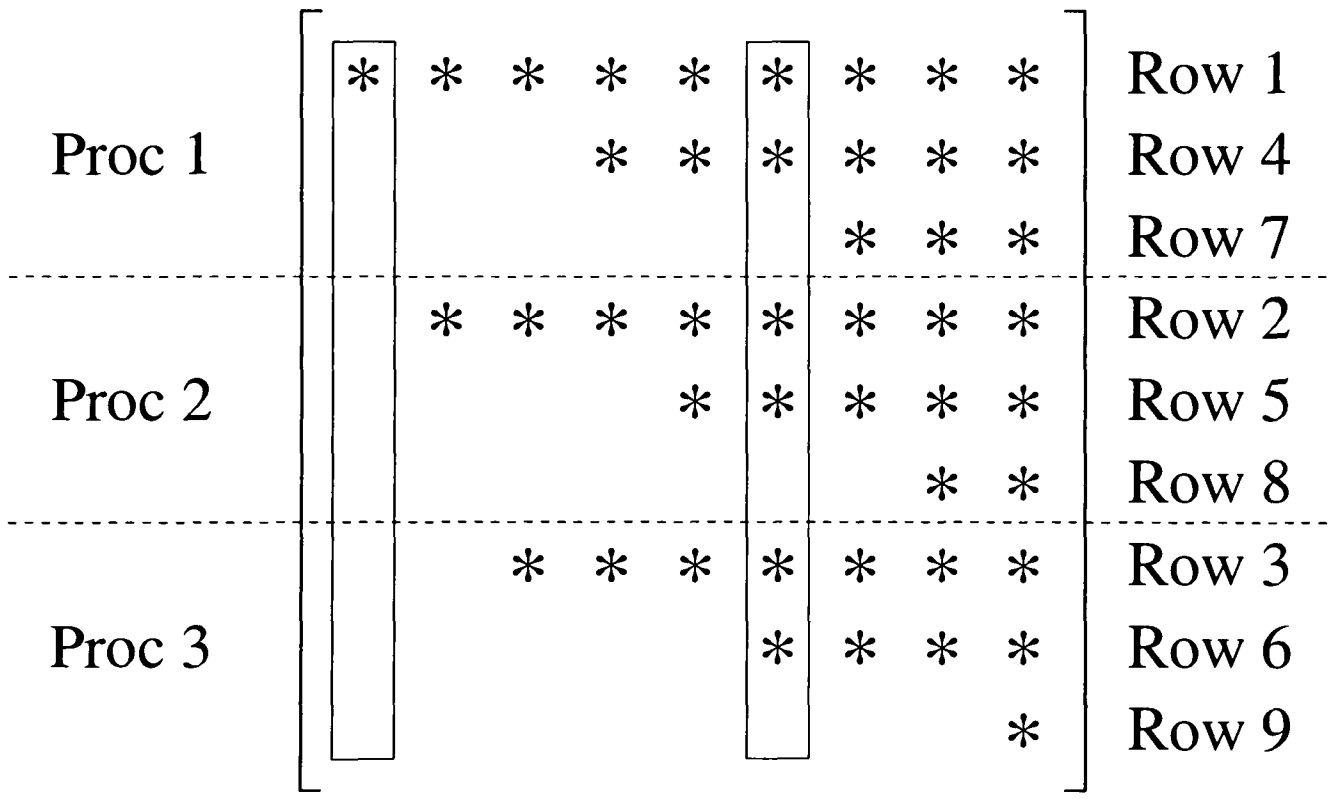


Figure 1.6 : Wrap Mapping of an Upper Triangular Matrix (N.B. * represents elements not processed)

Figure 1.6 shows that for the first column the workload is approximately equal. Looking at the sixth column now each processor has the same amount of work to accomplish. This method therefore provides a better load balance for upper or lower triangular matrices.

Another method that may be used is Block Cyclic which is a hybrid of the block and cyclic methods. For this method, the load imbalance is higher than it would be for the cyclic

method but lower than for the block method. This method does however have the potential for lower communication than the cyclic method but more than for the block method.

Graph Based Partitions involve taking a graph that represents the mesh of the problem and dividing the graph such that each processor has an equal amount of graph/mesh nodes whilst also minimising the number of graph edges cut. The greater the number of edges cut then the greater the volume of communication. There are several methods to accomplish this, some of which are : Greedy [38], Recursive Graph Bisection [39], Recursive Spectral Bisection [40] and Multilevel Recursive Spectral Bisection [41].

There are at present several software tools available to provide a graph decomposition. These include Scotch [42], Metis [43], Chaco [44] and JOSTLE [45, 46, 47].

JOSTLE [45, 46, 47] developed at the University of Greenwich decomposes unstructured meshes by first of all using the Greedy method to provide an initial partition. It will then reduce the problem size by using a Recursive greedy algorithm before applying further optimisation heuristics.

1.6 Communication Utilities.

In this work the Computer Aided Parallel Tools communication library is used. These are high level generic communication calls developed at the University of Greenwich [48, 49]. These map onto the low level communication calls of either machine specific communications such as Cray SHMEM [50], Inmos CToolset or onto communications libraries such as PVM [51] or MPI [52]. The communications have been designed to function on various processor topologies such as 1-D, 2-D or 3-D grids of processors, rings of processors or a full processor interconnection. They have been employed successfully and efficiently in the parallelisation of numerous Computational Mechanics (CM) codes [26]. The communications have been designed to be simple with the minimum number of parameters required. This allows the communications to be comprehendible in understanding the nature of the parallel code. They are easily portable to other parallel machines using the above mentioned communication libraries and can easily be adapted for use with any other communication library or low level communications.

Examples of these high level communication calls are CAP_SEND, CAP_RECEIVE and CAP_EXCHANGE which will respectively send, receive or perform a pairwise parallel

exchange of the required data between the required processors as stated in the communication calls parameter lists.

The parameter lists for these high level generic communications for the synchronous communications 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*, *Direction*)

where the *Send Address* is the start address of the data to send; the *Receive Address* is the start address to receive the data; the *Length* is the amount of data to be communicated; the *Type* is an integer value representing the type of data to be communicated e.g. integer, real, etc; and the *Direction* is the processor or neighbour to communicate with. The *Direction* definition for a 1-D grid of processors (pipeline/chain), for example, may be CAP_LEFT or CAP_RIGHT which simply states that the data is to be communicated from processor p to processor p-1 or processor p+1, respectively. For a 2-D grid of processors the *Direction* would be either CAP_LEFT, CAP_RIGHT, CAP_UP and CAP_DOWN. This is extended to a 3-D array of processors by introducing CAP_TOP and CAP_BOTTOM.

The CAP_SEND and CAP_RECEIVE communications will always work in tandem. Consider the following code:

```
CALL CAP_SEND(A(1), 10, 1, CAP_LEFT)
CALL CAP_RECEIVE(A(1), 10, 1, CAP_RIGHT)
```

The first communication will send 10 values of data beginning at the array address A(1) of the data type 1 (which represents an INTEGER) to the processor to its left. The second communication will then receive the 10 values of data from its right into the array with the array address beginning at A(1).

An example of an exchange communication as used in a code is as follows:

```
CAP_EXCHANGE(A(100), A(200), 100 ,2, CAP_RIGHT)
```

The processor will receive 100 data items from its right in to the array address A(100) of the data type 2 (which represents a REAL). The communication will also send back 100 data items in the opposite direction (i.e. to the left) from the array address beginning at A(200). The advantage of the CAP_EXCHANGE communication is that each processor will perform the communication at the same time in the same direction in parallel. The time required to exchange data is thus independent of the number of processors.

Another high level generic communication call commonly used is the CAP_COMMUATIVE that allows each processor to calculate its own local commutative operation (e.g. minimum, maximum or sum calculation, etc) before communicating with all other processors and returning the global value to each processor. The parameter list for this communication is as follows :

CAP_COMMUTATIVE(*Value*, *Type*, *Function*)

Where *Value* is the local contribution to the value to which the commutative operation is to be applied, and *Function* is the binary commutative function to be used, e.g. MAX, ADD, etc.

An example of a commutative communication is as follows :

CAP_COMMUTATIVE(MAXNUM, 2, CAP_RMAX)

Each processor will provide its own local value of MAXNUM and will, based on the function CAP_RMAX which finds the maximum value of MAXNUM, will return it as the global value.

In the CAPTools library every processor knows its position in the processor grid and knows which processors are its neighbours. This includes knowing that a neighbour processor does not exist in a certain direction if it is a processor on the edge of a grid.

1.7 Domain Decomposition of a 1-Dimensional Jacobi Solver.

To demonstrate the parallelisation of a code using domain decomposition and the Computer Aided Parallelisation Tools communication library consider the following problem. The simple diffusion Jacobi problem being solved is :

TNEW(I) = (T(I-1)+T(I+1))/2 where I=2,999

and the boundary conditions are

T(1) = 1 T(1000) = 100

The serial algorithm for solving this problem would be as follows :

```
DO I = 2,999
    TNEW(I) = (T(I-1)+T(I+1))/2
ENDDO
```

The mesh for the problem is 1-dimensional consisting of 1000 elements. To decompose this mesh onto, say N processors using Block Mapping (Section 1.5) would involve distributing 1000/N elements to each processor. For example if there were 4 processors then the number of

elements distributed to each processor would be 250. Table 1.1 shows the range of data each processor would operate on.

Processor	Low Range	High Range
1	1	250
2	251	500
3	501	750
4	751	1000

Table 1.1 : Data ranges for four processors.

Each processor has its own unique low and high range limit to operate upon. These low and high ranges values will differ depending on the number of processors. For example on 2 processors the ranges is shown in Table 1.2.

Processor	Low Range	High Range
1	1	500
2	501	1000

Table 1.2 Data ranges for two processors.

These low and high range values are dependent on the number of processors. Since the problem may be ran with varying number of processors then these low and high range values must be dynamically generated at the beginning of the parallel code. These low and high ranges are calculated for each processor at runtime based on the problem size (often read in) and the user specified number of processors. These low and high ranges are allocated the variable names CAP_LOW and CAP_HIGH and are unique to each processor. These variables may then be used to partition the solver loop as follows :

```
DO I = MAX(2,CAP_LOW), MIN(999, CAP_HIGH)
        TNEW(I) = (T(I-1)+T(I+1))/2
ENDDO
```

The MAX and MIN functions ensure that the original limits of the problem are not exceeded.

To calculate the value of TNEW for each processors range requires values from its neighbouring processors. Consider the 4 processor case again. On processor 2 the range of I will be from 251 to 500. To calculate the value of TNEW(251) requires the value of T(250) and

T(252). Processor 2 clearly does not own the value of T(250) as it is owned by processor 1. Processor 1 on the other hand requires the value of T(251) for the calculation of TNEW(250). An exchange of data is therefore required between processor 1 and 2. The same is also true for the other processors. This may be accomplished by means of two CAP_EXCHANGE communications as can be seen in the following code:

```
CALL CAP_EXCHANGE(T(CAP_HIGH+1), T(CAP_LOW), 1, 2, CAP_RIGHT)
CALL CAP_EXCHANGE(T(CAP_LOW-1), T(CAP_HIGH), 1, 2, CAP_LEFT)
DO I = MAX(2,CAP_LOW), MIN(999, CAP_HIGH)
     TNEW(I) = (T(I-1)+T(I+1))/2
ENDDO
```

The first CAP_EXCHANGE will receive into its upper overlap area (sometimes referred to as the halo region) the value of T(CAP_HIGH+1) from the processor to its right. It will then send the value of T(CAP_LOW) to the overlap area of the processor to its right. Similarly, the second CAP_EXCHANGE will receive into its lower overlap area the value of T(CAP_LOW-1) from the processor to its left. It will then send the value of T(CAP_HIGH) to the overlap area of the processor to its left. This partitioned data and exchange of data for 4 processors is illustrated in Figure 1.7.

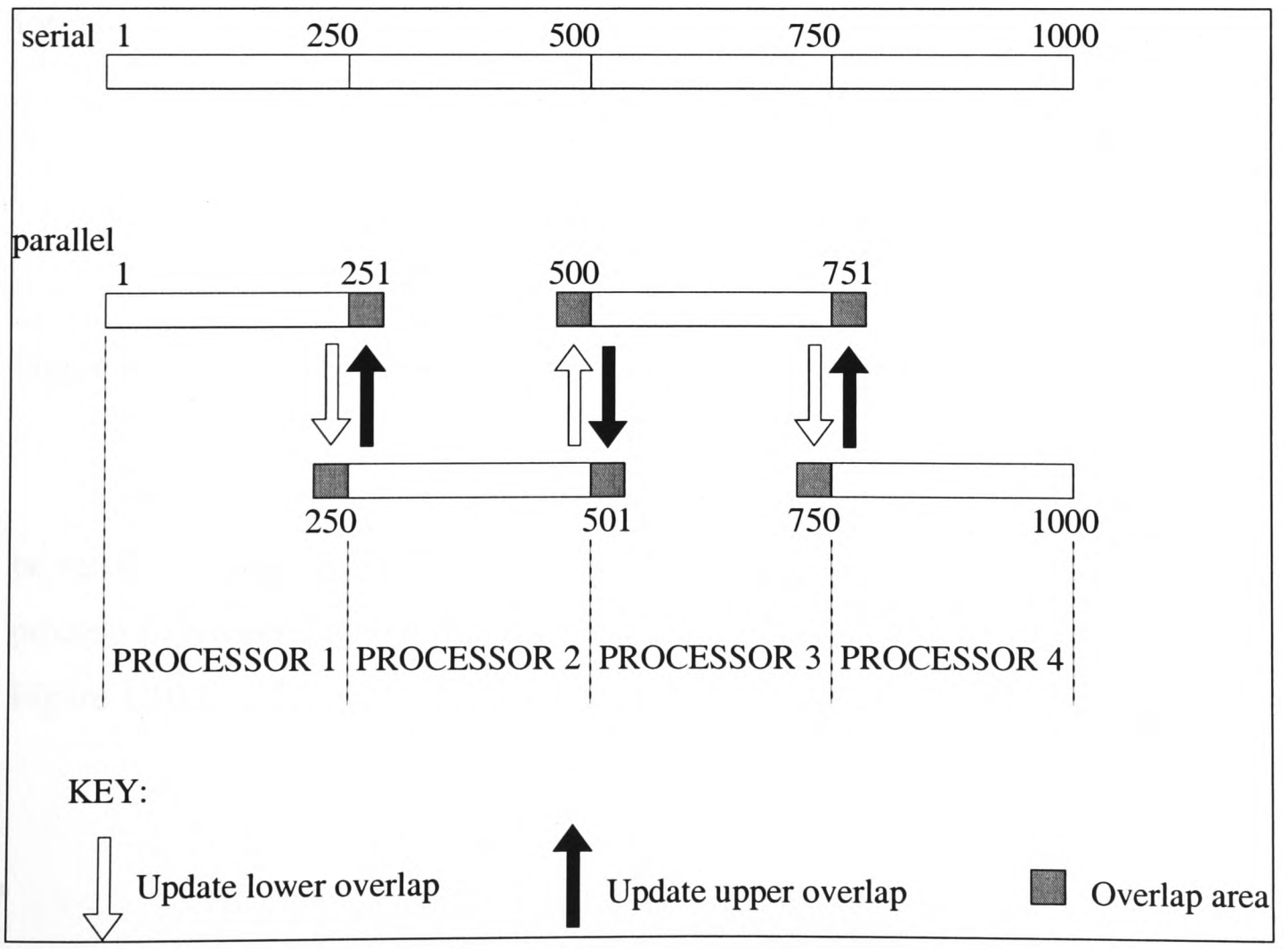


Figure 1.7 : Comparison of an Unpartitioned and Partitioned 1-D Array.

1.8 Implementing Recurrence Relations using Pipelines.

Consider the simple serial recurrence calculation in Figure 1.8.

```
A(1)=A_INITIAL
DO L=2,NZ
  A(L)=A(L)+A(L-1)
ENDDO
```

Figure 1.8 : A Simple Serial Recurrence.

To calculate the value of the array for a given index L requires the value of the present index and the value of the array at the previous index, i.e. it require A(L) and A(L-1). A recurrence occurs since the previous value A(L-1) is required to calculate the value of A(L). This recurrence is very much like a production line or pipeline where an entity is required from the previous stage of the pipeline. It is for this reason that a recurrence is often referred to as a pipeline.

Figure 1.9 shows how this pipeline is reliant on the previous index of the array. For example, to calculate the value of A(3) requires the value of the previous index A(2). This indicates that the computation must be done in this strict order to ensure correct results. This form of calculation is found regularly in most CFD codes.

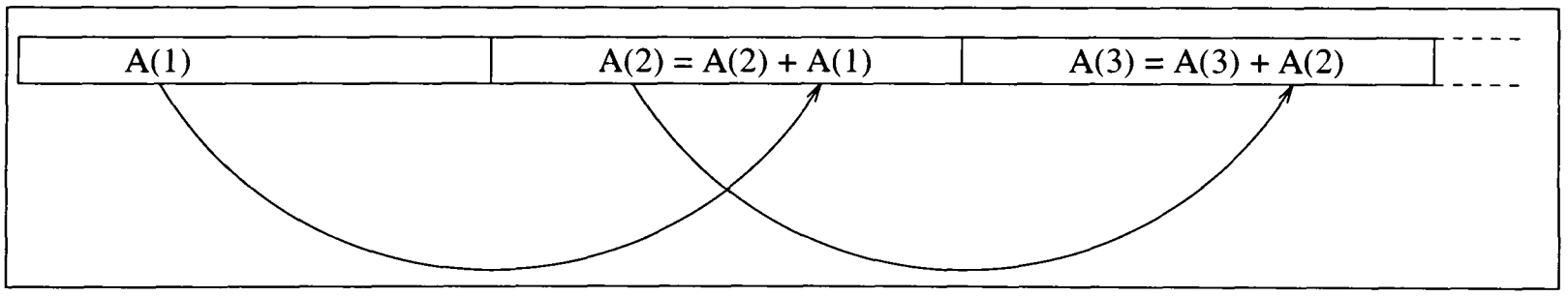


Figure 1.9 : The Use of Previously Calculated Data in a Pipeline

When this pipeline array is partitioned in parallel using a block partition, the array will be equally divided between the processors. If there are N number of processors and each of these processors is distributed with three indices of the array then the partition array will be as in Figure 1.10.

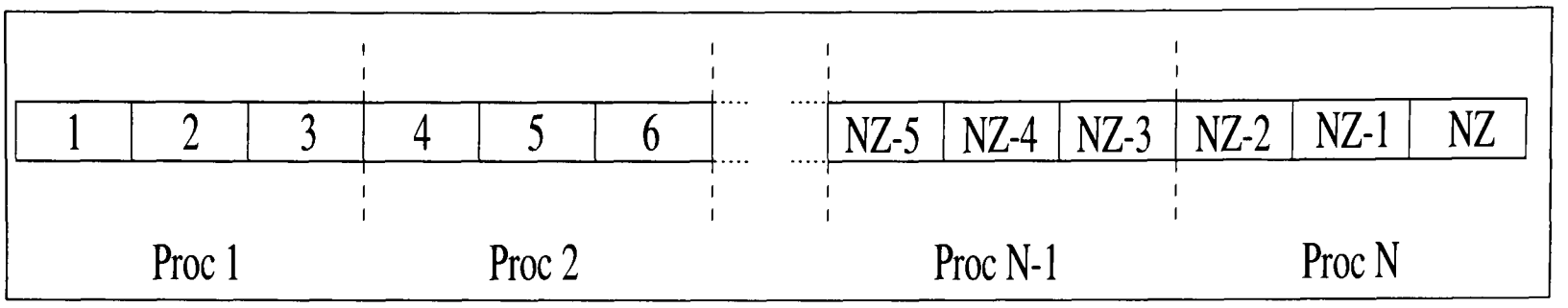


Figure 1.10 : A block partitoned pipeline.

Figure 1.11 shows the code for a simple serial recurrence (Figure 1.8) after parallelisation. The loop L has been partitioned using CAP_LOW and CAP_HIGH as described in Section 1.7. The aim of partitioning this loop is to allow each processor to calculate its own portion of the array concurrently and thus reduce the computation time. However, the calculation within this loop is dependent on having the value of the previous array index. This therefore condemns the processors to operate in a serial fashion. Figure 1.12 displays diagrammatically how this pipeline operates. The first processor will calculate for the range of indices it owns (i.e. 1 to 3) before communicating the required data (index 3) to the next processor. The next processor then receives this data before allowing calculation to be executed. Once this calculation has been executed the processor will then communicate to the next processor the data it requires.

```
A(1)=A_INITIAL
CALL CAP_RECEIVE(A(CAP_LOW-1),1,2,CAP_LEFT)
DO L=MAX(2,CAP_LOW),MIN(NZ,CAP_HIGH)
  A(L)=A(L)+A(L-1)
ENDDO
CALL CAP_SEND(A(CAP_HIGH),1,CAP_RIGHT)
```

Figure 1.11 : A Simple Serial Recurrence that has been parallelised.

The recurrence calculation causes the processors to operate serially. The pipeline will also take longer than the original serial code since there is now the additional overhead of communication between the processors. For a parallel machine with a high communication latency, the communication time will extend the total time of the pipeline significantly. If there is little computation within the loop then the communication time may completely dominate the time taken. Obviously if a parallel machine with a low communication latency is used then the time taken by the pipeline will be reduced, but will still take longer than the serial code. The

communication will never be instantaneous thus an additional time penalty will always be incurred when parallelising a recurrence calculation.

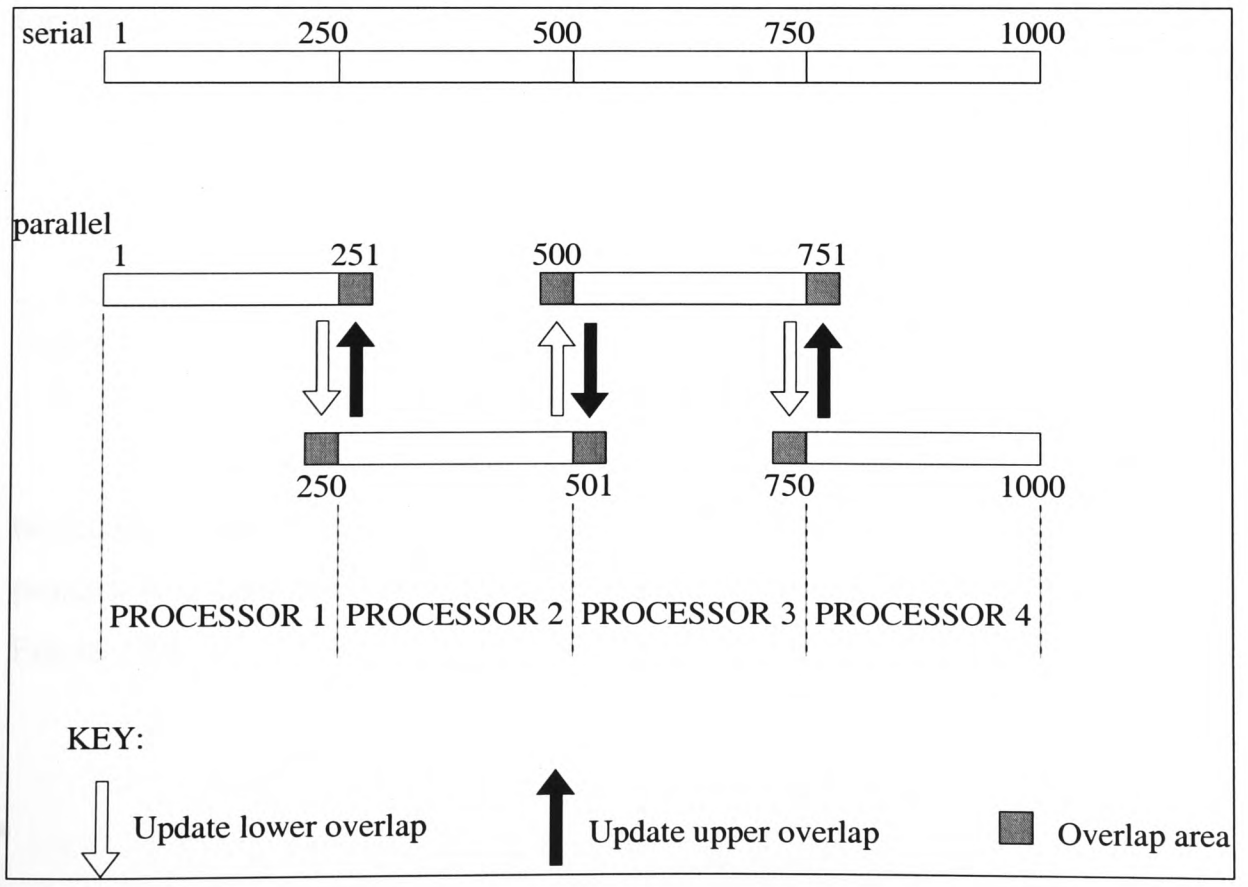


Figure 1.12 : A Serial Pipeline.

Each processor of the pipeline also has a substantial amount of idle time associated with it. The first processor of the pipeline (Figure 1.12) will remain idle once it has calculated and communicated its data to the next processor. Likewise the other processors will experience idle time whilst waiting to receive data from the preceding processor. This idle time incurred whilst the processor is waiting for the data is referred to as the pipeline start-up time, while the idle time after computation and communication is referred to as the pipeline shutdown time. This idle time contradicts the fourth main objective (Section 1.3) of parallelisation that is to make the maximum use of the available processing power.

If there are additional loops surrounding the pipeline (Figure 1.13) then there will be a succession of pipelines that will reduce the proportion of the idle time. It is from these additional surrounding loops that parallelism may be obtained. Figure 1.14 shows a succession of pipelines in operation on three processors. Comparing the proportion of idle time to the calculation and communication of the pipeline in Figure 1.14 against the pipeline in Figure 1.12 is now much less and a significant amount of parallelism may be exploited. If there are more iterations of the outer loops then this proportion will be reduced further.

```
A(1)=A_INITIAL
DO NITER = 1, TOTAL_NITER
      CALL CAP_RECEIVE(A(CAP_LOW-1),1,2,CAP_LEFT)
      DO L=MAX(2,CAP_LOW),MIN(NZ,CAP_HIGH)
        A(L)=A(L)+A(L-1)
      ENDDO
      CALL CAP_SEND(A(CAP_HIGH),1,CAP_RIGHT)
ENDDO
```

Figure 1.13 : A serial recurrence with surrounding loops.

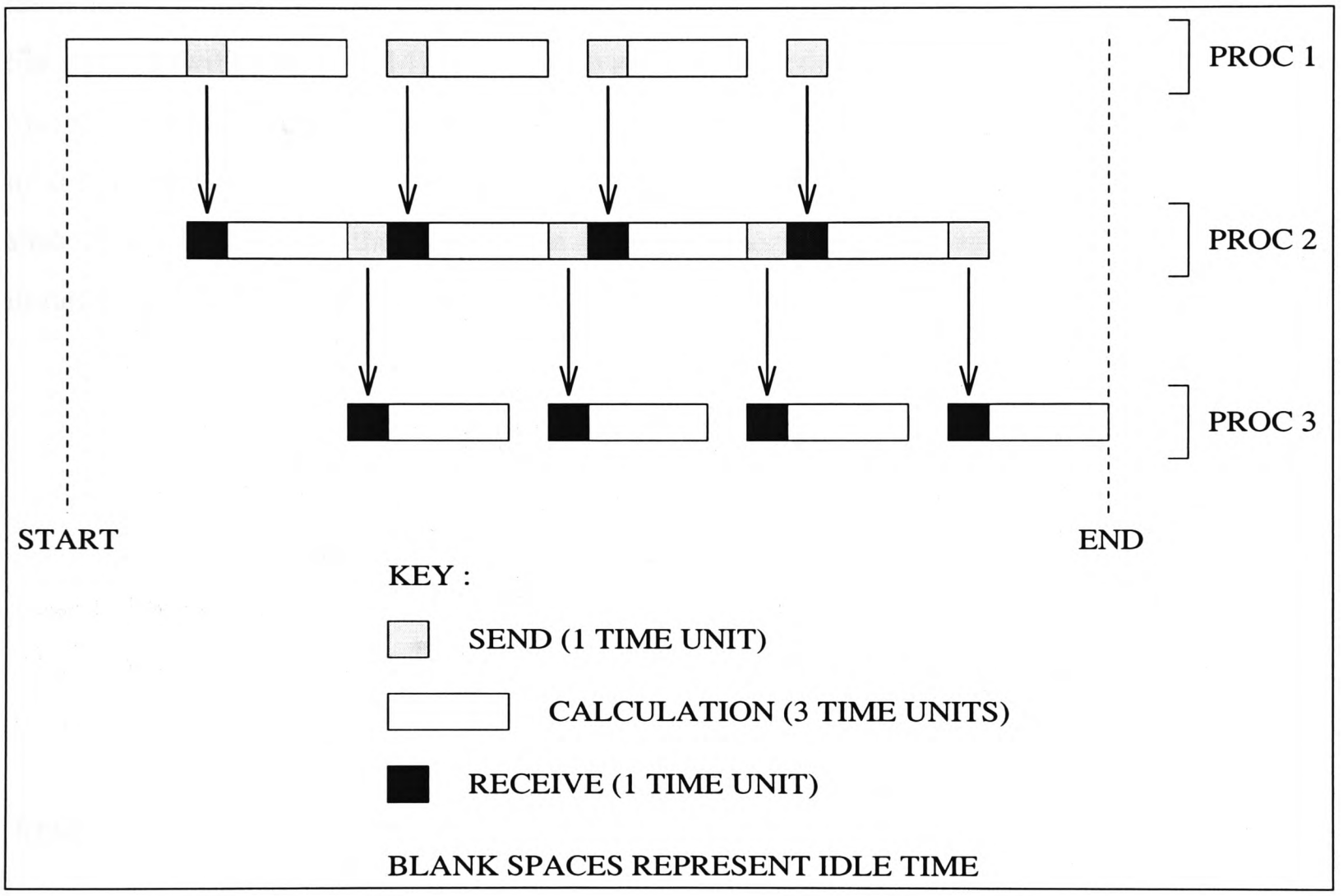


Figure 1.14 : A Succession of Pipelines.

These pipelines will always occur in parallel codes, in order to ensure that the correct results are obtained. They are therefore a necessary evil and this thesis will discuss in further detail, methods of reducing the overheads they incur (Section 4.8).

1.9 Iteration Grouping.

Iteration grouping is one method that may be applied to pipelines to reduce the number of communication start up latencies. This is achieved by increasing the number of calculations

per pipeline cycle. For example, if there were only one calculation per pipeline cycle then there would be one communication latency for every calculation; if there were two calculations per pipeline cycle then there would be only one communication latency for every two calculations; etc.

Figure 1.15 shows that doubling the number of calculations per cycle also causes the communication time to be doubled as well. However, the amount of communication start up latencies will now be half the amount it was before, i.e. 4 instead of 8 on processor 2 (cf. Figure 1.15 with Figure 1.14). One disadvantage of increasing the amount of calculation per cycle is that the actual pipeline start up and shutdown idle time increases. However, as the total number of cycles in the pipeline increases then the proportion of the start up and shutdown idle time to the calculation time will decrease. The application and results of iteration grouping are discussed later in Section 3.4.

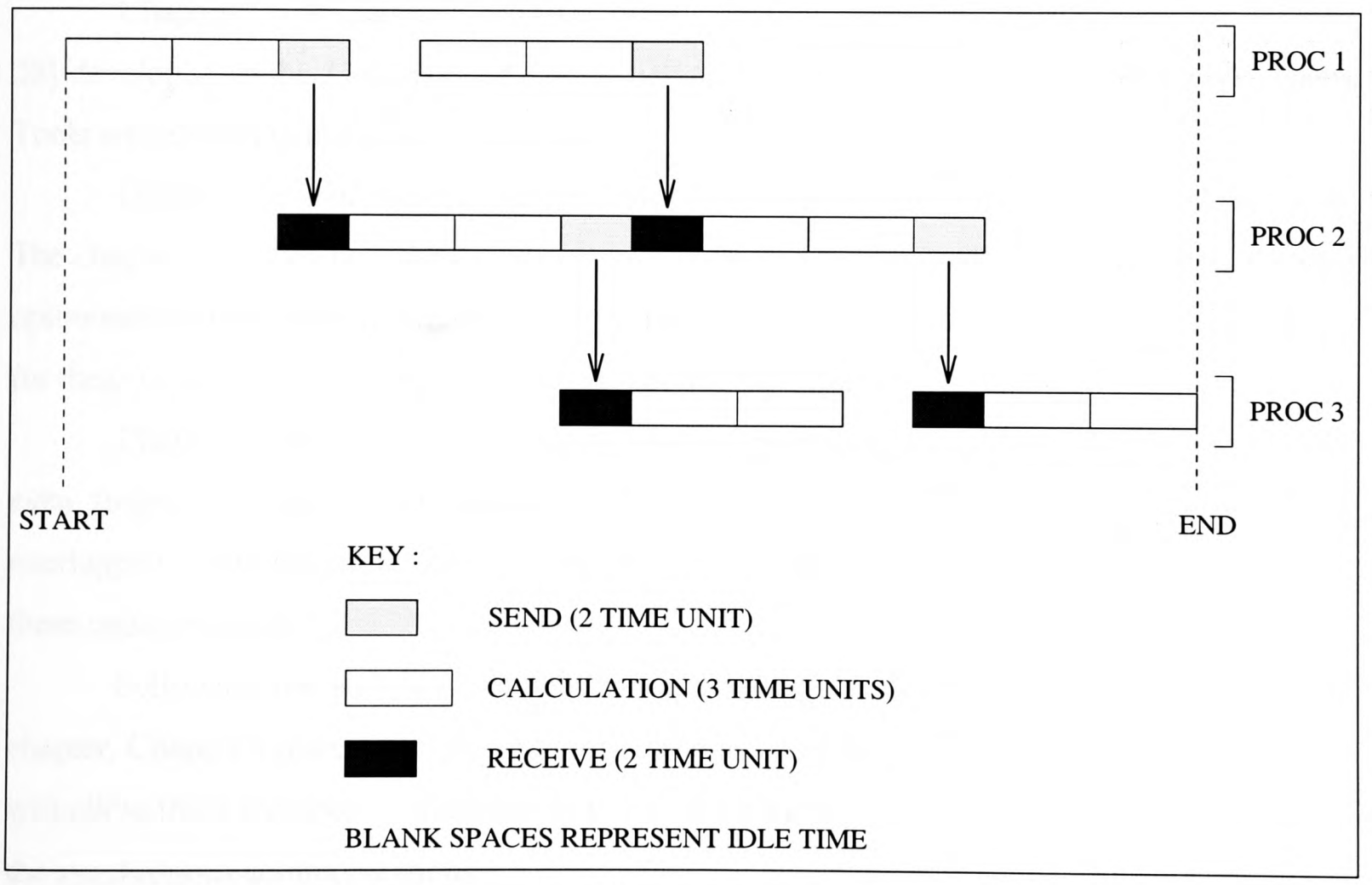


Figure 1.15 : Iteration Grouping in a Pipeline.

1.10 Research Objectives.

One of the main objectives of this research was to achieve improved performance from automatically generated parallel code from Computer Aided Parallelisation Tools

(CAPTools). In the first instance this was achieved by improving the automatically generated parallel code from CAPTools by applying by hand the overlapping of communications with calculation using asynchronous communications. This was tested on a test case of four codes. From this premise it was then possible to formulate a general formal model that was pursued in the incorporation of automatically generated overlapped communications as an additional stage within CAPTools. This additional stage within CAPTools was tested on several other codes. Initially these methods were applied for structured mesh based codes. A further objective was to investigate whether these methods were also applicable to unstructured mesh based codes.

1.11 Outline of Thesis.

Chapter 2 will discuss Computer Aided Parallelisation Tools (CAPTools) [26, 27, 28, 29] developed at the University of Greenwich to automatically generate parallel code. These Tools are referred to throughout this work.

Chapter 3 investigates the parallelisation of four structured mesh codes using CAPTools. The chapter discusses how these codes were parallelised using CAPTools along with any other optimisations that were applied to obtain improved efficiencies. Results will also be presented for these parallelisations using synchronous communications.

Chapter 4 moves on to investigate ways of increasing the performance of these codes even further by applying overlapped communications. Four different methods of applying overlapped communications were investigated and are discussed. These methods are applied to these codes by hand.

Following the successful application and testing of the four methods in the previous chapter, Chapter 5 discusses their implementation as an additional stage within CAPTools. This will allow these methods of overlapping to be automatically generated by CAPTools to replace the synchronous communications.

Chapter 6 provides the results obtained from CAPTools using synchronous communications and for overlapped communications.

Chapter 7 investigates the parallelisation of unstructured mesh codes. The chapter also discusses methods to automatically generate overlapping communications for these types of codes.

Chapter 8 provides a conclusion to the work investigated.

1.12 Conclusions.

This chapter has provided a basic understanding of the concepts of parallelising codes. It has also defined some of the problems associated with parallel processing. The subsequent chapters will attempt to resolve some of these problems, and be implemented for automatic generation within Computer Aided Parallelisation Tools.

Chapter 2

2 Computer Aided Parallelisation Tools (CAPTools).

This chapter explains in further detail the aims of the parallelisation tool, Computer Aided Parallelisation Tools (CAPTools) and the stages of the process of obtaining parallel code. Other parallelisation tools, as well as CAPTools, were discussed briefly in Section 1.2.

The whole process of automatically generating parallel code will be explained briefly along with a more in depth explanation of how each stage is accomplished. The embedding of the automatic generation of overlapping communications within CAPTools will require the use of various data structures from each of these stages.

2.1 CAPTools.

CAPTools [26, 27, 28, 29] is targeted at facilitating the generation of efficient parallel FORTRAN 77 code with explicit communication calls. Although the tools are designed for the parallelisation of any application, the initial focus of attention of CAPTools is for structured mesh based FORTRAN numerical codes such as Computational Fluid Dynamics, heat transfer and structural analysis.

The main aim of CAPTools is to produce a parallel code adhering to all the five requirements of parallel processing outlined in Section 1.3. Using CAPTools, it is possible to reduce the time taken to parallelise code from weeks or months, to just days or even hours.

2.2 Using CAPTools to parallelise a Structured Mesh Computational Mechanics Code.

The stages involved to produce a parallel code using CAPTools are as follows:

1. Serial Fortran code is loaded into CAPTools (Section 2.3).
2. A detailed dependence analysis of the serial code is calculated (Section 2.4).

3. A data partition for one array is prescribed by the user and inherited throughout the code (Section 2.6).
4. Execution control masks are generated (Section 2.7).
5. Calculation, Migration and Merging of communications (Section 2.8).
6. Generation of communications (Section 2.9).
7. Final code generation (Section 2.10).

Each one of these steps will be discussed in further detail in this Chapter.

2.3 Loading the Serial Code.

The very first stage in using CAPTools is to read in the serial Fortran 77 code. This will involve a basic parsing of the code and for a parse tree, symbol table, routine call graph and a control flow graph to be constructed.

The parse tree consists of nodes used to represent the source code being parallelised. The parse tree is constructed as binary trees with each node representing a symbol table entry (SYMBOL) with a left and right branch pointing to the next nodes. Each routine has its own symbol table. Figure 2.1 shows an example of a simple parse tree from CAPTools for an assignment statement A = A + 2.

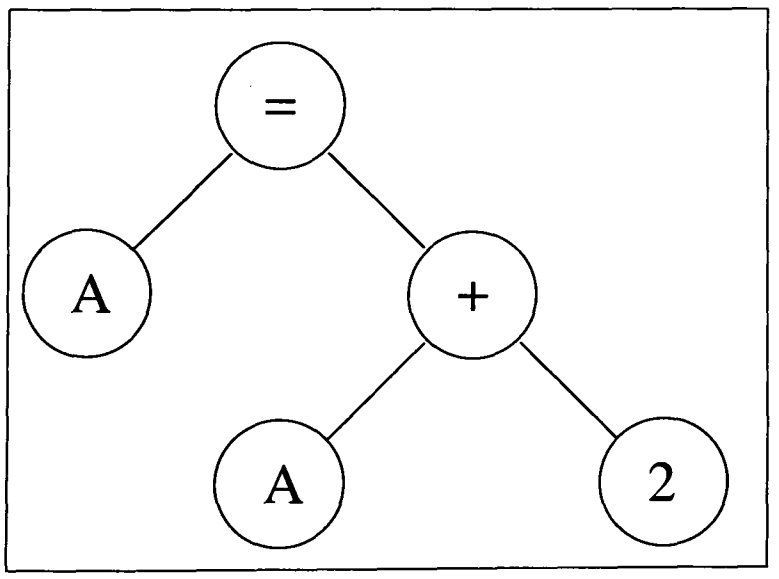


Figure 2.1 : A parse tree from CAPTools representing an assignment statement.

2.3.1 Call Graph.

The call graph consists of nodes each representing a routine. A node is connected to another node if a routine calls another routine. The call graph is assembled by identifying all calls to routines in the parse trees and matching them with the relevant routine header. The

strict order of the call graph is composed by performing a depth first search from the main program identifying every routine call. A routine is added to the ordered list only when every routine called by that routine has been processed. This provides a strict order of routine calls for the program that allows the Fortran code to be outputted in the same order as inputted. This strict order call graph may then be used for interprocedural analysis of the dependence graph. It will also be relevant when determining the path by which communications and any communication synchronisation points may proceed when being migrated (Section 2.8.3). The strict ordering of the routines is employed when traversing through the routine boundaries during an interprocedural traversal.

Each routine node (ROUTINE) holds information on other routines that a particular routine calls (CALLS) and a list of routines that have called this routine (CALLEDBY). Each ROUTINE also stores the next routine in the order as they were read from the input file (NEXT) and a strict order where each routine is listed only after all routines it references have already been listed (STRICT). The pseudo code in Figure 2.2 shows how CAPTools uses this data structure to traverse the call graph, which in this case are the routines being called.

```
CALLS := ROUTINE^.CALLS
WHILE (CALLS <> NIL) DO
    BEGIN
        :
    CALLS := CALLS^.NEXT
    END
```

Figure 2.2 : Pseudo code to traverse the call graph.

2.3.2 Control Flow Graph.

The control flow graph consists of nodes which represent a group/block of statements (known as a basic block) with directed control flow paths from one node to another [53]. These blocks of statements are stored within CAPTools as a BLOCK data structure. Each one of these BLOCK data structures will point to a list of these statements (COMMAND) that belongs to this BLOCK. These statements are grouped into blocks as follows: each DO or IF statement will be placed in a BLOCK of its own; while one or more consecutive assignment statements will be placed into one block. The pseudo code example in Figure 2.3 shows how it is possible to traverse through every statement of every block of every routine in the code as it was in the original input code read into CAPTools.

```
CURRENT_ROUTINE := ROUTINE
WHILE (CURRENT_ROUTINE <> NIL) DO
        BEGIN
        CURRENT_BLOCK := CURRENT_ROUTINE ^.BLOCKTOP
        WHILE (CURRENT_BLOCK <> NIL) DO
                BEGIN
                CURRENT_COMMAND := CURRENT_BLOCK^.COMMAND
                WHILE (CURRENT_COMMAND <> NIL) DO
                        BEGIN
                          :
                        CURRENT_COMMAND := CURRENT_COMMAND^.NEXT
                        END
                CURRENT_BLOCK := CURRENT_BLOCK^.NEXT
                END
        CURRENT_ROUTINE := CURRENT_ROUTINE ^.NEXT
        END
```

Figure 2.3 : Pseudo code to traverse every statement in the input code.

The first block of each routine is stored in the CAPTools data structure ROUTINE as the BLOCKTOP.

Each BLOCK possesses a HASFATHER and a HASCHILD data structure that represents a list of blocks from which flow can have reached a particular block and to which control can flow from the block respectively. Figure 2.4 shows the pseudo code that performs a depth first search (DFS) from a starting block (STARTBLOCK) passing through all blocks marking all reachable blocks down the control flow graph using the HASCHILD of each block. The blocks are marked using the MARKED field of BLOCK which is reserved specifically for this purpose. It is also possible to perform a depth first search up the control flow graph using the HASFATHER of the block

```
PROCEDURE BLOCKDFS(STARTBLOCK)
BEGIN
BLOCK^.MARKED := TRUE
BLOCKLIST := STARTBLOCK ^.HASCHILD
WHILE ( BLOCKLIST  <> NIL) DO
        BEGIN
        IF (NOT BLOCKLIST^.BLOCK^.MARKED) THEN
                BLOCKDFS(BLOCKLIST^.BLOCK)
        BLOCKLIST:=BLOCKLIST^.NEXT
        END
END
```

Figure 2.4 : Pseudo code showing a depth first search of the basic blocks.

Figure 2.5 shows how the statements are divided into blocks. The control flow graph (Figure 2.6) shows how the control flows from one block to another. For example, Block1

may either flow to Block2, i.e. another iteration of the I loop, or flow to block 5, i.e. there are no further iterations of loop I.

```
S1          DO I =                              Block 1
S2              DO J =                          Block 2
S3                  A(I,J)=                     Block 3
S4                  B(I,J)=                     Block 3
S5              ENDDO
S6              C(I)=                           Block 4
S7          ENDDO
S8          IF(CONDITIONAL) THEN                Block 5
S9              C(1)=                           Block 6
S10         ELSE
S11             C(1)=                           Block 7
S12         ENDIF
S13         IF (CONDITIONAL) THEN               Block 8
S14             GOTO 10                         Block 9
S15         ENDIF
S16         A(1,1)=                             Block 10
S17         A(1,N)=                             Block 10
S18     10  CONTINUE                            Block 11
S19         B(1,1)=                             Block 11
S20         B(1,N)=                             Block 11
```

Figure 2.5 : Code to Demonstrate Control Flow

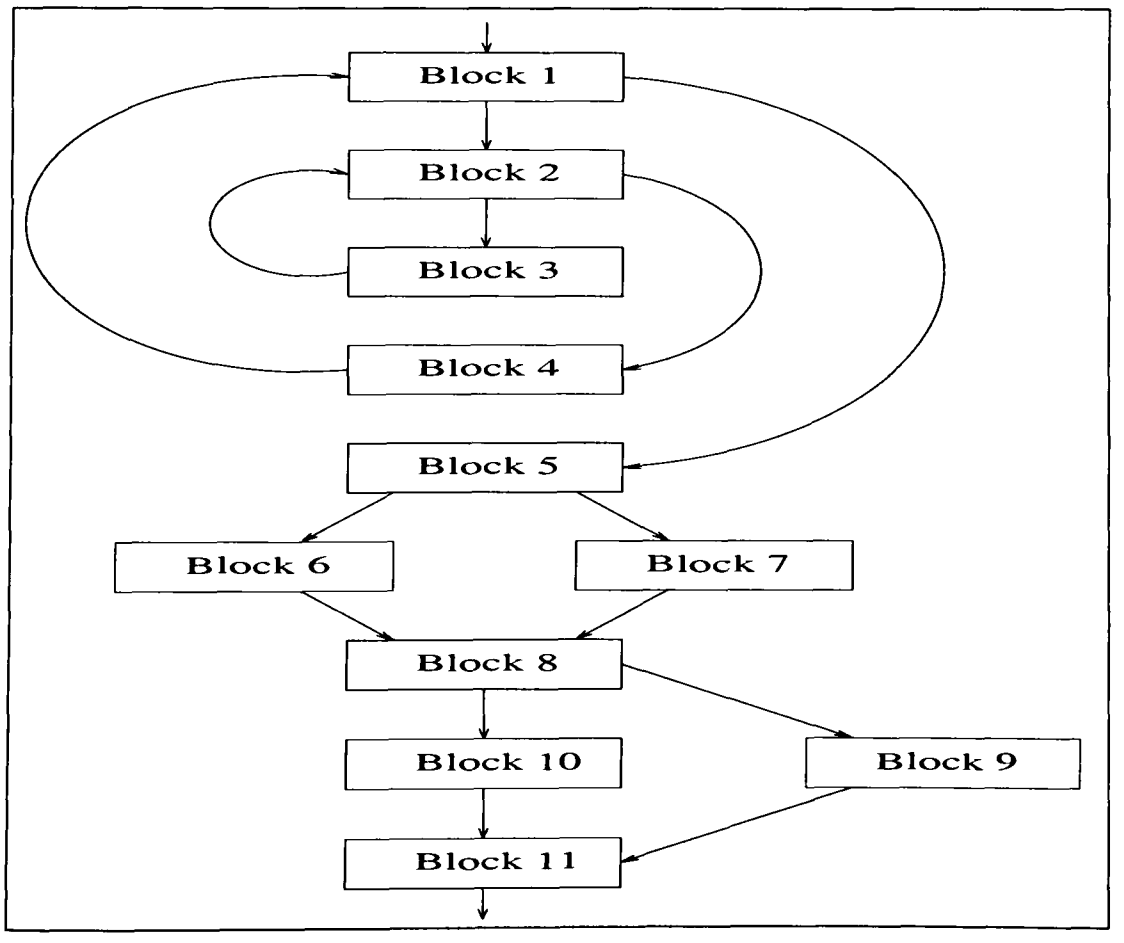


Figure 2.6 : Control Flow Graph.

Incorporated into the basic blocks are the post- and pre- dominator trees of the control flow graph. Post-domination indicates that a statement S_1 post-dominates a statement S_2 if no control flow path to the routine end exists from S_2 that does not pass through S_1. Predomination indicates that a statement S_1 pre-dominates statement S_2 if the control flow must pass through S_1 to reach S_2, i.e. no other route exists to S_2 that does not pass through S_1 [54]. The predomination graph and post-domination graph for the control flow graph in Figure 2.5 is shown in Figure 2.7 and Figure 2.8 respectively. Each block has its own unique immediate pre- and post- dominator. The pre-dominators and post-dominators of a block may be found, within CAPTools, by traversing up the appropriate pre or post dominator tree which is stored in the BLOCK data structure of CAPTools.

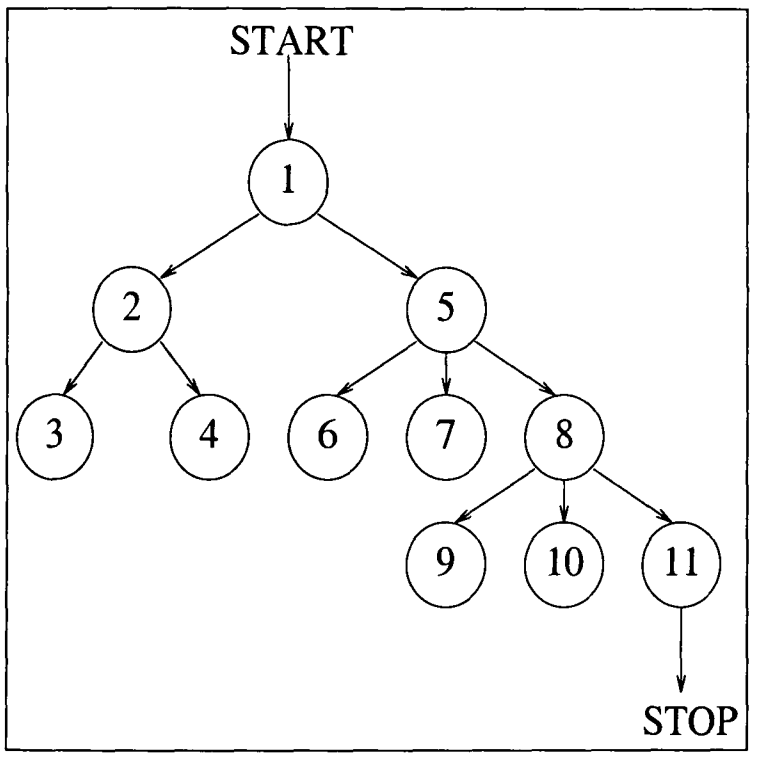


Figure 2.7 : Predomination Graph.

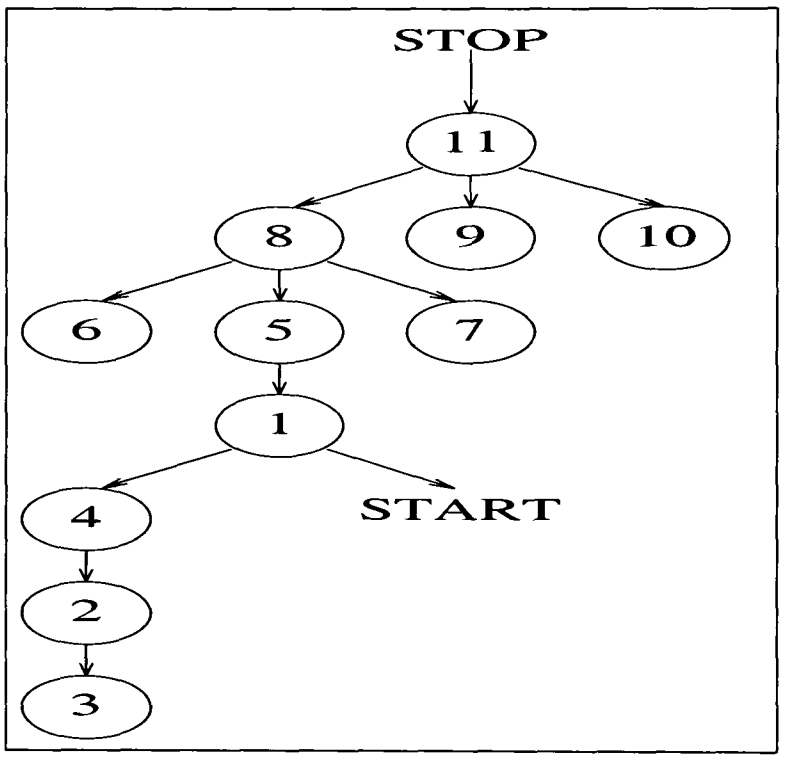


Figure 2.8 : Postdomination Graph

Using these graphs and data structures that are provided from CAPTools it is possible to traverse up the control flow graph to find the optimum position for placement of communications, synchronisation points, etc. For instance, if a communication were required before a certain statement then it would be migrated up the pre-dominator tree since this would guarantee execution before that statement. The pseudo code in Figure 2.9 shows how this would be accomplished within CAPTools.

```
PREDOMINATING_BLOCK := BLOCK
WHILE (PREDOMINATING_BLOCK <> NIL) DO
        BEGIN
          .
          .
        PREDOMINATING_BLOCK := PREDOMINATING_BLOCK^.PREDOM
        END
```

Figure 2.9 : Pseudo code showing a traversal of the pre-dominator graph in CAPTools.

If the BLOCK on the first line in Figure 2.9 is BLOCK11 from Figure 2.5 then the code will traverse the pre-domination graph (Figure 2.7) passing through BLOCK8 and BLOCK5 before reaching BLOCK1 where the command is a DO statement. A similar method may be used to traverse the post-domination graph.

During traversal, any barriers to movement (such as the assignments of the data to be communicated) must be detected between the current control flow graph block and its immediate pre-dominator in any control path before traversal is legal.

Each one of these blocks also holds a list relating to the loop nestings (NESTING) surrounding that basic block. For example, in Figure 2.5 Block 3 will have two loop nesting surrounding that block, i.e. Block 1 and Block 2. Figure 2.10 shows how this NESTING information is stored within CAPTools.

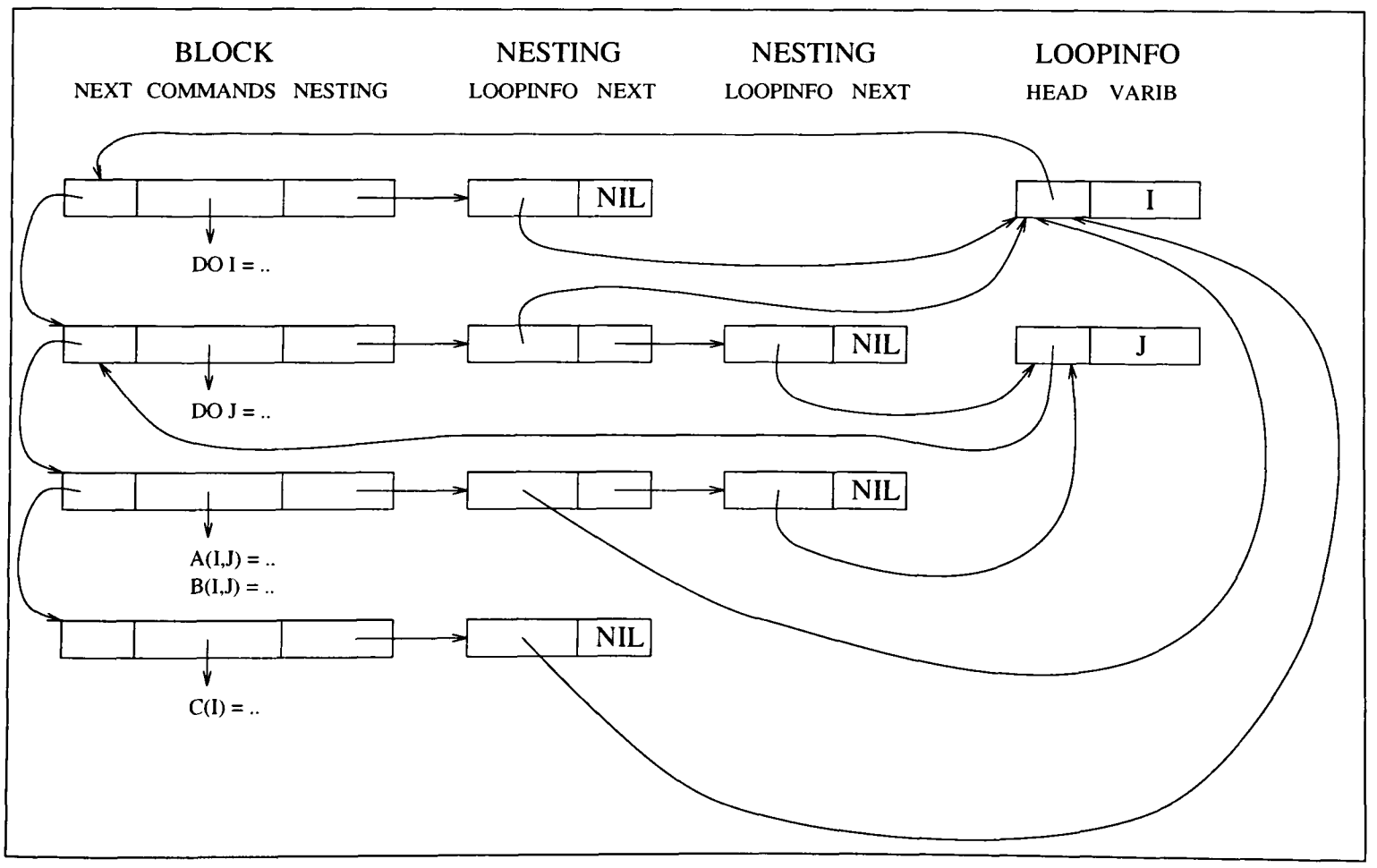


Figure 2.10 : The data storage of NESTING within CAPTools.

Each NESTING holds information regarding the surrounding loop (LOOPINFO) and a pointer to the next NESTING. The LOOPINFO holds information pointing to the loop limits LOWTREE and HIGHTREE and the step of the loop STEPTREE as well as the pointer to the HEAD – the basic block containing the loop.

2.4 Dependence Analysis.

The nucleus of CAPTools is its powerful dependence analysis that extracts and uses information from the code using the control statements and loop limits. The need for user interaction is imperative to ensure that the dependence analysis obtained is as accurate as possible. The user interaction is extensive so that the user may at any stage of the parallelisation, query and provide information. The user may, prior to the dependence analysis, submit any additional information such as the value of variables that are read in at runtime. This information may aid in minimising possible control flow paths and also remove any dependencies that would otherwise have been assumed to exist. This would lead to an enhanced dependence analysis.

From the dependence analysis a dependence graph is obtained. This dependence graph consists of nodes, which represent executable statements and directed edges that flow from node to node and represent the dependencies between statements

2.4.1 Dependence Types.

There are four basic types of dependencies [55]:

True Dependence - This is the data flow from the assignment statement (source) to the usage statement (sink). The source statement must obviously be executed before the sink statement. Consider the following example:

S_1 A(I) = ...

S_2 ... = A(I)

A true dependence exists between statement S_1 which assigns the values of A and statement S_2 which uses the data. A true dependence can also be marked as exact when every memory location accessed in the sink reference is also accessed in the source.

Anti Dependence - This is when the data is being reassigned from the usage of the same data. The statement that uses the data must therefore occur before the statement that

reassigns that data. The source is in effect overwritten by the sink. Consider the following example :

S_1 ... = A(I)

S_2 A(I) = ...

An anti dependence exists since the data used in statement S_1 is reassigned in statement S_2.

Output Dependence - This is when data is being reassigned after being previously assigned. Consider the following example :

S_1 A(I) = ...

S_2 A(I) = ...

The data in statement S_1 is simply reassigned in statement S_2. This is a common method used in many codes to reuse memory location to reduce the memory overheads.

Control Dependence - This is when a control statement, such as an IF, controls the execution of other statements. Consider the following example :

S_1 IF (conditional) THEN

S_2 A(I) = ...

The statement S_2 is controlled by the statement S_1. The statement S_2 may not execute until statement S_1 has been proved either true or false.

2.4.2 Depth dependence.

Another attribute of dependencies to consider is whether they are carried by loops. These are dependencies for data assigned in one iteration being used in a consequent iteration of the same loop.

Each dependence type also possesses a depth. A dependence may be **Loop Independent** if it exists within a single iteration of all surrounding loops. For example :

```
DO I=1,100
        DO J=2,99
                A(I,J) = ...
                ... = A(I,J)
        ENDDO
ENDDO
```

i.e. the value of A(I,J) was assigned and used in the same iteration.

If a dependence exists between iterations of the outermost loop of the surrounding statement/s then it is deemed to be **Level One**. For example :

```
DO K=1,100
        DO J=2,99
                DO I=1,5
                        A(I,J,K) = A(I,J,K-2)
                ENDDO
        ENDDO
ENDDO
```

i.e. the values used in each iteration was assigned two iterations earlier of the outermost K loop.

If a dependence exists between iterations of the next outermost loop of the surrounding statement/s then it is deemed to be **Level Two**. For example :

```
DO K=1,100
        DO J=2,99
                DO I=1,5
                        A(I,J,K) = A(I,J-1,K)
                ENDDO
        ENDDO
ENDDO
```

i.e. the value used in each iteration was assigned in a previous iteration of the J loop. This process may continue for every other loop.

These loop carried dependencies can cause a loop to be serial, often resulting in a pipeline (Section 1.8). These pipelines are caused (as mentioned earlier in Section 1.8) by the use of data calculated in a previous iteration. These serial loops are detected by CAPTools by the occurrence of a loop carried true dependence.

Within CAPTools data structures every executable statement (COMMAND) stores all dependencies for that statement. Each dependence data structure stores the information for its Level of dependence (DEPTH), its TYPE, i.e. True, Anti, etc, and the VARIABLE that causes that dependence.

2.4.3 Example of a Dependence Graph.

Figure 2.11 shows a dependence graph for a section of the Jacobi solver code (Section 1.7). There are 6 statements which have dependencies involving the array variable T. There are 5 loop independent true dependencies from the statement 9 (where the values of the array T are initialised) to statements 14, 16 17 and 20 (where the value of T is used in calculation). There are also 5 true dependencies from statement 24 to statements 14, 16, 17 and 20. These true dependencies are dependent on the outermost loop (40 CONTINUE), i.e. Level 1 loop dependence.

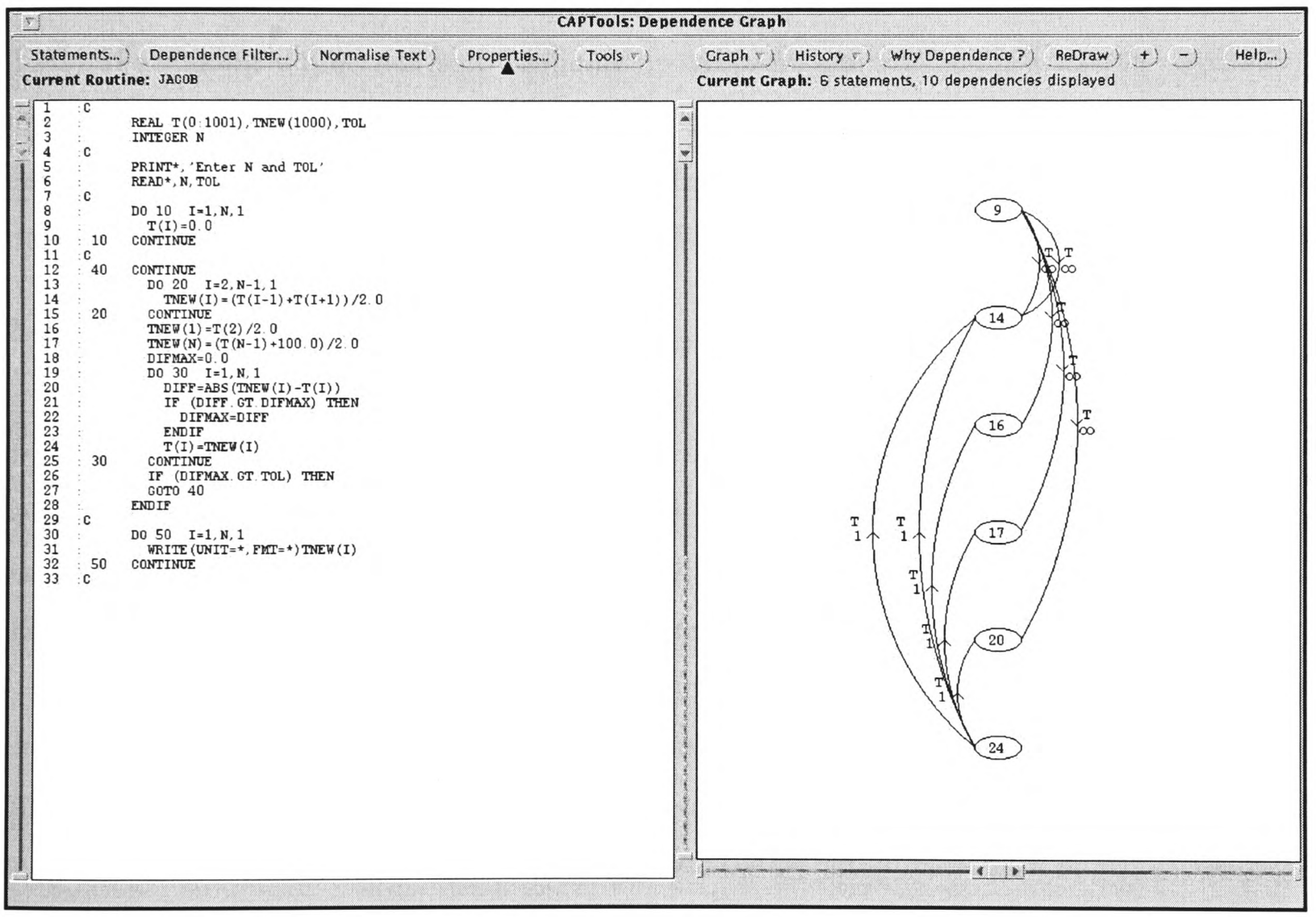


Figure 2.11 : Dependence Graph of the Jacobi Code.

2.4.4 Loop Normalisation.

Prior to the analysis of the dependence graph it is advantageous to normalise all DO loops. Normalisation consists of transforming all DO loops to start from 1 and to increment in steps of 1. The normalisation of these loops then leads to the need to transform variables, which have constant increments in every iteration of these normalised loops, to be a function of the normalised loop variable. Induction variables, i.e. variables which have constant

increments in every iteration of a particular loop, are identified and transformed to be functions of the loop variable concerned [56, 57, 58]. These transformations are not essential but they do simplify the analysis process, code generation and asynchronous code generation stages. These transformations are easily reversible during the code generation stage to ensure original code recognition [28].

2.4.5 Control Dependence Calculation.

Prior to the calculation of the dependence analysis the control dependencies (Section 2.4.1) are calculated using the post-domination graph of the control flow graph (Section 2.3.2). If a statement does not post-dominate its father statements then it is control dependent on those fathers [54]. The control dependence calculation algorithm searches up the post-domination graph until a common post-dominator is reached. All the blocks that were traversed then contain statements that are control dependent on the father block.

2.4.6 Dependence Analysis.

The dependence analysis first performs a basic dependence calculation. This analysis consists of a scalar and array analysis. A scalar variable can be a DO loop counter variable, whose value will always be defined within the loop, otherwise it is deemed as a nonloop variable. The values of the nonloop variables will always be defined by previous statements. These values may be determined by the true dependencies of the scalar dependence graph.

For an array analysis, the examination and determination of possible equality of array index expressions determines if a dependence exists. Due to the conservative nature of the algorithm to obtain a correct dependence graph, a dependence is set unless its non-existence may be proved. A dependence may be determined by the array references of the assignment and usage statements. From this, a set of equations and constraints may be determined, to which dependence tests are applied to attempt to prove that a dependence does not exist.

These tests include the Greatest Common Divisor test (GCD) [58], the Banerjee Inequality Test [58, 59, 60] and the Symbolic Inequality Disproof Algorithm (SIDA) [61, 27]. The GCD test obtains a solution to the equations based on the fact that they have only integer variables and integer coefficients. The Banerjee inequality test makes use of all variable range information to prove that a dependence does not exist.

The first two tests mentioned make use of information about the loop iteration variables. However, the inclusion of other variables, such as nonloop variables (also referred to as symbolic variables) in index expressions, loop limits and conditional statements, prevents an accurate analysis (Section 2.5). Consider the pseudo code in Figure 2.12 where the variable M used in statements S_4 and S_6 have the same defining statements but each has a different call path. The use of variable M in statement S_4 is defined in statement S_3 but has the call path S_1. Meanwhile, the use of the variable M in statement S_6 also has the defining statement S_3 but the call path of S_5. The two references to the variable M in subroutine SUB1 therefore have different values since their call paths to the defining statement of the variable are different.

```
S1      CALL INITIALISE (N)
S2      CALL SUB1(A, N)

        SUBROUTINE INITIALISE (K)
S3      K = ..
        END

        SUBROUTINE SUB1(A, M)
S4      A(M, 1) = ...
        DO J = 1, NJ
S5          CALL INITIALISE(M)
S6          A(M, J) = ...
        ENDDO
        END
```

Figure 2.12 : Pseudo code showing two different call paths for a defining statement.

2.4.6.1 Symbolic Inequality Disproof Algorithm

The Symbolic Inequality Disproof Algorithm (SIDA) test attempts to prove nonloop variables inequalities to be false. The algorithm makes use of information already known in a linear combination that matches the set of nonloop variables being tested. For example:

Test NONLOOPS + K >= 0 where K is a constant

A linear equation of known inequalities produces :

NONLOOPS + C <= 0 where C is a constant

The SIDA test can then be performed to eliminate all nonloop variables :

NONLOOPS + K >= 0 >= NONLOOPS + C

K >= C

If the inequality involving only known constants is true then the original test :

NONLOOPS >= 0 is false

For example, to prove that N + 2NM >= 10 requires the test to disprove that :

N + 2NM –10 < 0

The knowledge base consists of two relevant known inequalities N – 2 >= 0 and N – M <= 0. Since the original equation consists of a nonlinear term NM further inequalities are required to find a solution. These additional inequalities, for this example, may be obtained by multiplying the first inequality from the knowledge base with itself and multiplying the two inequalities together. This provides the following four inequalities :

N – 2 >= 0

N – M <= 0

N^2 – 4N + 4 >= 0

N^2 – NM – 2N + 2M <= 0

Taking these inequalities a matrix system A $\underline{k}$ = $\underline{b}$ is constructed. The vector $\underline{b}$ is constructed using the coefficients of the nonloop variables in the test inequalities. In the matrix A each column represents the coefficients of the nonloop variables in a known inequality, where each row represents a nonloop variable, matching those in the $\underline{b}$ vector. Any nonloop variables not in $\underline{b}$ are appended to $\underline{b}$ with a zero coefficient.

Also constructed are vectors $\underline{c}$ and $\underline{s}$ which store thes constants and signs respectively of each known inequality (i.e. columns of A) where (<=, =, >=) are represented by (-1,0,1) in $\underline{s}$.

For the four inequalities above the following matrix system and vectors are constructed :

$$\underset{A}{\begin{pmatrix} 1 & 1 & -4 & -2 \\ 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{pmatrix}} \underset{\underline{k}}{\begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{pmatrix}} = \underset{\underline{b}}{\begin{pmatrix} 1 \\ 2 \\ 0 \\ 0 \end{pmatrix}} \begin{matrix} N \\ NM \\ M \\ N^2 \end{matrix} \qquad \underline{c} = \begin{pmatrix} -2 & 0 & 4 & 0 \end{pmatrix} \quad \underline{s} = \begin{pmatrix} 1 & -1 & 1 & -1 \end{pmatrix}$$

The solution of this system produces the coefficients for the linear combination of the known inequalities required to eliminate the variables in the inequality being tested :

$$\underline{k} = \begin{pmatrix} 9 \\ -4 \\ 2 \\ -2 \end{pmatrix} \qquad \underline{s} * \underline{k} = \begin{pmatrix} 9 \\ 4 \\ 2 \\ 2 \end{pmatrix}$$

Since we are attempting to prove false a less than zero inequality, a positive combination is required. The $\underline{s}*\underline{k}$ vector above shows that all the contributions to the combination have the required sign , allowing the final test of the SIDA algorithm:

$$N + 2NM - 10 < 0 <= 9(N-2) - 4(N-M) + 2(N^2 - 4N + 4) - 2(N^2 - NM - 2N + 2M)$$

$$<= 9N - 18 - 4N + 4M + 2N^2 - 8N + 8 - 2N^2 + 2NM + 4N - 4M$$

$$N + 2NM - 10 < 0 <= N + 2NM - 10$$

Which provides :

$$-10 < -10$$

and thus the final test involves constants only and if false, proving that the original test is false and that the original inequality is true.

2.4.6.2 Inference Engine.

These dependence tests work well to exploit definitely true inequality information. However, much of this information, especially the execution control set of statements will often involve logical operations and logical variables. Vital information such as loop steps and division denominators can definitely never be zero and therefore cannot be used since the information is either greater than or less than zero.

Using an inference engine [62] in conjunction with the SIDA allows this information to be exploited. Every logical variable and inequality in the known information is used to form a literal when the logical expression is converted into clausal form. The inference engine then attempts to prove the clause list false by combining clauses that contain contradictory literals, performing a union on the remaining literals to form a new clause. The false conclusion is reached if an empty clause is formed when two clauses contain single literals that contradict each other. To calculate the contradictory literals between two inequalities to be false requires assuming one of these inequalities is true. Adding this inequality to the knowledge base then enables the other inequality to be proved false. This indicates that one of these literals, either the assumption or the second literal, to be false and satisfying the contradictory literal requirement.

Consider the following code example :

```
DO I = 2, N, S
        A(I) = A(I) + ...
ENDDO
```

Normalising this loop (Section 2.4.4) provides :

```
DO I = 1, (N-2)/S + 1
        A((I-1)*S + 2) = A((I-1)*S + 2) + ...
ENDDO
```

Applying the Banerjee Inequality Test [58, 59, 60] provides the following inequality :

$$-S - (S^- + S)^+ ((N-2)/S + 1) <= 0 <= -S + (S^+ - S)^+ ((N-2)/S + 1)$$

where

$S^+ = S$ if $S > 0$ or otherwise $S^+ = 0$

$S^- = S$ if $S > 0$ or otherwise $S^- = 0$

Therefore depending on the sign of the variable S :

$S > 0 : -S - S((N-2)/S + 1) <= 0 <= -S$

$S < 0 : -S <= 0 <= -S - S ((N-2)/S + 1)$

$S = 0 : 0 <= 0 <= 0$

The first two cases have contradictions that can be identified using the SIDA test (Section 2.4.6.1). The third case, however, cannot be disproved with the given current set of information. If S is zero then the location of array A used is the same for each iteration as the previous causing a loop carried loop dependence. Since S is the loop step and may not be equal to zero and also S is the denominator in a division the following clauses are added to the knowledge base :

$S > 0$ or $S < 0$

The inference engine is used during the Banerjee inequality to determine the possibility of the variable S having a value of zero. The first test provides the contradictory literals ($S = 0$) and ($S < 0$). A further second test provides the contradictory literals ($S = 0$) and ($S > 0$). The inference engine therefore proves that the variable S cannot be zero and thus the third case of $S = 0$ is removed from the set of Banerjee tests, enabling the non-existence of the loop carried true dependence to be proved.

2.4.6.3 Interprocedural Analysis.

The dependence analysis tests carried out are interprocedural. This is not done using the technique of inlining as this will change the structure of the code, which is in direct contradiction of rule 2 of the parallelisation objectives in Section 1.3. Instead, a mapping is executed between routines using a start and stop node of the routines involved. These start and stop nodes are added to the dependence graph after it has been constructed. All statements within a routine that uses variables that are not defined in that routine but passed in via either the parameter list or common block are joined in the dependence graph to the start node. Similarly, any statements that define variables that are passed out of this routine are connected to the stop node.

A further dependence test being incorporated into CAPTools is the OMEGA test [63]. This test uses the Fourier-Motzkin variable elimination [63, 64, 65] to attempt to determine precisely if a dependence exists. This method is slower, but more accurate, and for the majority of cases is not required.

2.5 Symbolic Variable Manipulation.

One of the most important features of CAPTools is its ability to manipulate symbolic variables. The dependence calculation algorithm makes use of loop iteration variables, but the inclusion of other non-loop variables could prevent accurate analysis.

To enable a more accurate comparison of these non-loop variables, they are defined not only in terms of the symbol of the variable but also as the defining statement of the variable along with the call path from the variable usage to the routine that assigns the variable.

Consider an array index expression, for an array

A((5*M*L)+(2*K)+J+6, (J*M)+5, IP(K+1)+J-1)

where J and K are loop variables and M and N are non-loop variables. This is stored within CAPTools as shown in Figure 2.13, Figure 2.14 and Figure 2.15.

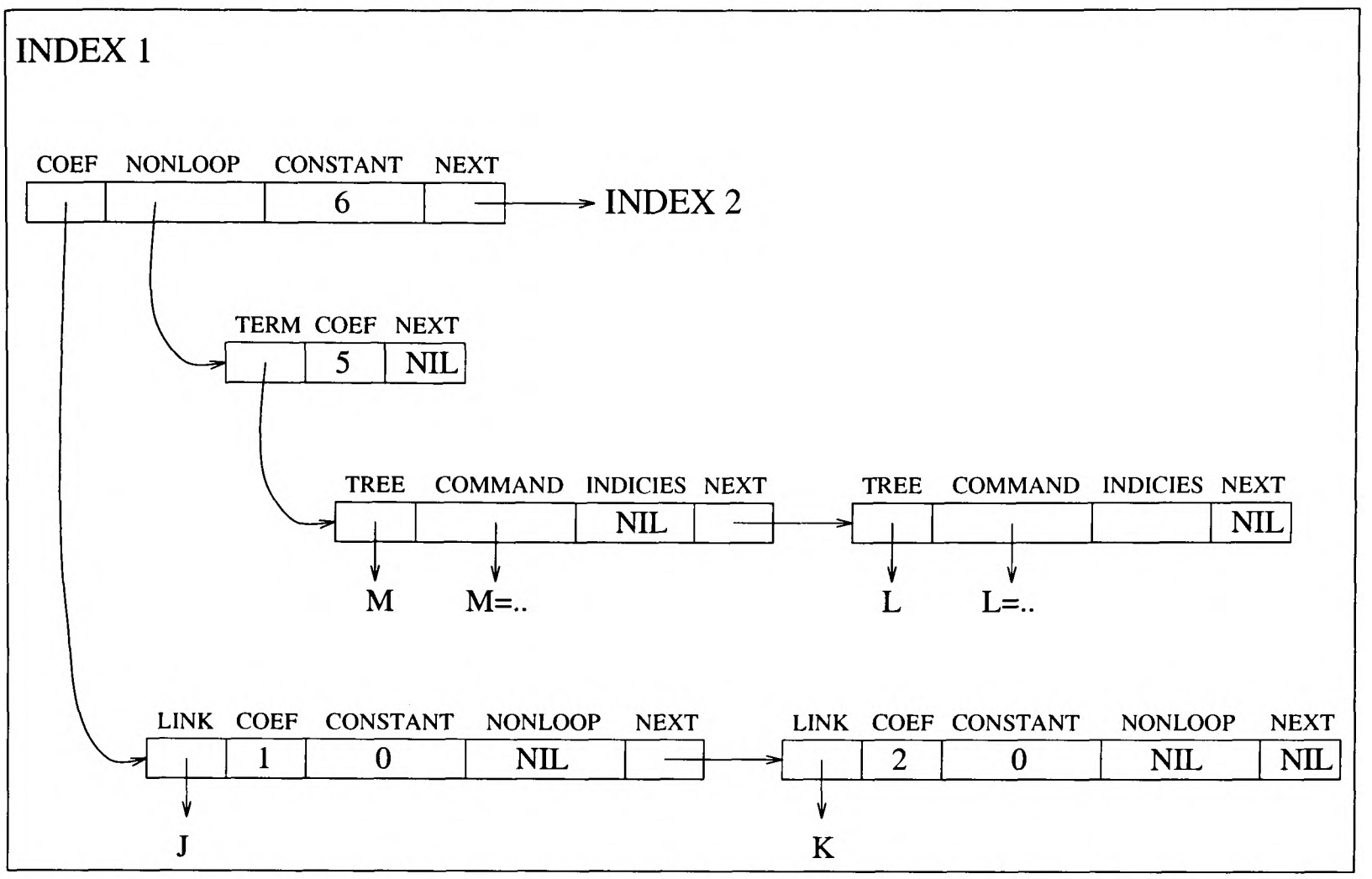


Figure 2.13 : The first index of array A stored within CAPTools.

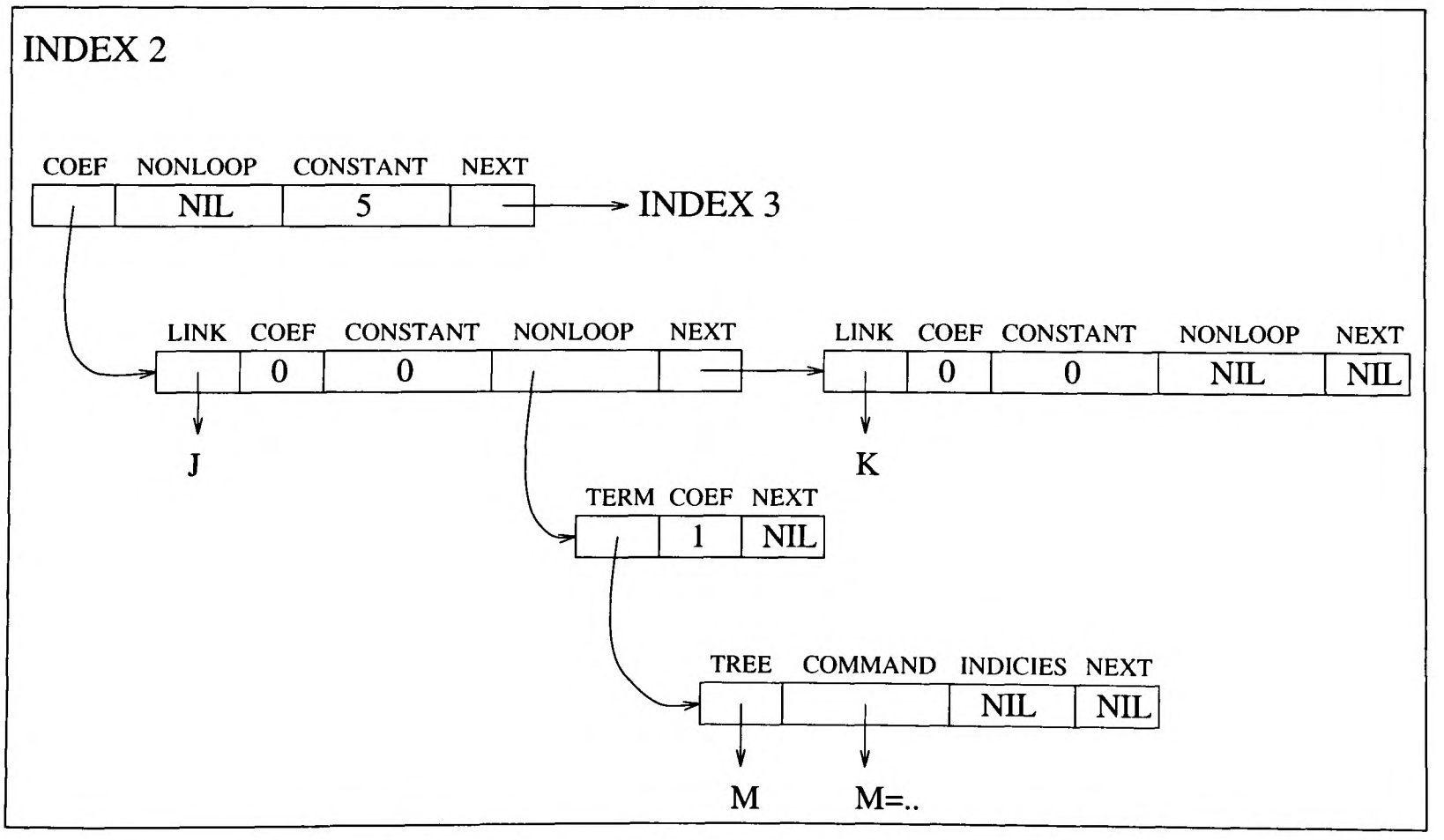


Figure 2.14 : The second index of array A stored within CAPTools.

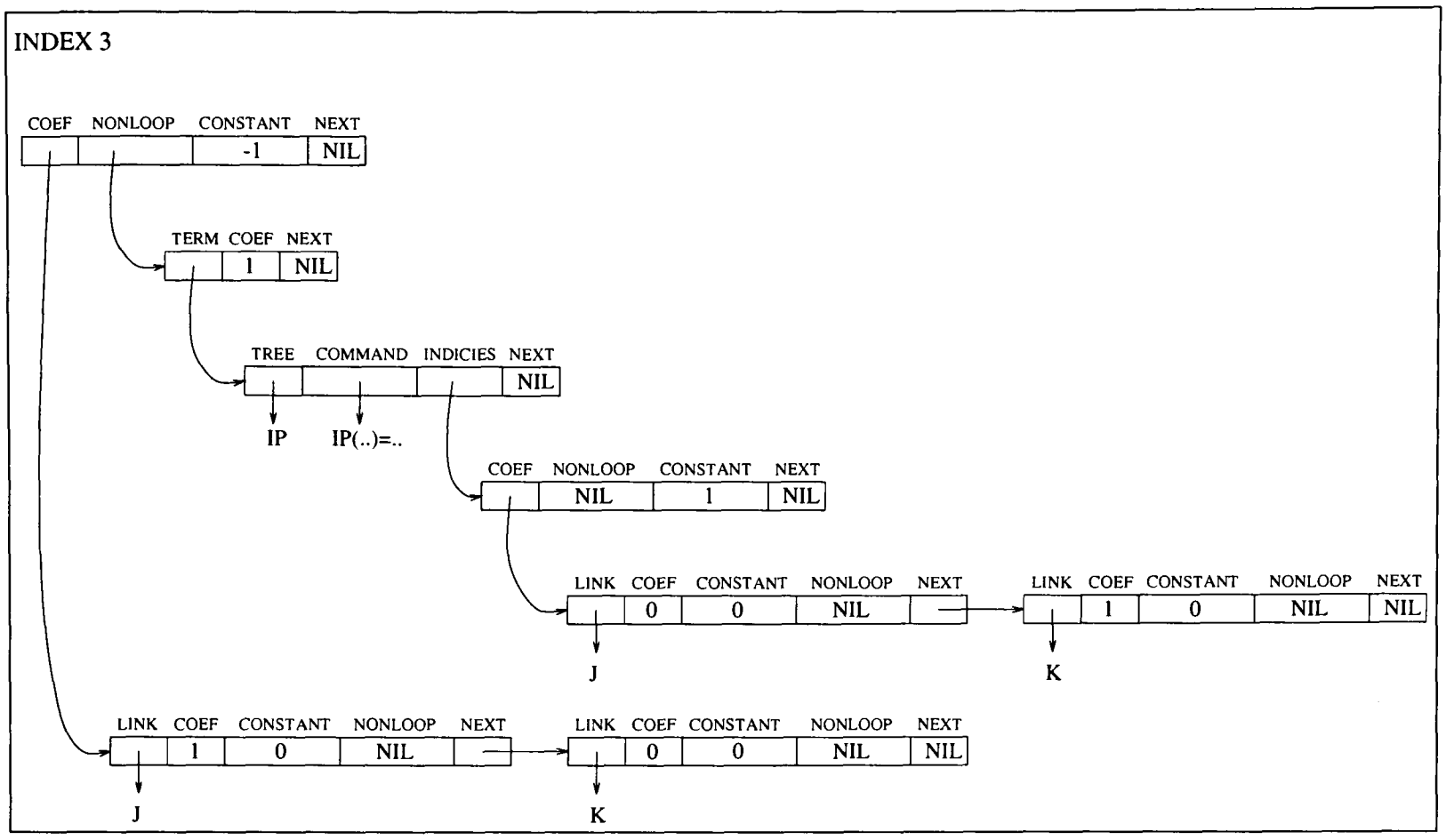


Figure 2.15 : The third index of array A stored within CAPTools.

Figure 2.13 shows the data structure for storing the first index (INDEX) for the array A. This INDEX data structure consists of a data structure for the loop variable coefficients (COEF), nonloop variables (NONLOOP) and any constants values (CONSTANT). For this index, the COEF data structure stores the value of K*2 and J*1; the NONLOOP data structure stores the value 5*M*L; and the CONSTANT stores the value 6. When these components are added together they provide the symbolic variable of the first index of array A i.e. (5*M*L)+(2*K)+J+6. The data is stored similarly for indices 2 and 3 in Figure 2.14 and Figure 2.15.

These symbolic variable data structure may be manipulated within CAPTools by iterating over each INDEX of an array, followed by each COEF and each NONLOOP. While iterating over each of these data structures the symbolic variables may be manipulated using some of the utilities in Section 2.5.1.

2.5.1 Symbolic Variable Manipulation Utilities.

There are several utilities within CAPTools that can be used to manipulate these symbolic variables and their data structures. Some of these routines are :

FORSUBSTITUTE – This utility processes an input symbolic expression or an input parse tree and performs a depth first search of the dependence graph substituting the original symbolic variables with the symbolic variables used to evaluate it. This allows symbolic variables to be converted into a more standard set of defining symbolic variables allowing a comparison between statements to eliminate any unknown symbolic variables;

ADDLIST – This allows two symbolic variable lists to be added or subtracted;

MULTLISTS – This allows two symbolic variable lists to be multiplied together;

LDISPROVE – This processes symbolic variable expressions to determine if they are true, false or cannot be resolved using the SIDA or OMEGA tests (Section 2.4.6);

EXTRACTLOOP – This allows loop variables to be extracted from a nonloop list;

CONTROLFACT – This extracts the control set in clausal form under which the input statement will execute;

FACTORISE – This symbolically factorises one symbolic expression by another symbolic expression

Using these utilities, the algorithms within CAPTools may be used to exploit the symbolic algebra.

2.6 Data Partitioning.

In CAPTools, a partition strategy for a structured mesh may be prescribed simply by defining a routine name, a variable array name and an index or subset of that array [28]. Figure 2.16 shows that for the Jacobi code (Section 1.7) the variable array TNEW, index 1 has been chosen as the base variable for partitioning. CAPTools will then produce a comprehensive decomposition of the mesh with automatic inheritance of partition information to all appropriate variables in all routines. Each processor will have its own partition range which is defined by the CAPTools generated variables CAP_LXXX and CAP_HXXX where XXX represent the variable that the partition is based upon. These partition range variables have their own unique values on each processor as mentioned in Section 1.7.

The inheritance of a data partition may be acquired by an array variable if it is assigned or used by a partitioned array if a linear relationship exists of the index expression

of the partitioned component of the array. This may be inferred either directly or transitively through dependencies. For example, consider the following statement:

A(I,K,J) = B(I,J,K)

If the array B had been partitioned in index 3 then the array A would be partitioned in index 2 due to the linear relationship that exists between them by the loop variable K.

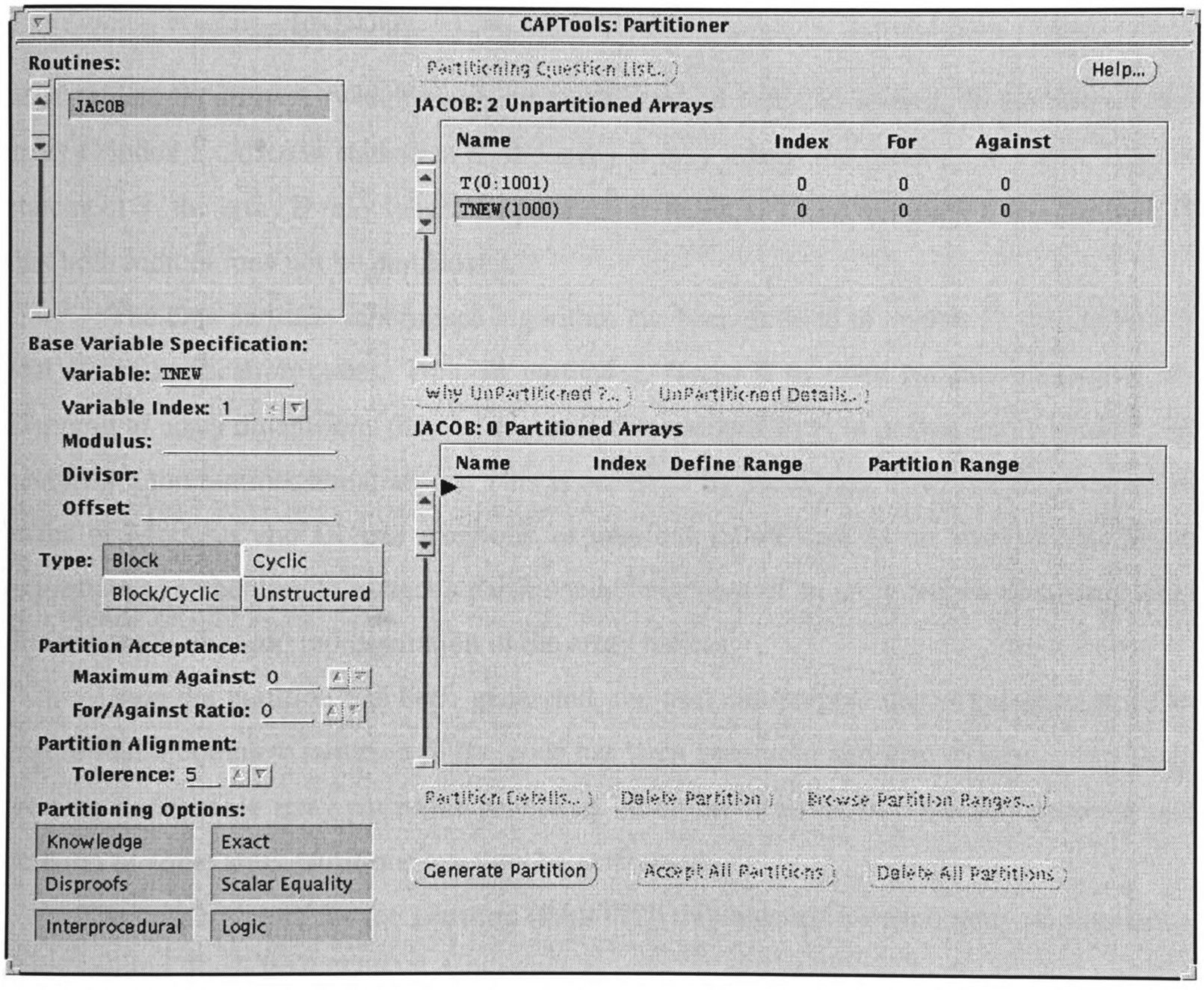


Figure 2.16 : Partitioning Window from CAPTools for the Jacobi Code.

The case may arise where an unpartitioned array may be partitioned in either index. Consider the following code where the variable array A is partitioned in index 2:

```
                DO I = 1, N
                    DO J= 1,100
S1                      B(J,I) = A(I,J)
                    ENDDO
                ENDDO
                DO I = 1, N
                    DO J= 1,100
S2                      C(I,J) = B(I,J) + A(I,J)
                    ENDDO
                ENDDO
```

In statement S_2 array A (which has been partitioned on index 2) inherits its partition to the array C index 2. Also, in statement S_2 the array B may inherit the partition in index 2 but in statement S_1 the array B may inherit its partition in index 1. There is clearly a conflict here in that both indices may not be partitioned.

The data partition inheritance algorithm has been devised to operate in the context of real world application codes. This, in particular, forces it to cater for interprocedural re-mapping of array dimensions (e.g. when a one-dimensional array is passed into a routine and becomes a three-dimensional array). This is achieved by specifying array decompositions in terms of Modulus and Divisor symbolic expressions rather than as an array index. These expressions can be used to extract a partitioned component of an array from a non-linear one-dimensionally mapped representation of the array indices.

Once the partition has been generated, the user can inspect the arrays partitioned to ensure that a complete partition of the code has been generated and also to ensure that there were no undesirable array partitions generated. Addition of arrays into the data partition and deletion of some array partitions can then be performed.

The data structure for the partition (PARTITION) is stored for each array partitioned in every routine. Each PARTITION data structure consists of a reference to the symbol variable (SYMBOL) that is partitioned; the index (INDEX) that is partitioned; the low and high partition range variables; the values of the Modulus and Divisor expression in terms of symbolic variables, i.e. NONLOOP and CONSTANT. It also stores whether the partition controls a disected subset of the array or the entire array. If the disected partition is present then any other disections of the partition are stored in the NEXT field.

2.7 Execution Control Masks.

The calculation and generation of execution control masks are required to ensure that the appropriate statements execute only on the processors that own the partitioned data [28, 31]. CAPTools generates a mask for every statement that requires one. These masks consist of an IF statement with a condition which states that only the processor which owns the data should execute the given statement. For example:

```
IF (CAP_LXXX <= I <= CAP_HXXX) A(I)=...
```

This execution CONTROL mask ensures that the value of A(I) is assigned only on the processor that owns the value of Ith entry of array A.

During the generation of these execution control masks, many statements will possess a mask. These execution control masks are however merged to reduce the overhead of calculating each mask. For example :

```
DO I = 1,100
      IF (CAP_LXXX <= I <= CAP_HXXX) A(I)=...
      IF (CAP_LXXX <= I <= CAP_HXXX) B(I)=...
            .
            .
      IF (CAP_LXXX <= I <= CAP_HXXX) Z(I)=...
ENDDO
```

In the above code the statements within the loop each possess the same execution control mask. These execution control masks may be reduced down to one execution control mask that surrounds all the statements :

```
DO I = 1,100
      IF (CAP_LXXX <= I <= CAP_HXXX) THEN
            A(I)=...
            B(I)=...
            .
            .
            Z(I)=...
      ENDIF
ENDDO
```

this execution control mask may then be transformed into the DO loop limits as follows:

```
DO I = MAX(1,CAP_LXXX),MIN(100,CAP_HXXX)
        A(I)=...
        B(I)=...
        .
        .
        Z(I)=...
ENDDO
```

This prevents the I loop iterating for all 100 iterations on each processor, while also avoiding the evaluation of the execution control mask the same number of times. Instead each processor will now only iterate over the partitioned data range allocated to that processor.

The main aim of the masking algorithm is to administer execution control masks to the maximum number of statements and thus allowing the maximum amount of parallelism to be obtained. Any statements that are not masked will be executed on every processor and can cause additional communications to be generated if, for example, an unmasked statement makes use of partitioned data.

There are four basic rules for generating execution control masks:

Rule 1. Statements that assign partitioned data.

Rule 2. Statements that use partitioned data.

Rule 3. Statements that assign values which are used by a masked statement.

Rule 4. Statements that use the data assigned by a masked statement.

The mask calculation algorithm attempts to mask as many statements as possible whilst minimising the communications required. Flexibility exists since statements may be controlled by a selection of execution control masks inherited via rules 2, 3 and 4. Some statements may be controlled by several execution control masks, for example, to calculate the data in overlap areas to avoid communication nested within many loops

Execution control mask statements may also be placed on call statements. This is possible if either all the statements within the called routine have the same mask or the called routine contains no masks.

To ensure a comprehensive set of masks has been set, they may be examined in the Mask Browser window.

Each execution control mask is stored in the CAPTools data structure MASK. This data structure contains a pointer to the symbolic expression for the execution control mask itself, and the partition that is relevant to this execution control mask. The MASK data structure is

assessable for each executable statement (COMMAND). This allows CAPTools to process each execution control mask for each executable statement. More than one execution control mask associated with the same statement is linked by means of the NEXT field.

2.8 Calculation, Migration and Merging of Communications.

When the execution control masks have been generated for the statements within the code, it is possible to determine whether any communications are required and their placement determined [28]. A communication is required if data used on a processor is not assigned on that processor. This is determined by comparing the execution control masks on the using statements with the location of the used data. This is achieved by the use of the symbolic inequality disproof algorithm (Section 2.4.6.1) comparing the partition range values designated by CAPTools in the partitioning stage with the execution control masks.

2.8.1 Commutative Operations.

Commutative operations, e.g. the summation or maxima of data, can exploit parallelism where a loop carried dependence appears to prohibit it. The loop carrying the dependence must be marked as partitioned during the masking stage (Section 2.7). There must also be no other assignments or usages of the data within the loop or any other loop carried true or control dependencies involving any other data in the statement.

This method allows each processor to calculate its own local values before communicating to all other processors and returning a global value. The value will be the same apart from some minor round-off error due to calculation reorder. This type of communication is accomplished using the CAP_COMMUTATIVE (Section 1.6) communication.

2.8.2 Calculation of Communication Requests.

Communication of partitioned data is necessary if the data required by one processor is calculated on another processor. The data range required by a processor can be determined by creating a clause list representing the control information for that range. The algorithm to calculate the communication control set is as follows :

1. Create two literals from the execution control mask of the statement, i.e.
   Mask_expression >= CAP_LOW AND Mask_expression <= CAP_HIGH.

2. Create a third literal for the same 'AND' set for the use of data before the set allocated to this processor, i.e.

   Usage_expression < CAP_LOW.

3. Duplicate the previous 'AND' set and place within an 'OR' list, changing the third literal to represent the use of data after the set allocated to this processor, i.e.

   Usage_expression > CAP_HIGH.

4. Normalise the 'AND' sets by setting the Mask_expression to be a function of the Usage_expression, creating a new variable to represent the Usage_expression.

5. Simplify the control list by using the inference engine within CAPTools and the control information of the statement.

Consider the following simple statement and its corresponding execution control mask :

IF (CAP_LOW <= I <= CAP_HIGH) A(I) = B(I–1)

where arrays A and B have the same partition. The communication control constructed for the usage of B was as follows :

((I >= CAP_LOW) AND (I <= CAP_HIGH) AND (I–1 < CAP_LOW)) OR

((I >= CAP_LOW) AND (I <= CAP_HIGH) AND (I–1 > CAP_HIGH)

After the normalisation (with the introduction of a new variable CAP_USAGE representing the index values involved in any communication) and simplification of the above control set by adding knowledge that the partition range on each processor is at least one or more, i.e.

CAP_HIGH – CAP_LOW >= Minimum SLAB NUMBER

where

Minimum SLAB NUMBER >= 1 (and may be set higher by the user)

provides the following :

((CAP_USAGE + 1 >= CAP_LOW) AND (CAP_USAGE < CAP_LOW))

which states that the values of array B to be received from another processor is for the low overlap area of B, i.e. B(CAP_LOW–1). If the control set is not empty a communication is required.

For unpartitioned data and scalars a similar algorithm is used where the ownership is not determined from a partition, but instead is based on the execution control mask of an assigning statement that is related to the usage via a dependence.

There are five distinct type of communications :

EXCHANGE – this involves the exchange of data between neighbouring processor as described in Section 1.6;

SEND/RECEIVE – the communication of points of data between processors which may not be neighbours;

BROADCAST – required when there is a conflict in data location or when no relationship may be determined between the usage and the assignment of data;

PIPELINES – communications involving a recurrence calculation (see Section 1.8);

COMMUTATIVES – communication of summation and maxima calculations (see Section 2.8.1).

Other communications may also be required for unpartitioned arrays and for all scalars.

If any DO, IF or CALL statements contains a communication and also has an execution control mask associated with it then the execution mask is removed to prevent any possible deadlock of the generated parallel code.

2.8.3 Migration and Merging of Communications.

Communications are generated for every statement that requires data from another processor. The communications are placed prior to the use of the communicated data. These communications may be placed within loops and be called several times more than desired, causing additional unnecessary start up latency. There may also be duplication in the communications of data between processors. To reduce this duplication the communications are migrated out of as many loops and routine boundaries as possible in order to maximise the possibility of merging these communications. This in turn leads to a further reduction in the number of communication calls.

The migration of the communications is achieved by traversing up the control flow graph via its predominator tree (Section 2.3.2) ensuring that there is no barrier to prevent the migration of the communications. A barrier which would prevent the migration of a communication are assigning statements or index assigners which may be determined using the true dependencies. If migration reaches a routine start then each reference to that routine call inherits the set of communications, and the communication continues from that point in the caller routine.

Communications will often migrate to the same point in the code. This is normally due to the barrier being the statement that assigns the data to be communicated. These

communications may be merged by calculating whether they are subsets or intersections of the same data using a symbolic comparison of the vector spaces [28].

When two requests for communication are merged, the original requesting statements are listed within the communications request list of the CAPTools data structure for communication (RECEIVE). The communication is referred to as the source while the statements requiring the communication are referred to as their sinks. A communication source may have several sinks due to the merger of communications as previously mentioned. Figure 2.17 shows the Communications Browser from CAPTools for the Jacobi code (Section 1.7). A communication has been selected and its associated statements using the communicated data (sinks) are shown. Each communication may also be interrogated to provide the user with additional information such as the barrier to the communication and details of why it is required.

The data structure for the communications within CAPTools (RECEIVE) has several fields of information. Each RECEIVE will store information for the symbol table of the variable (SYMBOL) that requires communications; the communication control information (Section 2.8.2); a pointer to the list of commands requesting the communication (COMMANDLIST); and a pointer to the list of commands assigning the communicated data (ASSIGNLIST). CAPTools can traverse through each routine, each communication and each executable statement that requested that communication.

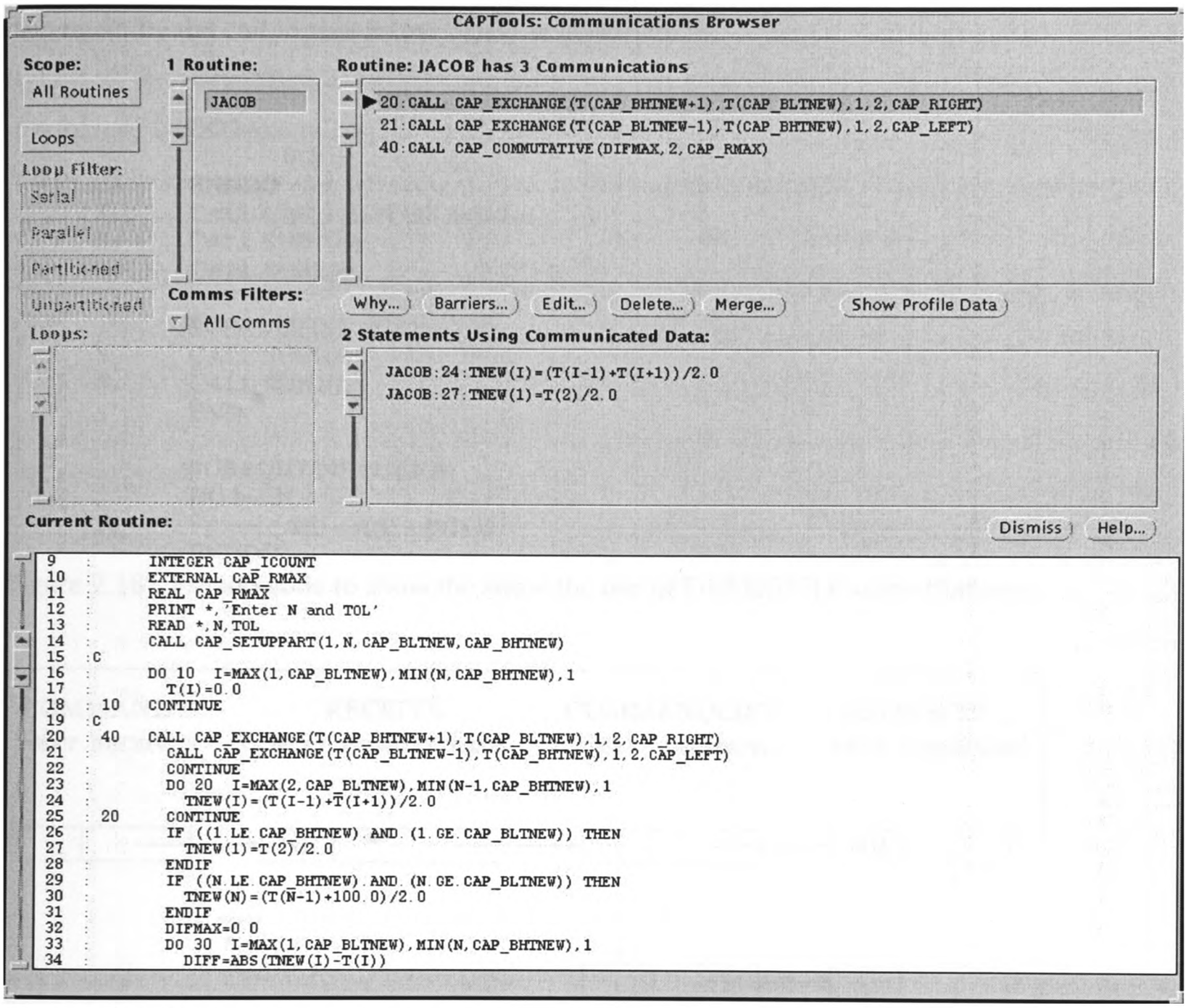


Figure 2.17 : Communication Browser from CAPTools.

Each COMMANDLIST also stores the paths (DEFROUTE) in the call graph that the communications migrated through to their optimum position. Consider the pseudo code in Figure 2.18. The data for the exchange communication S_1 is used in statement S_6 in subroutine SUB2 and the communication is prevented from migrating any further by the assignment of the data. There are three different paths by which this communication has migrated from subroutine SUB2 and Figure 2.19 shows the data structure that stores these paths. The first path is the call to subroutine SUB2 in statement S_3. The next call path is via the call to SUB2 in subroutine SUB1 at statement S_4 and then by the call to subroutine SUB1

at statement S_2. The third path is via the call to SUB2 in subroutine SUB1 at statement S_5 and then again by the call to subroutine SUB1 at statement S_2.

```
            DO I = ...
                    B(I) = ...
            ENDDO
S1          CALL CAP_EXCHANGE(B(I+1),....)
S2          CALL SUB1(B)
S3          CALL SUB2(B)

            SUBROUTINE SUB1
S4          CALL SUB2(B)
S5          CALL SUB2(B)
            END

            SUBROUTINE SUB2(B)
            DO I=1,N
S6                  A(I) = A(I) + B(I+1)
            ENDDO
```

Figure 2.18 : Pseudo code to show the show the use of DEFROUTE data structures.

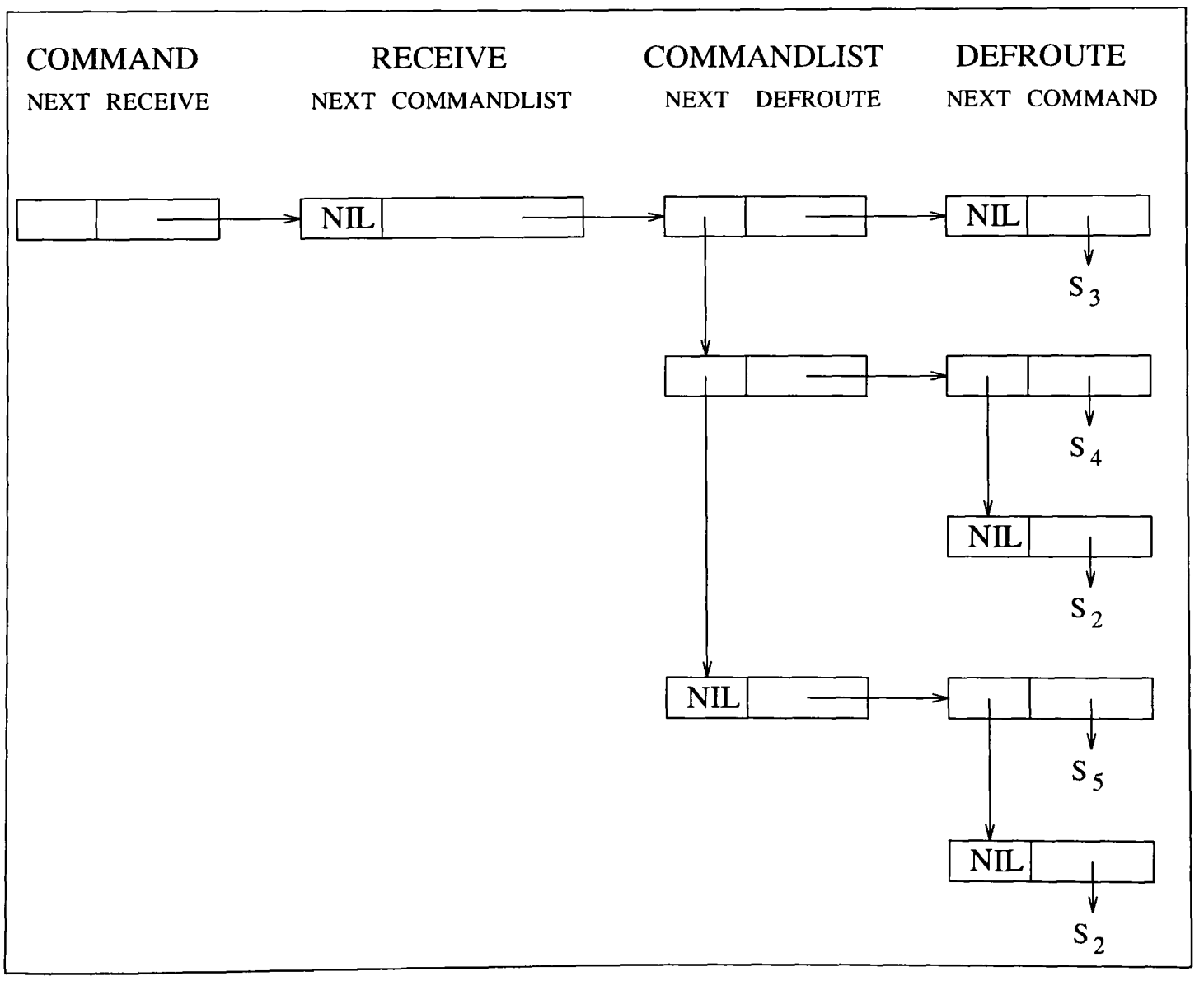


Figure 2.19 : Data structures for DEFROUTE for the example in Figure 2.18.

The information for the data storage of the communication within CAPTools is exploited for the calculation of the optimum position of synchronisation points for overlapping communications (Chapter 5).

2.9 Generation of Communications.

The communications generated by CAPTools are high level generic communication calls, developed by the University of Greenwich, which map onto the low level communication calls of either machine specific communications or communications (Section 1.6). The communications type generated depend upon that chosen by the user - either bulk or gather and scatter. The generated communications consist of exchanges, pipelines, constrained send/receive and broadcasts to all processors. Each of these communications may be generated with additional surrounding loops if required.

2.10 Final Code Generation.

This stage outputs the final parallel code generated by CAPTools. The parallel code will be generated with calls to routines to allow the setup of the partition of the data given the number of processors. The code will also consist of additional communications to distribute the initial data (i.e. from READ statements) and to collate the final results (i.e. for WRITE/PRINT statements).

The parallel code may then be compiled and linked with the pre-processed CAPLib communications library (Section 1.6). This library is pre-processed specifically for the parallel machine and the communications types required, e.g. it may be compiled for the Cray T3D using PVM, MPI or SHMEM communications or for the Transtech Paramid using PVM or Ctoolset communications.

2.11 Transformations.

At various stages throughout the parallelisation process within CAPTools the user may apply several automatic transformations to their source code. These are supplied within CAPTools to allow the user to obtain improved results form the parallel code by the application of very simple transformations. These transformations are :

Loop split – this allows for the splitting of loops. This can transform serialising loop carried dependencies within the original loop into loop independent dependencies between the two loops formed by splitting the original;

Loop Interchange – this allows loops to be interchanged to allow better parallelism to be obtained from a loop nesting;

Index Interchange – this allows the user to alter indices and, for example, to provide contiguous data for communication;

Routine Copy – this allows a routine to be copied;

Loop Movement – this allows a loop to be moved into or out of called routines.

All these transformations will modify statements, control flow graph and the dependence graph to ensure that correct code is generated.

2.12 Conclusions.

This chapter has provided a brief explanation of the parallelisation tool CAPTools and some of its data structures. These data structures provide a good basis for the work developed in Chapter 4 for the automatic generation of the overlapping communications within CAPTools in Chapter 5. CAPTools will also be employed in the parallelisation of several codes in Chapter 3.

Chapter 3

3 Parallelisation of Structured Mesh Computational Mechanics Codes.

3.1 Introduction.

This chapter sets out to obtain efficient parallel performance of four different codes. In the first instance all the codes were parallelised using Computer Aided Parallelisation Tools (Chapter 2). Each of the parallel versions of the codes was then scrutinised closely to detect if any further improvement in performance could be obtained on a parallel system. These codes were also examined with the aid of CAPTools.

Every communication within a parallel code incurs an overhead that varies from one parallel system to the next. It is therefore essential that the minimum number of communications be generated.

There are also different communication calls that may be used depending on both the data to be communicated and the nature of the algorithm. In the codes that are parallelised here there will be examples of pipelined communications (see Section 1.8) and also exchange of data between each processor (see Section 1.6 and 1.7).

Further optimisation such as Iteration Grouping is also investigated (Section 1.9).

3.2 2-D Heat Diffusion Code (FAB).

The in-house heat diffusion code from the University of Greenwich solves two dimensional head diffusion and conduction problems on a structured grid. The solver is based on the Gauss-Seidal Line Successive Over Relaxation (LSOR) algorithm which sweeps the domain in the j-direction. The code consists of approximately 700 lines of FORTRAN code.

Figure 3.1 shows a two dimensional mesh of dimension IN by JN. The LSOR solver sweeps the domain in the j-direction from 1 to JN solving for each i from 1 to IN. To calculate the line L_j requires the data from the previous line L_{j-1} and from the next line L_{j+1}. The physical model of the problem is as follows :

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

This is the two-dimensional Poisson equation to solve the temperature φ. This is a typical example of an elliptic partial differential equation. Applying a central difference approximation provides the equation :

$$a_{i,j-1}\phi_{i,j-1} + a_{i-1,j}\phi_{i-1,j} - a_{i,j}\phi_{i,j} + a_{i+1,j}\phi_{i+1,j} + a_{i,j+1}\phi_{i,j+1} = b_{i,j}$$

Where a denotes the coefficient at the respective nodal temperatures

Solving the problem using a line solver the solution of a line j is based on the approximations of lines j-1 and j+1. This provides a tridiagonal matrix system of equations to be solved for each line of the form:

$$a_{i-1,j}\phi_{i-1,j} - a_{i,j}\phi_{i,j} + a_{i+1,j}\phi_{i+1,j} = -a_{i,j-1}\phi_{i,j-1} - a_{i,j+1}\phi_{i,j+1} + b_{i,j}$$

For a single j line this, gives the matrix equation :

$$A\underline{\Phi} = \underline{B}$$

$$A = \begin{bmatrix} d_1 & u_1 & & & & & \\ l_2 & d_2 & u_2 & & & & \\ & l_3 & d_3 & u_3 & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & & & & l_{IN-} & d_{In-1} & u_{IN-1} \\ & & & & & l_{IN} & d_{IN} \end{bmatrix} \quad \Phi = \begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \Phi_3 \\ \cdot \\ \cdot \\ \cdot \\ \Phi_{IN-1} \\ \Phi_{IN} \end{bmatrix} \quad B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ \cdot \\ \cdot \\ \cdot \\ B_{IN-1} \\ B_{IN} \end{bmatrix}$$

Where

$$l_i = a_{i-1,j}$$

$$u_i = a_{i+1,j}$$

$$d_i = a_{i,j}$$

$$B_i = -a_{i,j-1}\phi_{i,j-1} - a_{i,j+1}\phi_{i,j+1} + b_{i,j}$$

This system of equations is then solved using the Thomas algorithm [66] which is a special case of Gaussian Elimination for tridiagonal matrices and consists of a forward elimination and a backward substitution. This method is often referred to as the Tridiagonal Matrix Algorithm (TDMA). This algorithm is implemented as two recurrences: one for the

forward elimination from I=1 to I=IN and the other for backward substitution from I=IN to I=1, as indicated in Figure 3.1.

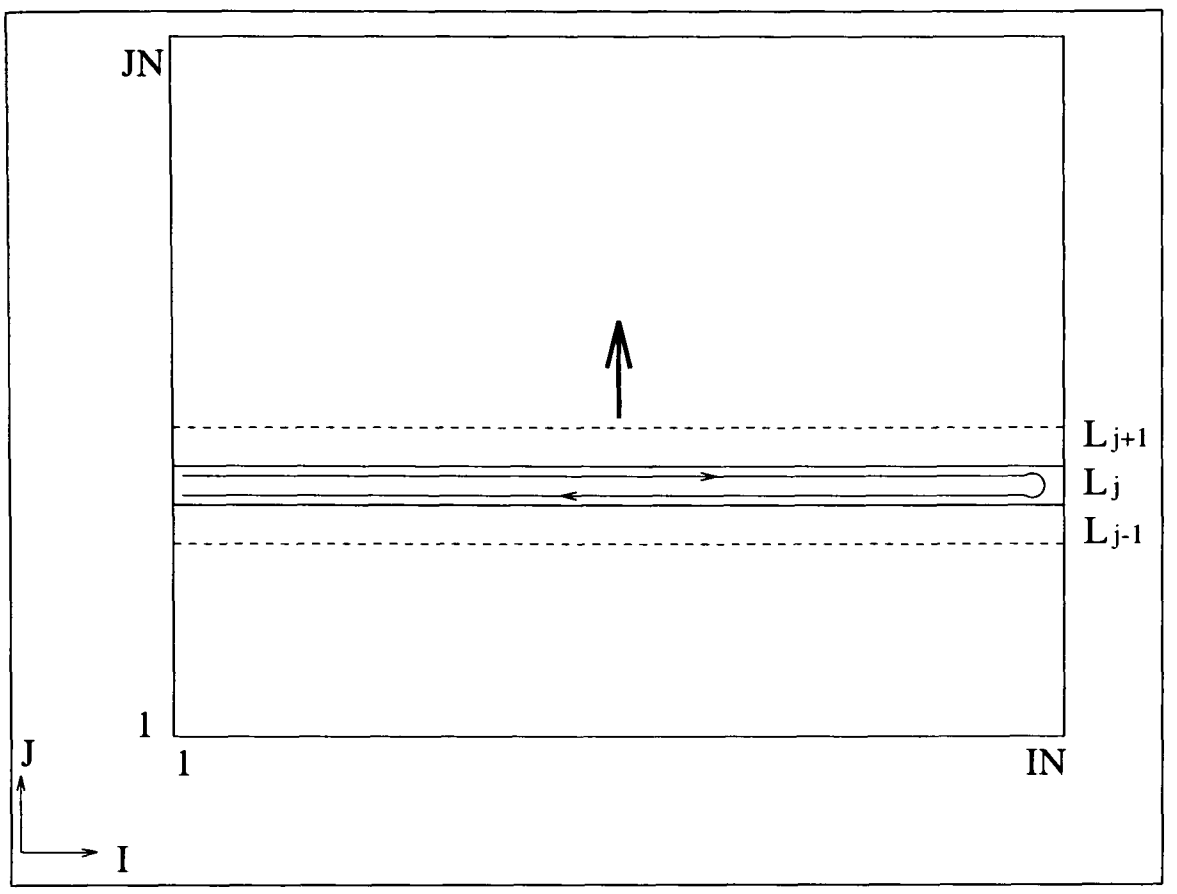


Figure 3.1 : The Line Successive Over Relaxation Algorithm in Serial.

An execution profile (Table 3.1) of the serial code ran on a SUN Workstation showed that over 97% of the computational effort occurred in the routine SOLVER (Figure 3.2) and the routines it called. The profile also showed that routines TDMA and RESIDUAL provided 35% and 16% of the total computation respectively. The routine SOLVER however called these two routines. The routine FAB was the main routine and obviously accounted for 100% of the computational time. Closer inspections of the loops within the routine SOLVER showed that there is a loop iterating over the j lines in the mesh of the problem (see Figure 3.2). This loop is surrounded by an additional iterative loop that controls the number of domain sweeps.

Function Name	Cost(seconds)	Cost(%)
SOLVER	225.56	97.59
TDMA	80.94	35.02
RESIDUAL	37.38	16.17
FAB	231.15	100.00
PROPS	0.98	0.42
TEMPER	0.12	0.05
SETTEMP	0.04	0.02
CONDUC	1.15	0.50

Table 3.1 : A profile of the serial FAB code for a 500x500 problem size.

```
C
C       This is the Main loop, it controls number of sweeps.
C
 40     CONTINUE
        RESID = 0.0
C
C       If max sweep reached print out results and quit.
C
        IF (ISWEEP .LE. MSWEEP) THEN
C
C               Start to sweep lines visiting each J line in domain once.
C
                TOP=0.0
                BOT=0.0
                DO 30 J = 2,JN-1
                        IF (GMOPT .EQ. 0) THEN
                                RBAR = 1.0
                        ELSE
                                RBAR = R(J)
                        ENDIF
                        DR = (R(J+1) - R(J-1)) / 2.0 * RBAR
C
C                       Construct coeff.(Gauss-Seidal iteration implemented so must use latest
C                       values of TNEW(I,J-1) for each line calculation.)
C
                        DO 10 I = 2,IN-1
                                LSWEEP(I) = TNEW(I,J)
                                DZ = (Z(I+1) - Z(I-1))*0.5
                                A(I) = WKSP(I,J) / DZ
                                C(I) = WKSP(I+1,J) / DZ
                                D(I) = -(A(I) + C(I) + FAC + (SK(I,J+1) + SK(I,J)) / DR )
                                B(I) = TOLD(I,J) * FAC + HFLX(I,J)
                                B(I) = -(B(I) + (TNEW(I,J+1) * SK(I,J+1) + TNEW(I,J-1) * SK(I,J))/DR)
 10                     CONTINUE
                        CALL TDMA(TNEW,IN,IN-1,J)
                        CALL RESIDUAL(LSWEEP,TNEW,IN-1,RESIDJ,J,JN-1,TOP,BOT)
 30             CONTINUE
                TOP = SQRT(TOP)
                BOT = SQRT(BOT) + 1.0
                RESIDJ = TOP/BOT
                RESID = RESIDJ
                ISWEEP = ISWEEP + 1
C
C               Is Problem converged? If no do another iteration.
C
                IF (MOD(ISWEEP,10).EQ.0) PRINT *, 'RESIDUAL = ',RESID,ISWEEP
                IF (RESID .GT. CON1) THEN
                        GOTO 40
                ELSE
                        PRINT*,'ITERATIONS:',ISWEEP
                        RETURN
                ENDIF
        ENDIF
```

Figure 3.2 : The Routine SOLVER from the serial FAB Code.

The code was analysed by CAPTools with a full power analysis. Using the Loop Browser within CAPTools, showed that the DO 30 J=2,JN-1 loop was serial. Interrogating the dependence graph reveals that there is a true dependence of the array TNEW between the source command :

CALL TDMA(TNEW,IN,IN-1,J)

and the sink command :

B(I) = -(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)

between iterations of the DO 30 J loop. This dependence is due to the calculation of TNEW(I,J) in one iteration of the J loop and its use in the next as TNEW(I,J-1), i.e. a recurrence. This recurrence occurs because the algorithm implements the Gauss-Seidal iteration which uses the latest values of TNEW(I,J-1) for each line calculation. On a serial machine this is often the best method to apply since it generally converges to the correct solution in fewer iterations than an explicit method.

Figure 3.3 shows the operation of this recurrence. The domain is partitioned in this case, onto three processors. To calculate each line requires the data from the previous line. To preserve the integrity of the algorithm in parallel it is necessary to calculate all the lines on processor 1 before communicating the values of the last line on processor 1 (L_j) to the second processor to calculate the values of the first line on that processor. The algorithm will therefore be serial in nature.

This recurrence causing the DO 30 loop to be serial compels the parallel code to be generated with a pipeline communication (Section 1.8) surrounding the DO 30 loop (Figure 3.4). The pipeline communication receives the values of TNEW(I,CAP_BLTNEW-1) for every sweep. A CAP_EXCHANGE communication has also been generated for the communication of the TNEW(I,J+1) in each domain sweep. Two CAP_COMMUTATIVE calls were also generated for the calculation of the global sum of TOP and BOT variables, that were summed in routine RESIDUAL, used for the calculation of the residuals for each sweep. These were the only communications generated for the main algorithm in routine SOLVER.

The solutions obtained from a parallel run of this code were the same as those obtained from the serial code. However, the speed up results obtained were poor. Table 3.2 shows that as the number of processor increase, the time taken does not decrease. Instead the actual time taken is increasing slightly as the number of processors increases. This is due to the additional communication latency incurred in the parallel version whilst no parallelism is being exploited due to the pipeline that is not surrounded by any parallel loops.

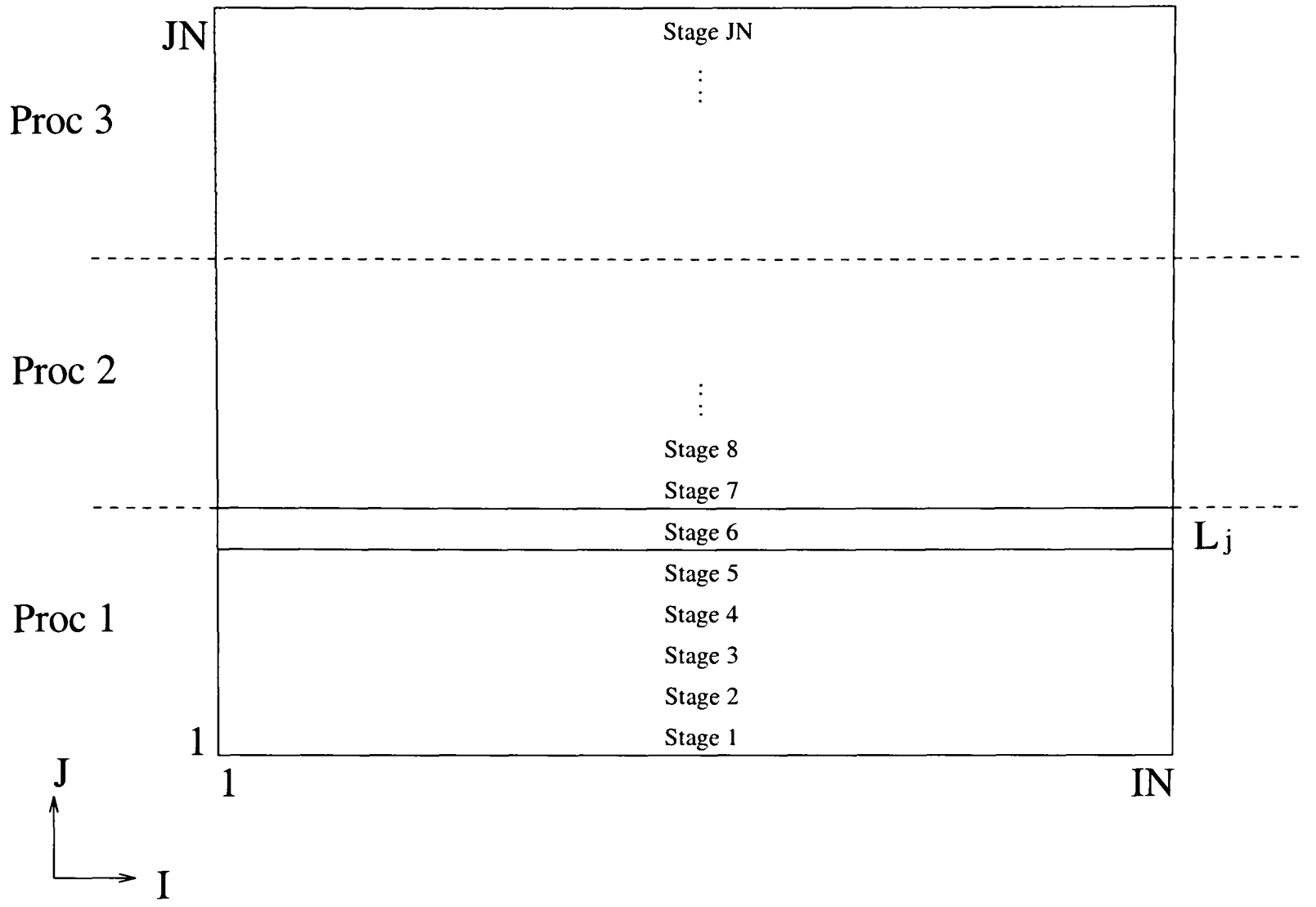


Figure 3.3 : Line successive over relaxation algorithm implemented as a pipeline in parallel.

```
C       This is the Main loop, it controls number of sweeps.
C
 40     CONTINUE
        RESID = 0.0
C
C       If max sweep reached print out results and quit.
C
        IF (ISWEEP .LE. MSWEEP) THEN
C
C               Start to sweep lines visiting each J line in domain once.
C
C               Exchange communication for the values of TNEW(I,J+1).
C
                CALL CAP_EXCHANGE(TNEW(2,CAP_BHTNEW+1),TNEW(2,CAP_BLTNEW),
     &                                  IN-2,CAP_RIGHT)
                TOP=0.0
                BOT=0.0
C
C               Receive communication of the pipeline for the values of TNEW(I,J-1).
C
                CALL CAP_RECEIVE(TNEW(2,CAP_BLTNEW-1),IN-2,CAP_LEFT)
                DO 30 J=MAX(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW),1
                        .
                        .
C
C                       Construct coeff.(Gauss-Seidal iteration implemented so must use latest
C                       values of TNEW(I,J-1) for each line calculation.)
C
                        DO 10  I=2,IN-1,I
                                .
                                .
                                B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
10                      CONTINUE
                        CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
                        CALL RESIDUAL(LSWEEP,TNEW,IN-1,RESIDJ,J,JN-1,
     &                                  TOP,BOT,CAP_LTNEW,CAP_HTNEW)
30              CONTINUE
C
C               Send communication of the pipeline for the values of TNEW(I,J-1).
C
                CALL CAP_SEND(TNEW(2,CAP_BHTNEW),IN-2,CAP_RIGHT)
C
C               Perform commutative operations to calculate global values of TOP and BOT.
C
                CALL CAP_COMMUTATIVE(CAP_TOP,CAP_RADD)
                TOP=TOP+CAP_TOP
                CALL CAP_COMMUTATIVE(CAP_BOT,CAP_RADD)
                BOT=BOT+CAP_BOT
                TOP=SQRT(TOP)
                BOT=SQRT(BOT)+1.0
                RESIDJ=TOP/BOT
                RESID=RESIDJ
                ISWEEP=ISWEEP+1
C
C               Is Problem converged? If no do another iteration.
C
                IF (MOD(ISWEEP,10).EQ.0) PRINT *, 'RESIDUAL = ',RESID,ISWEEP
                IF(RESID .GT. CON1) THEN
                        GOTO 40
                ELSE
                        PRINT*,'ITERATIONS:',ISWEEP
                        RETURN
                ENDIF
        ENDIF
```

Figure 3.4 : Communications in the Routine SOLVER for the Parallel FAB Code.

Number of Processors	Time Taken (seconds)
1	127.967
2	128.502
4	128.856
6	129.396
8	129.866

Table 3.2 : Initial timing results for parallel FAB on the Transtech Paramid.

The pipeline loop, DO 30, timed independently indicated that almost all the time was executed in this loop. Pipelines are essential to ensure correctness of the parallel code but are unfortunately highly inefficient due to communication start-up latency and also pipeline start-up and shutdown times. However, in this case the pipeline was also completely serial. Greater benefit could be obtained if the pipeline communication of TNEW(I,J-1) was replaced by a CAP_EXCHANGE communication similar to that applied to the TNEW(I,J+1).

To improve the performance of the parallel code, the true dependence carried by the DO 30 loop was removed. Reloading the earlier analysis database into CAPTools and by selecting and deleting the true dependence between the previously mentioned statements, using the CAPTools dependence graph browser, can remove this problem. The deletion of dependencies from a code can be highly dangerous since it changes the solution algorithm. However, in this case the removal of a dependence prevents the algorithm from using the latest available value of TNEW(I,J-1). Instead for each iteration the values of TNEW(I,J-1) from the previous iteration, i.e. the old value, are used for the first line of each processor. The removal of this dependence will allow CAPTools to no longer know that the value of TNEW(I,J-1) is dependent on a previous iteration. This allows the DO 30 loop to be parallel and for a more efficient parallel code to be generated. This is a slight alteration to the linear equation solver and will in no way alter the physics of the original problem. This is a well known method that is commonly used and is referred to as the Localised LSOR solver. The only disadvantage of altering the algorithm is that the convergence will now vary slightly from the serial algorithm due to a change in the calculation order [61].

The parallel code obtained (Figure 3.5) consists of an additional CAP_EXCHANGE communication for each sweep of the algorithm. This has replaced the pipeline communication. This communication has been migrated to its optimal position at the same point in the code as

the CAP_EXCHANGE communication of the TNEW(I,J+1), i.e. immediately after the sweep loop head.

```
C
C       This is the Main loop, it controls number of sweeps.
C
 40     CONTINUE
        RESID = 0.0
C
C       If max sweep reached print out results and quit.
C
        IF (ISWEEP .LE. MSWEEP) THEN
C
C               Start to sweep lines visiting each J line in domain once.
C
C               Exchange communication for the values of TNEW(I,J+1).
C
                CALL CAP_EXCHANGE(TNEW(2,CAP_BHTNEW+1),TNEW(2,CAP_BLTNEW),IN-2,CAP_RIGHT)
C
C               Exchange communication for the values of TNEW(I,J-1) which replaces the previous pipeline.
C
                CALL CAP_EXCHANGE(TNEW(2,CAP_BLTNEW-1),TNEW(2,CAP_BHTNEW),IN-2,CAP_LEFT)
                TOP=0.0
                BOT=0.0
                DO 30 J=MAX(2,CAP_BLTNEW),MIN(JN-1,CAP_BHTNEW),1
                        .
                        .
C
C                       Construct coeff.(Gauss-Seidal iteration implemented so must use latest
C                       values of TNEW(I,J-1) for each line calculation.)
C
                        DO 10  I=2,IN-1,1
                                .
                                .
                                B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
10                      CONTINUE
                        CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
                        .
                        .
30              CONTINUE
C
C               Perform commutative operations to calculate global values of TOP and BOT.
C
                CALL CAP_COMMUTATIVE(CAP_TOP,CAP_RADD)
                TOP=TOP+CAP_TOP
                CALL CAP_COMMUTATIVE(CAP_BOT,CAP_RADD)
                BOT=BOT+CAP_BOT
                TOP=SQRT(TOP)
                BOT=SQRT(BOT)+1.0
                RESIDJ=TOP/BOT
                RESID=RESIDJ
                ISWEEP=ISWEEP+1
C
C               Is Problem converged? If no do another iteration.
C
                IF (MOD(ISWEEP,10).EQ.0) PRINT *, 'RESIDUAL = ',RESID,ISWEEP
                IF(RESID .GT. CON1) THEN
                        GOTO 40
                ELSE
                        PRINT*,'ITERATIONS:',ISWEEP
                        RETURN
                ENDIF
        ENDIF
```

Figure 3.5 : The Routine SOLVER in FAB with the Pipeline Communications replaced by Exchange Communications.

The parallel code has been altered such that a Local LSOR (LLSOR) algorithm is implemented. This is a common parallel approach and Figure 3.6 shows how this is implemented. Each processor will now calculate a line of data on each processor at the same time. In the pipeline version the calculation of a line was dependent on knowing the values of the previous line. This dependence has now been removed and the data from the previous line is obtained from data calculated on the previous sweep of the domain. Thus the line L_2 on processor 2 will require the data from the previous sweep to be communicated to its processor by means of an exchange communication. The same is also true for the calculation of L_3 on processor 3.

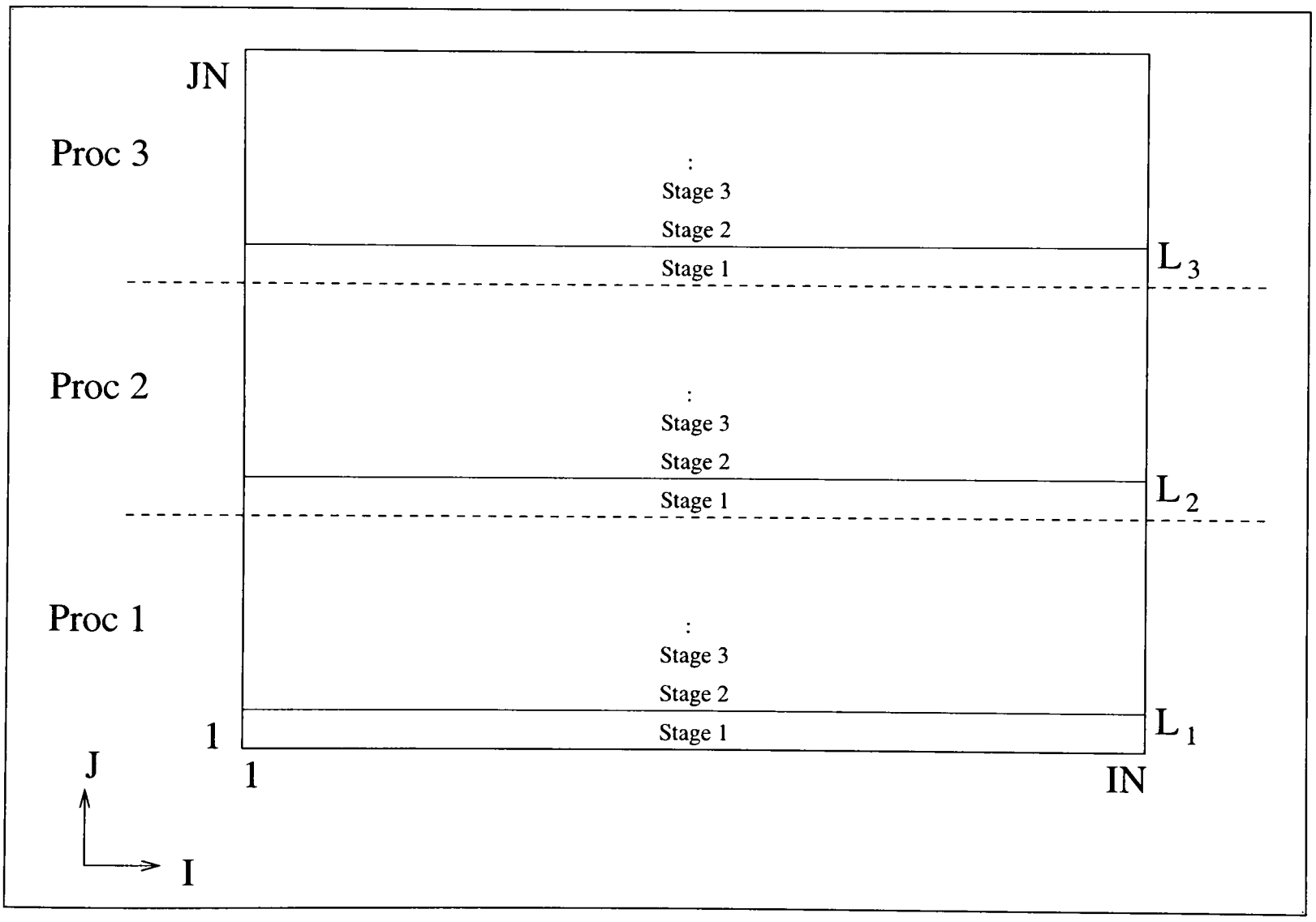


Figure 3.6 : The Gauss-Seidel Local LSOR in Parallel.

The performance results (Table 3.3) obtained were much better than the previous parallel code. A speed up of 7.33 was obtained on 8 processors. The solutions obtained did vary slightly as expected (within a specified tolerance) and were satisfactorily accurate. There was however, looking at the performance results, scope for further improvement. A further

profile of the parallel code revealed that the two CAP_EXCHANGE communications in the main algorithm were now taking a significant amount of time.

No. of Processors	Time Taken	Speed Up	Efficiency
1	127.967	-	-
2	65.277	1.96	98.0%
4	33.346	3.84	95.9%
6	22.833	5.60	93.4%
8	17.448	7.33	91.7%

Table 3.3 : Results for FAB with synchronous communications for the Transtech Paramid.

To measure the impact of these CAP_EXCHANGE a simple test was administered to establish how varying communication lengths affected the overall runtime of the code. The additional data communicated does not affect the usual calculation of the code. The test was executed on four i860 processors with a single communication of 500 reals, i.e. 2000 bytes of data. This communication length was multiplied by various factors and the results in Table 3.4 were obtained.

Communication Length Factor	Time (seconds)	Speed Up	Efficiency (%)
1	33.461	3.82	95.6
10	37.290	3.43	85.8
20	41.597	3.07	76.9
30	45.893	2.79	69.7
40	50.192	2.55	63.7
60	58.786	2.17	54.4
80	67.326	1.90	47.5
100	75.974	1.68	42.1

Table 3.4 : Results for varying communication lengths on four i860 processors.

The results in Table 3.4 show that as the amount of data being communicated (communication length factor) increased then so the efficiency of the problem decreased. It can therefore be concluded that as the communication lengths increased then so the total runtime of the parallel code will be extended and therefore the performance of the parallel code deteriorates. The communication time was plotted on a graph (Figure 3.7) against the communication length factor and linear regression was applied to obtain a 'best-fit' line. From the graph using linear regression it is estimated that if no data were communicated then the time for communication would be 33.013 seconds. This time is attributed to the total communication

start-up time for the whole run of the program. It would therefore be of advantage if the communication time and its associated start-up latency were concealed. Most parallel machines perform overlapping or asynchronous communications (Section 4.2). A method of improving these communications using overlapped communication will be discussed later in Chapter 4.

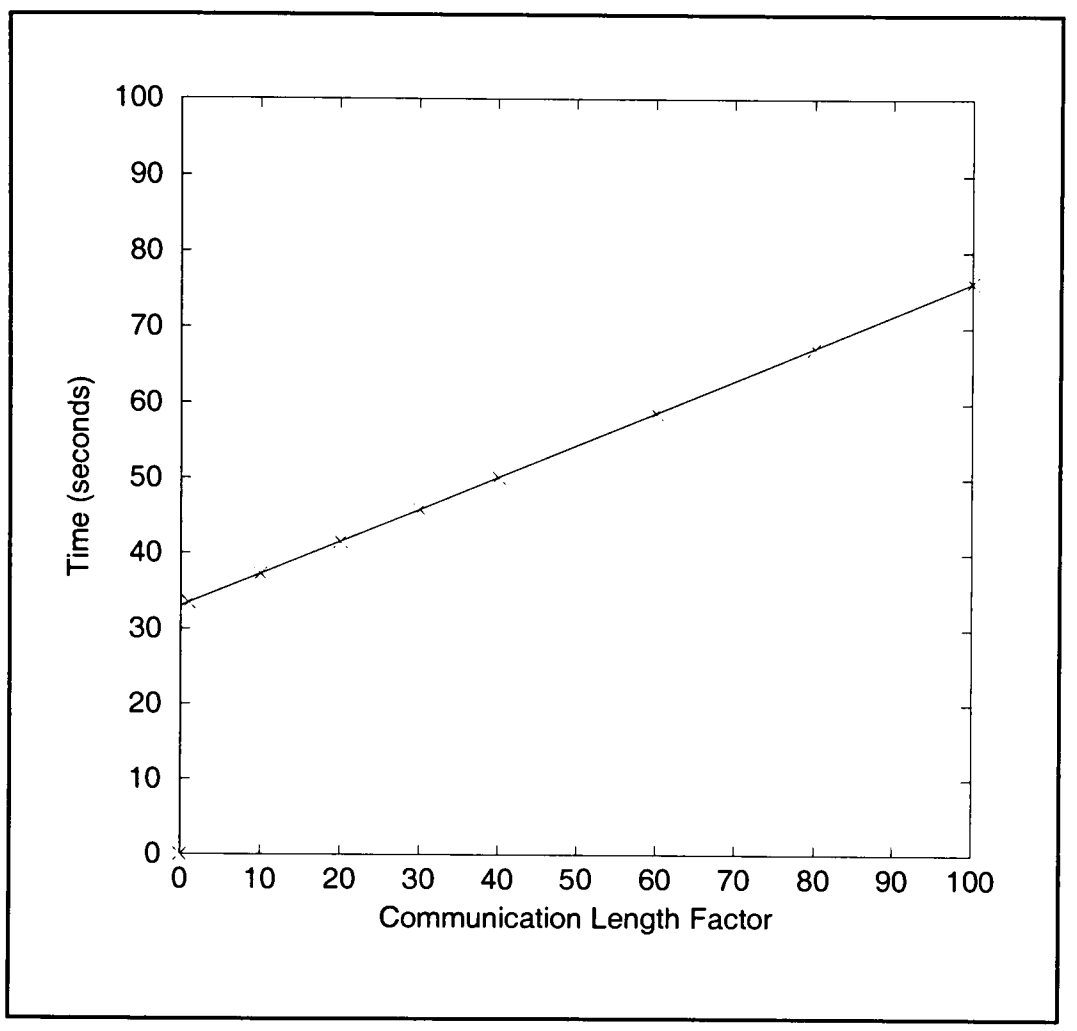


Figure 3.7 : Graph of Time taken against Communication Length Factor and a Best-Fit Line.

3.3 Teamke1.

Teamke1 is a two dimensional steady state flow prediction code from the University of Manchester Institute of Science and Technology (UMIST) [67]. It is a finite difference/finite volume technique with a k-ε turbulence model code using a structured Cartesian grid. The code may be applied to plane and axisymmetric flows, and laminar or turbulent flows. The convective terms may be discretised using either Quadratic interpolation (QUICK) or Power Law interpolation (PLDS).

A bi-directional Line Successive Over Relaxation (LSOR) solver is used to solve the linear equations. The FAB code (Section 3.2) consisted of a single line by line solver sweeping from top to bottom. The LSOR algorithm may sweep the domain from top to bottom, bottom to top, left to right, right to left or a combination of these. The advantage of using a combination of

these sweeping directions is that boundary effects can be conveyed throughout the domain at a faster rate than a single sweeping direction. TEAM allows the domain to sweep from left to right and then from bottom to top alternatively as shown in Figure 3.8.

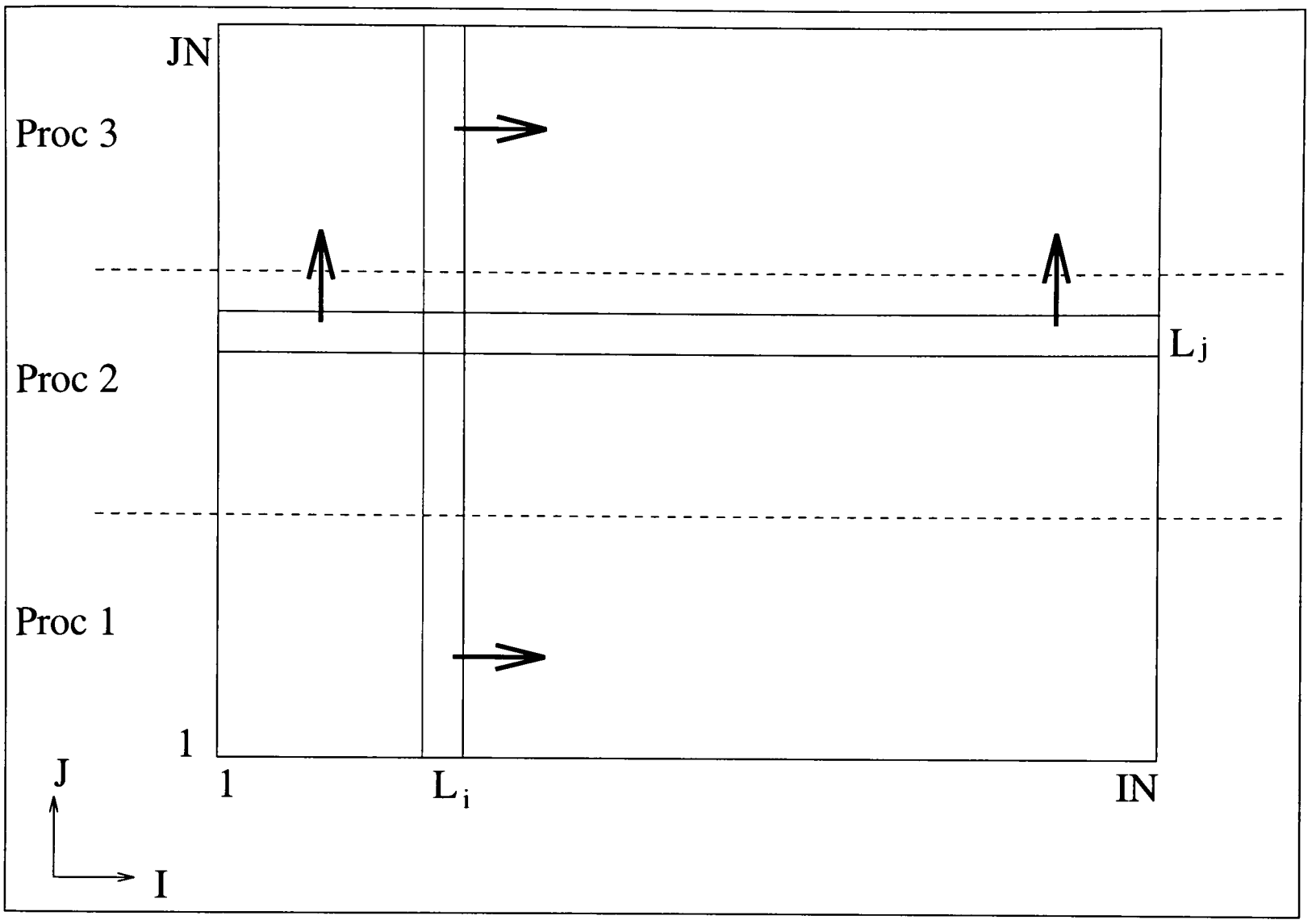


Figure 3.8 : The Bi-directional LSOR algorithm in the routine LISOLV.

The bi-directional solver is in the subroutine LISOLV (Figure 3.9). The bi-directional solver consists of sweeping through the i-direction (DO 100 loop) followed immediately by a sweep through the j-direction (DO 1000 loop). Figure 3.8 shows the domain being swept in the i-direction calculating a line at a time from 1 to IN. The domain is then swept in the j-direction line by line from 1 to JN. Each line requires the data from the previous line, i.e. line L_i requires the data from line L_{i-1} and likewise L_j requires data from line L_{j-1}. Each directional sweep consists of calculation of the coefficients and then an update of the solution variable PHI. For the i-direction the DO 101 loop is responsible for the coefficient calculation (forward elimination) and the DO 102 loop is responsible for updating (backward substitution). For the j-direction the DO 1010 loop is responsible for calculating the coefficients while the DO 1020 loop is

responsible for the updating. All of these four loops possess an implicit calculation which deem the loops serial. If these loops were parallelised then pipelines would be generated.

```
      SUBROUTINE LISOLV(...)
      .
      NIM1=NI
      NJM1=NJ
      JSTM1=JSTART-1
      ISTM1=ISTART-1
      DO 2000 IT=1,NSW
            A(JSTM1)=0.0
C
C           Commence W-E sweep.
            DO 100 I=ISTART,NIM1
                  C(JSTM1)=PHI(I,JSTM1)
C
C                 Commence S-N traverse.
                  DO 101 J=JSTART,NJM1
C
C                       Assemble TDMA coefficients.
                        A(J)=AN(I,J)
                        B(J)=AS(I,J)
                        C(J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
                        D(J)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D(J)-B(J)*A(J-1))
                        A(J)=A(J)*TERM
 101                    C(J)=(C(J)+B(J)*C(J-1))*TERM
C
C                 Obtain new PHI.
                  DO 102 JJ=JSTART,NJM1
                        J=NJ+JSTART-JJ
 102                    PHI(I,J)=A(J)*PHI(I,J+1)+C(J)
 100        CONTINUE
            A1(ISTM1)=0.0
C
C           Commence S-N sweep.
            DO 1000 J=JSTART,NJM1
                  C1(ISTM1)=PHI(ISTM1,J)
C
C                 Commence W-E traverse.
                  DO 1010 I=ISTART,NIM1
C
C                       Assemble TDMA coefficients.
                        A1(I)=AE(I,J)
                        B1(I)=AW(I,J)
                        C1(I)=AN(I,J)*PHI(I,J+1)+AS(I,J)*PHI(I,J-1)+SU(I,J)
                        D1(I)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D1(I)-B1(I)*A1(I-1))
                        A1(I)=A1(I)*TERM
 1010                   C1(I)=(C1(I)+B1(I)*C1(I-1))*TERM
C
C                 Obtain new PHI.
                  DO 1020 II=ISTART,NIM1
                        I=NI+ISTART-II
 1020                   PHI(I,J)=A1(I)*PHI(I+1,J)+C1(I)
 1000       CONTINUE
 2000 CONTINUE
```

Figure 3.9 : Routine LISOLV from Serial TEAMKE1 Code.

```
      SUBROUTINE LISOLV(...)
      .
      NIM1=NI
      NJM1=NJ
      IF (1.GE.CAP_LPHI.AND.1.LE.CAP_HPHI)JSTM1=JSTART-1
      ISTM1=ISTART-1
      DO 2000 IT=1,NSW,1
            IF (1.GE.CAP_LPHI.AND.1.LE.CAP_HPHI)A(JSTM1)=0.0
C
C           Commence W-E sweep.
            DO 100 I=ISTART,NIM1,1
                  IF (1.GE.CAP_BLT.AND.1.LE.CAP_BHT)C(JSTM1)=PHI(I,JSTM1)
C
C                 Commence S-N traverse.
                  CALL CAP_RECEIVE(A(CAP_BLT-1),1,CAP_LEFT)
                  CALL CAP_RECEIVE(C(CAP_BLT-1),1,CAP_LEFT)
                  DO 101 J=MAX(JSTART,JSTART+CAP_LPHI-2),MIN(NJM1,JSTART+CAP_HPHI-2),1
C
C                       Assemble TDMA coefficients.
                        A(J)=AN(I,J)
                        B(J)=AS(I,J)
                        C(J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
                        D(J)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D(J)-B(J)*A(J-1))
                        A(J)=A(J)*TERM
101                     C(J)=(C(J)+B(J)*C(J-1))*TERM
                  CALL CAP_SEND(A(CAP_BLT-CAP_LPHI+CAP_HPHI),1,CAP_RIGHT)
                  CALL CAP_SEND(C(CAP_BHT),1,CAP_RIGHT)
C
C                 Obtain new PHI.
                  CALL CAP_RECEIVE(PHI(I,CAP_HPHI+1),1,CAP_RIGHT)
                  DO 102 JJ=MAX(JSTART,JSTART-CAP_HPHI+NJ),MIN(NJM1,JSTART-CAP_LPHI+NJ),1
                        J=NJ+JSTART-JJ
102                     PHI(I,J)=A(J)*PHI(I,J+1)+C(J)
                  CALL CAP_SEND(PHI(I,CAP_LPHI),1,CAP_LEFT)
100               CONTINUE
                  CALL CAP_EXCHANGE(PHI(1,CAP_HPHI+1),PHI(1,CAP_LPHI),288,CAP_RIGHT)
                  A1(ISTM1)=0.0
C
C           Commence S-N sweep.
            CALL CAP_RECEIVE(PHI(1,CAP_LPHI-1),288,CAP_LEFT)
            DO 1000 J=MAX(JSTART,JSTART+CAP_LPHI-2),MIN(NJM1,JSTART+CAP_HPHI-2),1
                  C1(ISTM1)=PHI(ISTM1,J)
C
C                 Commence W-E traverse.
                  DO 1010 I=ISTART,NIM1,1
C
C                       Assemble TDMA coefficients.
                        A1(I)=AE(I,J)
                        B1(I)=AW(I,J)
                        C1(I)=AN(I,J)*PHI(I,J+1)+AS(I,J)*PHI(I,J-1)+SU(I,J)
                        D1(I)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D1(I)-B1(I)*A1(I-1))
                        A1(I)=A1(I)*TERM
1010                    C1(I)=(C1(I)+B1(I)*C1(I-1))*TERM
C
C                 Obtain new PHI.
                  DO 1020 II=ISTART,NIM1,1
                        I=NI+ISTART-II
1020                    PHI(I,J)=A1(I)*PHI(I+1,J)+C1(I)
1000        CONTINUE
            CALL CAP_SEND(PHI(1,CAP_HPHI),288,CAP_RIGHT)
2000  CONTINUE
```

Figure 3.10 : Routine LISOLV partitioned in the second dimensional index J.

```
      SUBROUTINE LISOLV(...)
      .
      .
      ISTM1=ISTART-1
      DO 2000  IT=1,NSW,1
C
C           Commence W-E sweep.
            DO 100  I=ISTART,NIM1,1
                  IF (1.GE.CAP_LPHI.AND.1.LE.CAP_HPHI)A(I,JSTM1)=0.0
                  IF (1.GE.CAP_BLT.AND.1.LE.CAP_BHT)C(I,JSTM1)=PHI(I,JSTM1)
C
C                 Commence S-N traverse.
                  CALL CAP_RECEIVE(BUF,2,CAP_LEFT)
                  IF (CAP_PROCNUM.NE.1)A(I,CAP_BLT-1)=BUF(1)
                  IF (CAP_PROCNUM.NE.1)C(I,CAP_BLT-1)=BUF(2)
                  DO 101  J=MAX(JSTART,JSTART+CAP_LPHI-2),MIN(NJM1,JSTART+CAP_HPHI-2),1
C
C                       Assemble TDMA coefficients.
                        A(I,J)=AN(I,J)
                        B(I,J)=AS(I,J)
                        C(I,J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
                        D(I,J)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D(I,J)-B(I,J)*A(I,J-1))
                        A(I,J)=A(I,J)*TERM
101                     C(I,J)=(C(I,J)+B(I,J)*C(I,J-1))*TERM
                  BUF(1)=A(I,CAP_BHT)
                  BUF(2)=C(I,CAP_BHT)
                  CALL CAP_SEND(BUF,2,CAP_RIGHT)
100         CONTINUE
C
C           Obtain new PHI.
            DO 200  I=ISTART,NIM1,1
                  CALL CAP_RECEIVE(PHI(I,CAP_HPHI+1),1,CAP_RIGHT)
                  DO 102  JJ=MAX(JSTART,JSTART-CAP_HPHI+NJ),MIN(NJM1,JSTART-CAP_LPHI+NJ),1
                        J=NJ+JSTART-JJ
102                     PHI(I,J)=A(I,J)*PHI(I,J+1)+C(I,J)
                  CALL CAP_SEND(PHI(I,CAP_LPHI),1,CAP_LEFT)
200         CONTINUE
            CALL CAP_EXCHANGE(PHI(1,CAP_HPHI+1),PHI(1,CAP_LPHI),288,CAP_RIGHT)
            CALL CAP_EXCHANGE(PHI(1,CAP_LPHI-1),PHI(1,CAP_HPHI),288,CAP_LEFT)
            A1(ISTM1)=0.0
C
C           Commence S-N sweep.
            DO 1000  J=MAX(JSTART,JSTART+CAP_LPHI-2),MIN(NJM1,JSTART+CAP_HPHI-2),1
                  C1(ISTM1)=PHI(ISTM1,J)
C
C                 Commence W-E traverse.
                  DO 1010  I=ISTART,NIM1,1
C
C                       Assemble TDMA coefficients.
                        A1(I)=AE(I,J)
                        B1(I)=AW(I,J)
                        C1(I)=AN(I,J)*PHI(I,J+1)+AS(I,J)*PHI(I,J-1)+SU(I,J)
                        D1(I)=AP(I,J)
C
C                       Calculate coefficients of recurrence formula.
                        TERM=1./(D1(I)-B1(I)*A1(I-1))
                        A1(I)=A1(I)*TERM
1010                    C1(I)=(C1(I)+B1(I)*C1(I-1))*TERM
C
C                 Obtain new PHI.
                  DO 1020  II=ISTART,NIM1,1
                        I=NI+ISTART-II
1020                    PHI(I,J)=A1(I)*PHI(I+1,J)+C1(I)
1000        CONTINUE
2000  CONTINUE
```

Figure 3.11 : Routine LISOLV from TEAMKE1 with loop splitting and array expansion.

The serial code was partitioned on the second index J since the data to be communicated will be stored contiguously in memory. The parallel code (Figure 3.10) contained three pipelines. There is a pipeline surrounding each of the DO 101, DO 102 and the DO 1000 loops. The DO 101 and DO 102 loops, although they are themselves serial, they are surrounded by a parallel loop DO 100 that allows parallelism to be exploited. Each iteration of the DO 100 loop consists of one execution of the DO 101 loop followed immediately by one execution of the DO 102 loop. This does not provide good parallelism, as is shown diagrammatically in the upper half of Figure 3.12, where there is clearly a lot of idle time on each processor. Greater parallelism may be obtained by applying a loop split (see Section 2.11) to the loop DO 100. This will allow the loops J (DO 101) and JJ (DO 102) to each have their own individual I loop and for much less idle time on each processor (lower half of Figure 3.12). A scalar expansion of the arrays A, B, C and D is also required to ensure correct results are obtained from the loop split. This loop split and scalar expansion are applied in Figure 3.11.

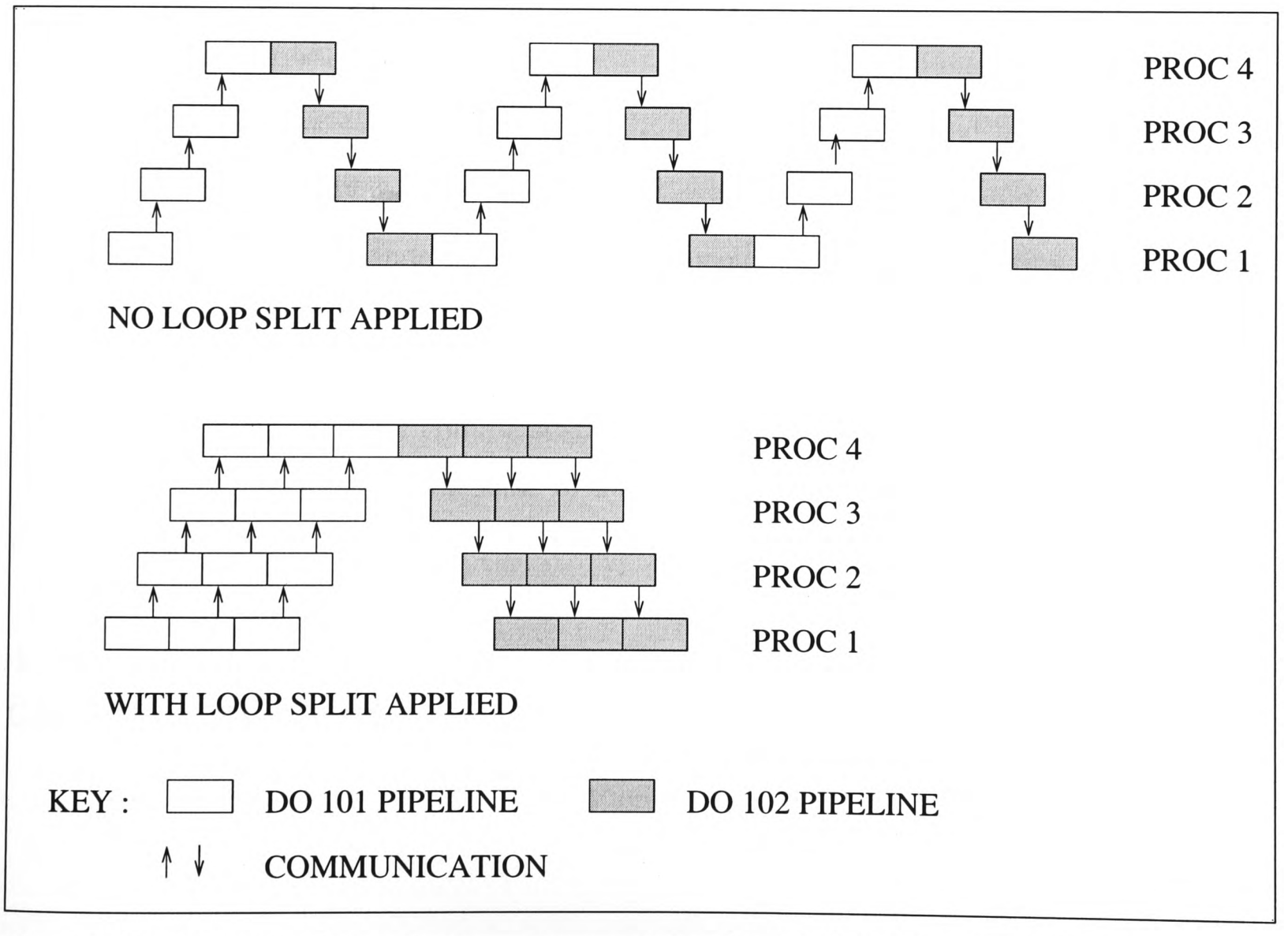


Figure 3.12 : Pipeline with and without loop splitting for LISOLV from TEAMKE1.

The DO 101 loop contains two single communications that both communicate only 1 element each for the arrays A and C. On a parallel machine with a very high communication start-up latency these calls could be very expensive. An obvious optimisation to reduce these communication start-up latencies is to buffer both the elements into a buffer array before communicating as a single communication. This leads to a halving of the communication start up latency for this particular loop. The DO 102 loop also only communicates a single element of the array PHI. This buffering of the data is applied in the code in Figure 3.11.

The third pipeline consists of the communication of a whole line of data. In this case, there is no parallel loop surrounding the DO 1000 loop, and therefore the pipeline will execute serially. This pipeline communication can however be modified in a similar fashion to the pipeline in FAB (Section 3.2). This may be accomplished by dependence deletion leading to the replacement of the pipeline communications CAP_RECEIVE/CAP_SEND with a CAP_EXCHANGE and by extending the coefficient arrays to a second dimension.

The routine MAIN (Figure 3.13) has numerous CAP_EXCHANGE calls. Using the Communications Browser in CAPTools it is possible to interrogate why these communications have been placed. On investigation the browser shows that the communications are required for use in calculation in routines called by the MAIN routine. The communications have been migrated from the commands that require the communicated data, through any surrounding loops and any routine boundaries to its optimal point in the code. During this migration numerous CAP_EXCHANGE calls were migrated to the same point in the code. These communications may require the communication of the same data but for use in different routines and statements. Since communications add to the total runtime of a parallel execution and the data communicated is the same or a subset then it would be of obvious advantage to merge the communication into one call. An illustration of this communication merging may be seen in Figure 3.14. The communication browser shows that the CAP_EXCHANGE of the density array DEN is providing data for 16 statements in 7 different routines (CALCT, CALCED, CACTE, CALCP2, CALCP1, CALCV and CALCU)

```
      PROGRAM TEAM
      .
      .
      CALL CAP_EXCHANGE(V(1,CAP_BLT-2/2-1),V(1,-2/2+CAP_BHT),2/2*288+288,CAP_LEFT)
      CALL CAP_EXCHANGE(U(1,CAP_BHT+1),U(1,CAP_BLT),2/2*288+288,CAP_RIGHT)
      CALL CAP_EXCHANGE(U(1,(CAP_BLT+0/2)-2),U(1,(0/2+CAP_BHT)-1),576,CAP_LEFT)
      CALL CAP_EXCHANGE(V(1,CAP_BHT+1),V(1,CAP_BLT),576,CAP_RIGHT)
      CALL CAP_EXCHANGE(P(1,CAP_BHT+1),P(1,CAP_BLT),288,CAP_RIGHT)
      CALL CAP_EXCHANGE(XPLUSE(CAP_BHT+1),XPLUSE(CAP_BLT),1,CAP_RIGHT)
      CALL CAP_EXCHANGE(TE(1,(CAP_BLT+0/2)-2),TE(1,(0/2+CAP_BHT)-1),576,CAP_LEFT)
      CALL CAP_EXCHANGE(TE(1,CAP_BHT+1),TE(1,CAP_BLT),2/2*288+288,CAP_RIGHT)
      CALL CAP_EXCHANGE(ED(1,(CAP_BLT+0/2)-2),ED(1,(0/2+CAP_BHT)-1),576,CAP_LEFT)
      CALL CAP_EXCHANGE(ED(1,CAP_BHT+1),ED(1,CAP_BLT),2/2*288+288,CAP_RIGHT)
      CALL CAP_EXCHANGE(T(1,(CAP_BLT+0/2)-2),T(1,(0/2+CAP_BHT)-1),576,CAP_LEFT)
      CALL CAP_EXCHANGE(T(1,CAP_BHT+1),T(1,CAP_BLT),2/2*288+288,CAP_RIGHT)
C
C === EDDY VISCOSITY FIELD
C
      CALL PROPS(CAP_BLT,CAP_BHT)
      CALL CAP_EXCHANGE(VIS(1,CAP_BHT+1),VIS(1,CAP_BLT),288,CAP_RIGHT)
      CALL CAP_EXCHANGE(VIS(1,CAP_BLT-1),VIS(1,CAP_BHT),288,CAP_LEFT)
C
C === VELOCITIES AND PRESSURES
C
      CALL CALCU(CAP_BLT,CAP_BHT)
      CALL CAP_EXCHANGE(U(1,CAP_BHT+1),U(1,CAP_BLT),288,CAP_RIGHT)
      CALL CALCV(CAP_BLT,CAP_BHT)
      CALL CAP_EXCHANGE(DV(1,CAP_BLT-1),DV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(V(1,CAP_BLT-1),V(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(AEV(1,CAP_BLT-1),AEV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(AWV(1,CAP_BLT-1),AWV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(ANV(1,CAP_BLT-1),ANV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(ASV(1,CAP_BLT-1),ASV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(APV(1,CAP_BLT-1),APV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(TAUE(CAP_BLT-1),TAUE(CAP_BHT),1,CAP_LEFT)
      CALL CALCP1(CAP_BLT,CAP_BHT)
      CALL CAP_EXCHANGE(DU(1,CAP_BHT+1),DU(1,CAP_BLT),288,CAP_RIGHT)
      CALL CAP_EXCHANGE(DU(1,CAP_BLT-1),DU(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(DV(1,CAP_BLT-1),DV(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(DV(1,CAP_BHT+1),DV(1,CAP_BLT),288,CAP_RIGHT)
      CALL CAP_EXCHANGE(DV(1,CAP_BLT-2),DV(1,CAP_BHT-1),576,CAP_LEFT)
      CALL CAP_EXCHANGE(U(1,CAP_BHT+1),U(1,CAP_BLT),288,CAP_RIGHT)
      CALL CAP_EXCHANGE(U(1,CAP_BLT-1),U(1,CAP_BHT),288,CAP_LEFT)
      CALL CAP_EXCHANGE(V(1,CAP_BLT-1),V(1,CAP_BHT),288,CAP_LEFT)
      CALL CALCP2(CAP_BLT,CAP_BHT)
C
C === TURBULENCE PARAMETERS
      CALL CALCTE(CAP_BLT,CAP_BHT)
      CALL CALCED(CAP_BLT,CAP_BHT)
C
C === TEMPERATURE
C
      CALL CALCT(CAP_BLT,CAP_BHT)
```

Figure 3.13 : Main Program for the Parallel TEAMKE1 Code.

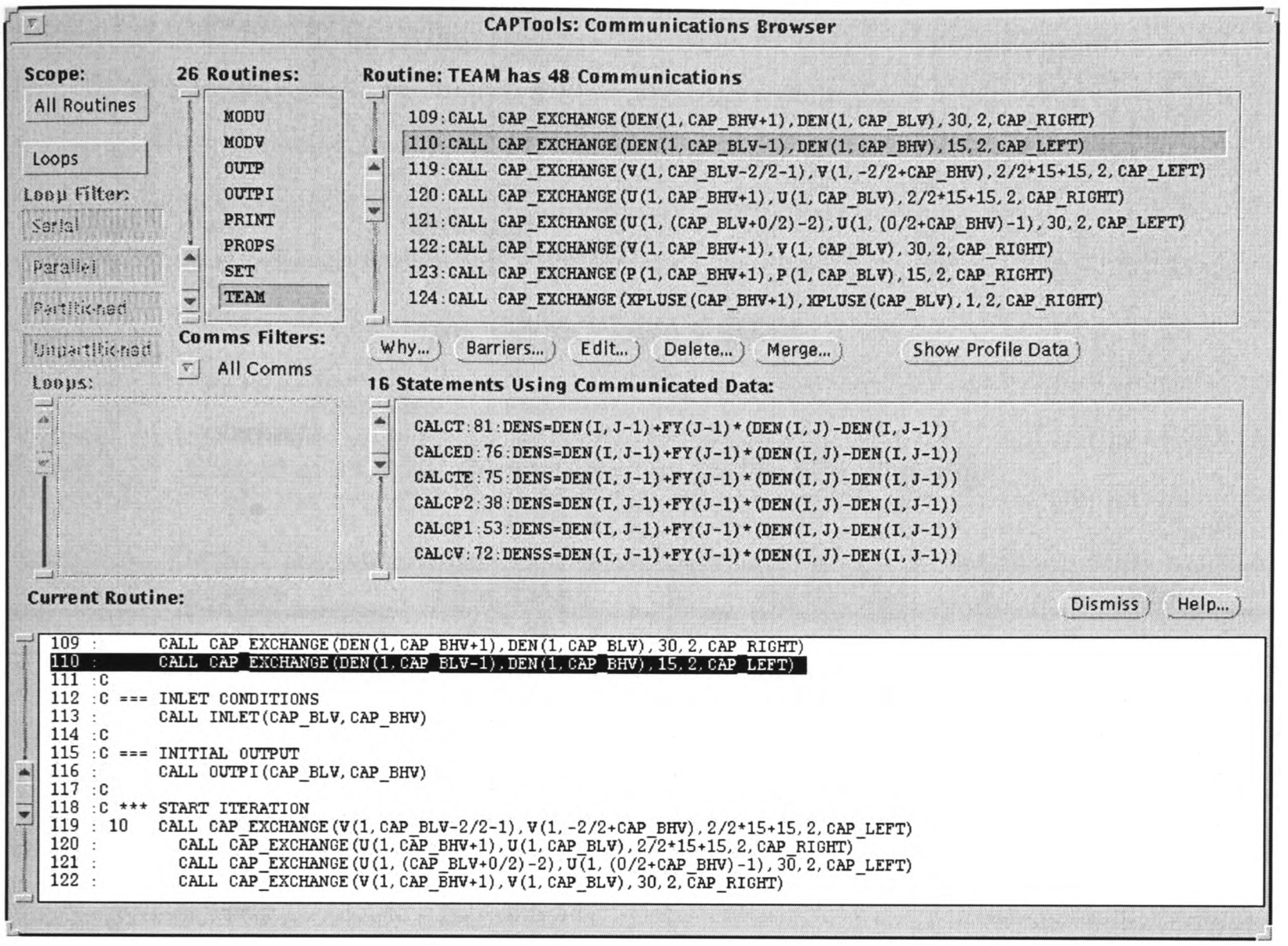


Figure 3.14 : CAPTools Communication Browser illustrating merged communications for the TEAMKE1 Code.

As part of the coefficient generation process in the CALC routines the source term array SUV is set in accordance with the Quick differencing scheme as in Figure 3.15. The values on the right hand side of Figure 3.15 show the possible values of the variables in the code on the left hand side. The migration of the communication of array V is blocked inside all loops by the assignment of array index LYP. Here the index LYP is dependent on the scalar NSVP, from this CAPTools can determine that NSVP can only ever be set to -1 or +1 due to the use of the intrinsic function SIGN. As a consequence, CAPTools can determine that LYP can only ever be set to J-2 or J+1. Therefore, two communications, one for each possible value of LYP are traversed and merged outside the nested loops, allowing bulk communications and further possible migration and merging.

```
            DO 100 I=2,NI,1
                   DO 100 J=2,NJM1,1                          POSSIBLE VALUES :
                                .                             ~~~~~~~~~~~~~~~~~~
                                .
                                NSVP=IFIX(SIGN(1.0,CS))       SIGN    +1.0    -1.0
                                .                             IFIX    +1      -1
                                .                             NSVP    +1      -I
                                .
222                             KYP=J-NSVP                    KYP     J-1     J+1
                                LYP=KYP-(1+NSVP)/2            LYP     J-2     J+1
                                .
                                .
                                SUV(I,J-1)=SUV(I,J-1)-TEMPM*V(I,LYP)
                                .
                                .
100                    CONTINUE
            CONTINUE
```

Figure 3.15 : Section of a CALC Routine showing the QUICK Algorithm.

No of Processors	Time Taken	Speed Up	Efficiency
1	513.41	-	-
2	267.23	1.92	96.1%
4	139.18	3.69	92.2%
8	72.86	7.05	88.1%
16	41.96	12.24	76.5%

Table 3.5 : Results for TeamKE1 with synchronous communications for the Transtech Paramid.

The results in Table 3.5 are for the TeamKE1 code for a problem size of 288x288 for 15 iterations. The speed up results were relative good up to eight processors (7.05 out of 8). However, for 16 processors the efficiency had dropped to 76.5%. This decrease in the speed up is due to the startup and shutdown idle times of the pipelines (in routine LISOLV) increasing as the number of processors increase.

3.4 APPLU.

APPLU is a code from the NASA Parallel (NAS-PAR) benchmark suite [68]. The NAS-PAR 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 factorization but instead implements a symmetric successive over-relaxation (SSOR) numerical scheme. This solves a regular-sparse, block lower and upper triangular system. These systems are obtained from an unfactored implicit finite difference discretisation of the Navier-Stokes equations in three dimensions.

The algorithm consists of four main steps:

1. Forming the right hand side vector;
2. Forming and solving the lower triangular system of equations;
3. Forming and solving the upper triangular system of equations;
4. Updating the solution.

These four stages are iterated until the problem converges to a solution.

The code consists of approximately 3300 lines of Fortran. The code was partitioned by means of a one dimensional partition using Computer Aided Parallelisation Tools. The parallel code obtained consisted of a few essential exchange communications within the main solver. However, the efficiencies obtained from the code were poor. The reasons for such poor efficiencies was investigated.

Some of the exchange communications generated were nested within extra generated loops. These communications were within loops because CAPTools had calculated the minimum data required to communicate for use in the calculation. Certain sections of an array were not communicated since they were not required by the calculation. Consider the following communication from the routine SSOR :

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

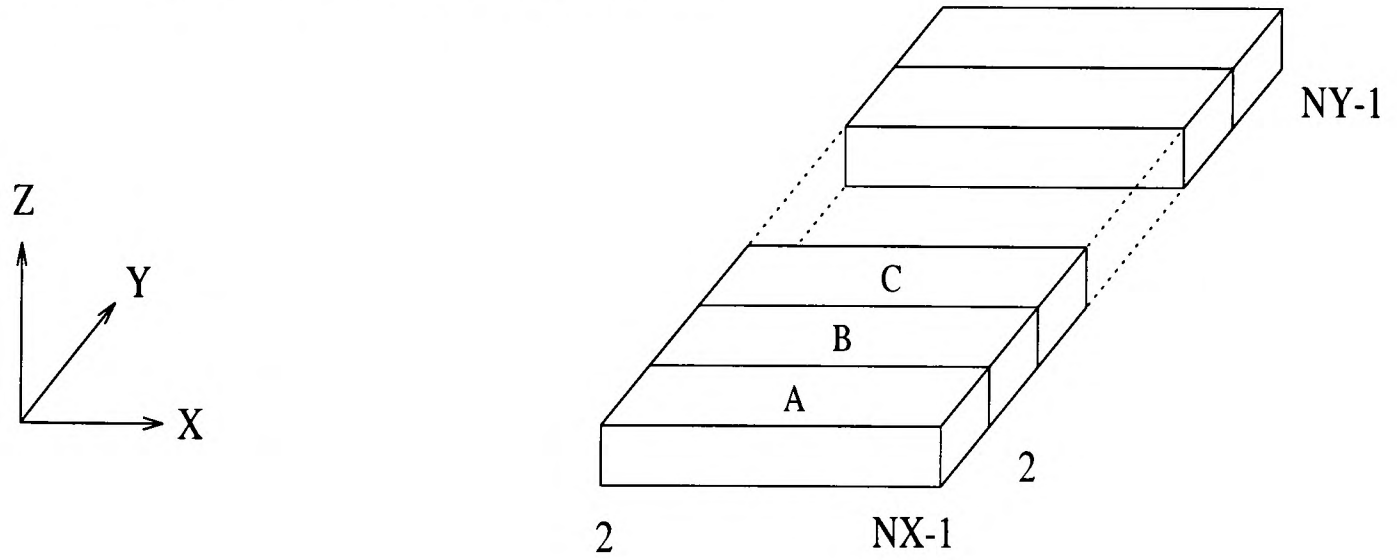
This communication exchanges 5 sets of double precision data from 2 to NX-1 (a total of NX*10-20 words) for each CAP_J from 2 to NY-1, i.e. a total of NY-2 calls to the CAP_EXCHANGE communication. Figure 3.16 shows each of the data blocks to be communicated from 2 to NY-1 as visualised on a cartesian grid and also how the data blocks would be stored in memory.

These data blocks, e.g. A, B and C are stored in memory as shown in the second half of the diagram. In memory either side of the data blocks A, B and C there are shaded areas 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 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 is however more efficient to communicate all the data, including the data not required for

calculation. This therefore necessitated only one communication of all the data as opposed to several calls and reduces the communication start-up latency incurred. This would, however, be ideal in the case where the non-required data 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 to be communicated may far exceed that required and thus extending the total runtime of the parallel code. The variable NX is entered at runtime. 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 be advantageous to convert the CAP_EXCHANGE communication to communicate the whole 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),CAP_LEFT)
```

This test has since been incorporated into CAPTools to attempt to minimise the number of communication calls. If the test could not determine the value of NX or was determined to be much less than that of the allocated memory space then the original conservative communication generated by CAPTools would be preserved.



The data is stored in memory as :

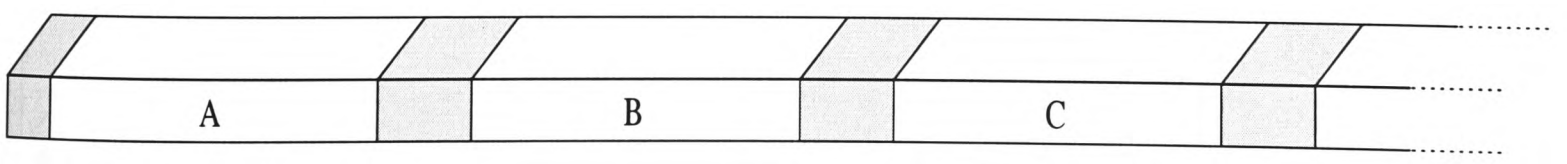


Figure 3.16 : Data Storage of arrays in Fortran.

Within the APPLU code there are two routines BUTS and BLTS that perform the computation of the Block Upper Triangular Solution and the Block Lower Triangular Solution respectively. Both parallel routines contained a pipeline (Section 1.8).

The pipeline code in routine BLTS (Figure 3.17) involves each processor performing its respective calculation sequentially before communicating the data to the next processor. For example processor 1 will execute its calculation (while all the other processors are idle) before communicating the data to the next processor (Figure 3.18). This second processor will then proceed with its calculation before communicating to the next processor. All this time only 1 processor is operating at any given time. It is of a much greater advantage if all the processors are active since idle time leads to inefficient use of parallel processing power.

```
CALL CAP_RECEIVE(V(1,1,1,CAP_BLD-1),5120,CAP_LEFT)
DO K=MAX(2,CAP_LD),MIN(NZ-1,CAP_HD),1
        DO J=2,NY-1,1
                DO I=2,NX-1,1
                  .
                  Calculation requiring communicated data.
                  .
                ENDDO
        ENDDO
ENDDO
CALL CAP_SEND(V(1,1,1,CAP_BHD),5120,CAP_RIGHT)
```

Figure 3.17 : All Pipeline code communicating all data for routine BLTS from APPLU.

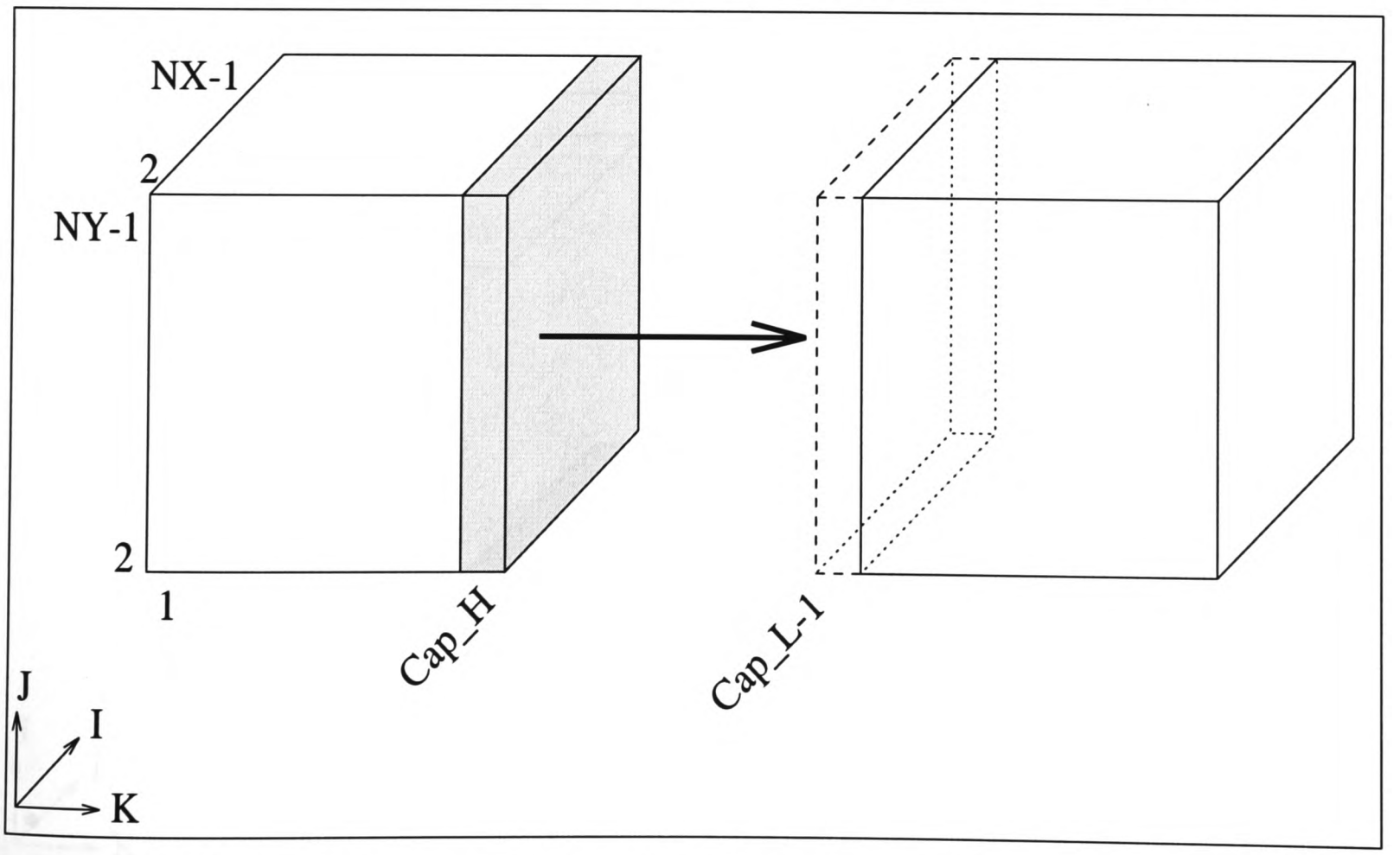


Figure 3.18 : Diagrammatic representation of Pipeline communicating all data.

To overcome this, a loop interchange (see Section 2.11) was applied to the J and K loops prior to the communication calculation (Figure 3.19). This leads to the communications to be generated within the outermost loop and for the data to be communicated as lines of data (Figure 3.20). This in turn allows the next processors in the pipeline to execute its calculation and communication. This provides some amount of parallelism to be introduced. There is however still an element of idle time due to the start up and shut down of the pipeline. Also there are several more communications in comparison with the communications of the original serial (Figure 3.18). This will mean that the number of communication start-up latencies will increase. On most, if not all, parallel systems this addition in communication start-up time will not exceed the time saved by the reduction of the pipeline start-up and shutdown and resultant parallelism.

```
DO J=2,NY-1,1
        CALL CAP_RECEIVE(V(1,1,J,CAP_BLD-1),160,CAP_LEFT)
        DO K=MAX(2,CAP_LD),MIN(NZ-1,CAP_HD),1
                DO I=2,NX-1,1
                  .
                  Calculation requiring communicated data.
                  .
                ENDDO
        ENDDO
        CALL CAP_SEND(V(1,1,J,CAP_BHD),160,CAP_RIGHT)
ENDDO
```

Figure 3.19 : Line Pipeline code communicating lines of data for routine BLTS from APPLU.

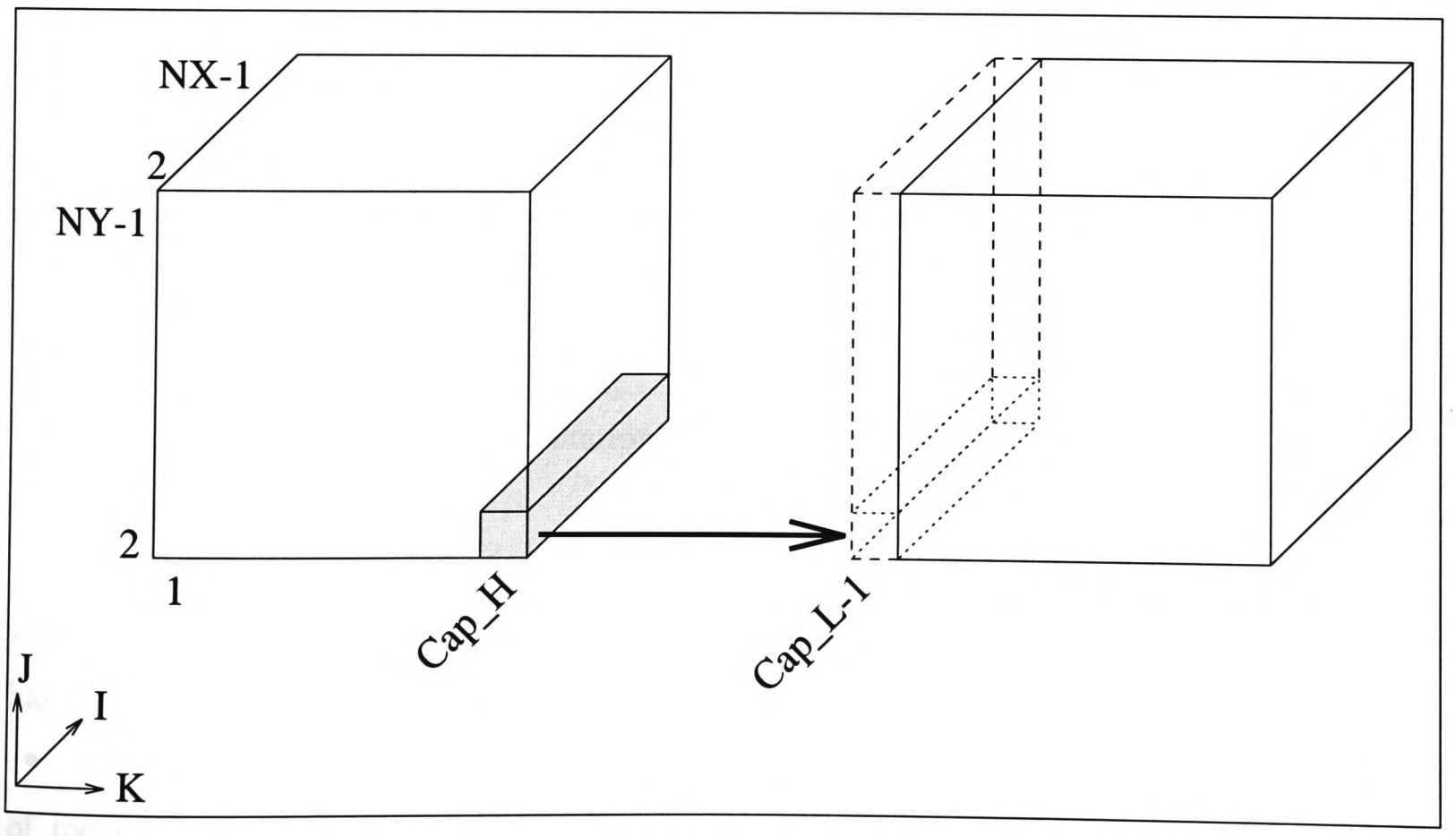


Figure 3.20 : Diagrammatic representation of Pipeline communicating line data.

A further loop interchange of loops I and K (Figure 3.21) - could be applied such that data is communicated as points of data (Figure 3.22). This pipeline of point values drastically reduces the amount of start-up and shutdown of the pipeline. However, the number of communications has also increased quite heavily. On most parallel machines this will lose efficiency not because of the pipeline start-up and shutdown but because of the number of communication start-up latencies.

```
DO J=2,NY-1,1
        DO I=2,NX-1,1
                CALL CAP_RECEIVE(V(1,I,J,CAP_BLD-1),10,CAP_LEFT)
                DO K=MAX(2,CAP_LD),MIN(NZ-1,CAP_HD),1
                  .
                  Calculation requiring communicated data.
                  .
                ENDDO
                CALL CAP_SEND(V(1,I,J,CAP_BHA),10,CAP_RIGHT)
        ENDDO
ENDDO
```

Figure 3.21 : Pipeline code communicating points of data for routine BLTS from APPLU.

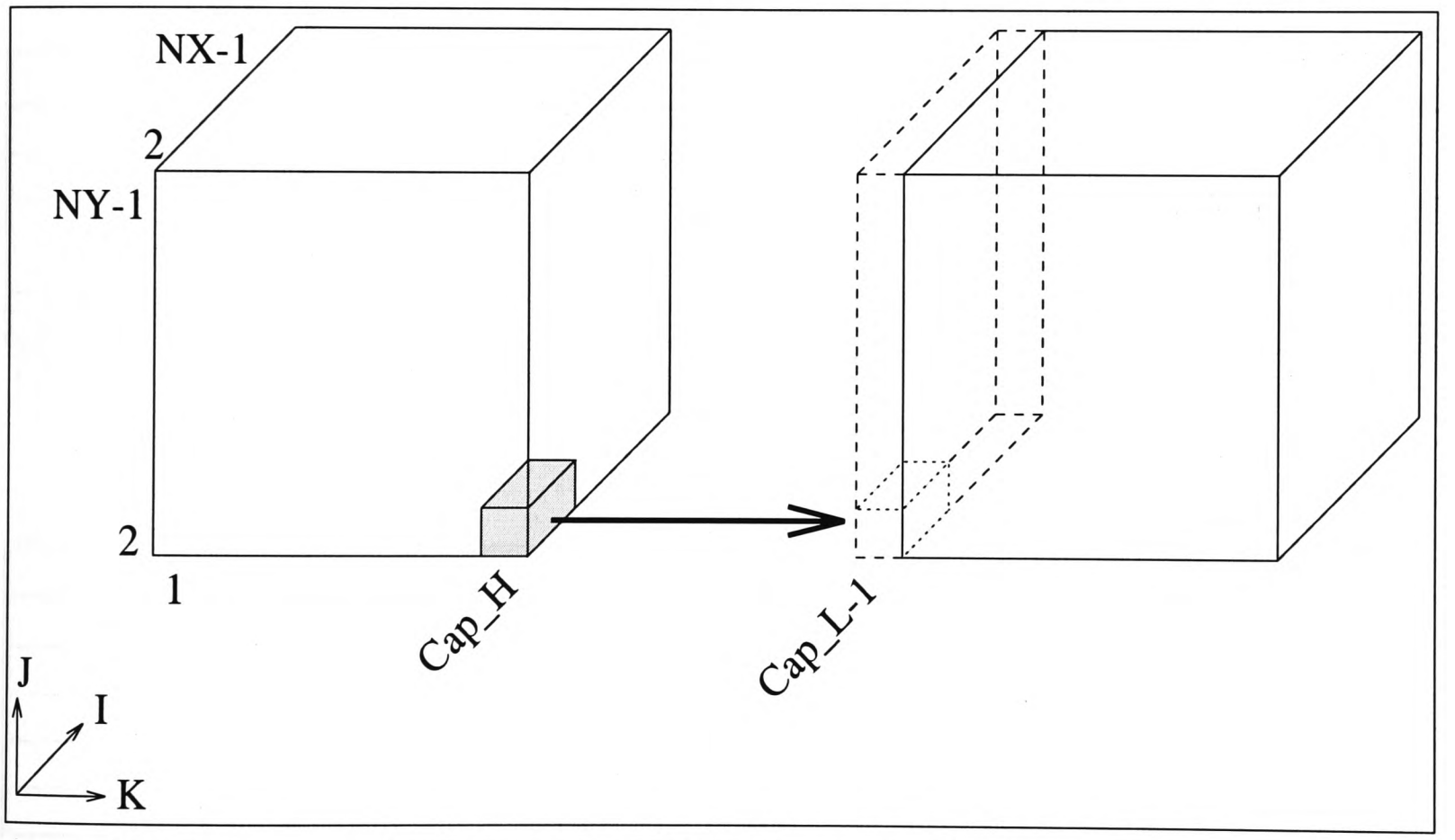


Figure 3.22 : Diagrammatic representation of pipeline communicating point data.

The amount of computation required for each communication is often referred to as the granularity of the pipeline [69]. The ALL pipeline has a coarse granularity; the LINE pipeline has a medium granularity; while the POINT pipeline has a fine granularity.

As mentioned earlier (Section 1.8), these pipelines caused a degradation in the efficiency of the code as the number of processors increased. There are two main reasons for this

degradation. The first, is that during the start-up and shutdown of the pipeline some of the processors will be idle. The second reason is due to the number of communications involved and their related start-up latencies. These pipelines often consist of a high ratio of communication to calculation.

No of Processors	Time Taken	Speed Up	Efficiency
1	345.90	-	-
2	274.98	1.26	62.9%
4	240.94	1.43	35.9%
8	231.26	1.50	18.7%
12	236.79	1.46	12.2%

Table 3.6 : Results for APPLU with ALL pipelines for the Transtech Paramid (32x32x32 Problem for 50 iterations).

No of Processors	Time Taken	Speed Up	Efficiency
1	345.29	-	-
2	187.36	1.84	92.1%
4	114.79	3.01	75.2%
8	74.74	4.62	57.7%
12	63.18	5.46	45.5%

Table 3.7 : Results for APPLU with LINE pipelines for the Transtech Paramid (32x32x32 problem for 50 iterations).

No of Processors	Time Taken	Speed Up	Efficiency
1	350.78	-	-
2	196.28	1.79	89.3%
4	124.09	2.83	70.7%
8	81.07	4.33	54.1%
12	69.83	5.02	41.9%

Table 3.8 : Results for APPLU with POINT pipelines for the Transtech Paramid (32x32x32 problem for 50 iterations)

The interchanging of the loops surrounding the pipeline did increase the efficiency of the code somewhat. However, there was still a substantial amount of efficiency lost due to the communication start-up latency times for communications especially in the pipelines where there was a great deal of communication involved.

The results for pipelines communicating all, lines and points of data are shown in Table 3.6, Table 3.7 and Table 3.8 respectively. The results are for 32x32x32 problem ran on the Transtech Paramid machine.

The use of POINT and LINE pipelines is clearly more effective than the ALL pipelines. The use of ALL pipelines cause the parallel code to be serial in these sections of code and thus reduces the efficiencies obtainable in the remainder of the parallel code. The use of LINE communication in the pipelines is clearly more effective than the POINT communications. The POINT pipelines require more communications than the LINE pipeline and causes more communication startup latencies.

The APPLU parallel code was also tested on the Parsys SN9500 machine using a LINE pipeline (Table 3.9) and a POINT pipeline (Table 3.10).

No of Processors	Time Taken	Speed Up	Efficiency
1	1593.83	-	-
2	872.27	1.83	91.4%
3	593.74	2.68	89.5%
4	454.21	3.51	87.7%
5	381.41	4.18	83.6%
6	314.28	5.07	84.5%

Table 3.9 : Results for APPLU with LINE pipelines for the Parsys SN9500 (24x24x24 problem for 50 iterations).

No of Processors	Time Taken	Speed Up	Efficiency
1	1595.84	-	-
2	874.62	1.82	91.2%
3	588.08	2.71	90.4%
4	444.41	3.59	89.8%
5	372.16	4.29	85.8%
6	300.03	5.32	88.6%

Table 3.10 : Results for APPLU with POINT pipelines for the Parsys SN9500 (24x24x24 problem for 50 iterations).

The results for the Parsys SN9500 show that the use of POINT pipeline is more advantageous than the LINE pipeline. This shows that the Parsys machine is more effective at communicating small amounts of data more often than communicating large amounts of data less often. This is due to the smaller amount of communication startup latency incurred.

The application of Iteration Grouping (discussed in Section 1.9) was also investigated on the pipelines in APPLU to determine if there was any advantage in their use. The results for the various number of iterations grouped as one communication, for 4 processors are shown in Table 3.11.

No of Processors	No of Iterations Grouped	Time Taken	Speed Up	Efficiency
1	1	350.78	-	-
4	1	115.46	3.04	75.9%
4	2	109.81	3.19	79.8%
4	3	107.93	3.25	81.2%
4	5	106.56	3.29	82.3%
4	6	106.25	3.30	82.5%
4	10	105.86	3.313	82.84%
4	15	105.92	3.311	82.79%
4	30	106.90	3.28	82.0%

Table 3.11 : Results for APPLU with POINT pipelines and Iteration Grouping for the Transtech Paramid on 4 processors (32x32x32 problem for 50 iterations).

Number of Processors	Number of Iterations Grouped
2	30
3	30
4	10
5	15
6	15
7	10
8	6
9	10
10	10

Table 3.12 : Number of Iteration Groupings required for varying Number of Processors for APPLU.

The results show that for 4 processors a much improved efficiency may be obtained from calculating 10 iterations before grouping the data from these iterations and communicating to the next processor. The number of iterations to group however varies depending on the number of processors involved (Table 3.12).

The number of iteration groupings also depends on the problem size, the parallel machine used and varies from code to code. It is therefore not very easy to determine the most effective number of iterations to group without first conducting several trial runs.

3.5 ARC3D.

ARC3D is a code from the Perfect Club Benchmark Suite [70] of codes and was developed by the NASA Ames Research Center. It is a three-dimensional Euler code solved with an implicit algorithm, central differences and full geometry. The code consists of approximately 3600 lines of code in 25 routines.

This code was also partitioned in the L dimension using CAPTools. During the parallelisation stage it was necessary to conduct a routine copy for the routines VPENTA and VPENTA3 using the Routine Copy transformation in CAPTools (see Section 2.11). The code in Figure 3.23 shows there are three calls to the routine VPENTA from routine STEPF3D. This routine is called three times, each time with a different orientation of the problem data being passed into the workspace. The first call operated on the J and K orientation, the second call on the K and J orientation and the third call on the L and J orientation. The first two calls did not involve the L-dimension and an execution control mask on their calls is sufficient. The third call operates on the partitioned L-dimension and it is desirable to allow masks to be applied to statements within this routine call. The routine VPENTA3 was also called three times by the routine STEPF3D in a similar fashion and required a routine copy.

```
      DO 241  K=2,KM,1
                 DO 241  J=2,JM,1
                            FR(J,K,1)=S(J,K,L,N)
241               CONTINUE
      CONTINUE
      CALL VPENTA(AR,BR,CR,DR,ER,WR1,WR2,FR,2,JM,2,KM)
                 .
                 .
      DO 432  K=2,KM,1
                 DO 432  J=2,JM,1
                            FR(K,J,1)=S(J,K,L,N)
432               CONTINUE
      CONTINUE
      CALL VPENTA(AR,BR,CR,DR,ER,WR1,WR2,FR,2,KM,2,JM)
                 .
                 .
      DO 533  L=2,LM,1
                 DO 533  J=2,JM,1
                            FR(L,J,1)=S(J,K,L,N)
533               CONTINUE
      CONTINUE
      CALL VPENTA(AR,BR,CR,DR,ER,WR1,WR2,FR,2,LM,2,JM)
```

Figure 3.23 : Section of code from STEPF3D routine of ARC3D code.

The two copied routines CAP_VPENTA1 and CAP_VPENTA31 both contain 2 separate pipelines. The original pipelines generated were serial. To overcome this a loop interchange was applied to both routines such that the DO K was the outermost loop instead of the DO J loop as shown in Figure 3.24 which shows the pipelines from routine CAP_VPENTA1. The first pipeline consists of three individual calls that communicate data for use in the pipeline calculation. To improve the performance of this pipeline, the data from the three communications was placed in a buffered array (similar to the TEAMKE1 optimisation in Section 3.3) and communicated as a single communication, to reduce the amount of communication start-up latency. The same was applied to the CAP_VPENTA31 routine.

```
      DO 11 K=KL, KU, 1
            CALL CAP_RECEIVE(X(CAP_BLS-2,K),2,CAP_LEFT)
            CALL CAP_RECEIVE(Y(CAP_BLS-2,K),2,CAP_LEFT)
            CALL CAP_RECEIVE(F(CAP_BLS-2,K),2,CAP_LEFT)
            DO 3 J=MAX(JL+2,CAP_BLS-2),MIN(JU-2,CAP_BHS-2),1
                  LD2=A(J,K)
                  LD1=B(J,K)-LD2*X(J-2,K)
                  LD=C(J,K)-(LD2*Y(J-2,K)+LD1*X(J-1,K))
                  LDI=1.D0/LD
                  F(J,K)=(F(J,K)-LD2*F(J-2,K)-LD1*F(J-1,K))*LDI
                  X(J,K)=(D(J,K)-LD1*Y(J-1,K))*LDI
                  Y(J,K)=E(J,K)*LDI
3           CONTINUE
            CALL CAP_SEND(X(CAP_BHS-1,K),2,CAP_RIGHT)
            CALL CAP_SEND(Y(CAP_BHS-1,K),2,CAP_RIGHT)
            CALL CAP_SEND(F(CAP_BHS-1,K),2,CAP_RIGHT)
11    CONTINUE
         .
         .
         .
      DO 15  K=KL,KU,1
            CALL CAP_RECEIVE(F(CAP_BHS+1,K),2,CAP_RIGHT)
            DO 4  J=MAX(2, JU-CAP_BHS), MIN(JU-JL, JU-CAP_BLS), 1
                  JX=JU-J
                  F(JX,K)=F(JX,K)-X(JX,K)*F(JX+1,K)-Y(JX,K)*F(JX+2,K)
4           CONTINUE
            CALL CAP_SEND(F(CAP_BLS,K),2,CAP_LEFT)
15    CONTINUE
```

Figure 3.24 : The pipelines in routine CAP_VPENTA1 in ARC3D code.

For the Transtech Paramid on a problem size of 40x33x40 the time taken on 8 processors was 398.08 seconds in relation to 1373.95 seconds in serial (Table 3.13). This produced a speed up of 3.45 on 8 processors equivalent to an efficiency of 43.1%. A timing profile of the pipeline routines (Table 3.14) revealed that they were slowing down as the number of processor increased. The time to communicate the data is much more than the calculation time required. The communications are therefore dominating the runtime in these routines and thus providing poor efficiencies to the whole program. The removal of the effect of the pipelines provided a speed up of 4.61 and an efficiency of 57.6% on 8 processors.

No of Processors	Time Taken	Speed Up	Efficiency
1	1373.95	-	-
2	830.71	1.65	82.7%
4	559.99	2.45	61.3%
6	465.85	2.95	49.1%
8	398.08	3.45	43.1%

Table 3.13 : Results for ARC3D on the Transtech Paramid (40x33x40 problem).

Number of Processors	Pipelines Time (seconds)
1	53.29
2	106.24
4	108.89
6	110.47
8	111.52

Table 3.14 : Time Taken by the Pipeline in ARC3D for a 40x33x40 problem.

3.6 Conclusion.

Using CAPTools provided a sound foundation for parallelising structured mesh codes. However, even better efficiencies could be achieved from these codes by applying some simple optimisations. The communications also reduced the efficiencies obtainable significantly. The dominant factor within the communications was the data transfer time with communication start-up latencies also significant. This was significant throughout the codes but especially in sections of pipelined code where there were large amounts of communication present.

The next chapter discusses methods applied to these codes by hand to increase their performance by overlapping the communications with unrelated calculation. This method of overlapping the communications is applied to the exchange of data communications as well as the pipelined communications.

Chapter 4

4 Application of Overlapping Communications for Structured Mesh Computational Mechanics Codes.

4.1 Introduction

This chapter investigates the methods for applying overlapped communications to several common circumstances that arise in parallel codes. These methods were tested on the codes previously parallelised in Chapter 3 by changing the code as necessary by hand. These methods of changing the code for overlapping communications were then applied as an automatic generation stage within CAPTools in the chapter that follows.

In all of the codes parallelised by CAPTools (Chapter 3) the communications have been migrated (Section 2.8.3) to their furthest point from the usage of the communicated data. This leads to the communications to be moved out of loops and wherever possible merged with other communications. This has been seen in all of the codes parallelised in Chapter 3.

The code between the communication of data and the calculation using that relevant data could be used to overlap the communication and thus hide the overheads incurred (Section 4.2). This chapter discusses methods used to hide these communications and therefore increase the performance of the parallel codes.

In order to apply overlapped communications to these codes it was first necessary to investigate the different methods of applying asynchronous communications (Section 4.2) available. It was also necessary to extend the current CAPLib communication library [48,49] to include asynchronous communications and their associated synchronisation points (Section 4.4).

The communications considered for overlapping were the CAP_EXCHANGE (Section 1.6) communications and the pipelines (Section 1.8). The exchange of data is frequently required and this is often the main method of communication within the codes previously parallelised in Chapter 3. These exchange communications were often migrated to some

distance from the statements that required the communicated data. They therefore offer good potential for increasing the efficiency of the code. The pipelines were also selected since they drastically reduced the performance of several of the parallel codes (Chapter 3). It would therefore be of great advantage to reduce the degradation of the efficiency of these parallel codes.

Three different methods were applied to the CAP_EXCHANGE communications: Simple overlapping (Section 4.5); Partial overlapping with loop unrolling (Section 4.6); and Partial overlapping using a conditional statement (Section 4.7). These methods will now be referred to in shorthand as SIMPLE, PARTIAL and UNROLL overlapping communications respectively.

The methods discussed in this chapter are also summarised in the paper referenced in [71].

4.2 Communications in Distributed Memory Systems.

Most parallel codes on Distributed Memory Systems (DMS) use communications that act synchronously. These communications, when initiated, halt computation until the communication has been completed. In some communications the time taken to move the data will be minimal and will not affect the total overall execution time drastically. However, as the amount of data to be communicated increases, so the time spent in this task will increase. Other factors to be taken into account are parallel systems where the communications have a high start-up latency. Here the high start-up latency will significantly increase the time taken for the communication, even if the actual data movement takes only a minimal amount of time.

The above two factors can extend the total execution time of a code considerably as the total number of communications and the amount of data to be communicated increases. It would be very advantageous to overcome this handicap if the communications were somehow concealed into the normal parallel execution time of the code. This can be achieved by using asynchronous communications. This type of communication allows the code execution to continue while the communication occurs concurrently. Unfortunately, the communication start up associated with each communication may not be overlapped.

There are two main methods of employing asynchronous communications in software:

Firstly, where communication occurs at the same time as the calculation, where the latter makes use of the latest available values, whether or not these are the same as those calculated in the serial case.

Consider the section of code in Figure 4.1. The values of A(J) are calculated for every iteration of the loop I before being communicated asynchronously. This form of communication allows the 'other code' to continue execution while the communication is proceeding. If this 'other code' is sufficient to allow the communication to complete before the values of A(J) are used then the latest values will be used in S_1. However, if there is insufficient code between the assignment and usage of A then the values from previous iterations of loop L_1 will be used.

This method is based on the original work of Bertsekas and Tsitsiklis [72] and has been proved successful in increasing the efficiencies of some suitable codes [73, 74, 75, 76].

```
            Initialise values of A(J)
L1          DO I= 1, NITER
                DO J =.   ..
                    A(J)=...
                ENDDO
                Asynchronous Communication of A(J)
                    .
                    .
                Other code
                    .
                    .
                DO J =...
S1                  ... = A(J)
                ENDDO
            ENDDO
```

Figure 4.1 : Pseudo Code of Asynchronous Communication.

However, this method, can affect the convergence rate and properties of a code in a non-deterministic manner. One of the requirements (Section 1.3) of the parallel code is that the results should be identical to that obtained by the serial code. It is for this reason that this method is considered inappropriate for this work, particularly as there is no restriction on the type of application code algorithms that may be parallelised.

The second method is to use overlapped communications with synchronisation points. This allows communications to be overlapped with calculations that do not require the use of the communicated data. Prior to the use of the communicated data within a calculation a synchronisation point must be set to ensure that the communication has completed before

allowing the calculation to continue. The synchronisation points ensure that the data communicated is the correct and latest data to be used in the calculation. Synchronisation compels the communications to be deterministic which allows the code to be reliable and for the same results as the serial code to be obtained, whilst still improving performance. Kennedy and Nedeljković have done some work in this area [77]. Consider the code in Figure 4.2.

```
Initialise values of A(J)
DO I= 1, NITER
    DO J =...
        A(J)=...
    ENDDO
    Asynchronous Communication of A(J)
        .
        .
    Other code
        .
        .
    Synchronisation of Communication.
    DO J =...
        ... = A(J)
    ENDDO
ENDDO
```

Figure 4.2 : Pseudo Code of Asynchronous Communications with Synchronisation Points.

The code is similar to that in Figure 4.1 except that there is now an additional 'synchronisation of communication' to ensure that the communication has completed before allowing the code to pass that point and for the communicated data to be used. This ensures that the usage has the most up to date calculated values.

In some cases it may not be possible to overlap the communications with the calculation because the data being communicated is required immediately afterwards for calculation. Since the synchronisation can also incur a time penalty then it can be concluded that if the communication cannot be overlapped with calculation then it should be left synchronous.

There has also been some work involving the use of multiple threads to overlap computation and calculation [78]. This method requires at least two sub-domains on each processor to allow two threads of execution to operate concurrently. The first row of processors in Figure 4.3 shows four sub-domains (A, B, C, D) distributed onto four processors. To distribute this data onto four processors with two multithreads each requires the sub-domains A, B, C and D to be split in two and distributed onto different processors (second row in Figure

4.3). When one sub-domain (thread) is communicating or is idle the second thread may execute and therefore reduce the processor idle time.

This method, however, does double the number of communications and the volume of data to be communicated. In the first row in Figure 4.3 the four processors have a total of 6 overlap areas of size N to be updated. For a multi-threaded execution (second row of Figure 4.3) on four processors the number of sub-domains is doubled to 8 and the number of overlap areas to be updated would be 14. This method is also reliant on context switching systems that are not widely available and typically incur a heavy overhead.

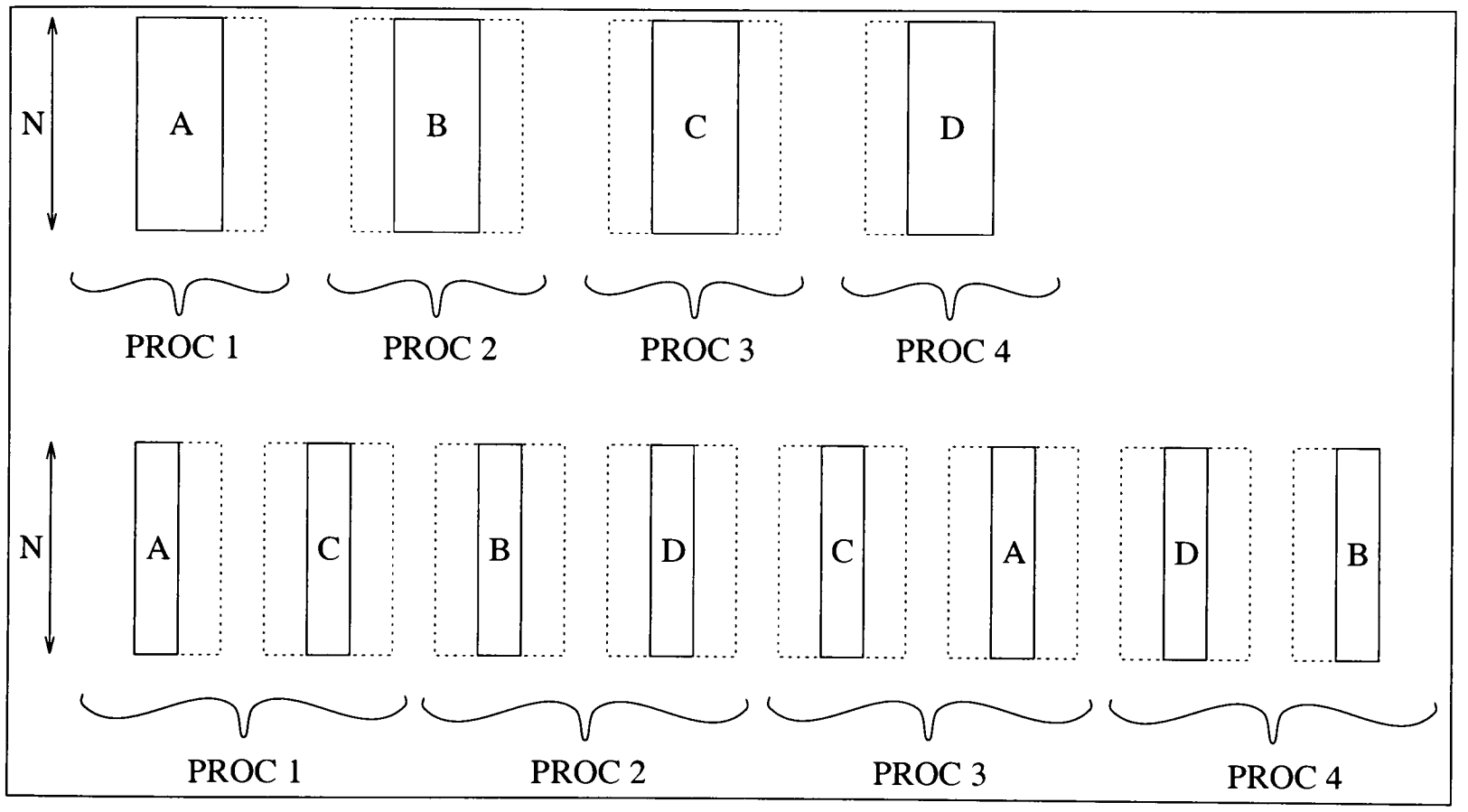


Figure 4.3 : Non-multithreaded and multithread distribution of data.

There has been some research into software-controlled prefetching [79]. This requires support from both hardware and software. The processor must provide a special prefctch instruction while the software uses this instruction to inform the hardware that it intends to use a particular data item. If the data is not currently in the cache memory then it is fetched from memory. While the memory services the data miss, the computation can continue to execute as long as it does not need the requested data. The memory accesses are therefore overlapped with computation. Better improvement may be obtained from using prefetching in conjunction with other optimisations such as blocking and loop transformations. Blocking, instead of operating on whole rows or columns of an array, operates on blocks of the array. Loop transformations that may be used to obtain improvement are loop interchange, loop skewing and loop reversal.

The use of prefetching requires the user or compiler to insert the prefetches in the correct place in the code. Also to obtain the most from prefetching some form of transformation might also need to be applied. There are also additional overheads involved with the additional instructions required in the machine code. This method unfortunately requires specialist hardware support such as fast context switching and prefetch instructions that may not be available on all parallel machines.

4.3 Hardware for Asynchronous Communications.

Whether or not asynchronous communications may be used is however hardware dependent. Many DM parallel systems such as the Intel Paragon [80] and Transtech Paramid [81] have compute nodes which consist of at least two processors - one for communication and one or more for computation. The Intel Paragon node consists of two Intel i860XP processors where one is concerned with the calculation while the other is concerned with communication. The Transtech Paramid on the other hand consists of two different processors: an Intel i860 for the computation and a T805 Transputer processor for communicating. This type of architecture facilitates the communication and calculation to be executed concurrently via the separate processors.

There are also processors available, such as the Inmos Transputer T9000 [6], as used in the Parsys SN9500 [82], which have an in-built facility to communicate while also administering calculation.

Figure 4.4 shows the architecture of a Transputer processor. The calculation is executed in the processor pipeline while any communication may be executed concurrently in one of the four link interfaces that connect to other processors.

However, there are DM parallel systems such as a workstation cluster where asynchronous communication is not available. Other parallel machines, such as the Cray T3D and the IBM SP2 have the potential of benefitting from asynchronous communication. Unfortunately, initial investigations displayed little improvement in the performance of parallel code after the application of asynchronous communications. There is at present little information on the successful use of asynchronous communication on these parallel systems.

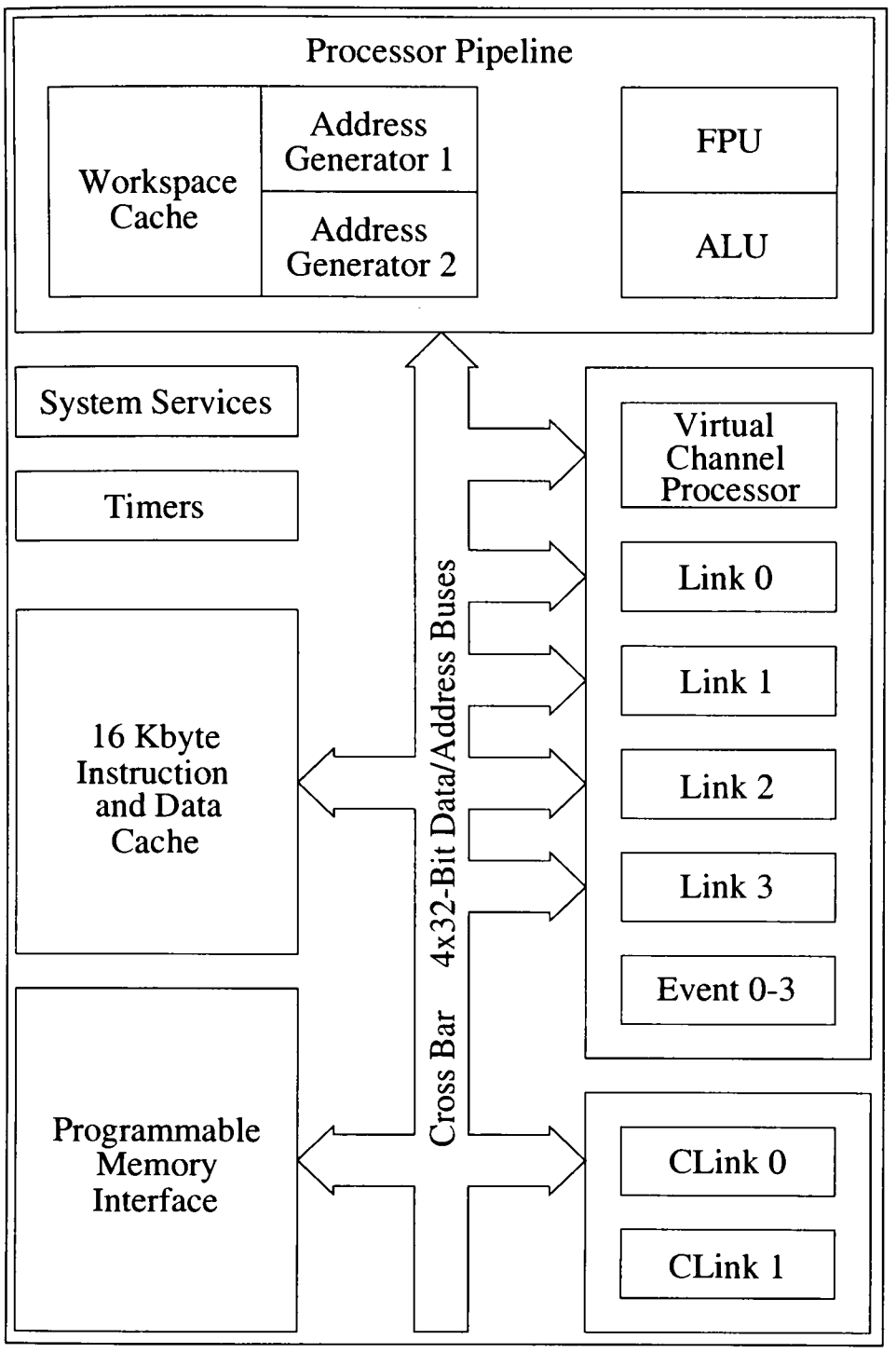


Figure 4.4 : The Transputer Architecture.

4.4 Asynchronous Communication Utilities.

To take advantage of this hardware for asynchronous communications required the extension of the CAPTools Communication Library (Section 1.6) to include asynchronous/overlapping communications. The requirements for these asynchronous/overlapping communications were specified for the developer of the communication library who then provided these as additional communications within the communication library.

The overlapping communication calls are very similar to their synchronous counterparts (Section 1.6) except that they have additional parameters to allow for synchronisation. The calls and parameter list for the overlapping communications 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*, *Receive Synchronisation*, *Send Synchronisation*)

where the *Send Synchronisation* is the synchronisation value for the send, the *Receive Synchronisation* is the synchronisation value for the receive.

Synchronisation points may also be required in association with the overlapping communications to ensure that the communication has completed before allowing the execution of the program to continue. Once the communication is completed, the synchronisation point will return and allow the program to continue. The calls and their parameters are as follows :

CAP_SYNC_SEND(*Direction, Send Synchronisation*)

CAP_SYNC_RECEIVE(*Direction, Receive Synchronisation*)

CAP_SYNC_EXCHANGE(*Direction, Receive Synchronisation, Send Synchronisation*)

The synchronisation values are set during the call to the overlapping communication and consist of a unique number for each communication. These values are used in the synchronisation calls to ensure that the communication has completed. The value of each synchronisation point is incremental. This allows the synchronisation point to synchronise on the highest synchronisation value to ensure that all previous communications in that direction have completed. The overlapping communications are also flexible enough to allow multiple synchronisation points for the same synchronisation value. This may reduce the parallel efficiencies obtained but it does allow the generation of overlapping communications to deal with cases where the communicated data may be required at different places in the code. If the synchronisation value is set to 0 then the synchronisation points will not synchronise. This can be very useful if, for instance, a synchronisation point is not required on the first iteration of a loop.

4.5 Simple Overlapping of Exchange Communications.

The pseudo code in Figure 4.5a shows an example of a typical synchronous EXCHANGE communication as generated by CAPTools for a parallel code. The pseudo code in Figure 4.5b shows the overlapping communication of the same piece of code as modified by hand to take advantage of the code, that does not use the communicated data, to overlap the communication.

```
DO I=1,NITER
  CALL CAP_EXCHANGE(A(Cap_H+1),A(Cap_L),..)

  {* Other code which does not *}
  {* use communicated data *}


  {* Calculation using communicated data *}
ENDDO
```

Figure 4.5a : Synchronous.

```
DO I=1,NITER
  CALL CAP_AEXCHANGE(A(Cap_H+1),A(Cap_L),..)

  {* Other code which does not *}
  {* use communicated data *}

  CALL CAP_SYNC_EXCHANGE(..)
  {* Calculation using communicated data *}
ENDDO
```

Figure 4.5b : Overlapping.

Figure 4.5 : Synchronous and overlapping pseudo code illustrating SIMPLE overlapping with unrelated code.

The application of an overlapping CAP_EXCHANGE to replace the synchronous EXCHANGE is straight forward, as long as there is code in between the communication start point (i.e. the CAP_AEXCHANGE call) and the synchronisation point (i.e the CAP_SYNC_EXCHANGE call) that does not use the data being communicated. CAPTools always generates a communication at a position as far as possible from the point that the data being communicated is being used in calculation. This is achieved by the migration of communications (Section 2.8.3). This will often provide an ample amount of code (more precisely the calculation time) to fully overlap the communication. Consider the routine SSOR of the APPLU code in Figure 4.6.

Using the communication browser (Figure 4.7) the data communicated (the array U) by the first exchange S_1 was not required for use in calculation until inside the routine JACLD. In between this communication call and the call to routine JACLD is a set of four nested loops that sets the values of the array RSD. These nested loops do not use the communicated data. At present the communications are synchronous (Section 1.6). These communications wait until they have completed exchanging their data before allowing the calculation to continue. If the amount of data to communicate is substantial then the time spent waiting for the completion could be very costly to the efficiency of the parallel code.

```
            DO ISTEP=1,ITMAX,1
S1              CALL CAP_EXCHANGE(U(1,1,1,CAP_BLD-1),U(1,1,1,CAP_BHD),1440,CAP_LEFT)
S2              CALL CAP_EXCHANGE(U(1,1,1,CAP_BHD+1),U(1,1,1,CAP_BLD),1440,CAP_RIGHT)
      C
                IF ((MOD(ISTEP,INORM).EQ.0).AND.(IPR.EQ.1)) THEN
                    IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=IOUT,FMT=1001)ISTEP
                ENDIF
      C
      C         Perform SSOR iteration
      C
                DO K=MAX(2,CAP_LD),MIN(NZ-1,CAP_HD),1
                    DO J=2,NY-1,1
                        DO I=2,NX-1,1
                            DO M=1,5,1
                                RSD(M,I,J,K)=DT*RSD(M,I,J,K)
                            ENDDO
                        ENDDO
                    ENDDO
                ENDDO
      C
      C         Form the lower triangular part of the Jacobian matrix
      C
                CALL JACLD(CAP_LD,CAP_HD)
      C
      C         Perform the lower triangular solution
      C
                CALL BLTS(ISIZ1,ISIZ2,ISIZ3,NX,NY,NZ,OMEGA,RSD,A,B,C,D,CAP_LD,CAP_HD)
      C
      C         Form the strictly upper triangular part of the jacobian matrix
      C
                CALL JACU(CAP_LD,CAP_HD)
```

Figure 4.6 : Section of code from routine SSOR in the APPLU code with synchronous communications.

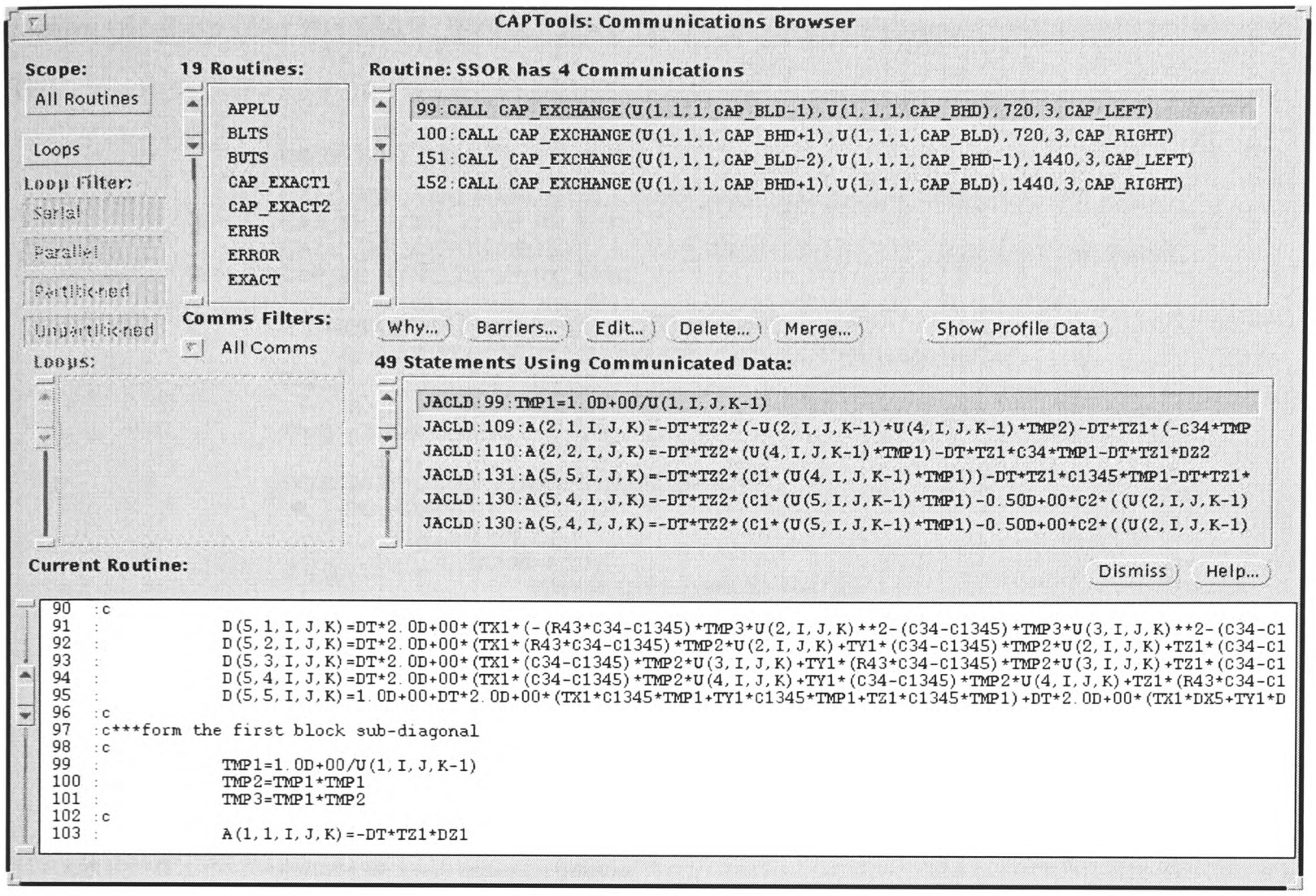


Figure 4.7 : Communication Browser showing where the communicated data is required.

In the code in Figure 4.6, the communication is some distance from the routine that requires the data having been migrated to its furthest legal most point from the usage of the data. It would be an advantage if the communication were overlapped with the nested loops prior to the call to routine JACLD. This would allow the communication time to be hidden and thus reduce the runtime of the code. A similar situation arises for the second CAP_EXCHANGE (S_2) communication with the communicated data required in this case in the routine JACU.

To apply overlapping communications to the code in Figure 4.6, the call name CAP_EXCHANGE is changed to be CAP_AEXCHANGE (Section 4.4) and the addition of two additional parameters to its parameter list. These two additional parameters are required as identifiers in the call to CAP_SYNC_EXCHANGE to ensure that the correct communication has completed its exchange of data. The CAP_SYNC_EXCHANGE itself may then be placed prior to the call that requires the communicated data (see Figure 4.8). For the communication S_1 the CAP_SYNC_EXCHANGE (S_3) is placed prior to the call to routine JACLD; for the second

communication (S_2) the CAP_SYNC_EXCHANGE is placed prior to the call to routine JACU (S_4).

```
                DO ISTEP=1,ITMAX,1
S1              CALL CAP_AEXCHANGE(U(1,1,1,CAP_BLD-1),U(1,1,1,CAP_BHD),1440,CAP_LEFT,
     &          CAP_SE_SYNC_1,CAP_RE_SYNC_1)
S2              CALL CAP_AEXCHANGE(U(1,1,1,CAP_BHD+1),U(1,1,1,CAP_BLD),1440,CAP_RIGHT,
     &          CAP_SE_SYNC_2,CAP_RE_SYNC_2)
C
                IF ((MOD(ISTEP,INORM).EQ.0).AND.(IPR.EQ.1)) THEN
                    IF (CAP_PROCNUM.EQ.1)WRITE(UNIT=IOUT,FMT=1001)ISTEP
                ENDIF
C
C               Perform SSOR iteration
C
                DO K=MAX(2,CAP_LD),MIN(NZ-1,CAP_HD),1
                    DO J=2,NY-1,1
                        DO I=2,NX-1,1
                            DO M=1,5,1
                                RSD(M,I,J,K)=DT*RSD(M,I,J,K)
                            ENDDO
                        ENDDO
                    ENDDO
                ENDDO
C
C               Form the lower triangular part of the jacobian matrix
C
S3              CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC_1,CAP_RE_SYNC_1)
                CALL JACLD(CAP_LD,CAP_HD)
C
C               Perform the lower triangular solution
C
                CALL BLTS(ISIZ1,ISIZ2,ISIZ3,NX,NY,NZ,OMEGA,RSD,A,B,C,D,CAP_LD,CAP_HD)
C
C               Form the strictly upper triangular part of the jacobian matrix
C
S4              CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC_2,CAP_RE_SYNC_2)
                CALL JACU(CAP_LD,CAP_HD)
```

Figure 4.8 : Section of code from SSOR with an overlapping communication.

In the code in Figure 4.8 the communications were overlapped with calculation prior to the routine calls. It might be the case that there is more code to overlap inside the routines themselves. The Communications Browser (Figure 4.7) shows that the first request for the data in calculation is required at line 99. It may be possible to place the synchronisation call CAP_SYNC_EXCHANGE immediately prior to this command. However, closer inspection of the code revealed that this command was within 3 loops. If the synchronisation points was placed inside these loops, it would be called for every iteration of these loops. This would cause the CAP_SYNC_EXCHANGE (which incurs a time overhead) to be called many more times than necessary. It is therefore preferable to place the synchronisation point prior to these loops to reduce the time overhead. If the synchronisation point was placed prior to these loops then for this example there would only be three assignment commands from the head of the routine

JACLD to the synchronisation point. The synchronisation point has now gained some additional code processing to overlap the communication. A similar situation occurs within the JACU call.

4.6 Partial Loop Overlapping using Loop Unrolling.

In the simple overlapping case it was possible to take advantage of sections of code that did not use the communicated data to overlap the communication and thus hide the communication time. In some cases, however, this will not be possible.

The use of data in a loop immediately after its communication prohibits simple overlapping of communications. Loop unrolling [83] may be applied to allow overlapping. If the data to be exchanged in a communication is to be used in calculation in the first iteration of the loop then that iteration of the loop may be stripped from the loop. The calculation loop would then execute from the second to the last iteration of the original loop. After the execution of this loop a synchronisation point is inserted and the original first iteration is then calculated. It is essential to ensure that the data assigned in the first iteration is not required in a consequent iteration of the loop. The left hand side (Figure 4.9a) shows the synchronous exchange as generated by CAPTools. This is a simplification of a similar code in the routine SOLVER in FAB (Figure 3.5). The right hand side (Figure 4.9b) shows the overlapping version of the code modified for partial loop overlapping with loop unrolling.

Figure 4.9 shows that the data being exchanged is being received into A(Cap_L-1). The calculation involves the statement B(J) = A(J-1) where the range of J will be 1 or Cap_L to 200 or Cap_H, depending on whether it is the first (1 to Cap_H), last (Cap_L to 200) or intermediate (Cap_L to Cap_H) processor of a 1-D grid of processors. The communicated value of A, i.e. A(Cap_L-1), is required on the first iteration on each processor, apart from the first processor of a 1-D grid of processors. The first iteration may then be stripped out of the J loop.

The code in Figure 3.5 shows that the two CAP_EXCHANGE communications are communicating data which is required immediately within the DO 30 loop. Applying the simple overlapping strategy (Section 4.5) the synchronisation points would be placed immediately prior to the DO 30 J loop. This would provide 2 lines of code to overlap the communication. This clearly is an insufficient amount of calculation to overlap the communication.

```
DO I=1,NITER
  CALL CAP_EXCHANGE(A(Cap_L-1),A(Cap_H),..)

  DO J=MAX(1,Cap_L), MIN(200,Cap_H)
    {* Calculation using communicated data *}
    {* on first iteration *}
    B(J) = A(J-1)
  ENDDO




ENDDO
```

Figure 4.9a: Synchronous

```
DO I=1,NITER
  CALL CAP_AEXCHANGE(A(Cap_L-1),A(Cap_H),..)

  DO J=MAX(1,Cap_L)+1, MIN(200,Cap_H)
    {* Calculation NOT using communicated data *}

    B(J) = A(J-1)
  ENDDO

  CALL CAP_SYNC_EXCHANGE(..)

  DO J=MAX(1,Cap_L),MAX(1,Cap_L)
    {* Calculation using communicated data *}
    B(J) = A(J-1)
  ENDDO

ENDDO
```

Figure 4.9b: Overlapped

Figure 4.9 : Synchronous and overlapping pseudo code illustrating partial loop overlapping using loop UNROLLing.

However, the two CAP_EXCHANGE communications exchange data which lies on the boundary of each processors' allocated data partition. Removing the very first and last iteration of the DO 30 loop and placing their calculation immediately after the reduced DO 30 loop would allow sufficient code to conceal the communication time.

To perform this method of concealing the communication requires loop unrolling or loop stripping. This is a common optimisation which may be found in many compilers [83]. It is, however, essential that before administering loop unrolling that the data from the first iteration of the DO 30 loop is not required in any subsequent iterations of the loop. For the final iteration of the loop this would also apply. For the DO 30 loop this is not a problem since no dependencies are carried by that loop (after the dependence deletion) and it is therefore possible to unroll the first and last iterations. This is illustrated in the code in Figure 4.10.

The original DO 30 loop (Figure 4.10) has now been adjusted (unrolled) such that the first and last iteration are no longer executed. The code now also contains two copies of the code within the DO 30 loop to allow the calculation of the stripped loops after synchronisation of the overlapped communication. When the modified loop has completed its execution, the overlapped communications are synchronised to ensure that they have completed. It is then legal to execute the calculation for the first and last iteration of the original loop.

```
40    CONTINUE
      RESID = 0.0
      IF (ISWEEP .LE. MSWEEP) THEN
            CALL CAP_AEXCHANGE(TNEW(2,CAP_BHTNEW+1),TNEW(2,CAP_BLTNEW),
+                 IN-2,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            CALL CAP_AEXCHANGE(TNEW(2,CAP_BLTNEW-1),TNEW(2,CAP_BHTNEW),
+                 IN-2,CAP_LEFT,CAP_SE_SYNC2,CAP_RE_SYNC2)
            TOP=0.0
            BOT=0.0
C
C           The first & last iterations have been stripped.
C
            DO 30 J=MAX(2,CAP_BLTNEW)+1,MIN(JN-1,CAP_BHTNEW)-1,1
                  .
                  DO 10  I=2,IN-1,1
                        .
                        B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
10                CONTINUE
                  CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
                  CALL RESIDUAL (...)
30          CONTINUE
C-----------------------------------------------------------------------------------------------------------------------
C           Calculation for the first iteration of the original J loop
C
            CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC_2,CAP_RE_SYNC_2)
            J=MAX(2,CAP_BLTNEW)
            .
            DO 110  I=2,IN-1,1
                  .
                  B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
110         CONTINUE
            CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
            CALL RESIDUAL(...)
C-----------------------------------------------------------------------------------------------------------------------
C           Calculation for the last iteration of the original J loop
C
            CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC_1,CAP_RE_SYNC_1)
            J=MIN(JN-1,CAP_BHTNEW)
            .
            DO 210  I=2,IN-1,1
                  .
                  B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
210         CONTINUE
            CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
            CALL RESIDUAL(...)
C-----------------------------------------------------------------------------------------------------------------------
            CALL CAP_COMMUTATIVE(CAP_TOP,CAP_RADD)
            TOP=TOP+CAP_TOP
            CALL CAP_COMMUTATIVE(CAP_BOT,CAP_RADD)
            BOT=BOT+CAP_BOT
            TOP=SQRT(TOP)
            BOT=SQRT(BOT)+1.0
            RESIDJ=TOP/BOT
            RESID=RESIDJ
            ISWEEP=ISWEEP+1
            IF (MOD(ISWEEP,10).EQ.0) PRINT *, 'RESIDUAL = ',RESID,ISWEEP
            IF(RESID .GT. CON1) THEN
                  GOTO 40
            ELSE
                  PRINT*,'ITERATIONS:',ISWEEP
                  RETURN
            ENDIF
      ENDIF
```

Figure 4.10 : The routine SOLVER in FAB with partial loop overlapping with loop UNROLLing applied.

4.7 Partial Loop Overlapping with a Conditional Statement.

The application of loop unrolling causes large amounts of additional code to be generated that breaks the second objective in Section 1.3. If the data being communicated is not required for calculation until the final few iterations of the loop then it could be possible to overlap the communication by using a conditional statement. The synchronisation point would be placed within the loop with a conditional statement that would be applicable only on the appropriate iteration of that loop. The pseudo code in Figure 4.11a shows the synchronous exchange as generated by CAPTools. Figure 4.11b shows the overlapping version of the code modified by hand for Partial Loop Overlapping using a conditional statement.

```
DO I=1,NITER
  CALL CAP_EXCHANGE(A(Cap_H+1),A(Cap_L),..)

  DO J=MAX(1,Cap_L), MIN(199,Cap_H)




    {* Calculation using communicated *}
    {* data only on last iteration *}
    B(J) = A(J+1)
  ENDDO
```

Figure 4.11a: Synchronous

```
DO I=1,NITER
  CALL CAP_AEXCHANGE(A(Cap_H+1),A(Cap_L),..)

  DO J=MAX(1,Cap_L), MIN(199,Cap_H)

    IF(J .EQ. Cap_H)THEN
      CALL CAP_SYNC_EXCHANGE(..)
    ENDIF

    {* Calculation using communicated *}
    {* data only on last iteration *}
    B(J) = A(J+1)
  ENDDO
```

Figure 4.11b: Overlapped

Figure 4.11 : Synchronous and overlapping pseudo code illustrating PARTIAL loop overlapping with a conditional statement.

Figure 4.11 shows that the data being exchanged is being received into A(Cap_H+1). The calculation involves the statement B(J)= A(J+1) where the range of J will be from 1 or Cap_L to 199 or Cap_H depending on whether it is the first (1 to Cap_H), last (Cap_L to 199) or intermediate (Cap_L to Cap_H) processor of a 1-D grid. As such, the communicated value of A, i.e. A(Cap_H+1), is not required until the last iteration on each processor, apart from the last processor in a 1-D grid of processors. The conditional statement required for the synchronisation point ensures that it is the last iteration for that processor before enforcing the synchronisation point. The code is also similar to the original with only minor changes.

Consider the code in Figure 4.12 from routine CALCU in TEAMKE1.

```
      CALL CAP_EXCHANGE(U(1,CAP_BHT+1),U(1,CAP_BLT),288,CAP_RIGHT)
C
C     Final Coefficient assembly and residual calculation.
C
      RESORU=0.0
      FAC=1.-URFU
      CAP_RESORU=0
      DO 420 I=2,NIM2,1
            DO 200 J=MAX(2,CAP_LVIS),MIN(NJM1,CAP_HVIS),1
                  APU(I,J)=AWU(I,J)+AEU(I,J)+ASU(I,J)+ANU(I,J)-SPU(I,J)
                  DU(I,J)=RSYCV(J)/APU(I,J)
S1                RESOR=ANU(I,J)*U(I,J+1)+ASU(I,J)*U(I,J-1)+AEU(I,J)*
     &                  U(I+1,J)+AWU(I,J)*U(I-1,J)-APU(I,J)*U(I,J)+SUU(I,J)
                  IF (-SPU(I,J).EQ.GREAT) THEN
                        RESOR=0.0
                  ENDIF
                  CAP_RESORU=CAP_RESORU+ABS(RESOR)
C
C                 Under-relaxation.
C
                  APU(I,J)=APU(I,J)/URFU
                  SUU(I,J)=SUU(I,J)+FAC*APU(I,J)*U(I,J)
                  DU(I,J)=DU(I,J)*URFU
C
C                 Solution of difference equations.
C
200         CONTINUE
420   CONTINUE
```

Figure 4.12 : Subroutine CALCU in TEAMKE1 with synchronous communications.

The exchange communication receives the data for use in statement S_1. There is clearly no other code that may be simply overlapped with the communication. The communication receives all the values of U for the J index CAP_BHT+1 from the next processor in a 1-D grid of processors. This communicated data in the exchange is not required until the final iteration of the DO 200 J loop.

The main philosophy of CAPTools is to generate parallel code that is still recognisable by the original code author with the minimum amount of changes or additional code. Placing a conditional within the loop that calls the synchronisation point on the last iteration loop prior to the use of the communicated data (Figure 4.13) reduces the amount of alteration to the code significantly in comparison with that obtained from loop unrolling. Loop unrolling would have necessitated a copy of the calculation within the DO 200 J loop being copied and placed immediately after the loop end.

It follows that the two loops unrolled for FAB in Figure 4.10 were not necessary. The loop unrolling of the first iteration of the loop was required along with a conditional for the partial loop overlapping of the data required for the last iteration of the loop.

```
      CALL CAP_AEXCHANGE(U(1,CAP_BHT+1),U(1,CAP_BLT),288,CAP_RIGHT,
     &                   CAP_SE_SYNC_1,CAP_RE_SYNC_1)
C
C     Final coefficient assembly and residual calculation.
C
      RESORU=0.0
      FAC=1.-URFU
      CAP_RESORU=0
      DO 420  I=2,NIM2,1
         DO 200  J=MAX(2,CAP_LVIS),MIN(NJM1,CAP_HVIS),1
            IF (J.GE.CAP_BHT) THEN
               CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC_1,CAP_RE_SYNC_1)
            ENDIF
            APU(I,J)=AWU(I,J)+AEU(I,J)+ASU(I,J)+ANU(I,J)-SPU(I,J)
            DU(I,J)=RSYCV(J)/APU(I,J)
S1          RESOR=ANU(I,J)*U(I,J+1)+ASU(I,J)*U(I,J-1)+AEU(I,J)*U(I+1,J)+AWU(I,J)*
     &              U(I-1,J)-APU(I,J)*U(I,J)+SUU(I,J)
            IF (-SPU(I,J).EQ.GREAT) THEN
               RESOR=0.0
            ENDIF
            CAP_RESORU=CAP_RESORU+ABS(RESOR)
C
C           Under-relaxation.
C
            APU(I,J)=APU(I,J)/URFU
            SUU(I,J)=SUU(I,J)+FAC*APU(I,J)*U(I,J)
            DU(I,J)=DU(I,J)*URFU
C
C           Solution of difference equations.
C
200      CONTINUE
420   CONTINUE
```

Figure 4.13 : Subroutine CALCU in TEAMKE1 with overlapped communication.

It is often the case that exchange communications emerge as pairs, requiring the boundary data of both its neighbouring processors. This is primarily due to the type of calculation required in Computational Mechanics codes. A certain point on a mesh may have, for instance, temperature calculated from the temperature of the points surrounding it. For example, The temperature value of T(I,J) could be dependant on the values of T(I+1,J), T(I-1,J), T(I,J+1) and T(I,J-1). It will therefore be the case that when two exchanges of boundary data are placed prior to the loop requiring the communicated data allowing no overlap will require both the partial loop overlap using loop unrolling and a conditional statement as is the case in Figure 4.14.

Figure 4.15 shows the new calculation order of the loop in Figure 4.14 after applying loop unrolling and partial overlapping. Originally, for processor 2, the order of the calculation would have been from J=10 to J=20. Applying the loop unroll strips the first loop, therefore the calculation now begins at J=11 up until J=20. The J=10 iteration is calculated after J=20. The J=10 calculation is only calculated after synchronisation point Sync2. Also the calculation of J=20 and J=10 are only possible after the synchronisation at the point Sync1 in Figure 4.15.

```
      40  CONTINUE
          IF (ISWEEP .LE. MSWEEP) THEN
              CALL CAP_AEXCHANGE(TNEW(2,CAP_BHTNEW+1),TNEW(2,CAP_BLTNEW),
     +        IN-2,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
              CALL CAP_AEXCHANGE(TNEW(2,CAP_BLTNEW-1),TNEW(2,CAP_BHTNEW),
     +        IN-2,CAP_LEFT,CAP_SE_SYNC2,CAP_RE_SYNC2)
              TOP=0.0
              BOT=0.0
              DO 30 J=MAX(2,CAP_BLTNEW)+1,MIN(JN-1,CAP_BHTNEW),1
                  IF (J.GE.CAP_BHTNEW) THEN
Sync1                 CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC_1,CAP_RE_SYNC_1)
                  ENDIF
                  .
                  DO 10  I=2,IN-1,1
                      .
                      B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
      10          CONTINUE
                  CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
                  CALL RESIDUAL (...)
      30      CONTINUE
C
C             Calculation for the first iteration of the original J loop
C
Sync2         CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC_2,CAP_RE_SYNC_2)
              J=MAX(2,CAP_BLTNEW)
              .
              DO 110  I=2,IN-1,1
                  .
                  B(I)=-(B(I)+(TNEW(I,J+1)*SK(I,J+1)+TNEW(I,J-1)*SK(I,J))/DR)
     110      CONTINUE
              CALL TDMA(TNEW,IN,IN-1,J,CAP_LTNEW,CAP_HTNEW)
              CALL RESIDUAL(...)
```

Figure 4.14 : Routine SOLVER in FAB with partial overlapping using loop unrolling and a conditional statement.

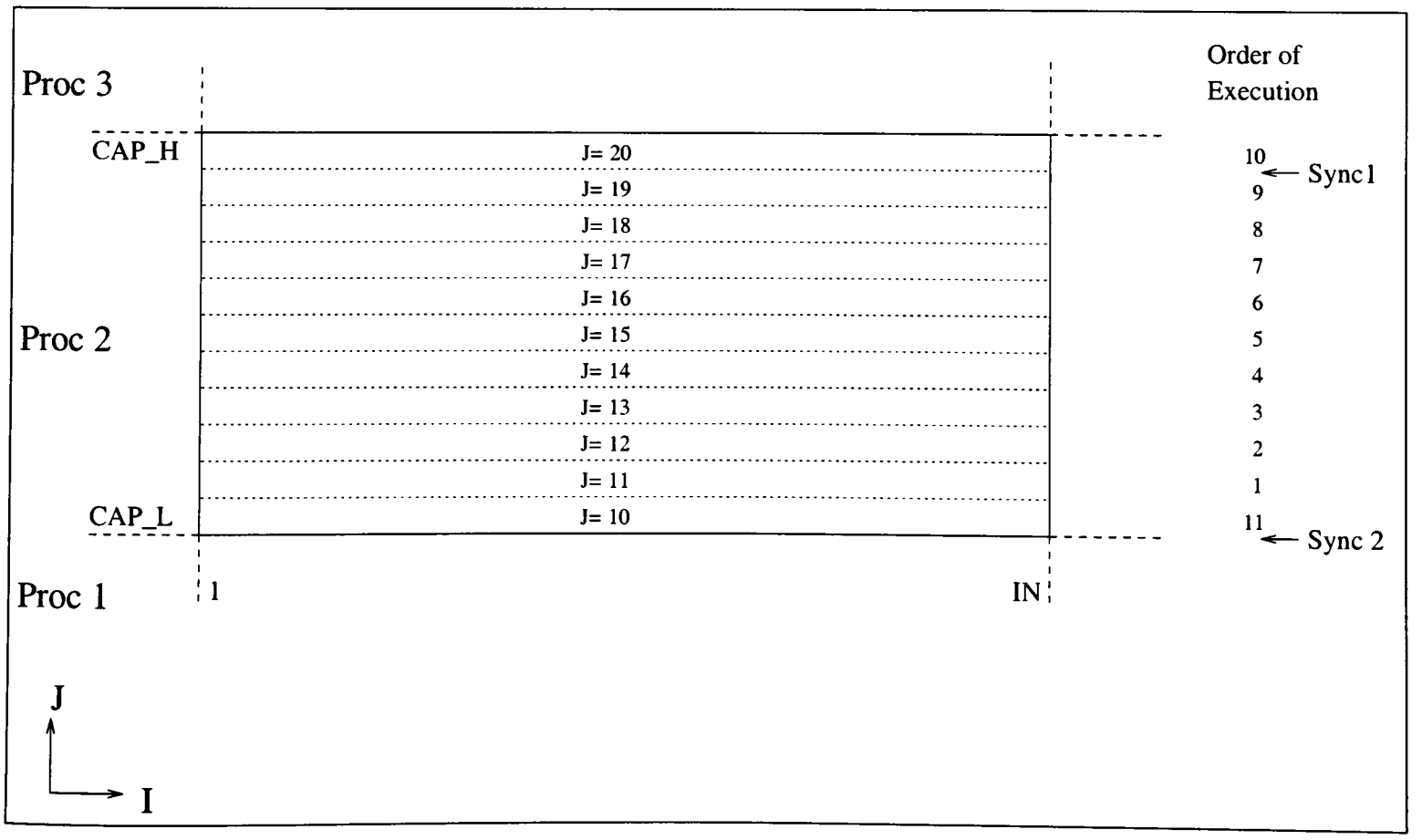


Figure 4.15 : Calculation order of code in Figure 4.14.

4.8 Pipelines.

From the codes that were parallelised in Chapter 3 it is clear that pipelines were necessary to ensure correct parallel code was generated. Experimentation with placing the communications surrounding different loops had provided some improvement in efficiencies, however, this still represents a significant overhead.

To improve these pipeline efficiencies it is necessary to hide the communication time. Unfortunately, none of the previous examples of overlapping communications are applicable to pipelines. Instead a different method has to be formulated.

The code for the pipeline communicating a line of data (Figure 3.19) for the BLTS routine in APPLU is represented diagrammatically in Figure 4.16. The first processor starts the pipeline by executing the calculation for J=2. The CAP_RECEIVE communication is not executed on the first processor since it has no processor to its left. When the calculation has completed a set of lines of data the last line calculated is sent to the next processor to the right. The first processor then repeats this process for the next J iterations until there is no more calculation in the pipeline. The next processor in the 1-D grid will start its process by receiving a line of data from the left. This processor then executes its calculation. When this calculation is completed the processor then sends data to the next processor to its right. All the intermediate processors of the 1-D grid will repeat this process for all J iterations until there is no more calculation to allow communication. The last processor in the 1-D grid will receive the communicated data from the previous processor and then execute its calculation. This final processor will not communicate to the right since this is the last in the 1-D grid.

The diagram (Figure 4.16) also shows the idle time the processors incur for the start-up and shutdown of the pipeline. There is also some idle time present for the first and last processor of the 1-D grid. It can clearly be seen that the communication extends the total runtime of the pipeline.

To overlap these communications the pipeline algorithm has be altered to take advantage of the calculation that is present. In Figure 4.16 the calculation and communication on each processor is linear, i.e. the send communication does not happen until the calculation has completed and the calculation does not happen until the receive communication has completed, etc. From the previous discussion in Section 4.2 these send and receive communications can occur in tandem (i.e. overlapped) with the calculation. Rearranging these communications in Figure 4.16 to overlap with the calculation provides us with the diagram in

Figure 4.17, which shows how all the communications, apart from the initial and final communications on each processor, have been overlapped with the calculation. With the aid of this diagram it was possible to transform the pipeline code from BLTS in APPLU (Figure 3.19) to operate as an overlapped pipeline code as in Figure 4.18.

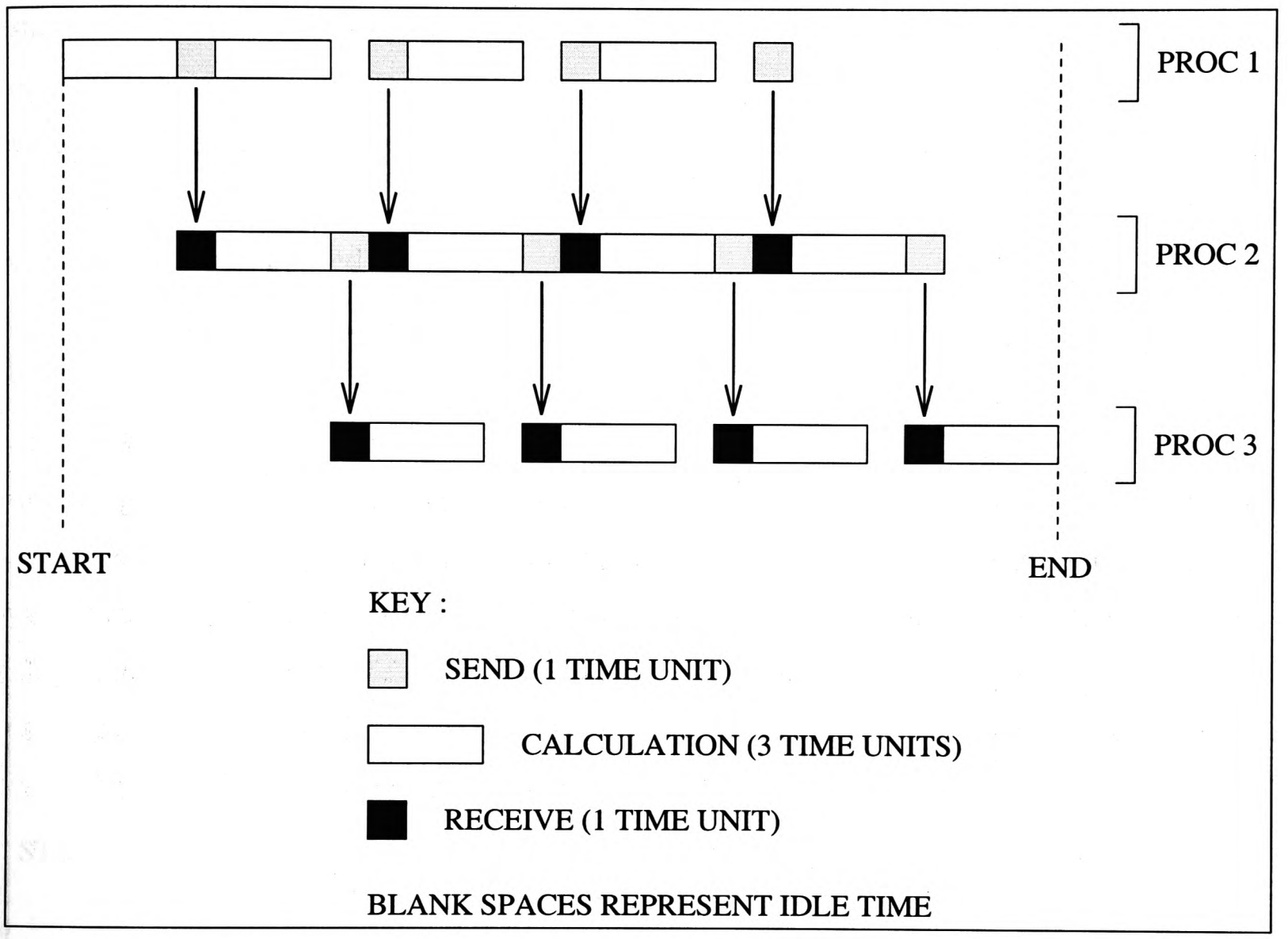


Figure 4.16 : Synchronous pipeline.

The pipeline has been changed such that all the synchronous communications are now overlapping communications. It has also been necessary to add some additional communications and synchronisation points. The most important factor of this overlapped pipeline is that the processors will start receiving the data for the next cycle of the pipeline whilst calculating for the present cycle of the pipeline.

The first processor of the pipeline will behave much like the synchronous pipeline. The first processor will execute its calculation before synchronising its send communication (to ensure that any previous overlap send has completed) before communicating the data using an overlapping send call. The intermediate processors of the pipeline will initialise their communications to receive the data at the same time as the previous processor is executing its

calculation for the previous J iteration. The processor will then receive the data before synchronising to ensure that the first communication has been completed. The intermediate processor will not immediately use the communicated data for calculation. It will first of all initialise a communication to receive the data for the next iteration. This communication statement will have a conditional statement attached to it to ensure that there is another iteration of the loop to receive data. Once this communication has been initialised the calculation on the intermediate processor may continue before passing the data to the next processor. The final processor will act in a similar fashion to the intermediate processor except it does not communicate its calculated data.

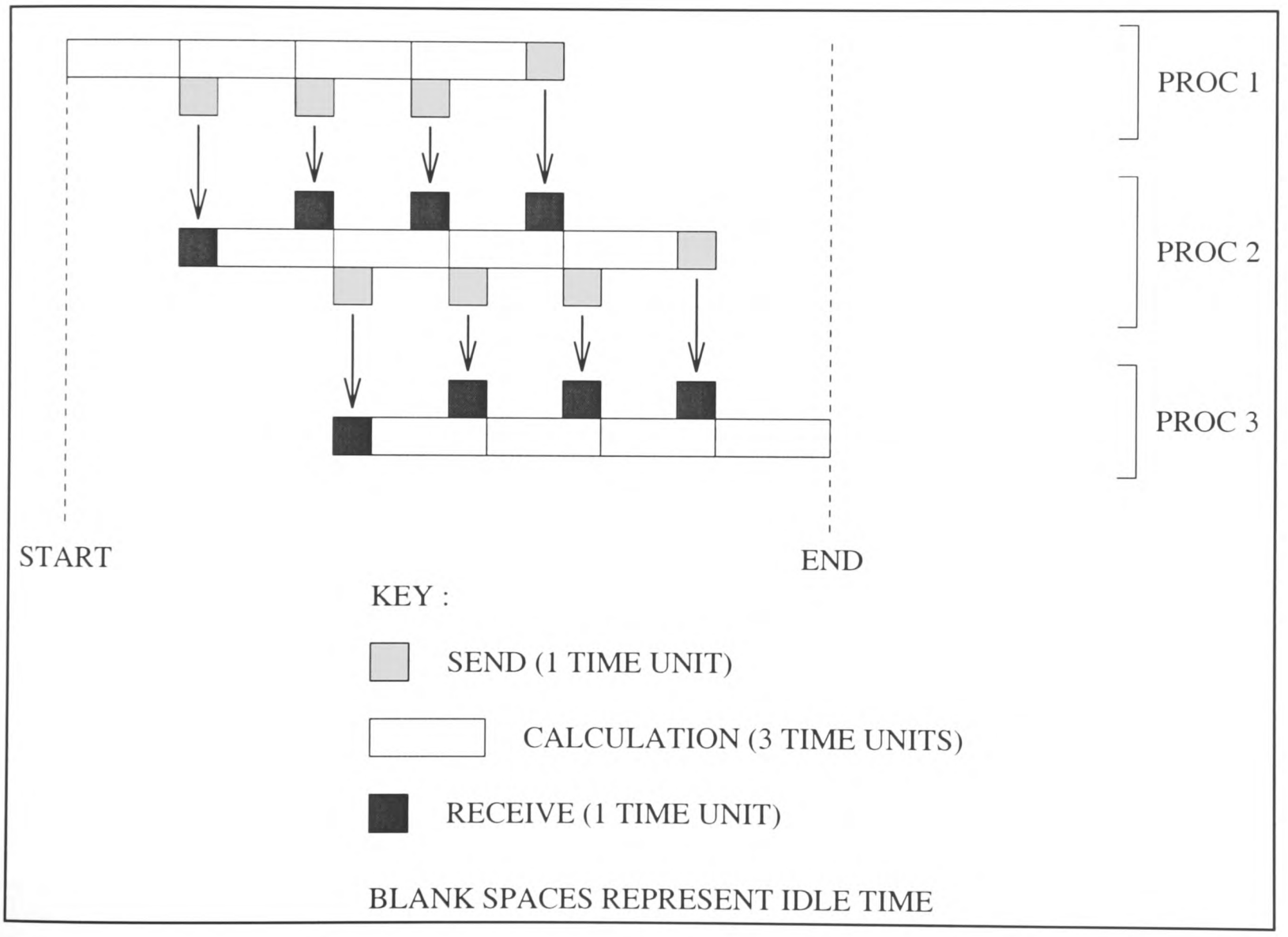


Figure 4.17 : Overlapping pipeline.

Comparing the diagrams of both the synchronous and overlapping pipelines (Figure 4.16 and Figure 4.17) also shows how the communications overlapped with the calculation and reduces the overall runtime of the pipeline. The idle time that was present on the first and last processors has also been eliminated.

```
CALL CAP_ARECEIVE(V(1,2,2,CAP_BLA-1),NX*10-20,CAP_LEFT,CAP_R_SYNC)
DO J=2,NY-1,1
      CALL CAP_SYNC_RECEIVE(CAP_LEFT,CAP_R_SYNC)
      IF (J+1.LE.NY-1) THEN
            CALL CAP_ARECEIVE(V(1,2,J+1,CAP_BLA-1),NX*10-20,CAP_LEFT,CAP_R_SYNC)
      ENDIF
      DO K=MAX(2,CAP_LLDZ),MIN(NZ-1,CAP_HLDZ),1
            DO I=2,NX-1,1
                  .
                  Calculation requiring communicated data.
                  .
            ENDDO
      ENDDO
      CALL CAP_SYNC_SEND(CAP_RIGHT,CAP_S_SYNC)
      CALL CAP_ASEND(V(1,2,J,CAP_BHA),NX*10-20,CAP_RIGHT,CAP_S_SYNC)
ENDDO
CALL CAP_SYNC_SEND(CAP_RIGHT,CAP_S_SYNC)
```

Figure 4.18 : Overlapping pipeline communicating line data for routine BLTS in APPLU.

These two pipelines were tested to establish how communicating varying amounts of data affected the obtainable efficiency. The pipelines were tested on the APPLU code for 6 i860 processors on the Transtech Paramid, the results of which are shown in Table 4.1.

	No Overlap in Comms & Calc			Overlapping Comms & Calc		
Nwords	Time	Speed Up	Efficiency	Time	Speed Up	Efficiency
120	41.5	3.49	58.1	29.7	4.87	81.1
360	47.8	3.02	50.4	30.4	4.76	79.4
600	54.2	2.67	44.5	31.0	4.66	77.6
840	60.5	2.39	39.8	31.7	4.56	76.0
960	63.7	2.27	37.9	32.2	4.49	74.8
1200	70.0	2.07	34.4	35.7	4.05	67.5
1440	76.4	1.89	31.6	39.2	3.69	61.5

Table 4.1 : Results showing how the efficiency vary with the amount of data communicated.

The results obtained (Table 4.1) showed that for the synchronous pipeline the efficiencies were falling dramatically as the amount of data being communicated increased. For the overlapped pipeline the efficiencies remained fairly constant as the amount of communicated data increased until the turning point at approximately 940 to 960 words of communicated data. There is some loss of efficiency up to this turning point due to contention between the two

processors (the i860 and the transputer) for access to the same data in memory. At the turning point, the efficiencies begin to fall rapidly for the overlapped pipeline. The graph, in Figure 4.19, clearly shows how the efficiency begins to decrease at this point. Up until the turning point the amount of time to communicate the data was always less than the time taken to execute the calculation. The communication time was effectively completely hidden by the calculation time. After the turning point the time taken to communicate the data was now greater than the communication time of the pipeline and the efficiencies were falling at the same rate as they would for a synchronous pipeline. The graph does, however, show that improved efficiencies are still obtainable for the overlapped pipeline in comparison with the synchronous pipeline even when the amount of communication is greater than the amount of communication to overlap.

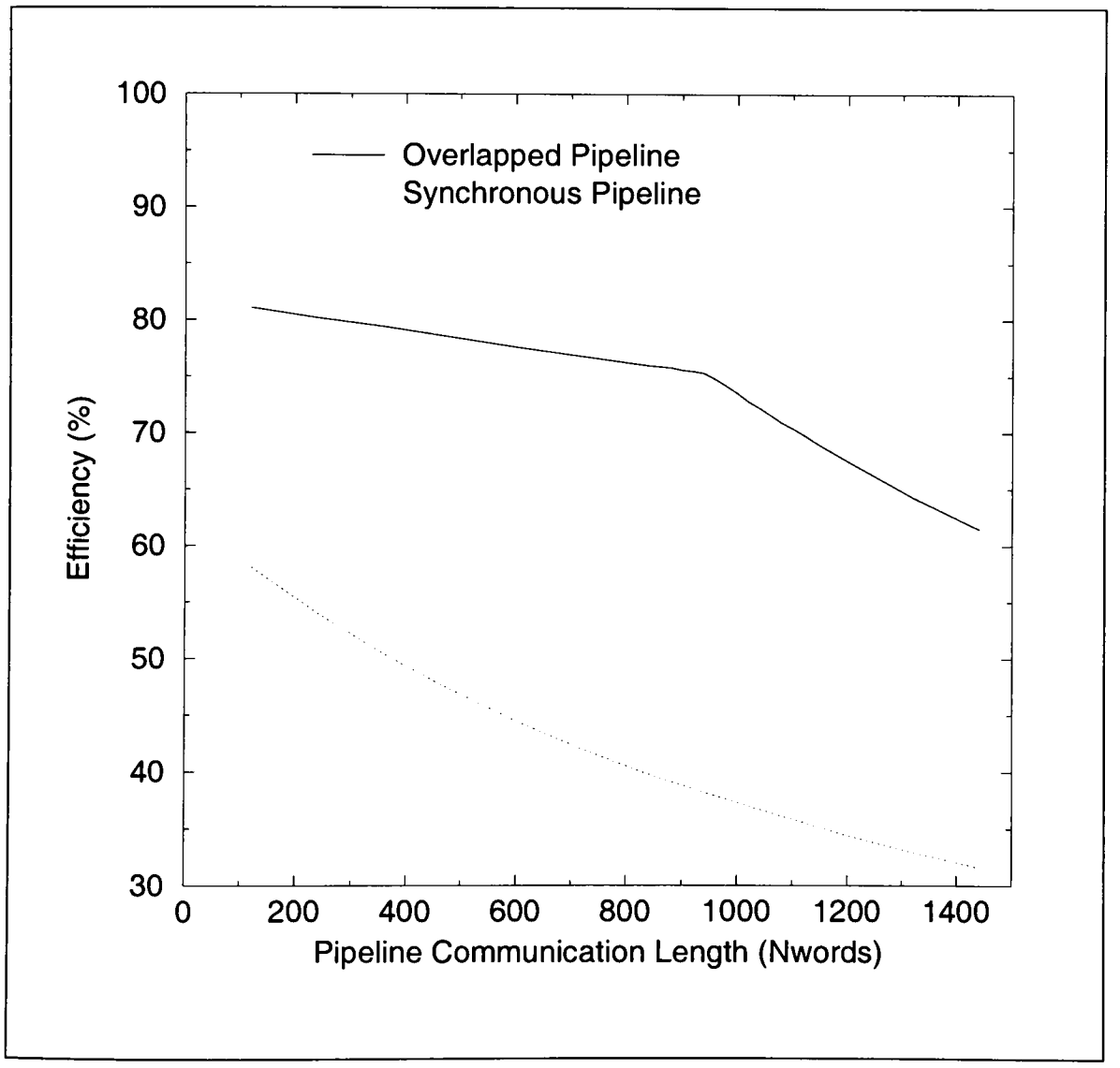


Figure 4.19 : Graph of synchronous versus overlapped communications.

The application of the overlapped communications to the two pipeline routines in APPLU provided a satisfactory increase in the efficiency of the code (see Section 6.4).

4.9 Conclusion.

The methods applied by hand to convert the synchronous communications to be overlapped proved effective in increasing the efficiencies of the parallel codes. This success provided the motivation for these methods to be incorporated and automatically generated within CAPTools. The application of these methods of overlapping the communications is discussed in the following chapter.

Chapter 5

5 The Automatic Code Generation of Overlapping Communications for Structured Mesh Computational Mechanics Codes.

5.1 Introduction.

The previous chapter demonstrated via manual parallelisation the methods that can be used to apply overlap communications. This process is now automated such that CAPTools will accomplish it automatically as an additional stage of the parallelisation. This requires a generic algorithm that makes use of many of the symbolic algebra tests and symbolic variable manipulation routines within CAPTools This additional stage is placed immediately after the Calculation and Generation of Synchronous Communications stage and prior to the Parallel Code Generation. This placement allows the exploitation of the synchronous communication information provided by CAPTools to assist in the generation of the overlapping communications.

The basic algorithm (Figure 5.1) shows how each command of the parallel code is traversed. Each command is inspected to determine if it is an exchange communication or a pipeline communication. Depending on the communication type the relevant tests are executed to discover their suitability for conversion. Synchronisation points and transformation requests are merged prior to final generation.

The methods discussed in this chapter are also summarised in the paper referenced in [71].

```
FOR (every COMMAND in the synchronous parallel code) DO
    BEGIN
    IF COMMAND = EXCHANGE COMMUNICATION THEN                          Section 5.2
        BEGIN
        {* The first stage of calculating CAP_EXCHANGE overlap. *}
        IF (test for SIMPLE is true) THEN                             Section 5.2.2
            {* SIMPLE overlap may be applied. *}
        ELSE IF (test for PARTIAL is true) THEN                       Section 5.2.3
            {* PARTIAL overlap may be applied. *}
        ELSE IF (test for UNROLL is true) THEN                        Section 5.2.4
            {* UNROLL overlap may be applied. *}
        ELSE
            {* Overlap may not be applied. *}
        END
    ELSE IF COMMAND = PIPELINE COMMUNICATION THEN                     Section 5.3
        BEGIN
        {* Generate Overlapped Pipeline. *}                          Section 5.3
        END
    END
{* Merge synchronisation points. *}                                  Section 5.2.5
{* The second stage of generating CAP_EXCHANGE overlap :    *}
{*     Generate UNROLL communications                       *}       Section 5.2.7.1
{*     Generate PARTIAL communications                      *}       Section 5.2.7.2
{*     Generate SIMPLE communications                       *}       Section 5.2.7.3
```

Figure 5.1 : The basic algorithm for the automatic generation of overlapping communications.

5.2 Exchange Communications.

The testing of the overlapped CAP_EXCHANGE communications consists of two stages. The first stage consists of calculating which CAP_EXCHANGE communications may be overlapped and which method is the most suitable to be applied. This stage tests which overlapping communications may be applied in the strict order of SIMPLE (Section 5.2.2), PARTIAL (Section 5.2.3) and UNROLL (Section 5.2.4). Each of these methods requires more additional or altered code than the previous method. It is obviously desirable to apply the simplest method when possible to maintain recognition of the original serial code. The second stage consists of converting these suitable communications to be overlapped using the appropriate method. This stage requires the generation of the overlapped communications methods in a strict order of UNROLL, PARTIAL and then SIMPLE. This method of ordering prevents any intervention from the previous methods applied. For example, the UNROLL method is the most disruptive in that it requires the unrolling of a loop. This involves the copying of the original loop and the adjustment of the both the original and copied loops. If the other two methods of overlapping had been applied prior to the loop copy then the situation may arise that any code alterations or additions from these methods will also be copied causing the algorithm to be incorrect or an unnecessary synchronisation call.

5.2.1 Selection of Overlapping Technique.

This primary stage consists of calculating which method of overlapping may be most effectively applied to the communications. The three methods of calculating the SIMPLE, PARTIAL and UNROLL are applied in a strict order. The SIMPLE (Section 5.2.2) overlapping is tested first since this is as the name suggests, the simplest method to apply and requires the minimum amount of change to the parallel code. The PARTIAL and UNROLL methods are tested in tandem. This is because they both involve similar tests. It is however preferable to use the PARTIAL method where possible since this causes the minimal code alteration (Section 5.2.3). The UNROLL is only applied when it is not possible to apply the PARTIAL method (Section 5.2.4)

5.2.2 Calculation of the Legality and Profitability of Simple Overlapping Communications.

The first test applied to ascertain if the CAP_EXCHANGE communication may be overlapped with calculation is the SIMPLE method. This is the simplest and most obvious method of overlapping the communication, if the circumstances allow. It will involve the least alteration and addition to the synchronous parallel code and can often enable the entire communication time to be overlapped. This can be achieved as long as there is sufficient code between the communication and the use of the communicated data.

As discussed in Section 4.5 this method takes advantage of executable code that exists between the communication and the command that uses the data. Since there is a slight additional overhead associated with overlapped communications it is essential to ensure that there is potentially enough calculation code between these two points. The executable code between these two points must not make use of this communicated data. CAPTools stores for each communication a list of commands that require the communicated data (Section 2.8.3). It is therefore possible to use the information from CAPTools and to manipulate it in the calculation of overlapping communications. This provides the position of the commands that require the communicated data in relation to its communication.

The position of the CAP_EXCHANGE is referred to as the source command (or source), while the command using the communicated data is referred to as the sink command (or sink). Each source command may have several sink commands. This is due to the migration of each

individual communication request from each sink command to the same position and the subsequent merger into one single communication (Section 2.8.3).

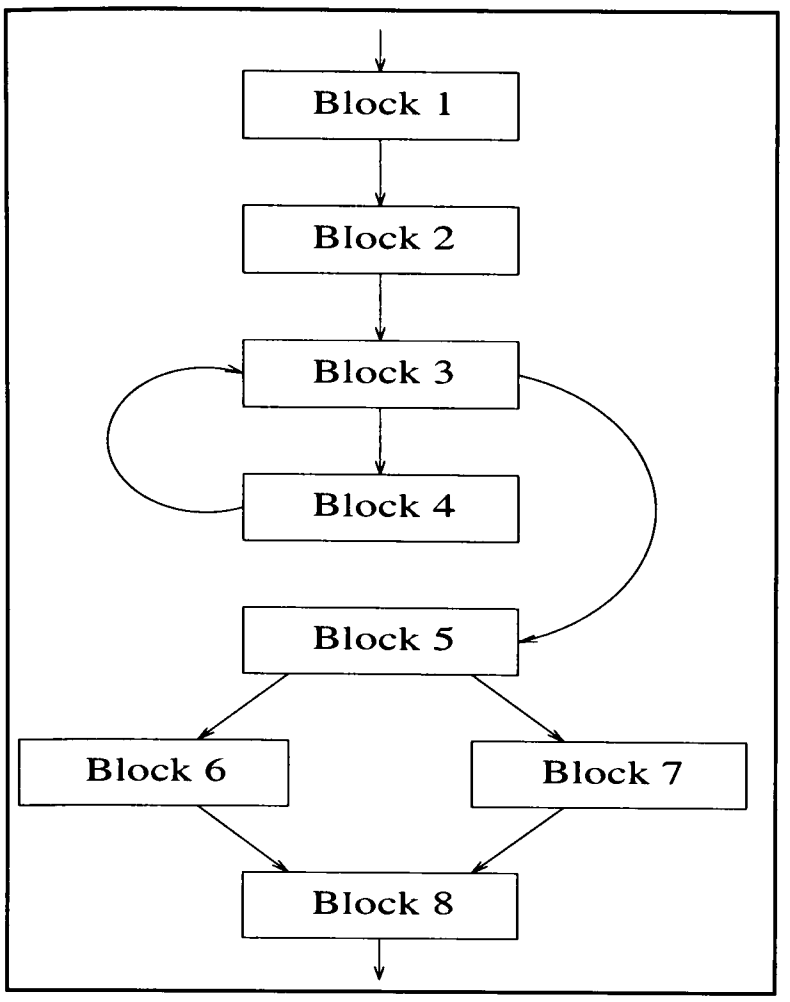
5.2.2.1 Sink Command with No 'Local' Surrounding Loops.

For this case the sink command has no 'local' surrounding loops. 'Local' loops in this context represent loops that surround only the sink command. It may be the case that both the sink and source commands also have common surrounding loops but the sink command itself does not have any 'local' loops.

The code in Figure 5.2 consists of an exchange communication which receives the value of A(CAP_H+1) which is required on this processor for calculation at the sink command. The sink command does not have any 'local' surrounding loops, but the pseudo code may or may not be surrounded by additional loops common to both the source and sink commands. Its associated control flow graph and pre-dominator tree are shown in Figure 5.3 and Figure 5.4 respectively.

```
CALL CAP_EXCHANGE(A(Cap_H+1)...)          } Block1 (SOURCE COMMAND Block)
CALL ROUTINE1(...)                        } Block2
IN1 = IN–1                                } Block2
DO I = 1, IN1                             } Block3
      B(..) = C(..)                       } Block4
ENDDO
CALL INTRINSIC FUNCTION(...)              } Block5
IF (INCREASE) THEN                        } Block5
      ISET = ISET + 1                     } Block6
ELSE
      ISET = ISET – 1                     } Block7
ENDIF
... = ...                                 } Block8
... = A(Cap_H+1)                          } Block8 (SINK COMMAND Block)
```

Figure 5.2 : Pseudo code of a sink command with no surrounding loop.



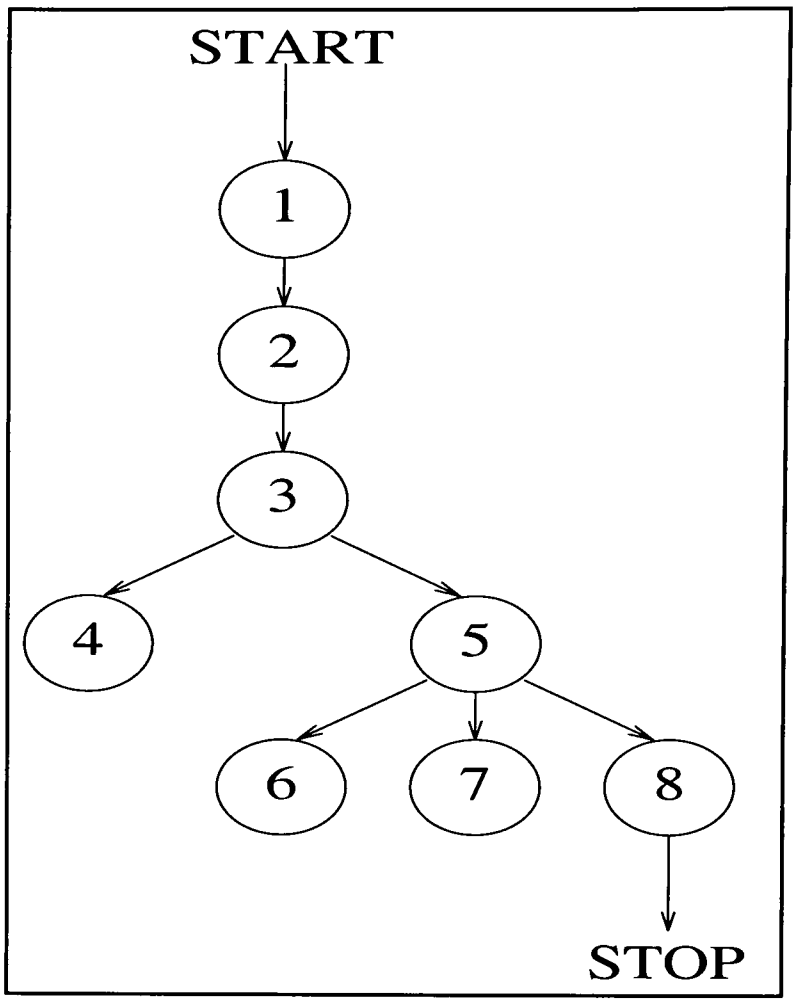


Figure 5.3: Control flow graph for Figure 5.2. Figure 5.4: Predominator tree for Figure 5.2.

The synchronisation point could be placed prior to this sink command. However, to ensure that the application of overlapping communication is advantageous a further test is applied to calculate if there is sufficient code between the source and sink command. This is accomplished by examining each command on the path traversed between the sink and source commands using an heuristic to determine whether any commands represent a significant computation overhead referred to in this work as a time consumer. Time consumers are defined as commands in the control flow graph (Figure 5.3) that are loops or calls to other routines. In the case of calls to routines, intrinsic routines are not considered, as they are relatively quick routines that would not provide sufficient code to overlap. Any loops considered as time consumers must not contain the sink command or source command. Examples of time consumers in the pseudo code in Figure 5.2 are :

CALL ROUTINE1	Block2
DO I=1,IN1	Block3

The algorithm to calculate a time consumer command shown in the function in Figure 5.5 simply involves the checking of a command, that is passed in, to see if it is a loop or a call to

a non-intrinsic routine. The function returns true or false depending on if a time consumer has been found.

```
FUNCTION IS_THERE_A_TIMECONSUMER(COMMAND)
{* Is this COMMAND a TIMECONSUMER, i.e. is *}
{* this COMMAND a non-intrinsic call or a loop.    *}
BEGIN
TIMECONSUMER = FALSE
IF (COMMAND = CALL to non-intrinsic routine) THEN
      {* The COMMAND is a CALL to a non-intrinsic routine *}
      {* therefore a TIMECONSUMER exists. *}
      TIMECONSUMER=TRUE
ELSE IF (COMMAND = LOOP) THEN
      IF (SINKCOMMAND not used in loop) THEN
            {* The COMMAND is a loop that does not contain the        *}
            {* SINKCOMMAND therefore a TIMECONSUMER exists. *}
            TIMECONSUMER = TRUE
      ENDIF
ENDIF
IS_THERE_A_TIMECONSUMER = TIMECONSUMER
END
```

Figure 5.5 : Pseudo code for detecting a time consumer command.

To find if a time consumer command exists in between the sink command and the source command requires the searching of each block and its commands between these two points. This may be performed by a depth first search (Section 2.3.2) up the control flow graph Figure 5.3), from the sink command block traversing through its predominating blocks (using the pre-dominator tree in (Figure 5.4) until any time consumers are found or the source command block is reached. This searches up the sub-graph between a block and its immediate predominating block. By its definition the predominating block is reached on all paths.

This in essence involves the depth first search of the control flow graph from a block to its immediate predominating block (the algorithm is shown in Figure 5.6). The predominator tree (Figure 5.4) is then traversed for each block from the sink command to the source command (the algorithm is shown in Figure 5.7).

The algorithm shown in Figure 5.6 passes into the function a block (BLOCK) and its predominating block (BLOCKPREDOM). A depth first search up the control flow graph is executed by using the HASFATHER (Section 2.3.2) of the BLOCK. This ensures that every possible route (i.e. every block) from the BLOCK to its BLOCKPREDOM is checked for a time consumer command. Each command in all the blocks is checked for time consumers using the algorithm in Figure 5.5.

```
FUNCTION IS_THERE_TIMECONSUMERS_ON_ALL_ROUTES(BLOCK,BLOCKPREDOM)
{* Check all routes from the current block of commands (BLOCK) to its      *}
{* predominating block of commands (BLOCKPREDOM) to see if there are  *}
{* any TIMECONSUMERS. All routes must have a TIMECONSUMER.     *}
BEGIN
SEARCHED := FALSE
TERMINATE := FALSE
{* Set up the list of the blocks that the control flow came from. *}
CBLOCKLIST:= BLOCK^.HASFATHER
TIMECONSUMER:=CBLOCKLIST <> NIL
{* Check all paths between the BLOCK and BLOCKPREDOM. *}
WHILE (CBLOCKLIST<>NIL) and (TIMECONSUMER) and (NOT TERMINATE) DO
      IF (CBLOCKLIST^.BLOCK = SOURCECOMMAND_BLOCK) THEN
            {* This BLOCK owns the SOURCECOMMAND. *}
            {* Terminate the search for TIMECONSUMERS.  *}
            {* All routes tested but no TIMECONSUMERS found. *}
            TIMECONSUMER = FALSE
            TERMINATE = TRUE
      ELSE
            IF (CBLOCKLIST^.BLOCK <> BLOCKPREDOM) THEN
                  {* Do NOT search the end pre-dominating block.*}
                  SEARCHED := TRUE
                  TIMECONSUMER = FALSE
                  FOR (each COMMAND in BLOCK) DO
                        {* Search for TIMECONSUMER. *}
                        TIMECONSUMER = IS_THERE_A_TIMECONSUMER(COMMAND)
                  ENDFOR
                  IF (TIMECONSUMER is not found) THEN
                        {*Recursively call this function  to search  up to BLOCKPREDOM. *}
                        TIMECONSUMER = IS_THERE_TIMECONSUMERS_ON_ALL_ROUTES
                                          (CBLOCKLIST^.BLOCK, BLOCKPREDOM)
                  ENDIF
            ENDIF
      ENDIF
      CBLOCKLIST:=CBLOCKLIST^.NEXT
ENDWHILE
{* Set the IS_THERE_TIMECONSUMER_ON_ALL_ROUTES to TRUE if a block  *}
{* of commands has been searched  and a TIMECONSUMER has been found.          *}
IS_THERE_TIMECONSUMER_ON_ALL_ROUTES=TIMECONSUMER AND SEARCHED
END
```

Figure 5.6 : Pseudo code for detecting time consumers between a block an its predominating block.

If we consider the code in Figure 5.2 then the first block to be checked would be the one containing the sink command (which is Block 8) and its predominating block in the predominator tree (Figure 5.4) would be Block 5. From the control flow graph (Figure 5.3) the Block 8 possesses two HASFATHER blocks – Block 6 and Block 7. Both routes from Block 8 to Block 5 via Block 6 or Block 7 are checked for time consumer commands.

The pseudo code in Figure 5.7 shows the algorithm to traverse through all the blocks between the sink command and the source command. This is achieved by traversing the predominator tree (Figure 5.3) for each of the blocks from the sink command block to the source command block. Each block and its predominating block and any blocks in between are checked for time consumers using the algorithm in Figure 5.6. For the example in Figure

5.2 the starting block will be Block 8; traversing through Blocks 5, 3 and 2 to the source command block (Block 1).

```
FUNCTION TIMECONSUMERS_IN_ROUTINE(SINKCOMMAND,SOURCECOMMAND)
{* Searches for a TIMECONSUMER between the SINKCOMMAND *}
{* and the SOURCECOMAND within a ROUTINE. *}
BEGIN
TIMECONSUMER = FALSE
BLOCK = SINKCOMMAND^.BLOCK
ENDBLOCK = SOURCECOMMAND^.BLOCK

{* Check for  time consumers in commands prior to the sink command in the block that owns the sink command. *}
CCOMMAND = BLOCK^.COMMAND
WHILE (CCOMMAND is not the SINKCOMMAND) DO
      TIMECONSUMER = IS_THERE_A_TIMECONSUMER(CCOMMAND)
      CCOMMAND = CCOMMAND^.NEXT
ENDWHILE

{* Check all routes from the block that owns the sink command. *}
WHILE (BLOCK^.PREDOM <> NIL)  AND  (BLOCK^.PREDOM <> ENDBLOCK) AND
        (TIMECONSUMER not found) DO
      {* Search all routes between a block and its predominating block until a time consumers is found. *}
      TIMECONSUMER = IS_THERE_TIMECONSUMERS_ON_ALL_ROUTES(BLOCK,BLOCK^.PREDOM)
      IF (TIMECONSUMER NOT FOUND) THEN
            TIMECONSUMER = FALSE
            FOR (each COMMAND in BLOCK) DO
                  {* Search for  TIMECONSUMER *}
                  TIMECONSUMER = IS_THERE_A_TIMECONSUMER(COMMAND)
            ENDFOR
      BLOCK = BLOCK^.PREDOM
ENDWHILE

IF (TIMECONSUMER not found) THEN
      {* The block containing the SOURCECOMMAND (ENDBLOCK)  *}
      {* must be checked for  TIMECONSUMERS if none already found. *}
      {* Check the remaining commands after the SOURCECOMMAND. *}
      CCOMMAND = SOURCECOMMAND^.NEXT
      WHILE (CCOMMAND  <> NIL) DO
            TIMECONSUMER = IS_THERE_A_TIMECONSUMER(CCOMMAND)
            CCOMMAND = CCOMMAND^.NEXT
      ENDWHILE
ENDIF

TIMECONSUMERS_IN_ROUTINE = TIMECONSUMER
END
```

Figure 5.7 : Pseudo code for detecting a time consumer between a sink command and its source command.

5.2.2.2 Sink Command with Surrounding Loops (Not Common to Source Command).

In the code in Figure 5.2 the sink command did not have any surrounding loops. In almost all cases in real codes the sink command will have some surrounding loops. Consider the code in Figure 5.8.

```
DO I=1,IN
      CALL CAP_EXCHANGE(A(I,2,CAP_H+1),.....          SOURCE COMMAND
      {* Sufficient code to overlap. *}
      DO J=2,JN
            DO K=MAX(1,CAP_L),MIN(KN,CAP_H)
                  .
                  .
                  B(J,K)= A(I,J,K+1)                  SINK COMMAND
                  .
            ENDDO
      ENDDO
ENDDO
```

Figure 5.8 : Code fragment of a sink command with surrounding loops.

In the example in Figure 5.2 the synchronisation point was placed immediately prior to the sink command. If the synchronisation point was placed prior to the sink command for the example in Figure 5.8 then it would have been called as many times as the number of iterations of the J and K loops for each call of the communication, representing a potentially significant overhead.

It is of greater advantage to place the synchronisation point outside any loops that surround the sink of the communication. This reduces the overhead of calling the synchronisation call (CAP_SYNC_EXCHANGE) for every iteration of the surrounding loop.

When the sink has been established for a communication it is necessary to ascertain if it has any surrounding loops. If the sink command has surrounding loops then each loop from the innermost to the outermost must be traversed to determine the outermost valid loop before which the synchronisation point may be placed. The loops traversed must not contain the source command. If this was allowed to occur then the synchronisation point could be placed prior to the loop containing the source command.

For Figure 5.8 the synchronisation point would be traversed through loop K and J before being placed prior to the loop J. The synchronisation point may not be placed prior to loop I since it would execute before the communication as previously mentioned.

Once the actual placement of the synchronisation point has been determined, the test for time consumers between the source command and the synchronisation point, as previously mentioned, can be applied as in Section 5.2.2.1.

5.2.2.3 Source Command and Sink Command in Different Routines.

The positioning for a synchronisation call must also be determined interprocedurally, i.e. where the sink is in a different routine to the source. For example, in the Fortran pseudo code (Figure 5.9) the sink for the CAP_AEXCHANGE source is in a different routine. Originally, the synchronisation point was placed at the statement prior to the sink. The synchronisation point has then traversed all loops surrounding the sink to its present location using the same communication migration call path as that followed by the communication. This communication migration call path is stored in the field DEFROUTE of the RECEIVE^.COMMANDLIST data structure (Section 2.8.3) for each sink of the communication, specifying precisely which call lead to this instance of the communication.

This is achieved by first traversing out of all the loops surrounding the sink in the routine where the sink exists ensuring any loops traversed do not contain the source command (as in Section 5.2.2.2). Once these loops have all been exited it is possible to traverse out of any loops surrounding the call to the sink routine. This will continue for every loop on the communication migration call path until the routine containing the source command is reached.

```
Source      CALL CAP_AEXCHANGE(A(1,1,Cap_HA+1),A(1,1,Cap_LA), 200, Right, Cap_r_sync, Cap_s_sync)
            {* Other Code which does not use communicated data *}
Synch       CALL CAP_SYNC_EXCHANGE(Right, Cap_r_sync, Cap_s_sync)
            DO I = 1, 100
              CALL ROUTINE1(I,A,B)
            ENDDO
            END

            SUBROUTINE ROUTINE1(I,A,B)
            DO J = 2,28
                  CALL ROUTINE2(I,J,A,B)
            ENDDO
            END

            SUBROUTINE ROUTINE2(I,J,A,B)
            DO K = MAX(1,Cap_L), MIN(98,Cap_H)
Sink1         B(I,J,K) = A(I,J,K+1)
            ENDDO
            END
```

Figure 5.9 : Code fragment showing interprocedural migration of the synchronisation point.

Once the actual placement of the synchronisation point has been determined, the search for any time consumers is required. The algorithm in Figure 5.7 must now traverse the pre-dominator tree from the synchronisation point to the communication interprocedurally. This may be achieved by modifying the function TIMECONSUMERS_IN_ROUTINE to traverse the communication migration call path.

Figure 5.10 shows that each routine in the communication migration call path (DEFROUTE) is traversed starting from the routine containing the synchronisation point (which is passed in as the SINKCOMMAND). When all command blocks in the synchronisation point routine have been traversed and no time consumers have been detected then the next routine in the DEFROUTE is traversed with SINKCOMMAND defined as the caller to the synchronisation point routine. This is repeated for each routine in DEFROUTE until: a time consumer is detected, there are no more routines in DEFROUTE or the source command has been reached.

If the synchronisation call is placed in a different routine to the communication then the two synchronisation parameters must be passed from the routine containing the communication via any intermediary routines to the routine containing the synchronisation. These parameters are passed between these routines by means of the parameter list ensuring that they do not already exist in the parameter list. The algorithm that allows this is explained in greater detail in Section 5.2.6.

```
FUNCTION TIMECONSUMERS_IN_ROUTINE(SINKCOMMAND,SOURCECOMMAND,CROUTE)
{* Searches for a TIMECONSUMER between the SINKCOMMAND *}
{* and the SOURCECOMAND within a ROUTINE. *}
BEGIN
TIMECONSUMER = FALSE
BLOCK = SINKCOMMAND^.BLOCK
ENDBLOCK = SOURCECOMMAND^.BLOCK

WHILE (SINKCOMMAND <> NIL) DO

    {* Check for time consumers in commands prior to the sink command in the block that owns the sink command. *}
    CCOMMAND = BLOCK^.COMMAND
    WHILE (CCOMMAND is not the SINKCOMMAND) DO
        TIMECONSUMER = IS_THERE_A_TIMECONSUMER(CCOMMAND)
        CCOMMAND = CCOMMAND^.NEXT
    ENDWHILE

    {* Check all routes from the block that owns the sink command. *}
    WHILE (BLOCK^.PREDOM <> NIL) AND (BLOCK^.PREDOM <> ENDBLOCK) AND
          (TIMECONSUMER not found) DO
        {* Search all routes between a block and its predominating block until a time consumers is found. *}
        TIMECONSUMER = IS_THERE_TIMECONSUMERS_ON_ALL_ROUTES(BLOCK,BLOCK^.PREDOM)
        IF (TIMECONSUMER NOT FOUND) THEN
            TIMECONSUMER = FALSE
            FOR (each COMMAND in BLOCK) DO
                {* Search for TIMECONSUMER *}
                TIMECONSUMER = IS_THERE_A_TIMECONSUMER(COMMAND)
            ENDFOR
        BLOCK = BLOCK^.PREDOM
    ENDWHILE

    IF (BLOCK = ENDBLOCK) THEN
        {* The block containing the SOURCECOMMAND (ENDBLOCK) *}
        {* must be checked for TIMECONSUMERS if none already found. *}
        {* Check the remaining commands after the SOURCECOMMAND. *}
        CCOMMAND = SOURCECOMMAND^.NEXT
        WHILE (CCOMMAND <> NIL) DO
            TIMECONSUMER = IS_THERE_A_TIMECONSUMER(CCOMMAND)
            CCOMMAND = CCOMMAND^.NEXT
        ENDWHILE
    ENDIF

    IF (TIMECONSUMER not found) AND (CROUTE <> NIL) THEN
        {* Time consumer not found *}
        {* CROUTE is not nil indicates that the communication is in a routine *}
        {* caller. Reassign for the routine caller. *}
        SINKCOMMAND = CROUTE^.COMMAND
        CROUTE = CROUTE^.NEXT
    ELSE
        {* Time consumer found or no more routine callers. *}
        SINKCOMMAND = NIL
    ENDIF
ENDWHILE

TIMECONSUMERS_IN_ROUTINE = TIMECONSUMER
END
```

Figure 5.10 : Interprocedural algorithm for calculating time consumers.

5.2.2.4 No Time Consumers outside Surrounding Loops.

If the command prior to the synchronisation point is the communication itself or if there is deemed to be insufficient code to overlap the communication then it would not be possible to apply simple overlapping. The next logical step would be to apply the PARTIAL and UNROLL tests. It may however, be possible to try to place the synchronisation point prior to the next innermost loop. Consider the code in Figure 5.11 where the outermost valid loop is the loop J. However, there are no time consumers between the source command and the start of the loop J to allow overlapping to be profitable. In this case there is additional code within this loop J and its next innermost loop K to allow the communication to be overlapped. This has the disadvantage that the synchronisation point will be called for every iteration of the outermost loop and thus extend the overall runtime of the code. It is however, a less intrusive alternative than to apply PARTIAL or UNROLL overlap transformations.

```
DO I=1,IN
      CALL CAP_EXCHANGE(A(I,2,CAP_H+1),.....                    SOURCE COMMAND
      {* NO code to overlap. *}
      DO J=2,JN
            {* Sufficient code to overlap. *}
            CALL CAP_SYNC_EXCHANGE(Right, Cap_re_sync, Cap_se_sync)   SYNCH COMMAND
            DO K=MAX(1,CAP_L),MIN(KN,CAP_H)
                  .
                  .
                  B(J,K)= A(I,J,K+1)                            SINK COMMAND
                  .
            ENDDO
      ENDDO
ENDDO
```

Figure 5.11 : Code fragment of a synchronisation point within a loop.

Another possibility to allow SIMPLE overlapping to be applied would be to perform a loop split transformation (Section 2.11). This could allow a communication (source command) to be split from a loop it shared with its sink command allowing the synchronisation call to migrate to the outside of a loop. To ensure correct results are obtained from the parallel code an array expansion (Section 2.11) of some arrays may be required in conjunction with the loop split transformation.

5.2.2.5 Testing of the Simple Overlapping Method.

The algorithm for the automatic generation of the SIMPLE overlapping was tested on several small test cases to ensure that the synchronisation points were generated in the most

profitable and correct position in the code. Figure 5.12 shows a few of these smaller tested cases.

```
CALL CAP_AEXCHANGE(A(..),...)
        .
        .
CALL CAP_SYNC_EXCHANGE()
...=A(..)
```

Figure 5.12a : No loops

```
CALL CAP_AEXCHANGE(A(..),...)
        .
        .
CALL CAP_SYNC_EXCHANGE()
DO I=
        DO J=
                DO K=
                        ..=A(..)
                ENDDO
        ENDDO
ENDDO
```

Figure 5.12b : Loops

```
CALL CAP_AEXCHANGE(A(..),...)
        .
        .
IF (...) THEN
        {* Other code *}
        CALL CAP_SYNC_EXCHANGE()
        ...=A(..)
ELSE
        {* Other code *}
        CALL CAP_SYNC_EXCHANGE()
                ...=A(..)
ENDIF
```

Figure 5.12c : Conditionals 1

```
CALL CAP_AEXCHANGE(A(..),...)
        .
        .
IF (...) THEN
        {* Other code *}
        CALL CAP_SYNC_EXCHANGE()
        ...=A(..)
ENDIF
CALL CAP_SYNC_EXCHANGE()
...=A(..)
```

Figure 5.12d : Conditionals 2

```
CALL CAP_AEXCHANGE(A(..),...)
        .
        .
IF (...) THEN
        CALL CAP_SYNC_EXCHANGE()
        DO I=
                DO J=
                        ...=A(..)
                ENDDO
        ENDDO
ENDIF
CALL CAP_SYNC_EXCHANGE()
...=A(..)
```

Figure 5.12e : Loops and Conditionals

Figure 5.12 : A sample of test cases used for testing.

The test code in Figure 5.12a is the simplest method to test and will simply place the synchronisation point prior to the sink command. The test code in Figure 5.12b has the sink command within a set of loops. The synchronisation point is simply migrated out of these loops and placed prior to the first loop.

Figure 5.12c shows a test code where there are two sink commands each lying within a separate conditional. Each conditional provides ample additional code to allow the communication to be overlapped. A synchronisation point is placed prior to each sink command to ensure that the communication is synchronised for whatever the result of the conditional.

Figure 5.12d shows a test code where there is a sink command within the conditional and outside the conditional. A synchronisation is required for both sink commands, since the conditional statement may not be true and the synchronisation point within may not be activated. A synchronisation is therefore required before the second sink command. If the conditional is true then both synchronisation points will be activated. As mentioned previously in Section 4.4 the overlapping communications are flexible enough to allow multiple synchronisation points for the same synchronisation value. This might lead to an additional unnecessary call but deals with the eventuality of the conditional being false.

The test case in Figure 5.12e is an extension of the previous test case but with the addition of loops. For this case the synchronisation points are generated much the same as the previous example. The only difference is that the synchronisation within the conditional has traversed to be outside the loops.

The test cases in Figure 5.12 are just some of the simple test cases used to ensure that the synchronisation points were generated in the correct positions for the simple methods. Several other test cases were also used to ensure that the synchronisation points were generated correctly interprocedurally such as the example in Figure 5.9 as well for more complex real codes.

5.2.3 Calculation of the Legality and Profitability of Partial Loop Overlapping.

Once simple overlapping is proven non-applicable, the PARTIAL loop overlapping method is considered. This, as discussed in Section 4.7, involves placing the synchronisation call within some of the surrounding loops of the sink controlled by a conditional statement. Of course, this method may only be applied if there are surrounding loops to the sink command. If there are no surrounding loops present then it may not be possible to overlap the communication for that particular sink command.

The PARTIAL method is effective if the communicated data is only required during the final iterations of a surrounding loop. Consider the synchronous code in Figure 5.13.

For the communication Comm1 each processor (apart from the last one) requires the 2 items of data from its processor to its right for use in the command Sink1. This data is placed into the array A at the address starting at CAP_BHA+1. Since there are two items of data being communicated they will be placed into the array addresses CAP_BHA+1 and CAP_BHA+2. Figure 5.14 shows the partitioned array on one processor with the partition range of CAP_BLA to CAP_BHA. The loop is increasing and will sweep through the array A from CAP_BLA to CAP_BHA. The values of CAP_BHA+1 and CAP_BHA+2 are required for calculation in the Sink1 when J equals CAP_BHA-1 and CAP_BHA respectively. It is therefore required to synchronise the communication before the calculation of the first iteration that uses the communicated data, i.e. when J is equal to CAP_BHA-1.

```
                DO IT= 1,NITER,1
Comm1               CALL CAP_EXCHANGE(A(CAP_BHA+1),A(CAP_BLA),2,CAP_RIGHT)
                    DO J=MAX(1,CAP_BLA),MIN(198,CAP_BHA),1
Sink1                   B(J)=A(J+2)
                    ENDDO
                    DO J=MAX(1,CAP_BLA),MIN(200,CAP_BHA),1
                        A(J)=B(J)
                    ENDDO
                ENDDO
```

Figure 5.13 : Pseudo code of a synchronous communication requiring partial loop overlapping.

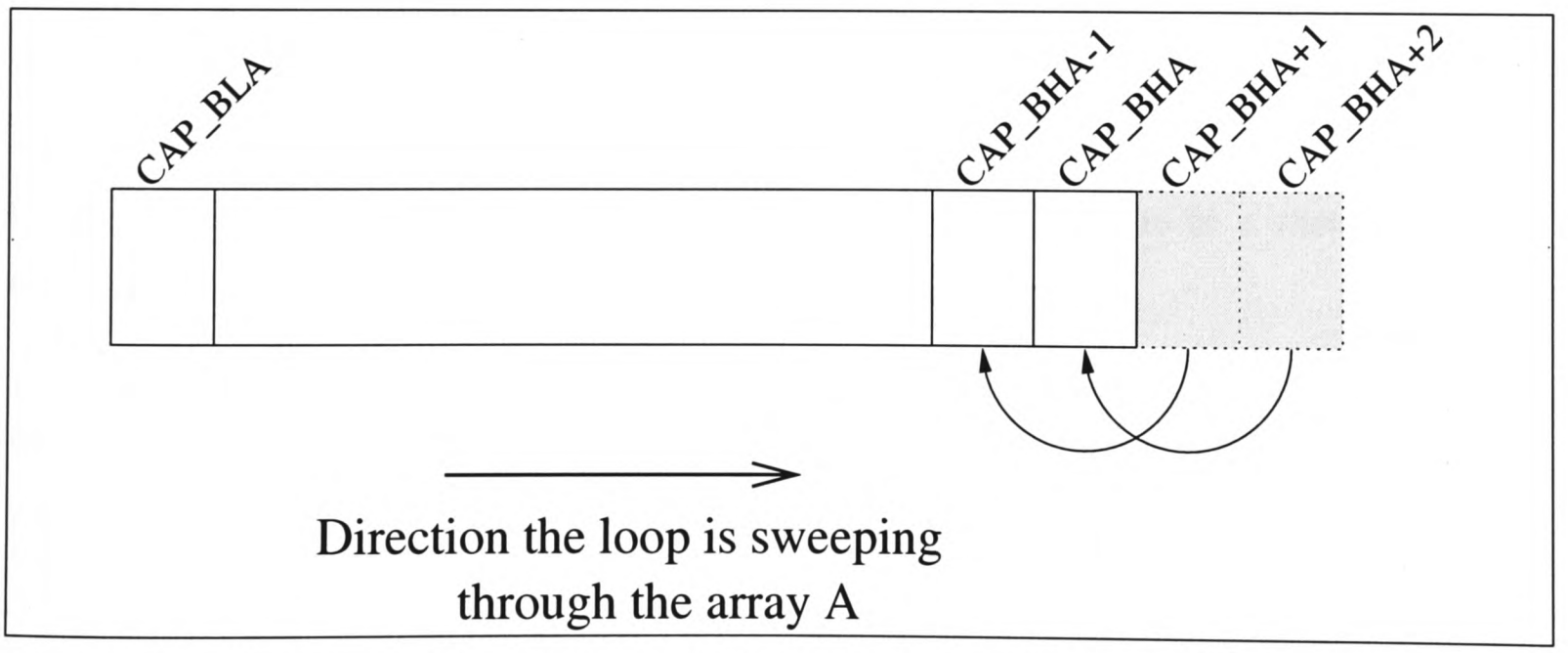


Figure 5.14 : Diagram of the loop sweep for the pseudo code in Figure 5.13.

To calculate if PARTIAL overlapping may be applied requires determining whether the communicated data is required on the final iterations of the loop. The iteration that executes the array usage being considered must be controlled by an execution control mask (Section 2.7). It is

the relationship between the execution control masks (Section 2.7) of the usage statement, the partitioned index component of the array usage and the communication control set (Section 2.8.2) that will determine which iteration of the loop will first use the communicated data.

To identify the direction of the movement of the array usage (i.e. is it increasing or decreasing) requires the symbolic expression (Section 2.5) of the partitioned index to be symbolically factorised by the execution control mask expression (Section 2.7). It is important that the remainder of this factorisation is loop invariant otherwise it is not possible to determine at which iteration the communicated data is required. Consider the formal model for a potential PARTIAL loop overlapped communication in Figure 5.15.

```
DO i1 = 1, u1, 1
    DO i2 = 1, u2, 1
        DO i3 = 1, u3, 1
            CALL CAP_EXCHANGE(A(C), ……..)
            ..
            DO in = 1, un, I
                IF (CAP_L <= m0+ m1i1+m2i2 + m3i3 … + mnin<= CAP_H) THEN
                    ….. = A(a0+a1i1+a2i2+a3i3+ … +anin)
                ENDIF
            ENDDO
            ..
        ENDDO
    ENDDO
ENDDO
```

Figure 5.15 : Formal model for a potential PARTIAL loop overlapping communication.

From the formal model (Figure 5.15) all the loops are normalised and there are a total of n surrounding loops from i_1, the outermost loop, to i_n, the innermost loop.

Let the index expression for A, the usage of the communicated data, to be a linear function of these loops where a is the loop variable coefficients for each loop:

$$A = a_0 + a_1 i_1 + a_2 i_2 + \ldots + a_n i_n$$

and the execution control mask expression M where m is a loop variable coefficient of the mask to be represented as :

$$M = m_0 + m_1 i_1 + m_2 i_2 + \ldots + m_n i_n$$

$$CAP\_L <= M <= CAP\_H$$

and C represents the partitioned component of the start address of the data that is received. This can be calculated from the communication control through the bounds of the CAP_USAGE (Section 2.8.2) variable.

Applying symbolic factorisation of the index expression, A, by the execution control mask expression, M, provides the factor, f, and remainder, r, such that :

$$A = M f + r$$

Therefore

$$A_j = f m_j, \; j = 1(1)n$$

$$a_0 = f m_0 + r$$

Using these definitions it is possible to develop an algorithm (Figure 5.16) to determine if the array usage is increasing or decreasing for a loop k, which is the outermost loop within which the execution control mask varies. The execution control mask M is determined depending on the array accesses using the factor and remainder. The algorithm calculates the first access of the communication requesting array partitioned component. The tolerance is the minimum number of iterations required per overlap and multiplying this by the relevant loop variable coefficient, in this case a_k, determines the number of iterations to be overlapped.

```
IF (f * m_k >= 0) THEN
    {* The array access is increasing with i_k*}
    M_LOWER = Cap_L
    A_LOWER = M_LOWER * f + r
    Test the Inequality : C_LOWER – A_LOWER >= Tolerance * abs(a_k)
ELSE
    {* The array access is decreasing with i_k*}
    M_UPPER = Cap_H
    A_UPPER = M_UPPER * f + r
    Test the Inequality C_UPPER – A_UPPER  >= – Tolerance * abs(a_k)
ENDIF
IF (Inequality Test is TRUE) AND (C_LOWER/UPPER – A_LOWER/UPPER<>0) THEN
    {*Generate PARTIAL overlapping*}
ELSE
    {*Test if UNROLL can be applied (See Section 5.2.4) *}
ENDIF
```

Figure 5.16 : Pseudo code algorithm for determining whether PARTIAL overlap may be applied.

This algorithm to calculate if PARTIAL overlapping may be applied (Figure 5.16) was incorporated within CAPTools using several of the symbolic variable manipulation utilities mentioned in Section 2.5.1. The symbolic factorisation is accomplished using the CAPTools utility FACTORISE (Section 2.5.1) while the multiplication and addition of the symbolic

variables is accomplished using MULTLISTS and ADDLISTS (Section 2.5.1). The test that determines if PARTIAL overlap may be applied is achieved using the CAPTools utility LDISPROVE (Section 2.5.1).

The examples that follow show how this algorithm is used to determine if PARTIAL overlap may be applied.

Example 1.

Consider the pseudo code in Figure 5.17.

```
                DO IT= 1,NITER,1
Comm1               CALL CAP_EXCHANGE(A(CAP_BHA+1),A(CAP_BLA),1,2,CAP_RIGHT)
Comm2               CALL CAP_EXCHANGE(A(CAP_BLA-1),A(CAP_BHA),1,2,CAP_LEFT)
                    DO J=MAX(1,CAP_BLA),MIN(199,CAP_BHA),1
Sink1                       B(J)=A(J+1)+A(J-1)
                    ENDDO
                    DO J=MAX(1,CAP_BLA),MIN(200,CAP_BHA),1
                        A(J)=B(J)
                    ENDDO
                ENDDO
```

Figure 5.17 : Code for Example 1 and Example 2.

Examining the first communication (Comm1) for the synchronous code in Figure 5.17 the communication control set is :

((CAP_USAGE >= CAP_BHA+1) AND (CAP_USAGE <= CAP_BHA+1))

The mask for the sink command (Sink1) is as follows :

(J.GE.CAP_BLA).AND.(J.LE.CAP_BHA)

The value of A, which is the value of the index of the array in the sink command using the communicated data, in this case is J+1. The value of M, which is the mask, in this case is J. Factorising the A by M provides:

f = 1

r = 1

i.e. A = M * f + r

J+1 = J * 1 + 1

The value of M is the first iteration that will trigger the sink command, i.e. the lower bound of the execution control mask :

M_{LOWER} = CAP_BLA

The value of C_{LOWER} is the data being communicated, i.e. the lower bound of communication control set :

$$C_{LOWER} = CAP\_BHA+1$$

The test is then executed for the conditional in the algorithm in Figure 5.16 where the array accesses are increasing :

$$A_{LOWER} = M_{LOWER} * f + r$$
$$= CAP\_BLA * 1 + 1$$
$$= CAP\_BLA + 1$$

Test $C_{LOWER} - A_{LOWER} >= Tolerance * abs(a_k)$

$CAP\_BHA + 1 - (CAP\_BLA + 1) >= 1 * 1$

$CAP\_BHA - CAP\_BLA >= 1$

$CAP\_BHA - CAP\_BLA - 1 >= 0$

This inequality may be proved true by proving the 'opposite inequality':

$$CAP\_BHA - CAP\_BLA - 1 < 0$$

is false using the symbolic inequality test (Section 2.4.6) which has known information (Section 2.8.2) that :

$$CAP\_BHA - CAP\_BLA - (Minimum\ SLAB\ NUMBER) >= 0$$

Using this information it can disprove the above 'opposite inequality' proving it to be FALSE. Since the 'opposite inequality' is FALSE then the 'original inequality' is TRUE and PARTIAL overlapping may be applied.

Example 2.

Examining the second communication (Comm2) the communication control set is :

$$((CAP\_USAGE <= CAP\_BLA-1)\ AND\ (CAP\_USAGE >= CAP\_BLA-1))$$

The mask for the sink command (Sink1) is the same as previously :

$$(J.GE.CAP\_BLA).AND.(J.LE.CAP\_BHA)$$

The value of A, the index of the array in the sink command using the communicated data, is J-1 and the value of M, the execution control mask, is J. Factorising A by M provides :

$$f = 1$$

$$r = -1$$

i.e. $A = M * f + r$

$$J-1 = J * 1 + (-1)$$

The value of M_{LOWER} is the first iteration that will trigger the sink command, since the loops are increasing for this case it will be the lower bound of the execution control mask:

$$M_{LOWER} = CAP\_BLA$$

The value of C_{LOWER} is the first value of data being communicated, i.e. the lower bound of communication control set :

$$C_{LOWER} = CAP\_BLA-1$$

The test is then executed for the conditional in the algorithm in Figure 5.16 where the array accesses are increasing :

$$A_{LOWER} = M_{LOWER} * f + r$$
$$= CAP\_BLA * 1 + (-1)$$
$$= CAP\_BLA - 1$$

Test $C_{LOWER} - A_{LOWER} >= Tolerance * abs(a_k)$

$$CAP\_BLA -1 - (CAP\_BLA -1) >= 1 * 1$$
$$0 >= 1$$

To prove this inequality true requires the testing of its 'opposite inequality' :

$$0 < 1$$

to be false using the symbolic inequality test (Section 2.4.6). This 'opposite inequality' is proved to be TRUE therefore proving the 'original inequality' to be FALSE which would not allow PARTIAL overlapping to be applied. Also, the value of $C_{LOWER} - A_{LOWER}$ is 0 and signifies that the synchronisation is required at the start of the loop. This communication will require additional testing to determine if UNROLL overlapping may be applied (Section 5.2.4).

Example 3.

The example in Figure 5.17 showed the case where the loop was increasing and the usage indices were also increasing. An example for the loop decreasing while the usage indices were also decreasing would also be very similar to those in Examples 1 and 2 apart from the fact that the 'ELSE' statements of the algorithm in Figure 5.16 would be executed. A more interesting example would be where the loop is increasing but the usage indices are decreasing. Consider the example in Figure 5.18.

```
                DO IT= 1,NITER,1
Comm1               CALL CAP_EXCHANGE(A(CAP_BLA-1),A(CAP_BHA),1,CAP_LEFT)
                    DO J=MAX(1,200-CAP_BHA),MIN(199,200-CAP_BLA),1
Sink1                   B(200-J)=A(200-J-1)
                    ENDDO
                    DO J=MAX(1,CAP_BLA),MIN(200,CAP_BHA),1
                        A(J)=B(J)
                    ENDDO
                ENDDO
```

Figure 5.18 : Pseudo code for Example 3.

Examining the communication (Comm1) the communication control set is :

((CAP_USAGE <= CAP_BLA-1) AND (CAP_USAGE >= CAP_BLA-1))

The mask for the sink command (Sink1) is as follows :

(200-J.GE.CAP_BLA).AND.(200-J.LE.CAP_BHA)

The value of A, the index of the array in the sink command using the communicated data, in this case is 200-J-1. The value of M, the execution control mask, in this case is 200-J. Factorising A by M provides :

$f = 1$

$r = -1$

i.e. $A = M * f + r$

$200\text{-}J\text{-}1 = (200\text{-}J) * 1 + (-1)$

The value of M is the first iteration that will trigger the sink command, since the loops are decreasing for this case it will be the upper bound of the execution control mask:

$M_{UPPER} = CAP\_BHA$

The value of C_{UPPER} is the data being communicated, i.e. the upper bound of communication control set :

$C_{UPPER} = CAP\_BLA\text{-}1$

The algorithm is then executed for the conditional where the array accesses are decreasing :

$A_{UPPER} = M_{UPPER} * f + r$

$= CAP\_BHA * 1 + (-1)$

$= CAP\_BHA - 1$

Test $C_{UPPER} - A_{UPPER} >= -Tolerance * abs(a_k)$

$CAP\_BLA - 1 - (CAP\_BHA - 1) >= -1 * 1$

$CAP\_BLA - CAP\_BHA + 1 >= 0$

This inequality may be proved true by proving the 'opposite inequality':

CAP_BLA - CAP_BHA + 1 < 0

is false using the symbolic inequality test (Section 2.4.6) which disproves the above 'opposite inequality' proving it to be FALSE. Since the 'opposite inequality' is FALSE then the 'original inequality' is TRUE and PARTIAL overlapping may be applied.

5.2.4 Calculation of Partial Loop Overlapping with Loop Unrolling.

If partial overlapping may not be applied it could be due to the use of the communicated data during the opening iterations of the loop. The test for partial overlapping using loop unrolling must then be applied.

Consider once again Example 2 in Section 5.2.3. The second communication Comm2 for the code in Figure 5.17 failed to have partial overlapping applied because the value of ($C_{LOWER} - A_{LOWER}$) was not greater or equal than the Tolerance*abs(a_k). This signified that the synchronisation is required at the start of the loop and requires additional testing to determine if a partial overlap with loop unrolling may be applied.

To determine if loop unrolling is allowed requires that the partitioned loop surrounding the sink command has no loop carried true dependencies (Section 2.4.1). This ensures that the data assigned in any iteration is not required in a subsequent iteration of the loop to be unrolled. If loop unrolling is possible then the number of iterations to be unrolled must be calculated. The algorithm is shown in Figure 5.19.

$C_{CONSTANT}$ = Constant of upper bound of communications control
No_of_Iterations = 0
Factorise A by M to obtain the remainder $r_{CONSTANT}$
IF ($C_{CONSTANT} > 0$) and ($r_{CONSTANT} < C_{CONSTANT}$) THEN
    {* Increasing loop *}
    No_of_Iterations = $C_{CONSTANT} - r_{CONSTANT} + 1$
ELSE IF ($C_{CONSTANT} < 0$) and ($r_{CONSTANT} >= C_{CONSTANT}$) THEN
    {* Decreasing loop *}
    No_of_Iterations = $C_{CONSTANT} - r_{CONSTANT} + 1$
ENDIF

Figure 5.19 : Algorithm to calculate how many iterations to UNROLL.

The communication control set for Comm2 in Figure 5.17 is :

((CAP_USAGE <= CAP_BLA-1) AND (CAP_USAGE >= CAP_BLA-1))

and the upper bound of the communication control set is :

CAP_BLA–1

which has a $C_{CONSTANT}$ value of –1. The value of A, the index of the array in the sink command using the communicated data, is J–1 which when factorised by the execution control mask for the sink command (Sink1) command :

(J.GE.CAP_BLA).AND.(J.LE.CAP_BHA)

provides the $r_{CONSTANT}$ of –1

i.e. $A = M * f + r_{CONSTANT}$

$J-1 = J * 1 + (-1)$

Since the $C_{CONSTANT}$ is negative the value of the No_of_Iterations is calculated to be :

$$\text{No\_of\_Iterations} = C_{CONSTANT} - r_{CONSTANT} + 1$$
$$= -1 - (-1) + 1$$
$$= 1$$

which represents the number of iterations that are required to be unrolled from the loop.

5.2.5 Merging of Synchronisation Points.

Due to the merging of communications in the generation of synchronous parallel code (Section 2.8.3) each communication may have several sinks associated with it. During the calculation stage, a synchronisation point was calculated for each individual sink. It is required to merge as many of these synchronisation points as possible. This reduces the number of calls to possible duplicated synchronisation points that are at the same position in the code. This in turn reduces any latency incurred by these synchronisation calls. It will also, in the case of the PARTIAL overlapping, allow more efficient and neater code to be generated. In the case of the UNROLL overlapping the non-merger of these synchronisation points may cause the loop to be stripped incorrectly.

Prior to the merging of the synchronisation points each one is assigned a unique synchronisation number that will correspond with the overlapped communication. If the communication has several usage points due to the merging of communications then it is essential to ensure that they all have the same synchronisation values.

Figure 5.20 shows a small section of parallel code generated by CAPTools. The code contains four CAP_EXCHANGE communications. The communication Comm1 receives the data C(CAP_H+1) from the processor to the right. This data is required in the sink commands Sink1 and Sink2 for C(I+1). The data is required for use in the sink command Sink1 twice and

once in the sink command Sink2. This communication has three sinks since their original communication requests have migrated to the same position in the code and have therefore been merged. The second communication, Comm2, exchanges the data C(CAP_L-1) with its processor to its left. This data is also required in the sink command Sink1 for the sink C(I-1). The third communication, Comm3, exchanges the value of D(CAP_L-1) with the processor to its left. This data is required by the sink D(I-1) in the sink command Sink2. Communications Comm1, Comm2 and Comm3 have been migrated to their optimal position in the code, i.e. immediately after their initialisation. The fourth communication, Comm4, exchanges two values from the start address CAP_H+1, i.e. CAP_H+1 and CAP_H+2 are exchanged, with the processor to the right. These communicated values are required in the sink command Sink1. There are three sinks in the sink command for Comm4 : two for A(I+1) and one for A(I+2). Once again these sinks' original communications have migrated to the same point and merged. The communication Comm4 cannot be migrated outside of the DO ITER loop because the exchange of the data is required at the start of each iteration due to the assignment command, Assign1, which assign the values of the array A during every iteration of that loop.

```
                READ*,(A(I),C(I),D(I),I=1,10)
Comm1           CALL CAP_EXCHANGE(C(CAP_H+1),C(CAP_L),1,CAP_RIGHT)
Comm2           CALL CAP_EXCHANGE(C(CAP_L-1),C(CAP_H),1,CAP_LEFT)
Comm3           CALL CAP_EXCHANGE(D(CAP_L-1),D(CAP_H),1,CAP_LEFT)
                    .
                    {* Other code NOT using communicated data *}
                    .
                DO ITER=1,1000,1
Comm4               CALL CAP_EXCHANGE(A(CAP_H+1),A(CAP_L),2,CAP_RIGHT)
                    DO I=MAX(2,CAP_L),MIN(9,CAP_H)
Sink1                   B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
Sink2                   B(I)=B(I)+C(I+1)+D(I-1)
                    ENDDO
                    DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign1                 A(I)=B(I)
                    ENDDO
                ENDDO
```

Figure 5.20 : Synchronous parallel code requiring SIMPLE and PARTIAL overlapped communications.

Figure 5.21 shows the overlapped communications and the synchronisation points generated if no merging was applied. Communications Comm1, Comm2 and Comm3 are SIMPLE overlapped communications while communication Comm4 is a PARTIAL overlapped communication.

```
              READ*,(A(I),C(I),D(I),I=1,10)
C             SIMPLE Overlapped Exchanges.
Comm1         CALL CAP_AEXCHANGE(C(CAP_H+1),C(CAP_L),2,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Comm2         CALL CAP_AEXCHANGE(C(CAP_L-1),C(CAP_H),1,CAP_LEFT,CAP_SE_SYNC2,CAP_RE_SYNC2)
Comm3         CALL CAP_AEXCHANGE(D(CAP_L-1),D(CAP_H),1,CAP_LEFT,CAP_SE_SYNC3,CAP_RE_SYNC3)
              .
              {* Other code NOT using communicated data *}
              .
Synch1        CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch2        CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch3        CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch4        CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC2,CAP_RE_SYNC2)
Synch5        CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC3,CAP_RE_SYNC3)
              DO ITER=1,1000,1
C                   PARTIAL Overlapped Exchange.
Comm4               CALL CAP_AEXCHANGE(A(CAP_H+1),C(CAP_L),2,CAP_RIGHT,
     &                                 CAP_SE_SYNC4,CAP_RE_SYNC4)
                    DO I=MAX(2,CAP_L),MIN(9,CAP_H)
                        IF (I.GE.CAP_H) THEN
Synch6                      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC4,CAP_RE_SYNC4)
                        ENDIF
                        IF (I.GE.CAP_H) THEN
Synch7                      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC4,CAP_RE_SYNC4)
                        ENDIF
                        IF (I+1.GE.CAP_H) THEN
Synch8                      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC4,CAP_RE_SYNC4)
                        ENDIF
Sink1                   B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
Sink2                   B(I)=B(I)+C(I+1)+D(I-1)
                    ENDDO
                    DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign1                 A(I)=B(I)
                    ENDDO
              ENDDO
```

Figure 5.21 : Overlapped parallel code with no merged synchronisation points.

For the communication Comm1 there are three synchronisation points, Synch1, Synch2 and Synch3 at the same point in the code. These three synchronisation points are generated for the two sinks C(I+1) in Sink1 and the one sink C(I+1) in Sink2. Evidently not all three synchronisation points are required at this point in the code. The last pair is merely repeating the same synchronisation that the first has already executed. This will add to the total run time of the parallel code so it is more efficient to merge these three synchronisation points together.

The communication Comm2 has only one synchronisation point (Synch4) since it has only one usage in the command Sink1. The communication Comm3 also only has one synchronisation point (Synch5) which requires the data in the sink command Sink2. Both of these synchronisation points are synchronising in the same direction, i.e. from the left. Since they are synchronising in the same direction then one of the synchronisation calls is superfluous. Only the synchronisation point Synch5 is required since its associated communication Comm3 post-dominates the communication Comm2. The values of the synchronisation variables CAP_SE_SYNC3 and CAP_RE_SYNC3 will ensure that all prior synchronisation variables for

the left direction are synchronised (Section 4.4). If the synchronisation point Synch4 were generated then only communications with synchronisation values prior to CAP_SE_SYNC2 and CAP_RE_SYNC2 would be synchronised and thus Comm3 would not be synchronised.

For the PARTIAL overlapped communication, Comm4, there are three synchronisation points: Synch6, Synch7 and Synch8. These synchronisation points each require a conditional IF statement in order to trigger the synchronisation point for the correct iteration of the loop it is partially overlapping. The synchronisation points Synch6 and Synch7 both have the same conditional (I.GE.CAP_H) which ensures that this is the last iteration. The second synchronisation point is therefore redundant and may be merged into the synchronisation point Synch6. The synchronisation point Synch8 has a conditional statement (I+1.GE.CAP_H) which ensures that this is the second from last iteration. Since the conditional for Synch8 will cause the synchronisation to be executed before the merged synchronisation point for Synch6 and Synch7, they become redundant and may be merged into the synchronisation point Synch8. The PARTIAL synchronisations may be merged depending on which has the highest number of iterations to be overlapped, i.e. Synch6 and Synch7 partially overlapped 1 iteration only while the synchronisation Synch8 partially overlapped 2 iterations.

After the merger of the synchronisation points the code in Figure 5.21 will be generated as in Figure 5.22.

```
          READ*,(A(I),C(I),D(I),I=1,10)
C         SIMPLE Overlapped Exchanges.
Comm1     CALL CAP_AEXCHANGE(C(CAP_H+1),C(CAP_L),2,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Comm2     CALL CAP_AEXCHANGE(C(CAP_L-1),C(CAP_H),1,CAP_LEFT,CAP_SE_SYNC2,CAP_RE_SYNC2)
Comm3     CALL CAP_AEXCHANGE(D(CAP_L-1),D(CAP_H),1,CAP_LEFT,CAP_SE_SYNC3,CAP_RE_SYNC3)
          .
          {* Other code NOT using communicated data *}
          .
Synch1/2/3 CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch4/5  CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC3,CAP_RE_SYNC3)
          DO ITER=1,1000,1
C               PARTIAL Overlapped Exchange.
Comm4           CALL CAP_AEXCHANGE(A(CAP_H+1),C(CAP_L),2,CAP_RIGHT,
     &                                CAP_SE_SYNC4,CAP_RE_SYNC4)
                DO I=MAX(2,CAP_L),MIN(9,CAP_H)
                    IF (I+1.GE.CAP_H) THEN
Synch6/7/8                CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC4,CAP_RE_SYNC4)
                    ENDIF
Sink1               B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
Sink2               B(I)=B(I)+C(I+1)+D(I-1)
                ENDDO
                DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign1             A(I)=B(I)
                ENDDO
          ENDDO
```

Figure 5.22 : Code from Figure 5.21 after merging synchronisation points.

The code in Figure 5.23 shows another example with the synchronous parallel code before applying loop unrolling. It consists of an exchange communication involving two data values from the start address A(CAP_L-2). This is communicating the values of A(CAP_L-2) and A(CAP_L-1) which are required for calculation during the first two iterations of the loop. These two values are required since there are two sinks A(I-1) and A(I-2) in the sink command Sink1. The sink A(I-1) will require a loop unrolling for the first iteration only while the sink A(I-2) will require the first and second iteration to be unrolled. If these iterations were stripped from the original loop independently then the code obtained would be incorrect, since the for the first iteration would be calculated twice. These two synchronisation points must be merged before loop unrolling is performed.

```
            READ*,(A(I),I=1,10)
            DO ITER=1,1000,1
Comm1           CALL CAP_EXCHANGE(A(CAP_L-2),A(CAP_H-1),2,CAP_LEFT)
                DO I=MAX(2,CAP_L),MIN(9,CAP_H)
Sink1               B(I)=B(I)+A(I-1)+A(I-2)
                ENDDO
                DO I=MAX(1,CAP_L),MIN(10,CAP_H)
                    A(I)=B(I)
                ENDDO
            ENDDO
```

Figure 5.23 : Synchronous parallel code requiring UNROLL overlapped communication.

The code in Figure 5.24 shows that the loop unrolling is applied only for the sink that requires the most iterations stripped. The loop unrolling for the other sink, which strips only the first iteration is not applied since that iteration of the loop has already been unrolled from the original loop.

```
            READ*,(A(I),I=1,10)
            DO ITER=1,1000,1
Comm1           CALL CAP_AEXCHANGE(A(CAP_L-2),A(CAP_H-1),2,CAP_LEFT,Cap_R_Sync,Cap_S_Sync)
                DO I=MAX(2,CAP_L)+2,MIN(9,CAP_H)
Sink1               B(I)=B(I)+A(I-1)+A(I-2)
                ENDDO
                CALL CAP_SYNC_EXCHANGE(Cap_R_Sync,Cap_S_Sync)
                DO I=MAX(2,CAP_L),MIN(9,CAP_H,MAX(2,CAP_L)+1)
                    B(I)=B(I)+A(I-1)+A(I-2)
                ENDDO
                DO I=MAX(1,CAP_L),MIN(10,CAP_H)
                    A(I)=B(I)
                ENDDO
            ENDDO
```

Figure 5.24 : Code from Figure 5.18 with UNROLL overlapped communications.

The basic algorithm for merging the synchronisation points for PARTIAL and UNROLL synchronisation is summarised in Figure 5.25.

```
IF (SYNCH_POINT = PARTIAL) THEN
    {* Find MATCHING_SYNCH_POINT which has:                    *}
    {*          - same COMMUNICATION DIRECTION;                *}
    {*          - same SURROUNDING LOOP.                       *}
    IF (MATCHING_SYNCH_POINT FOUND) THEN
        {* Find the earliest iteration that requires synchronisation *}
        IF (LOOP INCREASING) THEN
            {* Calculate which SYNCH_POINT has the most number *}
            {* of positive iterations to be partially overlapped.    *}
        ELSE IF (LOOP DECREASING) THEN
            {* Calculate which SYNCH_POINT has the most number *}
            {* of negative iterations to be partially overlapped.    *}
        ENDIF
    ENDIF
ELSE IF (SYNCH_POINT = UNROLL ) THEN
    {* Find MATCHING_SYNCH_POINT which has same SURROUNDING LOOP.   *}
    IF (MATCHING_SYNCH_POINT FOUND) THEN
        {* Find the synchronisation that requires the most iteration to be unrolled *}
        IF (LOOP INCREASING) THEN
            {* Calculate which SYNCH_POINT has the most   *}
            {* number of positive iterations to be UNROLLED *}
            MAX_UNROLL_COMM = comm with most number of iterations to unroll
        ELSE IF (LOOP DECREASING) THEN
            {* Calculate which SYNCH_POINT has the most   *}
            {* number of negative iterations to be UNROLLED *}
            MAX_UNROLL_COMM = comm with most number of iterations to unroll
        ENDIF
    ENDIF
ENDIF
{* Remove any duplicate redundant UNROL synchronisations *}
```

Figure 5.25 : Basic algorithm for merging PARTIAL and UNROLL synchronisation points.

When the PARTIAL and UNROLL synchronisation points have been merged the SIMPLE synchronisation points may be merged with each other (as previously mentioned in this Section). Two SIMPLE synchronisations may only be merged together if they are synchronising in the same direction. If two synchronisation points also have the same unique synchronisation values set, then evidently they have the same source command communication and may be merged (cf. Synch1, Synch2 and Synch3 in Figure 5.21).

If, however, there are two synchronisation points at the same point in the code with the same direction but different unique synchronisation values then their communication call paths (Section 2.8.3) will have to be traversed. If the call paths are the same or along the same call path then it may be possible to merge the synchronisation points if one of the communications post-dominates (Section 2.3.2) the other.

Figure 5.26 has a subroutine SUB3 with two synchronisation points. Traversing the pre-dominator tree (Section 2.3.2) and the call graph (Section 2.3.1) for Synch1 will pass through

subroutine SUB2 to the communication Comm1 in subroutine SUB1. For Synch2 the pre-dominator tree is traversed to the communication Comm2 in subroutine SUB2. This is a subset of the previous path traversed for the other synchronisation point. The communication Comm1 is clearly post-dominated by the communication Comm2 and the synchronisation points may be merged.

```
            SUBROUTINE SUB1
Comm1       CALL CAP_AEXCHANGE(A(CAP_H+1),A(CAP_L),1,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            CALL SUB2(A,B,CAP_SE_SYNC1,CAP_RE_SYNC1)
            END

            SUBROUTINE SUB2(A,B,CAP_SE_SYNC1,CAP_RE_SYNC1)
Comm2       CALL CAP_AEXCHANGE(C(CAP_H+1),C(CAP_L),1,CAP_RIGHT,CAP_SE_SYNC2,CAP_RE_SYNC2)
            CALL SUB3(A,B,C,CAP_SE_SYNC1,CAP_RE_SYNC1,CAP_SE_SYNC2,CAP_RE_SYNC2)
            END

            SUBROUTINE SUB3(A,B,C,CAP_SE_SYNC1,CAP_RE_SYNC1,CAP_SE_SYNC2,CAP_RE_SYNC2)
Synch1      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch2      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC2,CAP_RE_SYNC2)
            DO I=1,10
                B(I)= B(I)+A(I)+A(I+1)+C(I+1)
            ENDDO
            END
```

Figure 5.26 : Two synchronisation points with different synchronisation values

5.2.6 Passing of Synchronisation Values between Routines.

If the synchronisation call is placed in a different routine to the communication then the two synchronisation parameters must be passed from the routine containing the communication to the routine containing the synchronisation.

The easiest method of achieving this, from an automation point of view, would be to use COMMON blocks. The disadvantage of this method is that it may make the code less generic. For example, consider the code example in Figure 5.27 where there is a synchronisation point in subroutine SUB2 for three separate communications Comm1, Comm2 and Comm3. For this example each of the communications has different synchronisation point values. If these values were passed via a Common block then all three pairs of synchronisation values would have to be passed between subroutine SUB1 and SUB2.

However, the synchronisation values are passed between subroutines using the parameter list for the subroutine SUB2. This allows only the one pair of synchronisation points to be passed to the subroutine SUB2 thus preserving the generic nature of the subroutine.

```
            SUBROUTINE SUB1
                  :
Comm1       CALL CAP_AEXCHANGE(C(CAP_H+1),C(CAP_L),1,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            CALL SUB2(A,B,C,CAP_SE_SYNC1,CAP_RE_SYNC1)
                  :
                  :
Comm2       CALL CAP_AEXCHANGE(D(CAP_H+1),D(CAP_L),1,CAP_RIGHT,CAP_SE_SYNC2,CAP_RE_SYNC2)
            CALL SUB2(A,B,D,CAP_SE_SYNC2,CAP_RE_SYNC2)
                  :
                  :
Comm3       CALL CAP_AEXCHANGE(E(CAP_H+1),E(CAP_L),1,CAP_RIGHT,CAP_SE_SYNC3,CAP_RE_SYNC3)
            CALL SUB2(A,B,E ,CAP_SE_SYNC3,CAP_RE_SYNC3)
            END


            SUBROUTINE SUB2(A,B,C,CAP_SE_SYNC1,CAP_RE_SYNC1)
Synch       CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            DO I=1,10
                  B(I)= B(I)+A(I)+C(I+1)
            ENDDO
            END
```

Figure 5.27 : Example showing the passing of synchronisation values between routines.

This method will also require the passing of these parameters through any intermediary routines. This is accomplished by once again traversing the communication migration path stored in DEFROUTE (Section 2.8.3).

There is, however, the need to find other calls that are not on the DEFROUTE path to ensure that the additional dummy parameters are added to all calls of the routines on the DEFROUTE. Consider the code in Figure 5.28.

```
Comm1       CALL CAP_AEXCHANGE(A(CAP_H+1),A(CAP_L),1,2,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
Comm2       CALL CAP_AEXCHANGE(C(CAP_H+1),C(CAP_L),1,2,CAP_RIGHT,CAP_SE_SYNC2, CAP_RE_SYNC2)
            {* Other code to overlap *}
Call1       CALL ASSIGNB(1,A,B,CAP_SE_SYNC1,CAP_RE_SYNC1)
Call2       CALL ASSIGNB(2,C,B,CAP_SE_SYNC2, CAP_RE_SYNC2)

            SUBROUTINE ASSIGNB(OPTION,X,Y,CAP_SE_SYNC1,CAP_RE_SYNC1,CAP_SE_SYNC2, CAP_RE_SYNC2)
            IF (OPTION.EQ.1) THEN
Synch1                  CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1, CAP_RE_SYNC1)
                        DO I = 1, 10
                                X(I) = Y(I) + Y(I+1)
                        ENDDO
            ELSE IF (OPTION.EQ.2) THEN
Synch2                  CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC2, CAP_RE_SYNC2)
                        DO I = 1, 10
                                X(I) = X(I+1) + Y(I)
                        ENDDO
            ENDIF
```

Figure 5.28 : Example showing the need to find all callers to a routine.

In Figure 5.28 the first call to the routine ASSIGNB (Call1) passes in the value of the array A and passes in the parameters for the synchronisation values of its corresponding exchange communication, Comm1 for use in the synchronisation point Synch1. The second

call to the routine ASSIGNB (Call2) passes in the value of the array C and passes in the parameters for the synchronisation values of its corresponding exchange communication, Comm2 for use in the synchronisation point Synch2.

For the synchronisation point Synch1 the DEFROUTE path to Call1 places the synchronisation values at the end of the parameter list. There is however an additional call to the same routine in Call2 which does not contain the same synchronisation values since they are in different DEFROUTE paths. It is therefore necessary to ensure that all other calls to the same routine have the same parameter list.

5.2.7 Generation of Overlapped Communication.

The completion of the calculation of which communications may be overlapped and the merger of the synchronisation points allows the overlapped communications to be generated. This generation will be done in the prescribed order mentioned earlier in Section 5.2 - UNROLL, PARTIAL and finally the SIMPLE overlapped communication. The generation of the communication in this order is vital to ensure that during loop unrolling the effects of the partial and simple overlapped communication are not also unrolled and therefore cause synchronisation points and conditional statements to be copied erroneously.

The first task during generation is to convert the present exchange communication commands to be overlapping communication calls. This requires a simple change of the call name from CAP_EXCHANGE to CAP_AEXCHANGE (Sections 1.6 and 4.4). Two additional parameters are also appended to the parameter list. These two parameters are the synchronisation values for both the receive and send of the exchange communication. These parameters will take the form of CAP_RE_SYNC_XXX and CAP_SE_SYNC_XXX, where the XXX represents a number that is uniquely linked to each synchronisation point.

Generating the synchronisation point CAP_SYNC_EXCHANGE requires three parameters (Section 4.4) - the direction which the communication data is being communicated, and a synchronisation values for both the receive and send of the exchange communication. The first parameter can be identified from the exchange communication itself. The other two parameters are generated for their related communications and will have the same values as all the relevant communications.

If the synchronisation call is placed in a different routine to the communication then the two synchronisation parameters must be passed from the routine containing the communication

via any intermediary routines to the routine containing the synchronisation. These parameters are passed between these routines by means of the parameter list ensuring that they do not already exist in the parameter list. The algorithm that allows this is explained in greater detail in Section 5.2.6.

5.2.7.1 Generation of Partial Loop Overlapping with Loop Unrolling.

During the generation of UNROLL overlapped communications the start and end of the loop to be unrolled must be established and a copy of that loop placed immediately after the present position of the loop. The loop limits of this loop must then be adjusted. The algorithm to allow this is summarised in Figure 5.29.

```
FOR (every UNROLL communication) DO
    IF (MAX_UNROLL_COMM) THEN
        {* Unroll communication with maximum number of iterations: *}
        {* Find start and end of loop to be unrolled. *}
        {* Generate the CALL CAP_SYNC_EXCHANGE after original loop end. *}
        {* Copy loop and place after the current loop end and the synchronisation call. *}
        {* Adjust the loop limits of the original loop: *}
            IF (LOOPDIRECTION > 0) THEN
                {* Increment lower limit of loop by NO_OF_ITERATIONS. *}
            ELSE
                {* Decrement lower limit of loop by NO_OF_ITERATIONS. *}
            ENDIF
        {* Adjust limits of copied loop: *}
            {* Copy the upper limit of original loop. *}
            {* Add a third parameter to the upper limit of the loop : *}
            IF (LOOPDIRECTION > 0) THEN
                {* Add lower limit of original loop - I *}
            ELSE
                {* Add lower limit of original loop + I *}
            ENDIF
        {* Adjust any loop labels copied to avoid conflict. *}
    ELSE
        {* Generate CAP_SYNC_EXCHANGE for other UNROLL communications *}
        {* whose loops have been already unrolled by the MAX_UNROLL_COMM. *}
    ENDIF
ENDFOR
```

Figure 5.29 : Algorithm to generate partial loop overlapping with loop unrolling.

If the loop has increasing iterations (Figure 5.30) then the lower limit of the original loop is incremented and the upper limit of the copied loop must be adjusted. The lower limit of the original loop is adjusted such that it will now start on the first iteration that does not require the communicated data, in this case MAX(2,CAP_L)+2. The upper limit of the copied loop is adjusted such that the loop will iterate from the original initial iteration to the final iteration that requires the communicated data or the last iteration of the original loop, whichever is lowest, in this case MIN(9,CAP_H,MAX(2,CAP_L)+2-1). This will ensure that the correct loop range is

executed for all loop unrolling cases including the possible case where the values of CAP_L and CAP_H may be equal.

```
                READ*,(A(I),I=1,10)
                DO ITER=1,1000,1
Comm1                   CALL CAP_AEXCHANGE(A(CAP_L-2),A(CAP_H-1),2,CAP_LEFT,
     &                                     CAP_SE_SYNC1,CAP_RE_SYNC1)
                        DO 10 I=MAX(2,CAP_L)+2,MIN(9,CAP_H)
Sink1                       B(I)=B(I)+A(I-1)+A(I-2)
          10            CONTINUE
                        CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC1,CAP_RE_SYNC1)
                        DO 100 I=MAX(2,CAP_L),MIN(9,CAP_H,MAX(2,CAP_L)+2-1)      } Copied
Sink1                       B(I)=B(I)+A(I-1)+A(I-2)                              } Loop
          100           CONTINUE                                                 }
                        DO I=MAX(1,CAP_L),MIN(10,CAP_H)
                            A(I)=B(I)
                        ENDDO
                    ENDDO
```

Figure 5.30 : Loop with increasing iterations.

If the loop is decreasing (Figure 5.31) then the low of the original loop MIN(9,CAP_H)-2 is decreased and the high of the copied loop, MAX(2,CAP_L,MIN(9,CAP_H)-2+1) is also decreased.

```
                READ*,(A(I),I=1,10)
                DO ITER=1,1000,1
Comm1                   CALL CAP_AEXCHANGE(A(CAP_H+1),A(CAP_L),2,CAP_LEFT,
     &                                     CAP_SE_SYNC1,CAP_RE_SYNC1)
                        DO I=MIN(9,CAP_H)-2,MAX(2,CAP_L),-1
Sink1                       B(I)=B(I)+A(I+1)+A(I+2)
                        ENDDO
                        CALL CAP_SYNC_EXCHANGE(CAP_LEFT,CAP_SE_SYNC1,CAP_RE_SYNC1)
                        DO I=MIN(9,CAP_H),MAX(2,CAP_L,MIN(9,CAP_H)-2+1),-1       } Copied
Sink1                       B(I)=B(I)+A(I+1)+A(I+2)                              } loop
                        ENDDO                                                    }
                        DO I=MAX(1,CAP_L),MIN(10,CAP_H)
                            A(I)=B(I)
                        ENDDO
                    ENDDO
```

Figure 5.31 : Loop with decreasing iterations.

It is often the case when unrolling loops that the loop labels are copied from the original loops. All the labels within this copied loop must be changed to ones that do not already exist in that routine. These labels may occur for DO, CONTINUE and GOTO commands and their target commands.

5.2.7.2 Generation of Partial Loop Overlapping

When all the Partial overlapping with Loop Unrolling communications have been generated the Partial Overlapping communications may be generated. This involves the generation of a conditional IF statement containing the synchronisation call. The algorithm to perform this is shown in Figure 5.32.

```
FOR (Every PARTIAL overlapping communication) DO
      {* Create a conditional IF inside partitioned loop. *}
      IF (LOOPDIRECTION > 0) THEN
            {* Increasing index. *}
            {* Set conditional to be .GE. *}
      ELSE
            {* Decreasing index. *}
            {* Set conditional to be .LE. *}
      ENDIF
      {* Generate r.h.s. of conditional. *}
      {* Generate l.h.s. of conditional. *}
      {* Generate CAP_SYNC_EXCHANGE. *}
      {* Generate ENDIF. *}
ENDFOR
```

Figure 5.32 : Algorithm to generate the conditional synchronisation call for the partial loop overlapping.

The conditional command generated will be dependant on whether the surrounding loop is increasing or decreasing. The conditionals greater than or equal (.GE.) or less than or equal (.LE.) are used as opposed to the conditional equals to (.EQ.) to ensure that the conditional is triggered correctly. If the loop has a step of 1 then the .EQ. would be sufficient but if the loop had any other step then it could be possible that the conditional is not triggered.

```
DO I = 1,50
  CALL CAP_EXCHANGE(A(Cap_H+1), A(Cap_L),..)
  DO J = MAX(2,Cap_L),MIN(59,Cap_H),1



     B(J) = A(J+1) + A(J+2)
  ENDDO
ENDDO
```

a) Synchronous

```
DO I = 1,50
  CALL CAP_AEXCHANGE(A(Cap_H+1), A(Cap_L),..)
  DO J = MAX(2,Cap_L),MIN(59,Cap_H),1
      IF (J+2.GE.Cap_H+1) THEN
        CALL CAP_SYNC_EXCHANGE(Right,..)
      ENDIF
      B(J) = A(J+1) + A(J+2)
  ENDDO
ENDDO
```

b) Overlapped

Figure 5.33 : Synchronous and overlapping code applying partial loop overlapping.

In the synchronous pseudo code (left hand side in Figure 5.33) the coefficients of the partitioned loop variable J are positive in both usages. A synchronisation point is required for when either of these indices (J+2 or J+1) is greater than or equal to the lower bound of the

communication range (Cap_H+1). The synchronisation point is placed immediately after the loop head with a conditional for J+2 >= Cap_H+1, as shown in the overlapping code (right hand side in Figure 5.33).

5.2.7.3 Generation of Simple Overlapping.

Generating the synchronisation call for the Simple Overlapped communication requires the CAP_SYNC_EXCHANGE to be placed at the most advantageous point as calculated in Section 5.2.2 and the communication call is changed to be a CAP_AEXCHANGE as opposed to CAP_EXCHANGE. Two additional synchronisation values which correspond to those of the CAP_SYNC_EXCHANGE are also added to the parameter list.

5.2.7.4 Communications with Several Sinks using Different Overlapping Methods.

It is possible for one communication to possess different sinks that requires a selection of different overlapping methods. As long as every sink may be overlapped then it is possible to apply the overlapping methods. Consider the code in Figure 5.34.

```
          READ*,(A(I),C(I),D(I),I=1,10)
          DO ITER=1,1000,1
Comm1     CALL CAP_EXCHANGE(A(CAP_H+1),A(CAP_L),2,2,CAP_RIGHT)
               IF (TEST.EQ.1) THEN
                    {* Lots of code to overlap - SIMPLE *}
                    DO I=MAX(2,CAP_L),MIN(9,CAP_H)
Sink1               B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
                    ENDDO
                    DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign1                  A(I)=B(I)
                    ENDDO
               ELSE IF (TEST.EQ.2) THEN
                    {* No code to overlap - PARTIAL *}
                    DO I=MAX(2,CAP_L),MIN(9,CAP_H)
Sink2               B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
                    ENDDO
                    DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign2                  A(I)=B(I)
                    ENDDO
               ELSE
                    {* No code to overlap - UNROLL *}
                    DO I=MIN(9,CAP_H),MAX(2,CAP_L),-1
Sink3               B(I)=B(I)+A(I+1)+A(I+1)+A(I+2)+C(I+1)+C(I+1)+C(I-1)
                    ENDDO
                    DO I=MAX(1,CAP_L),MIN(10,CAP_H)
Assign3                  A(I)=B(I)
                    ENDDO
               ENDIF
          ENDDO
```

Figure 5.34 : Communication with several sinks using different overlapping methods.

The code in Figure 5.34 consists of a single communication with three different sink commands. Either one of these sink commands may be executed depending on the outcome of the conditional statements. For the first conditional there is ample amount of code to overlap to allow the SIMPLE method to apply. For the second conditional there is no other code to overlap. Since the data is not required until the final iteration then the PARTIAL method is applied. In the final conditional the communicated data is required in the earlier iterations of the loop and the UNROLL method has to be applied. If for any reason any one of the sink commands could not be overlapped then none of the other sink commands could be overlapped either. For instance, if the first two sink commands could be overlapped then if their corresponding conditionals were triggered then the communication could be synchronised. However, if the third conditional were true then the overlapped communication could not synchronise and the sink command could use the incorrect data.

5.2.8 Validation of the Overlapping Communications Generation.

All the methods were tested on small test cases before being tested on real codes. The results for these real codes are collated and discussed in Chapter 6. No problems were encountered when running these codes. Correct results were obtained for all the codes apart from one. Investigation of this code revealed that there was a minor flaw in the generation of the PARTIAL overlapping. Consider the code in Figure 5.35.

```
          DO ITER = 1, NITER
Comm1         CALL CAP_AEXCHANGE(A(J,CAP_H+1),A(J,CAP_L),1,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
              DO K=MAX(2,CAP_L),MIN(KMAX,CAP_H)
                  IF (K.GE.CAP_H) THEN
Synch1                CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
                  ENDIF
                  DO J=3,JMAX
Usage1                ...=A(J,K+1)
                  ENDDO
              ENDDO
              DO K=MAX(2,CAP_L),MIN(KMAX,CAP_H)
                  DO J=3,JMAX
Assign1               A(J,K) = ....
                  ENDDO
              ENDDO
          ENDDO
```

Figure 5.35 : Original partial overlapped code generated.

In the code in Figure 5.35 the value of A(J,CAP_H+1) is being communicated by the communication Comm1. This communication is synchronised within the loop at Synch1 before being used in calculation at the command Usage1. This communication occurs for every

iteration of the loop ITER. Within this same loop the value of the array A is being reassigned at Assign1. The data being communicated asynchronously could also be overwritten by the reassignment of the same data. It might be the case, for example that the last processor may not synchronise due to the original loop limits KMAX being less than CAP_H. This will lead to that processor arriving at the command Assign1 before the communication has been completed. To avoid this occurrence an additional synchronisation point is placed immediately after the loop surrounding the partial conditional (Synch2 in Figure 5.36). This ensures that all processors are synchronised.

```
        DO ITER = 1, NITER
Comm1       CALL CAP_AEXCHANGE(A(J,CAP_H+1),A(J,CAP_L),1,CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            DO K=MAX(2,CAP_L),MIN(KMAX,CAP_H)
                IF (K.GE.CAP_H) THEN
Synch1              CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
                ENDIF
                DO J=3,JMAX
Usage1              ...=A(J,K+1)
                ENDDO
            ENDDO
Synch2      CALL CAP_SYNC_EXCHANGE(CAP_RIGHT,CAP_SE_SYNC1,CAP_RE_SYNC1)
            DO K=MAX(2,CAP_L),MIN(KMAX,CAP_H)
                DO J=3,JMAX
Assign1             A(J,K) = ....
                ENDDO
            ENDDO
        ENDDO
```

Figure 5.36 : Modified partial overlapped code now generated.

5.3 Pipelines.

From the discussion in Section 4.8 it may be observed that there is a general pattern for both synchronous and overlapping communication pipelines which may be formulated as a formal model. To achieve this requires the definition of the functions required for the formal model. Consider the simple general model in Figure 5.37.

```
DO i1 = 1, u1, 1
    DO i2 = 1, u2, 1
        ..
        DO iN = 1, uN, 1
            A( g1(i1,i2,...,iN), g2(i1,i2,...,iN),..., gM(i1,i2,...,iN) ) = ....
        ENDDO
        ..
    ENDDO
ENDDO
```

Figure 5.37 : A simple general model.

From the simple general formal model (Figure 5.37) all the loops had been normalised and that there were a total of N surrounding loops from i_1, the outermost loop, to i_N, the innermost loop. The reference to array A takes the form :

$$A(g_1,g_2,..,g_M)$$

Each array index of the array A is a function of g for all the loop iteration counters from i_1 to i_N. Each index function g is defined as a linear function of these loop iteration counters, e.g. for an array index g_j :

$$g_j(i_1,i_2,..,i_N) = g_{j,0} + g_{j,1}i_1 + g_{j,2}i_2 + ... + g_{j,N}i_N$$

where $g_{j,0}$ represents the constant term and $g_{j,1}$ the outermost loop variable coefficient up to $g_{j,N}$ the innermost loop variable coefficient. Each $g_{j,k}$ is a constant which can consist of both an integer part and a symbolic part, e.g.

$$g_{j,k} = (N*M) + N + 5$$

Where N and M are loop invariant in all the loops in the pipeline.

Using these definitions it is possible to develop a general formal model for both the synchronous and the overlapping pipeline as can be seen in Figure 5.38 and Figure 5.39 respectively.

In Figure 5.38 the synchronous pipeline is similar to the example of the synchronous code illustrated in Figure 3.19. Loops have been incorporated into the model to anticipate all possible loop cases, for example, there are additional loops placed around the communications due to additional loops within the pipeline surrounding the calculation. The innermost loop of the pipeline is indicated by PIPELEVEL on the loop I_P. It is outside of this PIPELEVEL loop that the communications are placed for a synchronous communication pipeline.

The general formal model for the overlapping pipeline in Figure 5.39 follows the same outline as the overlapping pipeline code illustrated in Figure 4.18. Once again all possible additional loops have been incorporated into the model. The innermost loop of the pipeline is also signified by PIPELEVEL on the loop I_P. It is also required to signify the outermost valid loop of the pipeline denoted by OUTERLEVEL on the loop I_A. Other loops that surround OUTERLEVEL, which are not valid loops of an overlapping pipeline are also incorporated. There is a conditional statement for each loop from the OUTERLEVEL loop to the loop prior to the PIEPLEVEL loop. Each of the overlapping receive communications within these conditionals will also possess the same surrounding loops as the overlapping receive prior to the OUTERLEVEL loop.

The first objective of the conversion of the pipeline from synchronous to overlapping communication is to find the innermost loop PIPELEVEL (P) and the outermost loop OUTERLEVEL (A) of the pipeline. The loop P is the partitioned loop that surrounds the pipeline calculation and it is prior to this loop that a synchronous receive communication will be placed by the synchronous code generation of CAPTools. To determine the position of the OUTERMOST loop A it is necessary to apply three conservative tests.

```
DO I1=1,U1,1
  DO I2=1,U2,1
    ..
    DO IA-1=1,UA-1,1
      DO IA=1,UA,1
        DO IA+1=1,UA+1,1
          DO IP-2=1,UP-2,1
            DO IP-1=1,UP-1,1
              DO IX=1,HX,1
                DO IY=1,HY,1
                  Call Cap_Receive( A(g1(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IX,..,IY),..,
                                     gn(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IX,..,IY)),1,Left)
                ENDDO(IY)
              ENDDO(IX)
              DO IP=1,UP,1                                    <-------- PIPELEVEL (P)
                ..
                DO IP+1=1,UP+1,1
                  DO IX=1,UX,1
                    DO IY=1,UY,1
                      (* Pipeline Calculation *)
                    ENDDO(IY)
                  ENDDO(IX)
                ENDDO(IP+1)
                ..
              ENDDO(IP)
              DO IX=1,HX,1
                DO IY=1,HY,1
                  Call Cap_Send( A(g1(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IX,..,IY),..,
                                  gn(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IX,..,IY)),1,Right)
                ENDDO(IY)
              ENDDO(IX)
            ENDDO(IP-1)
          ENDDO(IP-2)
        ENDDO(IA+1)
      ENDDO(IA)
    ENDDO(A-1)
    ..
  ENDDO(I2)
ENDDO(I1)
```

Figure 5.38 : Formal model for a synchronous pipeline.

```
DO I1=1,U1,1
  DO I2=1,U2,1
    .
    DO IA-1=1,UA-1,1
      DO IX=1,HX,1
        DO IY=1,HY,1
          Call Cap_AReceive( A(g1(I1,I2,..,IA-1,1,1,1,1,IP,IP+1,IX,..,IY),..,
                    gn(I1,I2,..,IA-1,1,1,1,1,IP,IP+1,IX,..,IY)),1,Left,Cap_R_Sync )
        ENDDO(IY)
      ENDDO(IX)
      DO IA=1,UA,1                                    <------ OUTERLEVEL (A)
        DO IA+1=1,UA+1,1
          DO IP-2=1,UP-2,1
            DO IP-1=1,UP-1,1
              Call Cap_Sync_Receive(Left,Cap_R_Sync)
              IF(IP-1+1.LE.UP-1)THEN
                DO IX=1,HX,1
                  DO IY=1,HY,1
                    Call Cap_AReceive( A(g1(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1+1,IP,IP+1,IX,..,IY),..,
                              gn(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1+1,IP,IP+1,IX,..,IY)),1,Left,Cap_R_Sync )
                  ENDDO(IY)
                ENDDO(IX)
              ELSEIF(IP-2+1.LE.UP-2)THEN
                DO IX=1,HX,1
                  DO IY=1,HY,1
                    Call Cap_AReceive( A(g1(I1,I2,..,IA-1,IA,IA+1,IP-2+1,1,IP,IP+1,IX,..,IY),..,
                              gn(I1,I2,..,IA-1,IA,IA+1,IP-2+1,1,IP,IP+1,IX,..,IY)),1,Left,Cap_R_Sync )
                  ENDDO(IY)
                ENDDO(IX)
              ELSEIF(IA+1+1.LE.UA+1)THEN
                DO IX=1,HX,1
                  DO IY=1,HY,1
                    Call Cap_AReceive( A(g1(I1,I2,..,IA-1,IA,IA+1+1,1,1,IP,IP+1,IX,..,IY),..,
                              gn(I1,I2,..,IA-1,IA,IA+1+1,1,1,IP,IP+1,IX,..,IY)),1,Left,Cap_R_Sync )
                  ENDDO(IY)
                ENDDO(IX)
              ELSEIF(IA+1.LE.UA)THEN
                DO IX=1,HX,1
                  DO IY=1,HY,1
                    Call Cap_AReceive( A(g1(I1,I2,..,IA-1,IA+1,1,1,1,IP,IP+1,IX,..,IY),..,
                              gn(I1,I2,..,IA-1,IA+1,1,1,1,IP,IP+1,IX,..,IY)),1,Left,Cap_R_Sync )
                  ENDDO(IY)
                ENDDO(IX)
              ENDIF
              DO IP=1,UP,1                            <------ PIPELEVEL (P)
                .
                DO IP+1=1,UP+1,1
                  DO IX=1,HX,1
                    DO IY=1,HY,1
                      (* Pipeline Calculation *)
                    ENDDO(IY)
                  ENDDO(IX)
                ENDDO(IP+1)
                .
              ENDDO(IP)
              Call Cap_Sync_Send(Right,Cap_S_Sync)
              DO IX=1,HX,1
                DO IY=1,HY,1
                  Call Cap_ASend( A(g1(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IP,IP+1,IX,..,IY),..,
                            gn(I1,I2,..,IA-1,IA,IA+1,IP-2,IP-1,IP,IP+1,IX,..,IY)),1,Right,Cap_S_Sync)
                ENDDO(IY)
              ENDDO(IX)
            ENDDO(IP-1)
          ENDDO(IP-2)
        ENDDO(IA+1)
      ENDDO(IA)
      Call Cap_Sync_Send(Right,Cap_S_Sync)
    ENDDO(IA-1)
    .
  ENDDO(I2)
ENDDO(I1)
```

Figure 5.39 : Formal model for an overlapped pipeline.

The first conservative test, is to ensure that there are no loop exits from the pipeline loops. This is achieved by checking that the block of statements inside the loop are postdominated by the loop head. For example, in Figure 5.40 the statement S_1 is postdominated by the loop heads I_4, I_3 and I_2. However, the statement S_1 is not postdominated by the loop head I_1 due to a loop exit within the loop. This loop is thus not a valid loop for incorporation of the overlapping pipeline and the OUTERLEVEL loop is designated as the previous valid loop I_2.

```
      DO I1=...
        DO I2=...              <---- OUTERLEVEL (A=2)
          DO I3=...
            DO I4=...                  <---- PIPELEVEL (P=4)
              {* Pipeline Calculation *}
S1            .....=.....
            ENDDO
          ENDDO
        ENDDO
        IF (true) THEN GOTO 20
      ENDDO
20    CONTINUE
```

Figure 5.40 : Fortran pseudo code illustrating loop exits.

Secondly, to check that the data used in the overlapped calculation is not the same as the communicated data. There is a danger of the CAP_RECEIVE communication overwriting data before actually being used in calculation. To avoid this eventuality it is essential to ensure that the loop does not have any anti dependencies between the communication and the calculation. For example, in Figure 5.41 there is an anti dependence between statement S_1 and S_2. In the statement S_1 the value of V(5,J+1,Cap_LV-1) is being communicated while in statement S_2 the value V(M,J+1,K-1) is being used in the calculation. There is a potential danger that the data being used could also be overwritten simultaneously by the communication. For this reason, the overlapping pipeline code shown in Figure 5.41 would never be generated by CAPTools.

```
      CALL CAP_ARECEIVE(V(1,2,Cap_LV+1),...)
      DO J=2,JEND                                        <---- OUTERLEVEL
            CALL CAP_SYNC_RECEIVE()
S1          IF(J+1.LE.JEND)CALL CAP_ARECEIVE(V(1,J+1,Cap_LV-1),...)
                  DO K= Max(1,Cap_LV),Min(10,Cap_HV)                    <---- PIPELEVEL
                        DO M=1,5
                              {* Pipeline Calculation *}
S2                            V(M,J,K) = V(M,J,K-1) + V(M,J+1,K-1)
                        ENDDO
                  ENDDO
                  CALL CAP_SYNC_SEND()
                  CALL CAP_ASEND(V(1,J,Cap_HV),...)
            ENDIF
      ENDDO
      CALL CAP_SYNC_SEND()
```

Figure 5.41 : Illegal Fortran pseudo code illustrating an anti-dependence.

Thirdly, to check that the data being assigned in the overlapped calculation is not the same as that being communicated. There is a danger of overwriting data before it is sent. To avoid this occurrence it is essential to ensure that there are no output dependencies between the send communication and the calculation.

These two last rules are very conservative and it could very well be possible that another technique may be used, i.e. such as placing the communicated data into a buffer array.

The conversion of the present synchronous communications involves the generation of new communication calls and conditional statements to ensure that the code achieves its maximum efficiency. Finally, it is necessary to generate the synchronisation statements at the correct positions, as designated in the model, to ensure that the correct data values are used in the pipeline calculation.

The generation of the overlapping communications consists of three stages:

1. The generation of the conditional statements and their related overlapping RECEIVE communications;
2. The generation of the very first overlapping RECEIVE communication of the pipeline and the RECEIVE synchronisation point;
3. The generation of the overlapping SEND communication and the two SEND synchronisation points.

The positions of these communications are summarised in the pseudo code in Figure 5.42.

```
{* First Overlapping RECEIVE *}
DO I1=...                                  <---- OUTERLEVEL
      DO I2=...
            DO I3=...
                  {* RECEIVE Synchronisation Point *}
                  {* Conditional Overlapping RECEIVES *}
                  DO I4=.....                          <---- PIPELEVEL
                        {* Calculation *}
                  ENDDO
                  {* SEND Synchronisation Point *}
                  {* Overlapping SEND *}
            ENDDO
      ENDDO
ENDDO
{* SEND Synchronisation Point *}
```

Figure 5.42 : Fortran pseudo code illustrating the positions of overlapping communications within a pipeline.

5.3.1 Generation of the Conditional Statements and their Related Overlapped RECEIVE Communications.

In the overlapping pipeline model (Figure 5.39) there exists a conditional statement for every loop from loop A to loop P-1 of the pipeline. This ensures that another iteration of a loop exists and that data for the next iteration is received into the correct data address. The very first conditional loop ensures that there is another iteration of the innermost loop prior to the PIPELEVEL loop, i.e. the I_{P-1} loop. If there is another iteration of that loop then the model will receive the data for the next iteration into the array address $I_{P-1}+1$ for the loop index I_{P-1}. All other loop iteration counters of the surrounding loops prior to I_{P-1} remain unchanged. If there is not another iteration of I_{P-1} to be received then the model will execute the next conditional statement, which ensures that there is another iteration of the loop I_{P-2}. If there is another iteration of that loop then the model receives the data for the next iteration into the array address $I_{P-2}+1$ for the loop index I_{P-2}. All subsequent loop iteration counters for the pipeline loops are reinitialised to 1, i.e. in this case the index I_{P-1} would be reinitialised to 1 since when I_{P-2} is increased to the next iteration then the iteration of I_{P-1} will then intuitively start from 1. All other array index addresses of the surrounding loops prior to the I_{P-2} remain unchanged. This continues for all the conditionals up to the loop I_A where the array address for the index I_A will be reset to I_A+1 and all subsequent pipeline loops array addresses reinitialised to 1.

From the general formal model it is necessary to create additional specifications for use in an algorithm for automatic generation of the conditional statements for the overlapping pipeline. These consist of the specification of the indices for the next iteration of a loop and also the indices of the first iteration of a loop. These may be defined as follows :

Indices for next iteration of a k loop is :

$$g_j(i_1,i_2,..,i_k+1,..,i_N) = g_{j,0} + g_{j,1}i_1 +..+ g_{j,k}(i_k+1) +..+ g_{j,N}i_N$$
$$= g_j(i_1,i_2,..,i_k,..,i_N) + g_{j,k}$$

where j = 1 (1) M

Indices for the first iteration of the kth loop :

$$g_j(i_1,i_2,..,1,..,i_N) = g_{j,0} + g_{j,1}i_1 + g_{j,2}i_2 +..+ g_{j,k} +..+ g_{j,N}i_N$$
$$= g(i_1,i_2,..,0,..,i_N) + g_{j,k}$$

From the general formal model specification it is possible to obtain an algorithm for the automatic generation of the conditional statements of the overlapping pipeline (Figure 5.43).

Let P be the Pipelevel
A be the Outerlevel
M be the number of array indices

```
for k = P-1 down to A
    {* Process loop k*}
    generate "IF (lk+1 .LE. Uk) THEN"
    generate "CALL CAP_ARECEIVE()"
    generate "converted array indices" as follows:
     copy original indices
     for j = 1 to M
          {* Process index j *}
          {* Increase the loop iteration counter lk *}
          add gj,k to constant term
          for r = k+1 to P-1
               {* set loop iteration counter lr to zero *}
               add gj,r to constant term
               set gj,r to zero
          endfor
     endfor
    generate "length" and "direction"
    generate "sync_variable"

    if (k > A) then
          generate "ELSE"
    else
          generate "ENDIF"
    endif
endfor
```

Figure 5.43 : Pseudo code for the automatic generation of the conditional statements of the overlapping pipeline.

The algorithm consists of a loop that will generate a conditional statement for each valid loop of the overlapped pipeline from the loop prior to PIPELEVEL (i.e. P-1) to the OUTERLEVEL (i.e. A). Each conditional statement generated for each loop k will all generate the following basic statement:

```
IF (Ik+1 .GE. Uk) THEN
    CALL CAP_ARECEIVE( )
```

After the statement has been generated the parameter list for the receive communication is required. The first parameter is the data address to receive the communicated data. This involves calculating the correct array indices to receive the communicated data into the correct data address. Initially, this involves copying the original array indices of the synchronous receive communication. These array indices may then be adjusted to ensure that they receive the data into the array address of the next iteration. This will involve for each array index g_j the adding of a constant term $g_{j,k}$ (where k is the present loop being processed and j the array index). The adding of this constant term will ensure that for the loop k that data will be received into the data

region of the next iteration of loop k. All array index addresses after the index k are then set to their lowest value 1. In order to do this it is essential for each loop index from k+1 to P-1 (referred to as r) to add $g_{j,r}$ to the constant term and reset $g_{j,r}$ to zero.

Consider the generation of the CAP_ARECEIVE statement for the third conditional statement of the general model of the overlapped pipeline (Figure 5.39). The original indices copied from the CAP_RECEIVE statement of the general model of the synchronous pipeline (Figure 5.38) were as follows :

$$A(g1(I_1,I_2,..,I_{A-1},I_A,\mathbf{I_{A+1},I_{P-2},I_{P-1}},I_P,I_{P+1},I_X,..,I_Y),..,$$
$$gn(I_1,I_2,..,I_{A-1},I_A,\mathbf{I_{A+1},I_{P-2},I_{P-1}},I_P,I_{P+1},I_X,..,I_Y))$$

The present loop k being processed for this conditional statement is the loop I_{A+1}. The constant term is added to give the address of the next iteration of I_{A+1}, to give the $I_{A+1}+1$. All the loop iteration counters after I_{A+1} index up to and including index I_{P-1} are then set to their lowest value 1. The indices for the conditional will then be as follows :

$$A(g_1(I_1,I_2,..,I_{A-1},I_A,\mathbf{I_{A+1}+1,1,1},I_P,I_{P+1},I_X,..,I_Y),..,$$
$$g_n(I_1,I_2,..,I_{A-1},I_A,\mathbf{I_{A+1}+1,1,1},I_P,I_{P+1},I_X,..,I_Y))$$

Once the correct array address of the next iteration has been generated the communication length, the direction of the communication and the "sync_variable" are appended to the communication parameters list.

Finally if it is not the first conditional statement then an ELSE should be generated before the IF statement. An ENDIF statement must also be generated after the final conditional statement.

5.3.2 Generation of the First Overlapped RECEIVE Communication of the Pipeline and the RECEIVE Synchronisation Point.

This overlapping RECEIVE communication will be placed before the OUTERLEVEL loop. If the data being communicated is an array then the indices of this array, which also correspond to the valid loops of the pipeline, will be adjusted such that each index will be equal to the value of the first iteration of their corresponding loop. This is applied to all indices of the array that has a corresponding loop from OUTERLEVEL to the loop previous to the PIPELEVEL loop. The PIPELEVEL loop is not incorporated since it is the first loop of the calculation, i.e. the partitioned loop.

The adjustment of all the loop iteration counters to the value of the first iteration of each corresponding loop, may be applied using the previous specification for calculating the indices for the first iteration of a loop k. This simple specification is then applied for each array index j from 1 to M, for each loop index from the OUTERLEVEL loop (A) to the loop prior to the PIPELEVEL loop (P-1). The algorithm for this is shown in Figure 5.44.

```
generate "CALL CAP_ARECEIVE()"
generate "array indices"
 copy original indices
 for j = 1 to M
      for r = A to P-1
            add g_j,r to constant term
            set g_j,r to zero
      endfor
 endfor
generate "length" and "direction"
generate "sync_variable"
```

Figure 5.44 : Pseudo code for the adjustment of the loop iteration counters for the First Overlapped receive in the pipeline.

It will also be required to generate the synchronisation point for the overlapping RECEIVE communications. This synchronisation point must be placed prior to the conditional overlapping RECEIVE communications which are generated before the PIPELEVEL loop.

5.3.3 The Generation of the Overlapping SEND Communication and the SEND Synchronisation Points.

Now that the overlapping RECEIVE communications and synchronisation points have all been generated the overlapping SEND communication and related synchronisation points may be generated. The generation of the overlapping SEND is a straight forward operation which merely involves changing the name of the synchronous SEND call and the addition of the "sync_variable" to the parameters list. The generation of a synchronisation point is needed before the overlapping SEND communication, to ensure that the previous call to SEND has completed. There is also a need for an additional synchronisation point outside of the end of the OUTERLEVEL loop. This synchronisation point is necessary for the last iteration of the pipeline and is required for consistency to avoid potential overwriting of data still being communicated by the SEND.

5.4 Conclusions

The methods applied by hand in Chapter 4 were incorporated as an additional stage within CAPTools. This allowed the overlapped communications to be automatically generated at a fraction of the time it would take by hand. The transformed code generated was correct and tested on several small test cases before being tested on real codes. The results for these codes are provided in the next chapter.

Chapter 6

6 Results for Automatic Code Generation of Overlapping Communications for Structured Mesh Computational Mechanics Codes.

6.1 Introduction.

The automatic code generation methods applied within CAPTools (Chapter 5) were tested on several codes. These codes comprised of the four codes that were parallelised by hand in Chapter 3 and three additional codes. Two further codes from the NASPAR [68] benchmark suite of codes : APPSP and APPBT and a real world CFD code from a major industrial company was also processed.

The asynchronous codes automatically generated form CAPTools were tested on two parallel machines. These two machines used were the Transtech Paramid [81] and the Parsys SN9500 [82] whose architecture was briefly discussed in Section 4.3.

6.2 2-D Heat Diffusion Code (FAB).

The FAB code (described in Section 3.2) when parallelised using CAPTools generated 13 synchronous communications, 7 of which were exchange communications. There were no pipelines in this code. The results for this code utilising synchronous communications are shown in Table 6.1. The results obtained for synchronous communications are very favourable, with a speed up of 7.33 obtainable on 8 processors on the Transtech Paramid. This provided a very healthy efficiency of 91.7%. These results were good due to the small number of communications and because there were no pipelines to reduce the overall efficiencies.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	127.97	-	-	127.97	-	-
2	65.28	1.96	98.0%	64.74	1.98	98.8%
4	33.35	3.84	95.9%	32.54	3.93	98.3%
6	22.83	5.60	93.4%	21.98	5.82	97.0%
8	17.45	7.33	91.7%	16.63	7.69	96.2%

Table 6.1 : Results for FAB with synchronous and overlapped communications for the Transtech Paramid.

After applying the automatic generation of overlapped communications, five of the original seven synchronous communications were overlapped. The SIMPLE method of overlapping was applied to 2 communications; PARTIAL overlapping to 1 communication; and UNROLL to the remaining 2 communications. Two of the original synchronous exchange communications were not overlapped. Figure 6.1 shows a fragment of the code in the main routine FAB where these two communications exist. For the communication Comm1 the data being communicated has two sinks commands one of which is Sink1. The other sink for this communication is in another routine that has sufficient amount of code to overlap the communication. Unfortunately, Sink1 of this communication has insufficient amount of code to allow a SIMPLE overlap due to the lack of time consumers (Section 5.2.2). The generation of a PARTIAL or UNROLL communication is not possible since there are no loops surrounding the sink command Sink1. The same is also true for the communication Comm2 and its sink command Sink2.

The application of the overlapped communications provided an additional increase in the speed up of the FAB code (Table 6.1). The speed up for 8 processors has now increased from 7.33 to 7.69 allowing an increasing efficiency from 91.7% to 96.2%. The graph in Figure 6.2 shows that even though the time saved is minimal the improvement in the speed up (Figure 6.3) was significant.

```
Comm1           CALL CAP_EXCHANGE(R(CAP_BHTNEW+1),R(CAP_BLTNEW),1,2,CAP_RIGHT)
Comm2           CALL CAP_EXCHANGE(R(CAP_BLTNEW-1),R(CAP_BHTNEW),1,2,CAP_LEFT)
                CALL CAP_AEXCHANGE(SK(2,CAP_BHTNEW+1),SK(2,CAP_BLTNEW),IN-2,2,
              &         CAP_RIGHT,CAP_SE_SYNC_1,CAP_RE_SYNC_1)
 100            CONTINUE
 200            CONTINUE
C
C               CREATE INPUT DATA FILE
C
                REWIND(10)
                CON2=CON1
                WRITE(UNIT=10,FMT=*)TIME,TL,DT,PMON
                WRITE(UNIT=10,FMT=*)GMOPT,RIN
                WRITE(UNIT=10,FMT=*)IN,JN,IMON,JMON,HGT,WTH
                WRITE(UNIT=10,FMT=*)FACX,FACY
                WRITE(UNIT=10,FMT=*)T0
                WRITE(UNIT=10,FMT=*)CON2,PRIN
                WRITE(UNIT=10,FMT=*)HCF(0),HCF(1),HCF(2),HCF(3)
                WRITE(UNIT=10,FMT=*)TMBY(0),TMBY(1),TMBY(2),TMBY(3)
                WRITE(UNIT=10,FMT=*)KO
                WRITE(UNIT=10,FMT=*)RHO,CP,Q0
C
C               PRINT INITIAL FIELDS.
C
                IF ((2.LE.CAP_BHTNEW).AND.(2.GE.CAP_BLTNEW)) THEN
Sink2               RF=(R(1)+R(2))*0.5
                    CALL CAP_SEND(RF,1,2,CAP_RIGHT)
                ENDIF
                CALL CAP_RECEIVE(RF,1,2,CAP_LEFT)
                CALL CAP_SEND(RF,1,2,CAP_RIGHT)
                IF ((JN-1.LE.CAP_BHTNEW).AND.(JN-1.GE.CAP_BLTNEW)) THEN
Sink1               RL=(R(JN)+R(JN-1))*0.5
                    CALL CAP_SEND(RL,1,2,CAP_LEFT)
                ENDIF
                CALL CAP_RECEIVE(RL,1,2,CAP_RIGHT)
                CALL CAP_SEND(RL,1,2,CAP_LEFT)
```

Figure 6.1 : Code from FAB showing the two CAP_EXCHANGE communications that have not been overlapped.

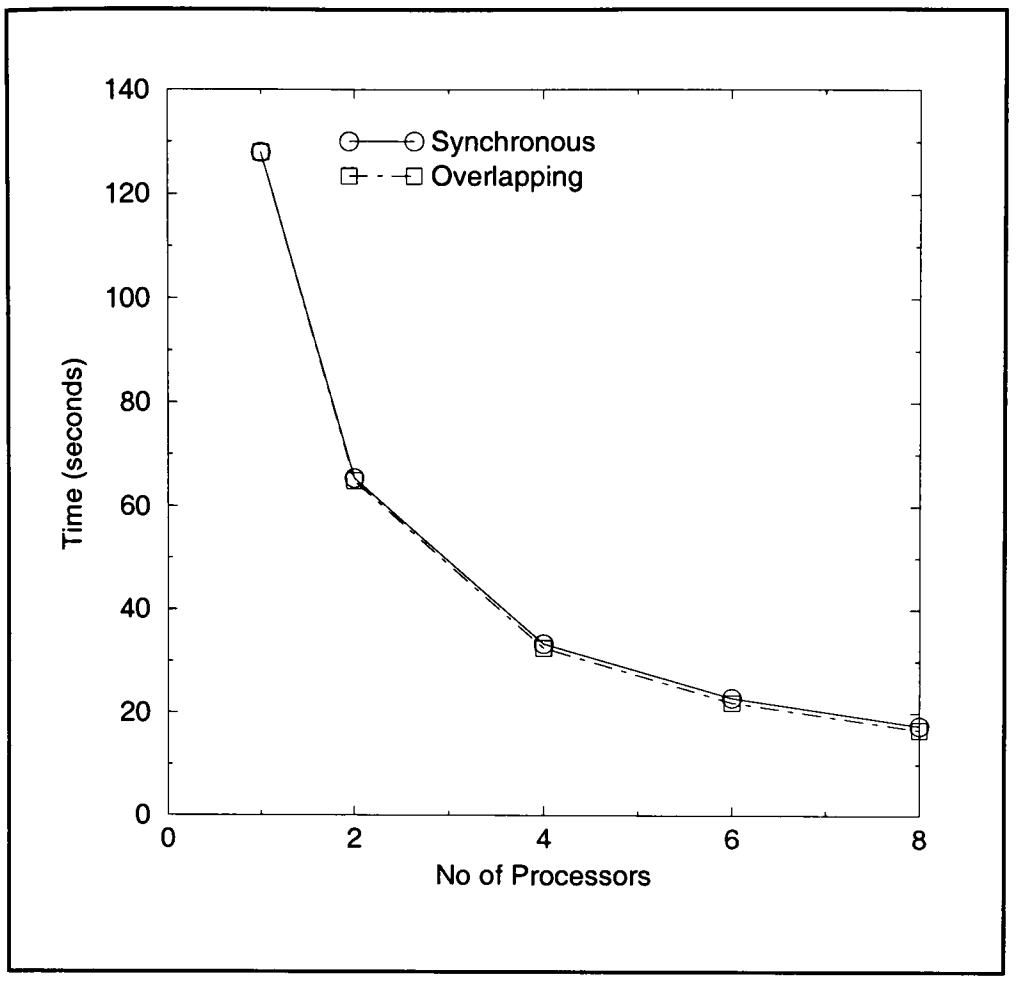


Figure 6.2 : Time graph of FAB on the Transtech Paramid.

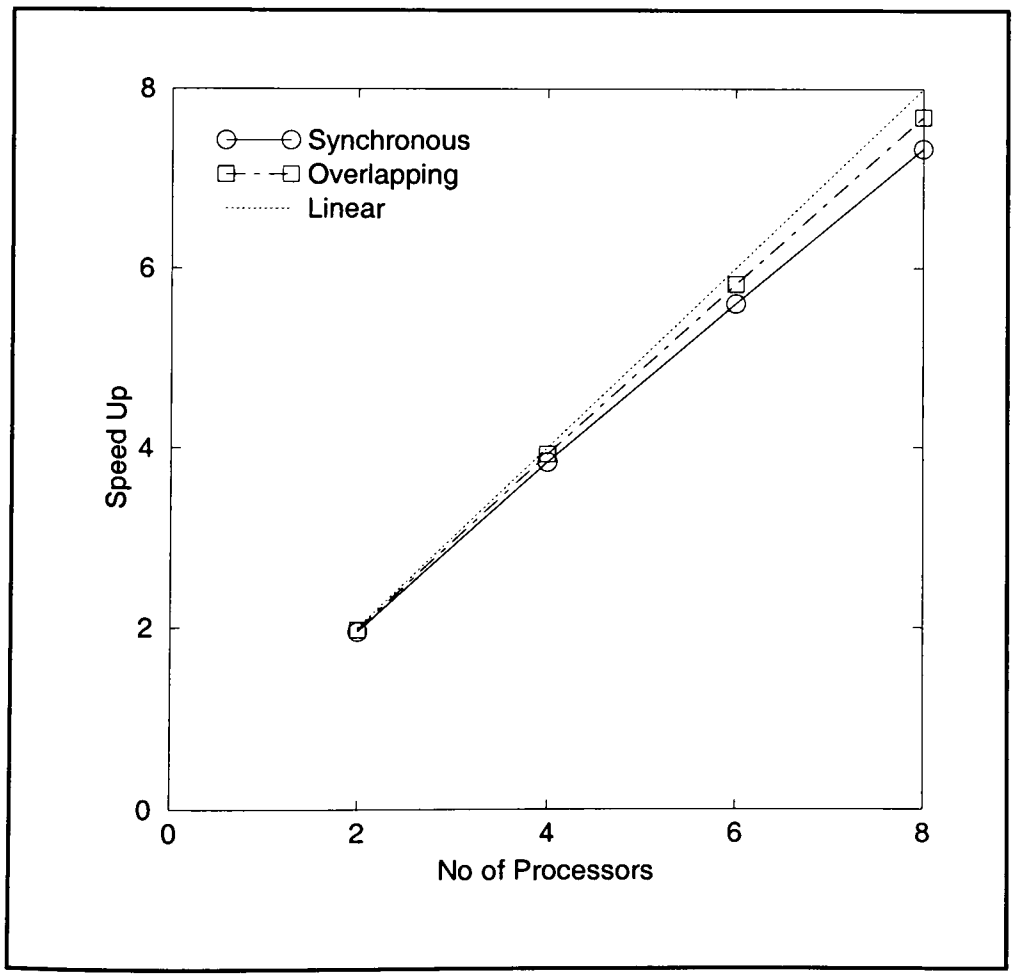


Figure 6.3 : Speed up graph of FAB on the Transtech Paramid.

6.3 TeamKE1.

The TeamKE1 [67] code (described in Section 3.3) consisted of 115 synchronous communications in the parallel code generated by CAPTools. These comprised of 62 exchange and 8 pipeline communications, the remainder being other communications.

The results for the parallel code with synchronous communications (Table 6.2) provide a speed up of 12.24 on 16 processors.

All of the 62 exchange communications were overlapped : 53 SIMPLE, 6 PARTIAL and 3 UNROLL communications. To allow the pipeline communications to be overlapped required the loop split and the scalar expansion mentioned in Section 3.3 for both the synchronous and overlapped codes.

The data communicated in the two single pipelines surrounding the DO 101 were also buffered and the pipeline communications in the DO 1000 loop were replaced by a CAP_EXCHANGE communication as mentioned in Section 3.3. This method was applied to both the synchronous and overlapped codes.

The results in Table 6.2, Figure 6.4 and Figure 6.5 show that the application of the overlapped communications have improved the speed up of the code providing a speed up of 12.69 on 16 processors.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	513.41	-	-	513.41	-	-
2	267.23	1.92	96.1%	268.37	1.91	95.6%
4	139.18	3.69	92.2%	138.4	3.71	92.7%
8	72.86	7.05	88.1%	72.23	7.11	88.8%
16	41.96	12.24	76.5%	40.45	12.69	79.3%

Table 6.2 : Results for TeamKE1 with synchronous and overlapped communications for the Transtech Paramid.

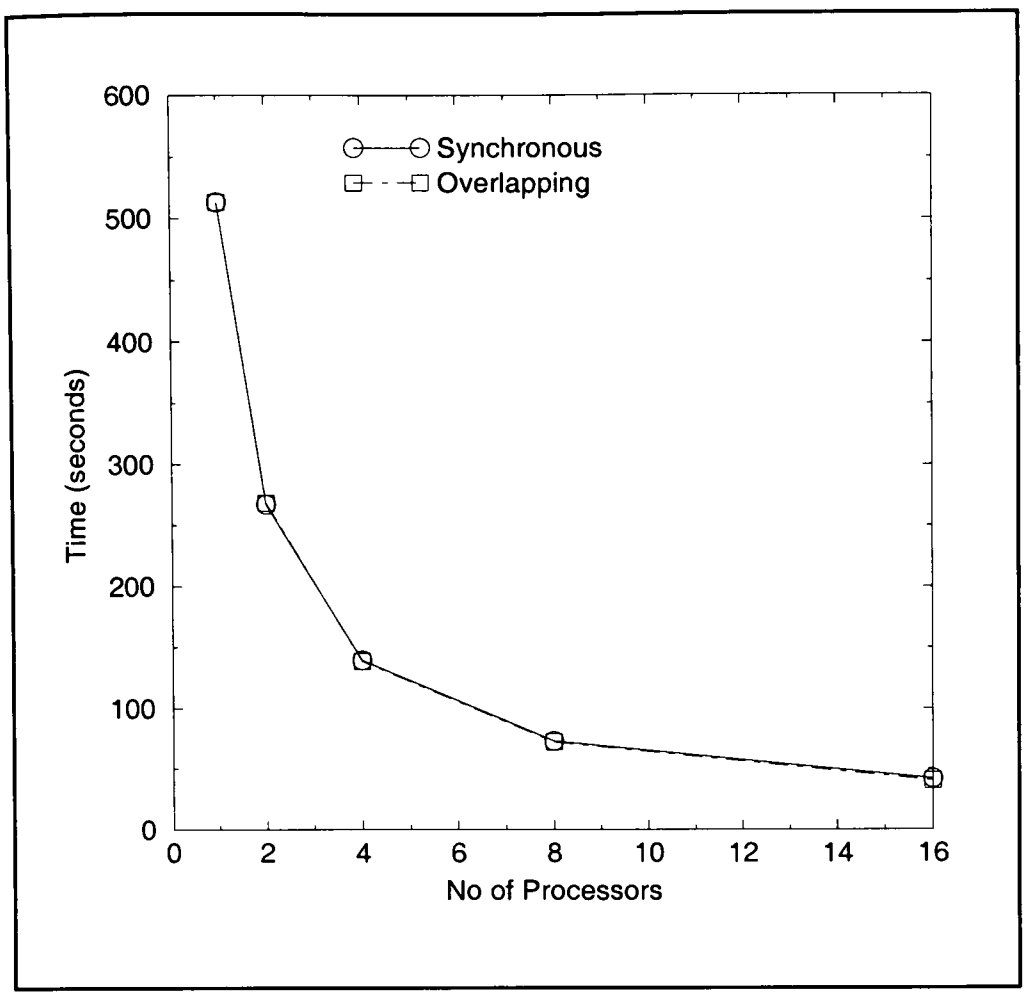


Figure 6.4 : Time graph of TEAMKE1 for the Transtech Paramid.

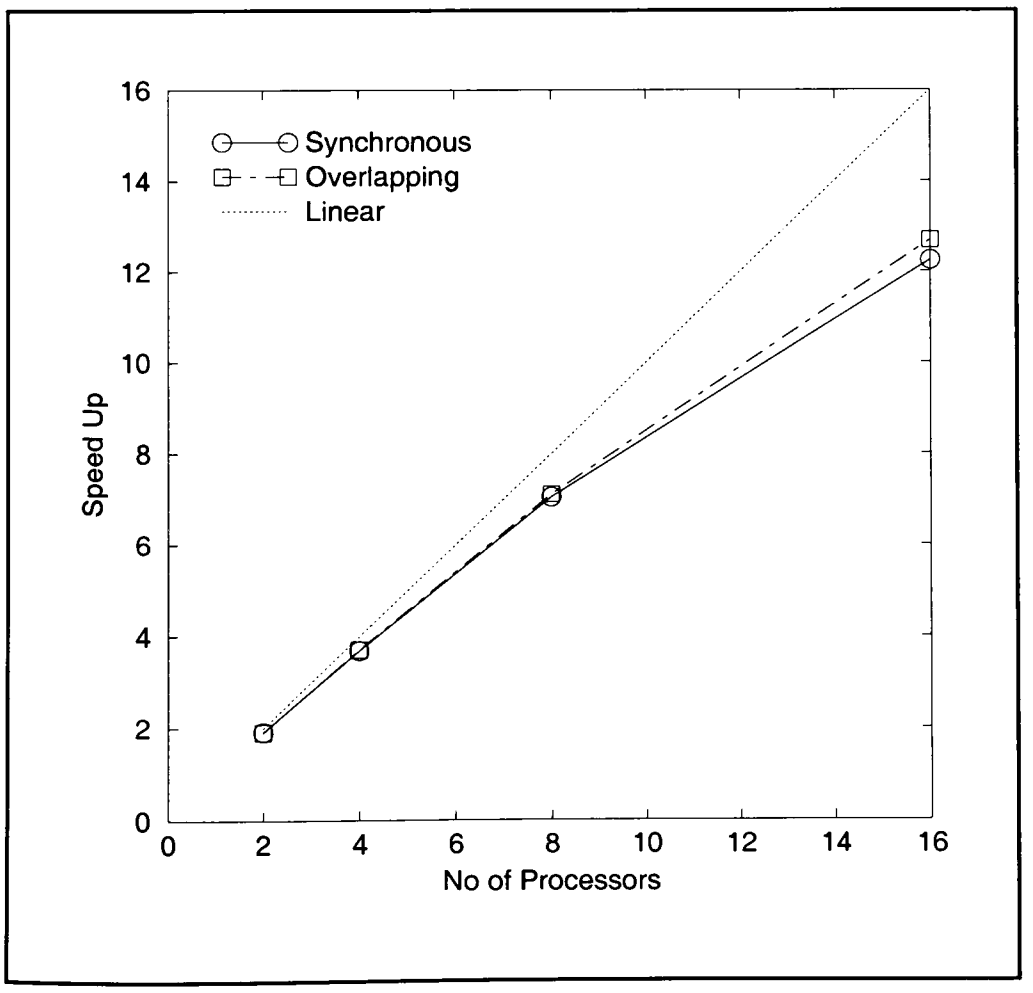


Figure 6.5 : Speed up graph of TEAMKE1 for the Transtech Paramid.

6.4 APPLU.

The APPLU [68] code (Section 3.4) with synchronous communications generated by CAPTools, consisted of 27 communications in total. Ten of these were exchange communications. The subroutines BUTS and BLTS also each contained a pipeline communication.

The synchronous results for the 32x32x32 problem on the Transtech Paramid (Figure 6.6) showed that the time taken was reduced from 343.94 seconds in serial to 53.67 seconds on 16 processors. This provided a speed up of only 6.41 on 16 processors (Figure 6.7) which was equivalent to only 40% efficiency (Table 6.3). The main reasons for this decrease in speed up as the number of processors increase is due to two factors. The communication of large amounts of data between processors using the CAP_EXCHANGE synchronous communication and the presence of two pipeline routines within the code.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	343.94	-	-	343.94	-	-
2	195.63	1.76	87.9%	184.36	1.86	93.3%
4	115.24	2.98	74.6%	97.09	3.54	88.6%
8	74.27	4.63	57.9%	54.04	6.36	79.6%
12	63.50	5.42	45.1%	43.35	7.95	66.2%
16	53.67	6.41	40.0%	34.40	10.0	62.5%

Table 6.3 : Results for the APPLU with synchronous and overlapped communications for the Transtech Paramid. (32x32x32 (line pipelines) problem)

The synchronous communication parallel code was processed through CAPTools and all the CAP_EXCHANGE communications were automatically converted to the CAP_AEXCHANGE overlapping communications. SIMPLE overlapping was applied to all of the CAP_EXCHANGE calls in the main iterative routine SSOR, all of which consisted of exchanging 5 whole slabs of data. Several other CAP_EXCHANGE communications, outside of the main iterative scheme, for broadcasting the initial values and the final results to all

processors were also overlapped successfully. All of the CAP_AEXCHANGE routines generated were found to have overlapped with sufficient amount of code.

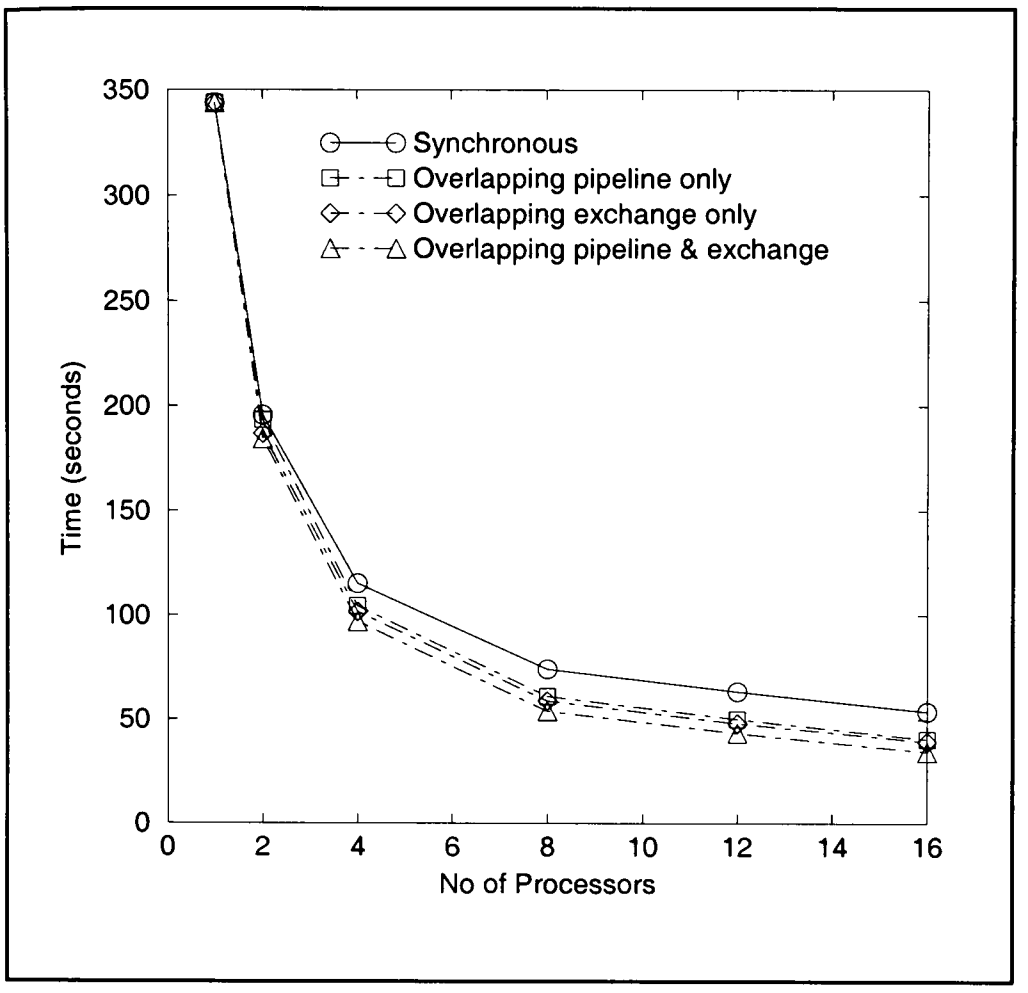


Figure 6.6 : Time graph for a 32x32x32 problem on the Transtech Paramid.

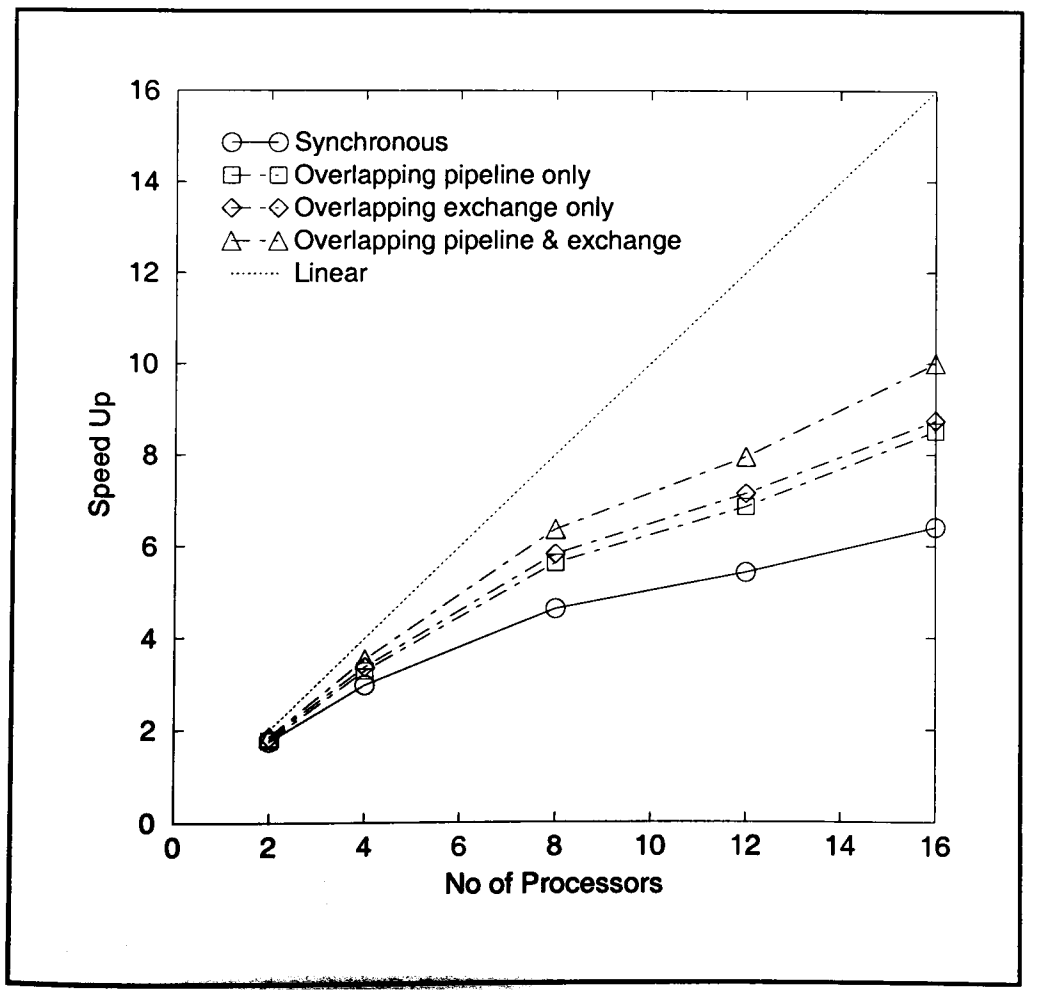


Figure 6.7 : Speed up graph of APPLU for a 32x32x32 problem on the Transtech Paramid.

The two subroutines BUTS and BLTS containing the pipeline algorithms were also automatically converted to contain overlapping communications. The overlapping pipelines generated were correct and there was a sufficient amount of calculation within the pipeline loop to allow the communication to be fully overlapped.

The effect of overlapping the exchange communications and pipelines is shown in Figure 6.6. Both show significant improvement in performance.

A favourable speed up of 10 was acquired on 16 processors (Figure 6.7) when the overlapping communications were applied. The results clearly demonstrate that efficiencies increase dramatically (Table 6.3) when applying overlapping communication. For more than 4 processors the time taken decreased by approximately 20 seconds and thus efficiency increased by approximately 20%.

The time saved by the exchanges on the Transtech Paramid was found to be constant as the number of processors increased. This is due to the nature of the CAP_AEXCHANGE and CAP_EXCHANGE routines. The CAP_EXCHANGE consists of exchanging data between two processors and this will occur concurrently between every other pair of processors. Thus, as the number of processors increases the number of communications per processor involved stays constant. Thus the speed up increases as the number of processors increase. The time saved by the pipelines was also found to be constant as the number of processors increase because the communication is overlapped completely with the calculation.

The results for the Parsys machine are for a smaller problem size of 24x24x24 since this is the maximum problem size that could be executed in serial on the processors available. The T9000 transputers process at a much slower rate hence the serial time of 1593.83 seconds as shown in Figure 6.8. The time of 314.28 seconds for 6 processors provided a healthy speed up of 5.07 (Figure 6.9) and an efficiency of 88.6% (Table 6.4). The efficiencies decrease less rapidly than for the 32x32x32 on the Transtech Paramid. Of course, if the smaller problem were run on the Transtech Paramid its parallel efficiencies would decrease even more rapidly. This does not occur with the Parsys machine since it has a low communication start-up latency.

The results for the Parsys SN9500 (Table 6.4) show a small increase to the efficiency of the parallel code (with point pipelines) when overlapped communications are applied. This is mostly due to the communications in the POINT pipeline. The small amount of communication start up latency and the small amount of data being communicated provides an insignificant amount of time to hide.

No. of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	1595.84	-	-	1595.84	-	-
2	874.62	1.82	91.2%	875.04	1.83	91.2%
3	588.08	2.71	90.4%	587.30	2.72	90.6%
4	444.41	3.59	89.8%	442.66	3.60	90.1%
5	372.17	4.29	85.8%	369.94	4.31	86.3%
6	300.027	5.32	88.6%	297.84	5.36	89.3%

Table 6.4 : Results for APPLU with synchronous and overlapped communications for the Parsys SN9500. (24x24x24 (point pipeline) problem)

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	1593.84	-	-	1593.84	-	-
2	872.27	1.83	91.4%	872.95	1.83	91.3%
3	593.74	2.68	89.5%	592.15	2.69	89.7%
4	454.21	3.51	87.7%	450.53	3.54	88.4%
5	381.41	4.18	83.6%	377.07	4.23	84.5%
6	314.28	5.07	84.5%	311.42	5.12	85.3%

Table 6.5 : Results for APPLU with synchronous and overlapped communications for the Parsys SN9500. (24x24x24 (line pipeline) problem)

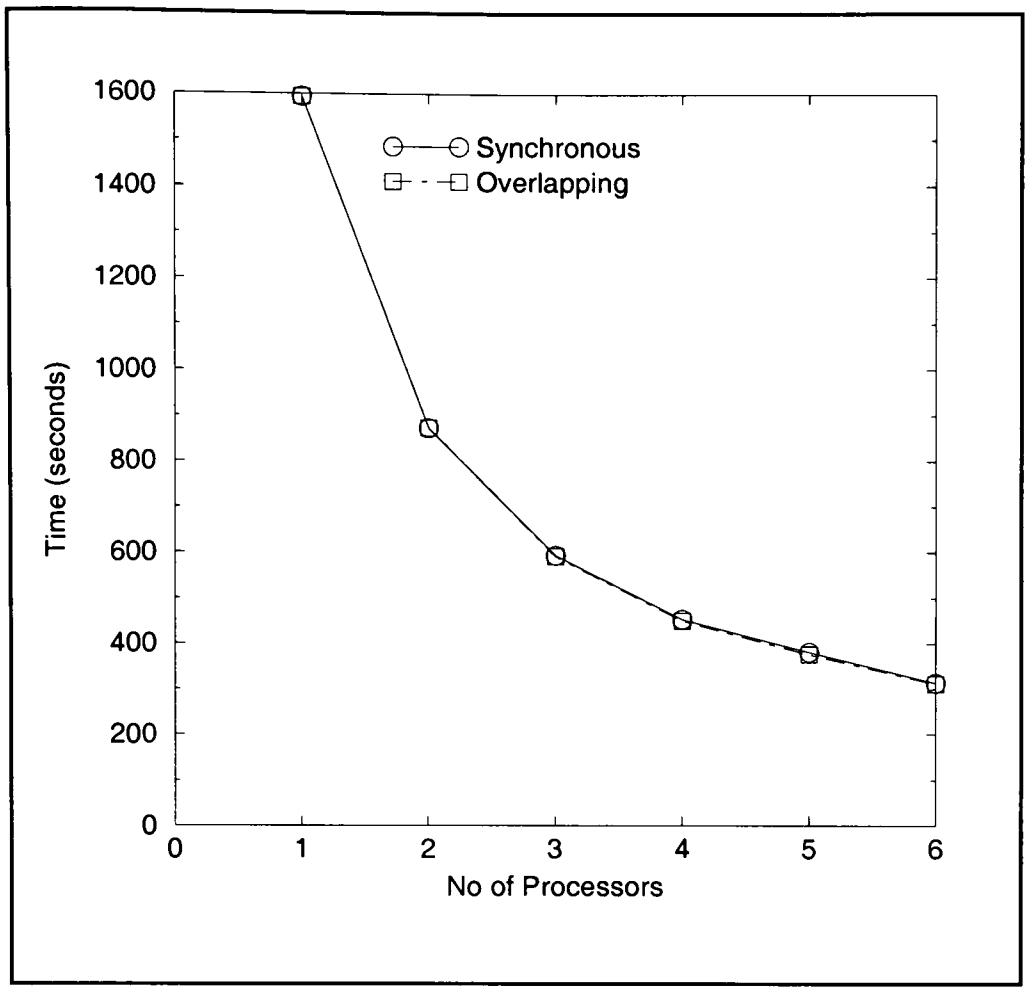


Figure 6.8 : Time graph of APPLU for a 24x24x24 problem on the Parsys SN9500.

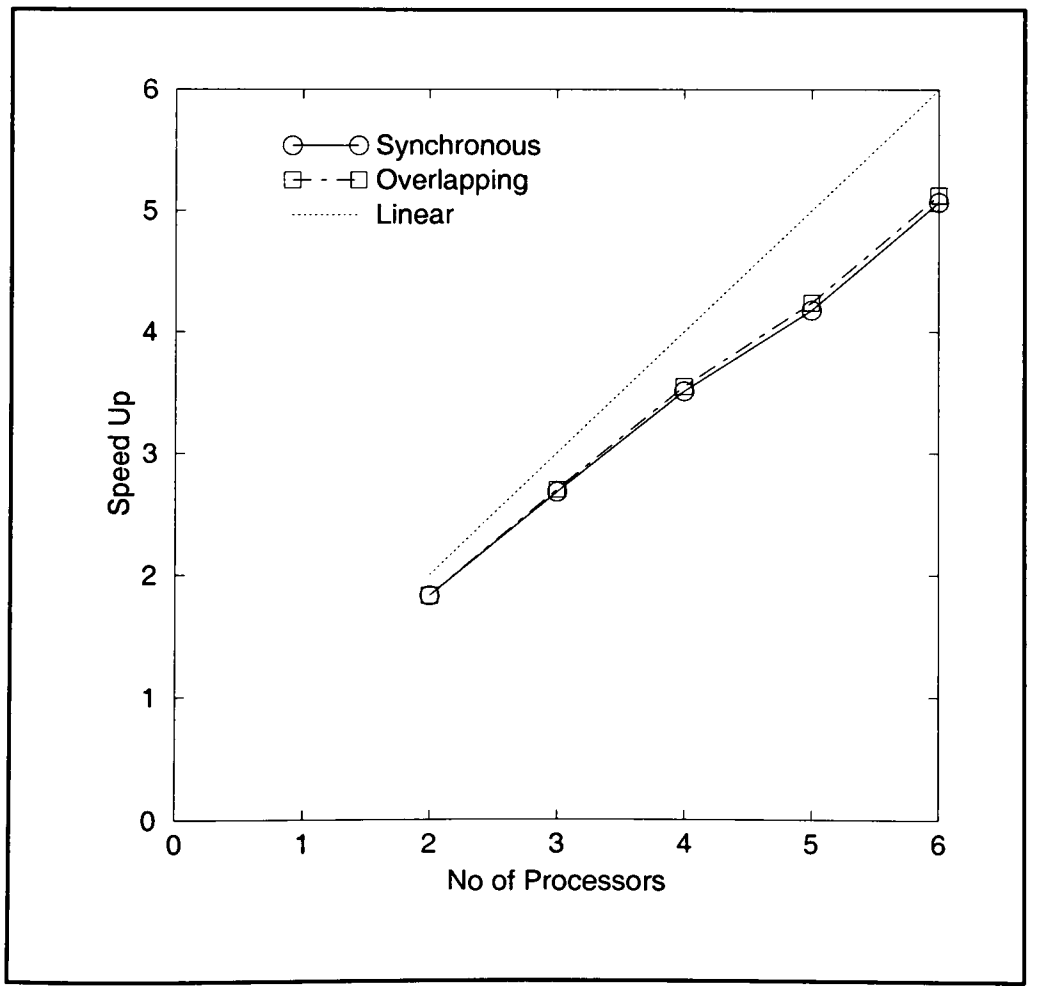


Figure 6.9 : Speed up graph of APPLU for a 24x24x24 problem on the Parsys SN9500.

The results for the APPLU code with LINE pipelines (Table 6.5) provide a slightly greater increase with the application of overlapping communications in comparison with the use of the POINT pipeline (Table 6.4). The amount of data being communicated in the LINE pipeline is greater than that of the POINT pipeline. This provides more communication time to be overlapped with the calculation time providing a better efficiency.

For the 24x24x24 problem on the Parsys SN9500 when overlapping communications is applied the decrease in time is marginal (Figure 6.8). There is therefore only a slight improvement in the speed up (Figure 6.9) when overlapping communications applied. The improvement in efficiency (Table 6.4 and Table 6.5) is not as dramatic as for the Transtech Paramid since the Parsys SN9500 already produced high efficiency results using synchronous communications.

6.5 ARC3D.

The ARC3D [70] code (Section 3.5) with synchronous communication generated by CAPTools consisted of 134 communications, 44 of which were exchange communication calls and there were also two routines containing pipelines. Both of these routines contained two sets of pipelines each. These pipelines consisted of very little calculation by comparison with the amount of data communicated.

For the Transtech Paramid on a problem size of 40x33x40 the time taken on 8 processors was 398.084 seconds in relation to 1373.949 seconds in serial (Figure 6.10). This produced a speed up of 3.45 on 8 processors (Figure 6.11) equivalent to an efficiency of 43.1% (Table 6.6). The effect of the pipelines was removed since it was noticed that the time of communication within the pipelines was much higher than the time of calculation. This produced an improved speed up of 4.61 (Figure 6.11) on 8 processors and an efficiency of 57.6% (Table 6.6).

For the Parsys SN9500 on a smaller size problem of 40x23x30 the time taken on 6 processors was 951.748 seconds as opposed to 4662.731 seconds in serial (Figure 6.12). This produced a speed up of 4.90 on 6 processors (Figure 6.13) equivalent to an efficiency of 81.6% (Table 6.7).

All the CAP_EXCHANGE communication calls within the code were automatically converted to overlapping communications within CAPTools. They mostly consisted of SIMPLE overlapping with unrelated code. There was also two cases of PARTIAL overlapping and three

cases of UNROLL overlapping. These occurred in routines VISRHS and MUTUR. All the SIMPLE overlapping communications and their synchronisation calls were separated by sufficient code to provide adequate overlapping.

The pipelines in routines VPENTA and VPENTA3 were also automatically converted to contain overlapping communications. The buffering of the pipelines in VPENTA3 was also conducted, as mentioned previously.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	1373.95	-	-	1373.95	-	-
2	830.71	1.65	82.7%	782.63	1.76	87.8%
4	559.99	2.45	61.3%	453.80	3.03	75.7%
6	465.85	2.95	49.1%	354.06	3.88	64.7%
8	398.08	3.45	43.1%	294.06	4.67	58.4%

Table 6.6 : Results for Arc3D with synchronous and overlapped communications for the Transtech Paramid.(40x33x40)

The results for the overlapping communication (Figure 6.10 and Figure 6.11) for the Transtech Paramid decreased the time taken on 8 processors to 294.060 seconds and increased the speed up to 4.67. This produced an efficiency of 58.4% on 8 processors (Table 6.6). The speed up for both the synchronous and overlapping communication are very poor. The predominant factors causing this decrease in efficiency were the pipelines in routines VPENTA and VPENTA3. Investigation of the time taken by both the synchronous and overlapping pipelines revealed that they were slowing down. This was due to the small amount of calculation involved in comparison with the amount of communication. The application of overlapping communication to the pipeline did cause a small increase in their efficiency, but could not eliminate the essential problem.

The speed up graph (Figure 6.11) shows that if the effect of the pipelines were removed for both the synchronous and overlapping then much better results are obtained. If the effect of the pipeline is ignored a efficiency of 57.6% (Table 6.8) is obtained with synchronous communications which will increase to 82.2% using overlapping communications. This

provides sufficient evidence that the exchange communications have been overlapped successfully and that the pipelines are the major cause of poor speed up and reducing the maximum possible efficiencies that could be obtained.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	4662.73	-	-	4662.73	-	-
2	2405.99	1.94	96.9%	2424.30	1.92	96.2%
3	1764.97	2.64	88.1%	1737.38	2.68	89.5%
4	1439.34	3.24	81.0%	1413.68	3.30	82.5%
5	1114.26	4.18	83.7%	1090.63	4.27	85.5%
6	951.75	4.90	81.6%	930.86	5.01	83.5%

Table 6.7 : Results for Arc3D with synchronous and overlapped communications for the Parsys SN9500.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	1320.66	-	-	1320.66	-	-
2	724.47	1.82	91.1%	687.04	1.92	96.1%
4	451.11	2.93	73.2%	368.43	3.58	89.6%
6	355.38	3.72	61.9%	265.22	4.98	83.0%
8	286	4.61	57.6%	200.86	6.58	82.2%

Table 6.8 : Results for Arc3D with synchronous and overlapped communications for the Transtech Paramid. (40x33x40 - pipeline effect removed)

The results (Figure 6.12 and Figure 6.13) for the Parsys SN9500 for the smaller size problem of 40x23x30 also showed an improvement in efficiencies (Table 6.7) when overlapping communication was applied. The improvement is very slight due to the nature of the Parsys SN9500 system.

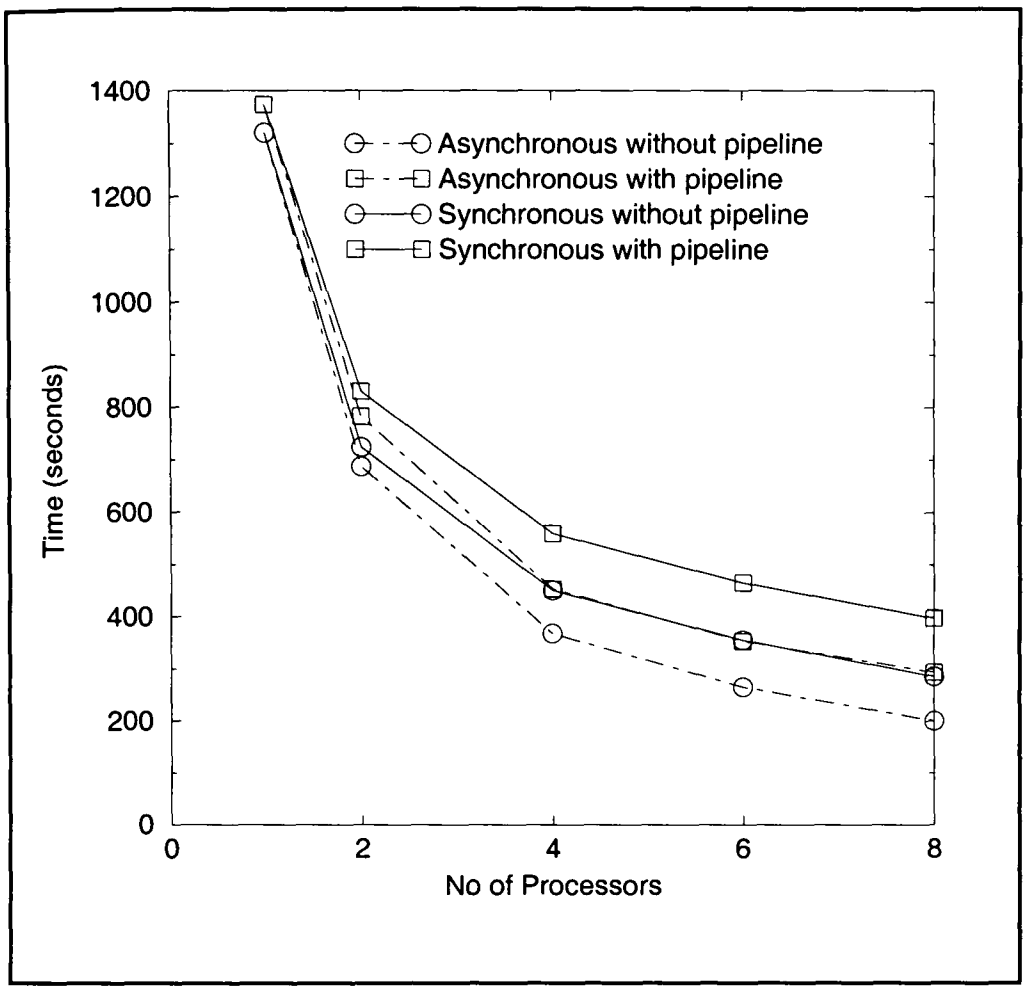


Figure 6.10 : Time graph of ARC3D for a 40x33x40 problem on the Transtech Paramid.

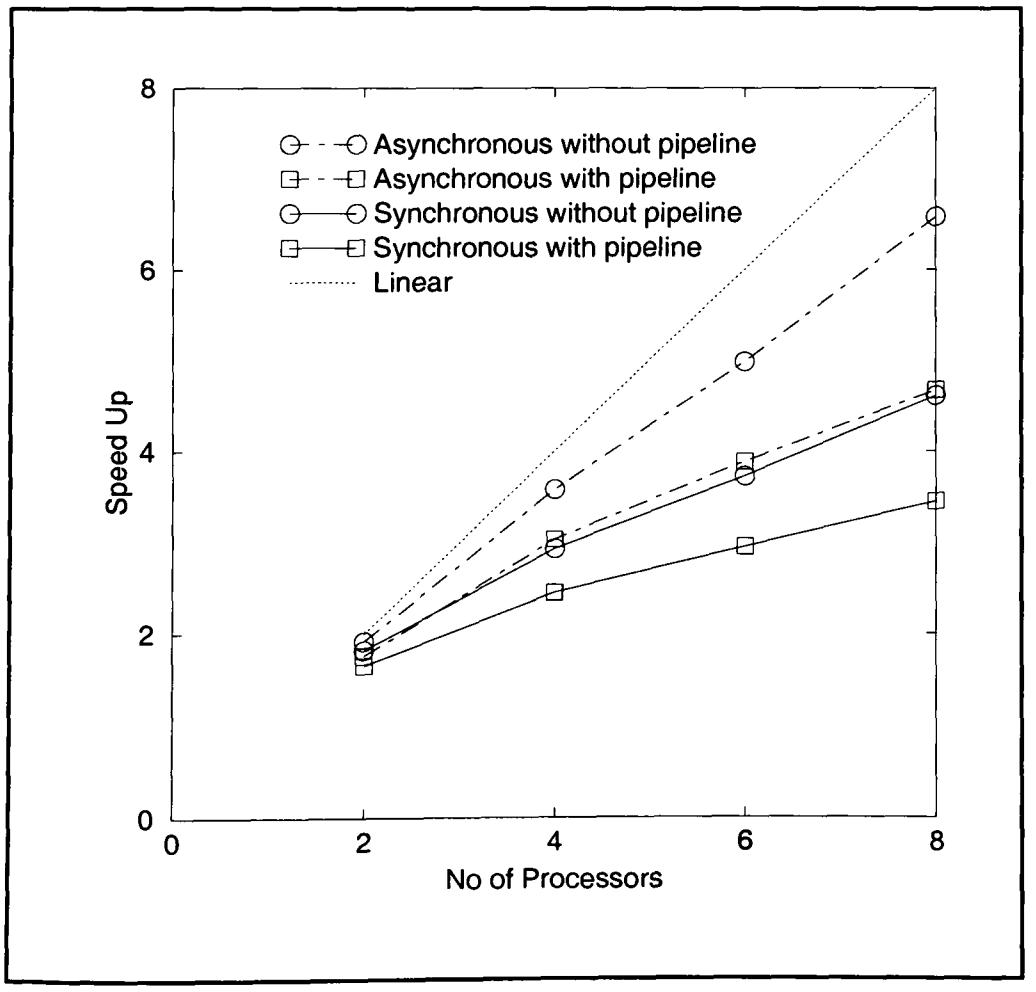


Figure 6.11 : Speed up graph of ARC3D for a 40x33x40 problem on the Transtech Paramid.

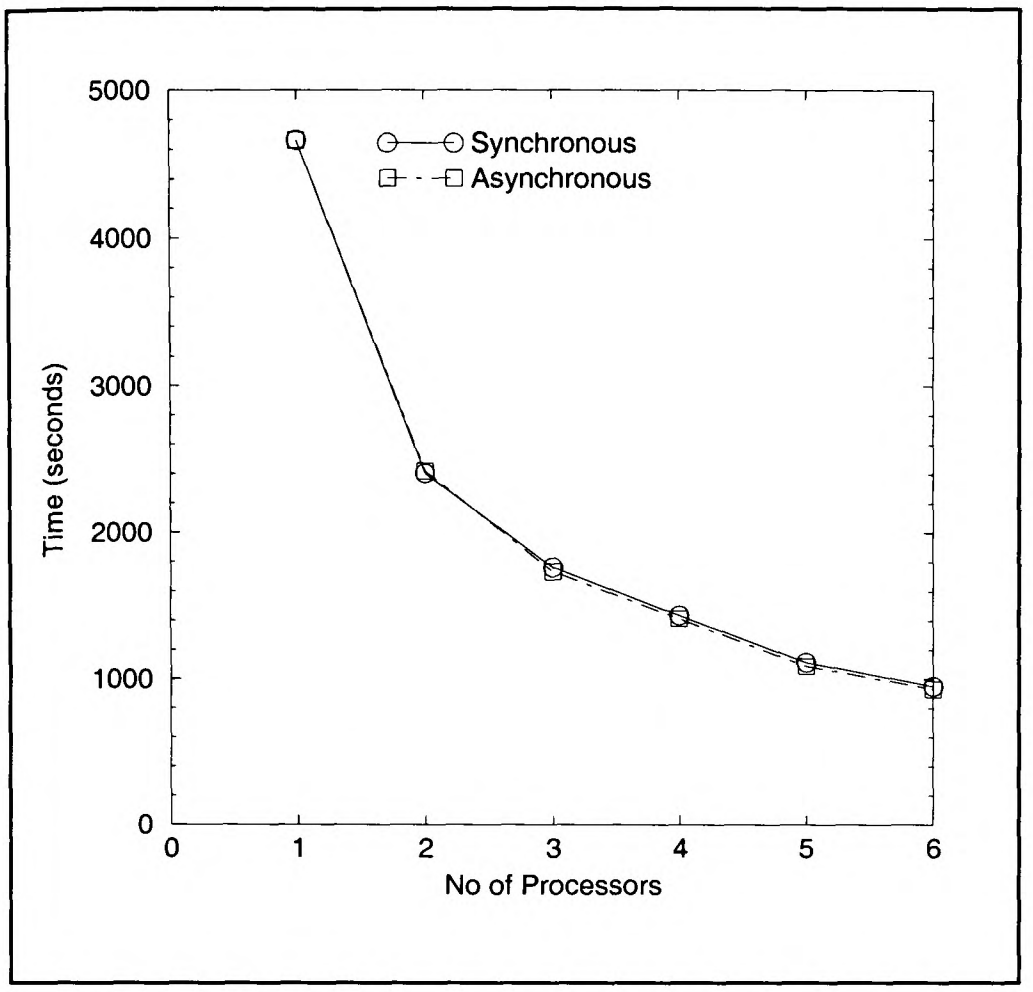


Figure 6.12 : Time graph of ARC3D for a 40x33x40 problem on the Parsys SN9500.

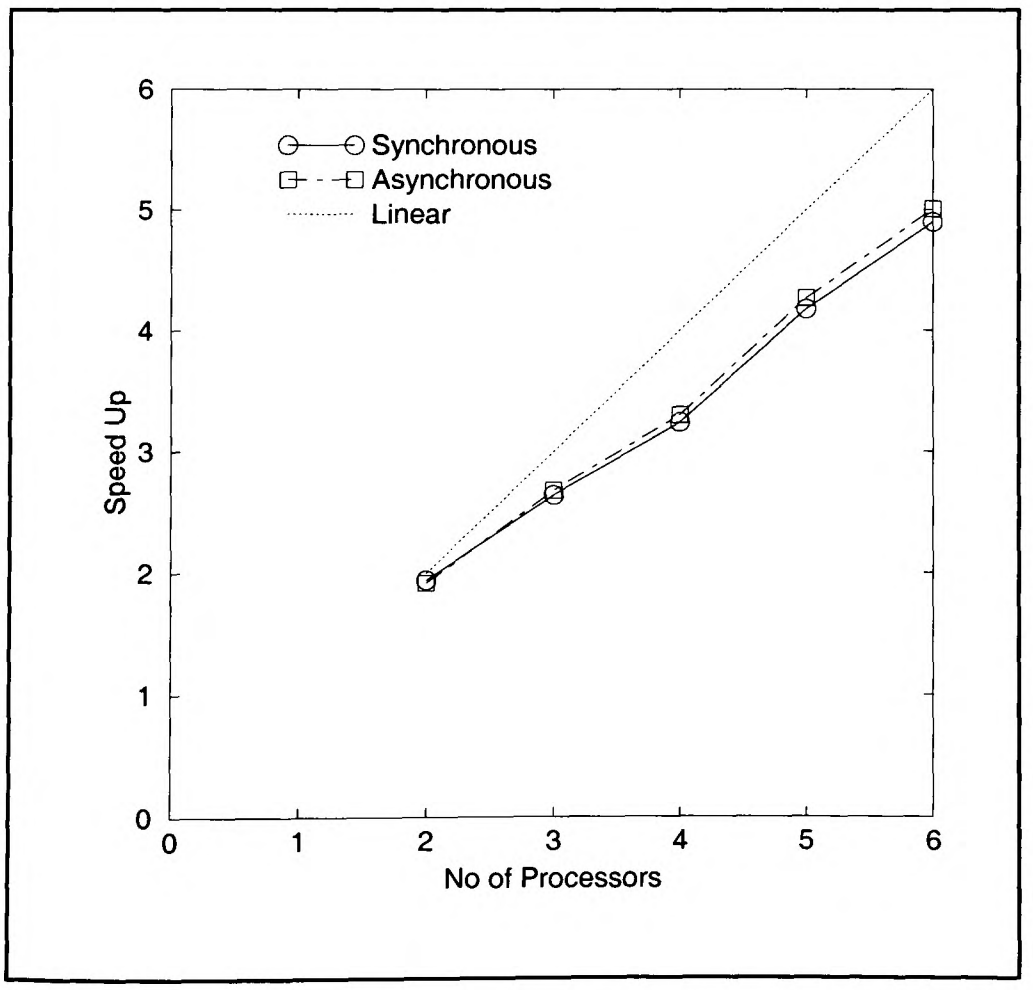


Figure 6.13 : Speed up graph of ARC3D for a 40x33x40 problem on the Parsys SN9500.

6.6 APPSP.

APPSP is also a code from the NAS-PAR benchmark suite [68]. The code uses an implicit algorithm to compute a finite difference solution to the 3D compressible Navier-Stokes equations. The solution is based on a Beam-Warming approximate factorisation. The approximate factorisation decouples the three dimensions leading to three sets of regularly structured systems of linear equations. The resulting equations are scalar pentadiagonal which are solved using the Thomas algorithm (Gaussian elimination) without pivoting of a banded system. The code is approximately 3500 lines of Fortran 77 code.

The synchronous code generated consisted of 50 communications of which 13 were exchange communications and 8 were pipeline communications. There were four pipeline communications each in routines SPENTAZ and SPENTAZ3. Similar to the VPENTA routines in Arc3d the pipelines of a set of three communications surrounding the same calculation loop. These three communications were therefore merged to allow the data to be communicated within a buffer. These alterations were also performed for the overlapping communications. The results for this code may be seen in Table 6.9. It can be seen that the results are not exceptionally good with only a speed up of 3.19 obtained on 16 processors (Figure 6.15).

After applying the automatic generation of overlapped communications, ten of the original thirteen synchronous communications were overlapped. The SIMPLE method of overlapping was applied to all the communications. Three of the exchange communications were not overlapped since they required the PARTIAL and UNROLL methods to be applied. The sink command of these communications lay within pipeline loops. It is not possible to apply PARTIAL and UNROLL to pipeline loops as this may cause incorrect data to be communicated and for incorrect results to be obtained. Furthermore, all of the pipelines were overlapped, the results of which can be seen to help reduce the run itme of the code in parallel (Figure 6.14). It can be seen that applying overlapped communications has caused an improved speed up of 5.30 (Figure 6.15) to be obtained for 16 processors.

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	613.43	-	-	613.43	-	-
2	380.24	1.61	80.7%	357.07	1.72	85.9%
4	281.30	2.18	54.5%	218.14	2.81	70.3%
8	218.22	2.81	35.1%	146.35	4.19	52.4%
12	205.27	2.99	24.9%	130.27	4.71	39.2%
16	192.16	3.19	19.9%	116.56	5.26	32.9%

Table 6.9 : Results for APPSP with synchronous and overlapped communications for the Transtech Paramid.

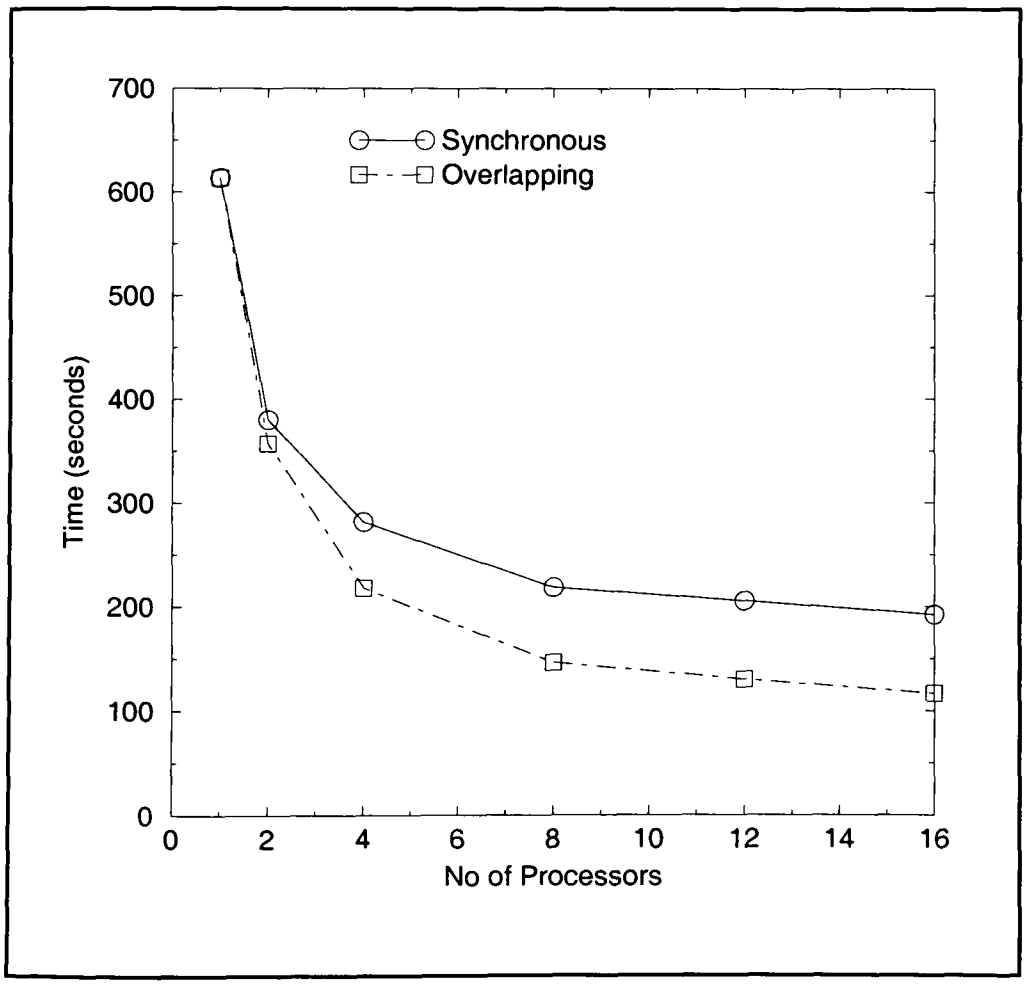


Figure 6.14 : Time graph of APPSP for the Transtech Paramid.

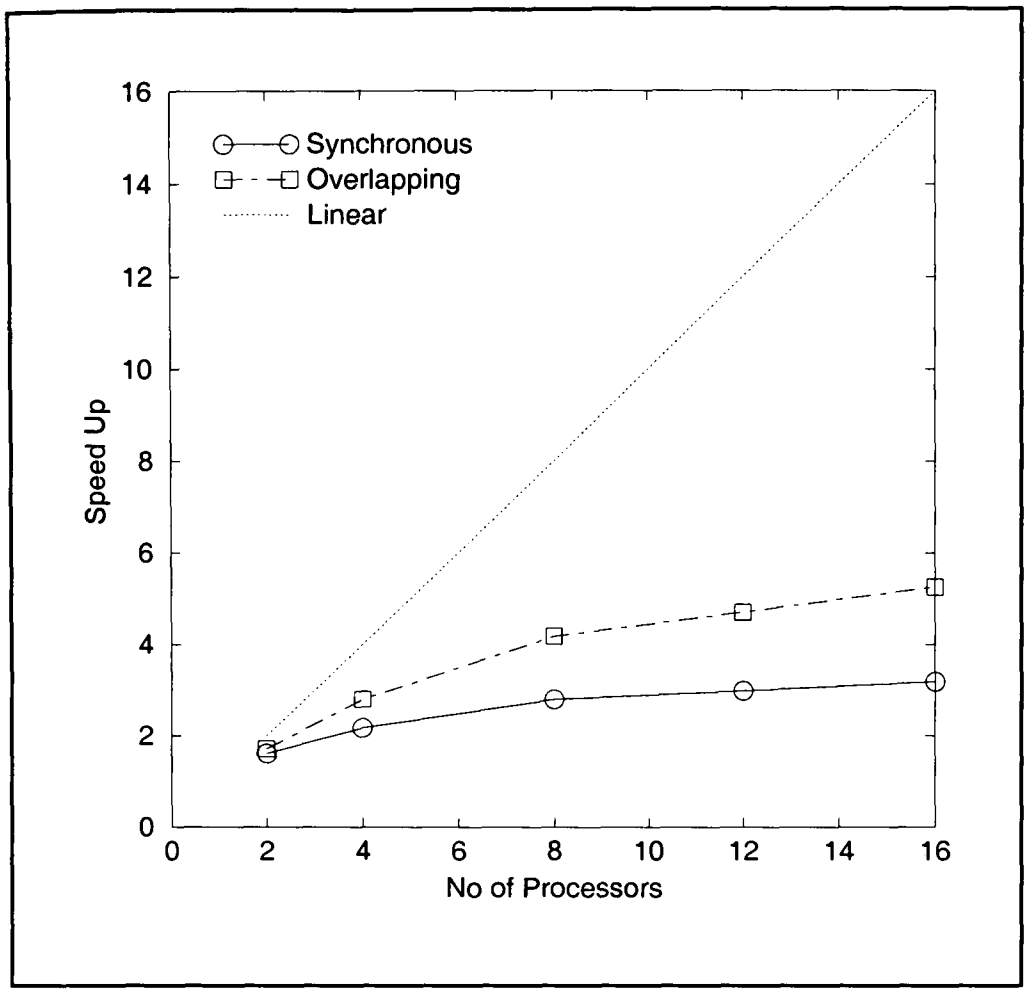


Figure 6.15 : Speed up graph of APPSP for the Transtech Paramid.

6.7 APPBT.

APPBT is also a code from NAS-PAR benchmark suite [68]. This code, similar to the APPSP code, uses an implicit algorithm to compute a finite difference solution to the 3D compressible Navier-Stokes equations. The solution is based on a Beam-Warming approximate factorisation. The approximate factorisation decouples the three dimensions leading to three sets of regularly structured systems of linear equations. The resulting equations vary from the APPSP code in that they are block tridiagonal but are also solved using the Thomas algorithm (Gaussian elimination) without pivoting of a banded system. The code is approximately 4500 lines of Fortran 77 code.

The synchronous code generated consisted of 32 communications of which 13 were exchange communications and 3 were pipeline communications. There were two pipeline communications in the routine BTRIDZ that surrounded the same calculation loop. These two communications were therefore merged to allow the data to be communicated within a buffer. These alterations were also performed for the overlapping communications. The results for this code may be seen in Table 6.10. It can be seen that the results are not exceptionally good with only a speed up of 4.29 obtained on 16 processors for the synchronous version (Figure 6.17).

After applying the automatic generation of overlapped communications, all 13 of the original synchronous communications were overlapped. The SIMPLE method of overlapping was applied to all these communications. The 3 pipeline communications were also overlapped successfully. The results obtained increase the speed up of the code significantly with a speed up of 7.31 obtainable for 16 processors (Figure 6.17) and also reduced the run time (Figure 6.16).

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	840.63	-	-	840.63	-	-
2	512.82	1.64	81.9%	477.25	1.76	88.1%
4	351.29	2.39	59.8%	275.21	3.05	76.4%
8	251.21	3.35	41.8%	168.53	4.99	62.3%
12	224.38	3.75	31.2%	141.17	5.95	49.6%
16	198.13	4.24	25.5%	114.99	7.31	45.7%

Table 6.10 : Results for APPBT with Synchronous and Overlapped Communications for the Transtech Paramid.

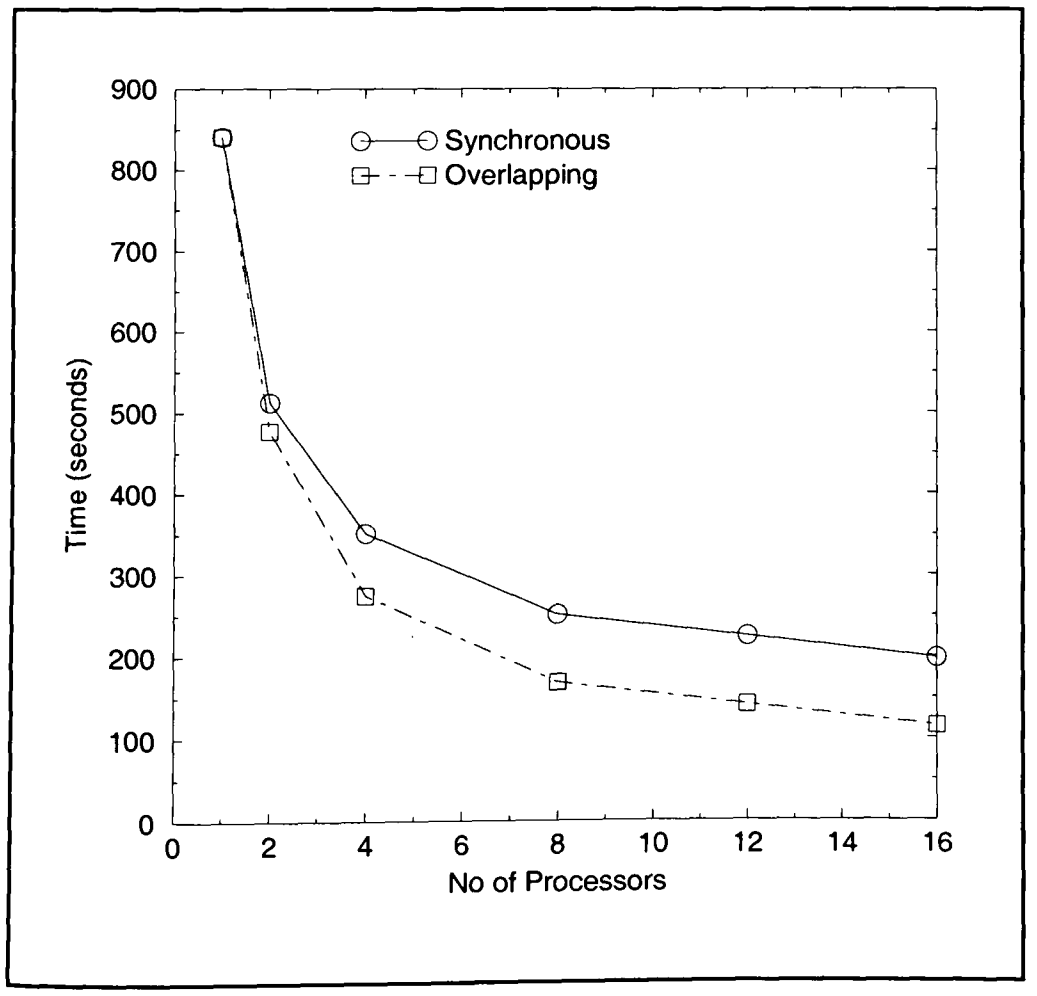


Figure 6.16 : Time graph of APPBT for the Transtech Paramid.

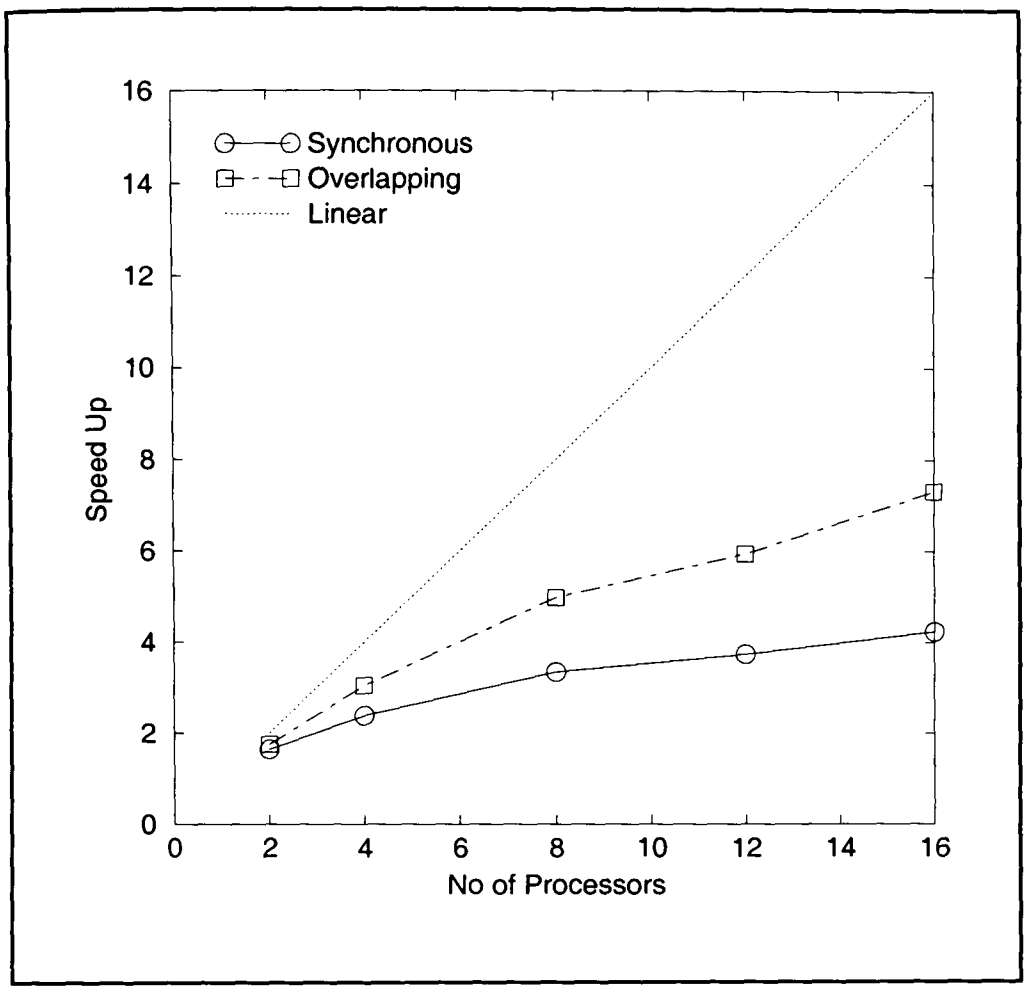


Figure 6.17 : Speed up graph of APPBT for the Transtech Paramid.

6.8 Industrial CFD Code

This code is a three-dimensional, transient, laminar and turbulent fluid flow code. The predictions are made on a structured mesh and the solutions are obtained using a two colour (checkerboard) ADI solver. There are also several upwind differencing calculations in the code. This technique is used in CFD codes to simulate the effect of high velocities affecting the influences on each cell in the mesh. The code was developed for, and is used by, a major industrial company. It consists of approximately 5000 lines of Fortran 77 and contains 40 subroutines.

The synchronous communications generated for this code by CAPTools consisted of 79 exchange communication calls and 5 pipeline communications. There are three pipelines in the main ADI solver routine two of which required the buffering and unbuffering of data to reduce the overheads of the pipeline communication. This buffering and unbuffering of data was applied to both the synchronous and overlapped versions of the code generated. The synchronous results for this code (Table 6.11, Figures 6.21 and 6.22) show a reasonably good speed up of 11.29 on 16 processors.

Applying automatic overlapping communications allowed 69 of the exchange communications to be overlapped and all but one of the pipelines were overlapped. The exchanges not overlapped are due to conflict with the upwinding scheme (Figure 6.18) that is similar to the QUICK differencing scheme for TeamKE1 (Section 3.3). The values communicated by the exchanges may not be overlapped using the SIMPLE method since there are no time consumers between the synchronisation point and the communication. When PARTIAL and UNROLL are tested they also fail since each one of the communications has two sinks which are dependent on the upwinding scheme. The values determined in the upwinding calculation may be increasing or decreasing for each iteration of the loops and it is not determinable. One pipeline was not overlapped since it involved the communication of a scalar variable.

```
CALL CAP_EXCHANGE(ADD_SRC(1,1,CAP_BHCOEFX_P+1),ADD_SRC(1,1,CAP_BLCOEFX_P),900,2,CAP_RIGHT)
CALL CAP_EXCHANGE(ADD_SRC(1,1,CAP_BLCOEFX_P-1),ADD_SRC(1,1,CAP_BHCOEFX_P),900,2,CAP_LEFT)
DO 35  K=2,N,1
        DO 34  J=2,M,I
                DO 31  I=MAX(3,CAP_BLCOEFX_P),MIN(L,CAP_BHCOEFX_P),1
                        IO=INT(-SIGN(I.,UVELOCITY(K,J,I)))
                                .
                                .
                                .
                        SRC_I(I) = (ADD_SRC(K,J,I) + ADD_SRC(K,J,I-IO))*.....
                                *(ADD_SRC(K,J,I) + ADD_SRC(K,J,I+IO))*.....
                                .
                                .
                                .
31                      CONTINUE
34              CONTINUE
35      CONTINUE
```

Figure 6.18 : Upwinding scheme from the Industrial CFD code.

The automatic generation of the overlapped communication provided an improvement in the speed up of 12.26 on 16 processors (Table 6.11, Figure 6.19 and Figure 6.20).

Number Of Processors	Synchronous Communications			Overlapped Communications		
	Time Taken	Speed Up	Efficiency	Time Taken	Speed Up	Efficiency
1	560.99	-	-	560.99	-	-
2	287.92	1.95	97.4%	282.30	1.98	99.4%
4	156.47	3.58	89.6%	150.75	3.72	93.0%
8	84.65	6.63	82.8%	79.79	7.03	87.9%
12	62.30	9.00	75.0%	58.08	9.66	80.5%
16	49.70	11.29	70.5%	45.75	12.26	76.6%

Table 6.11 : Results for an Industrial CFD code with synchronous and overlapped communications for the Transtech Paramid.

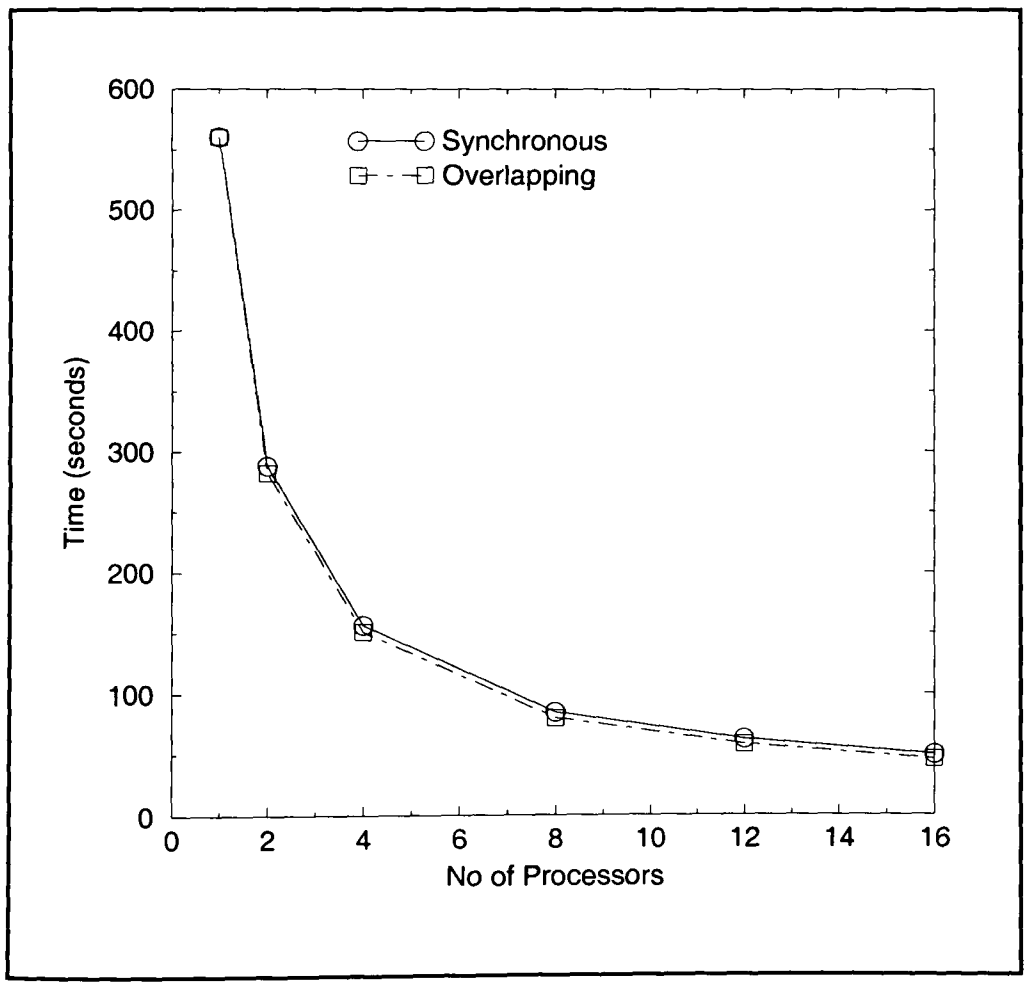


Figure 6.19 : Time graph of an Industrial CFD code for the Transtech Paramid.

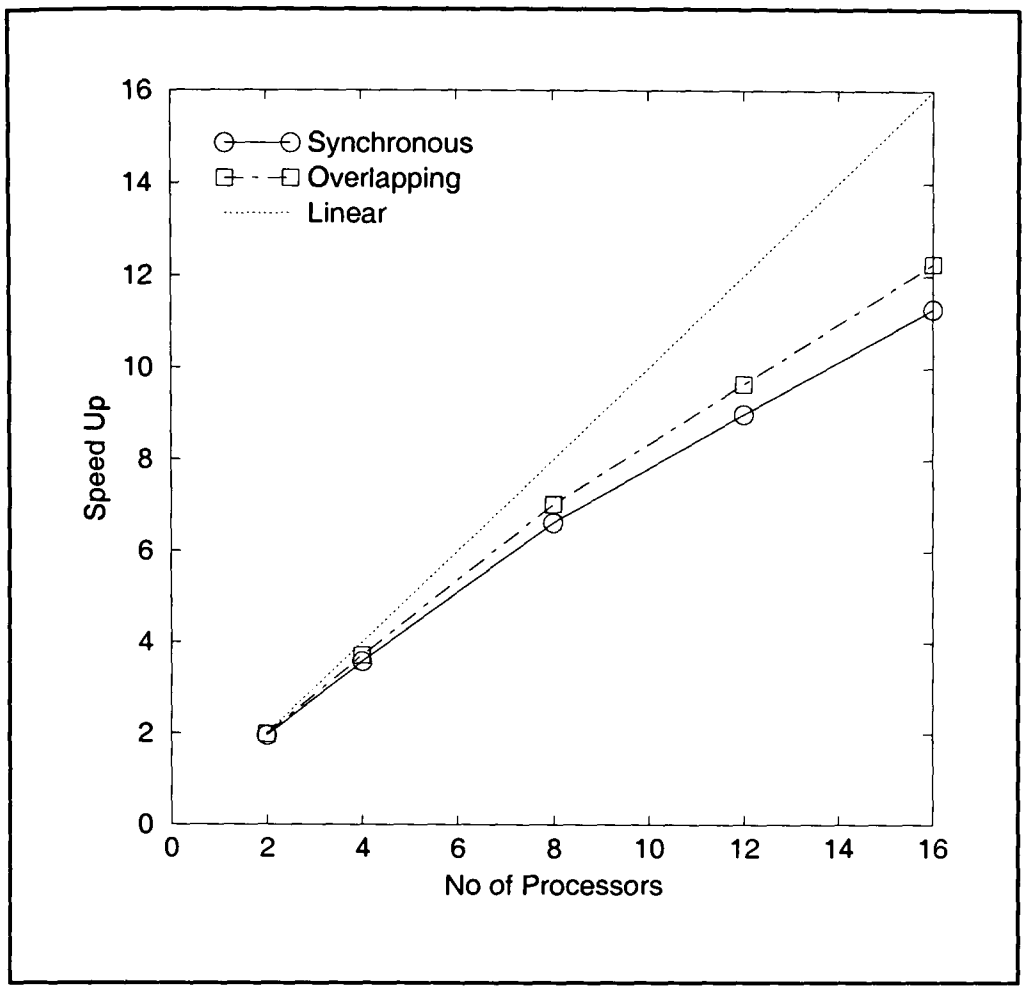


Figure 6.20 : Speed up graph of an Industrial CFD code for the Transtech Paramid.

6.9 Summary of Overlapped Communications Applied.

Table 6.12 summarises the number of exchanges overlapped using which method and the number of pipelines overlapped for each of the codes referred to in this chapter.

Code	Number of Exchange Comms.	Number of Simple Overlap	Number of Partial Overlap	Number of Unroll Overlap	Number of Pipelines Overlapped	Percentage improvement on 8 procs
FAB	7	2	1	2	-	4.5
TeamKE1	62	53	6	3	8 of 8	0.7
Applu	10	10	-	-	2 of 2	21.7
Arc3d	31	25	2	3	10 of 10	15.3
APPSP	13	10	-	-	8 of 8	17.3
APPBT	13	13	-	-	3 of 3	20.5
Industrial	79	69	2	-	4 of 5	5.1

Table 6.12 : Summary of overlap methods applied to each code.

The total number of Simple, Partial and Unroll communications may not equal the total number of Exchange communications since each communication may have more than one request for the communicated data, i.e. due to the merging of communications (Section 2.8.3).

The results show a significant improvement for most codes on 8 processors. The improvement, however, is not very impressive for the TeamKE1 code (0.7% improvement over the synchronous communications code). This is due to the high number of additional communications within the code that are not possible to overlap, i.e. the communications that are not CAP_EXCHANGE calls or pipelines. These communications cause the overlapped communications to synchronise before their respective synchronisation points.

6.10 Conclusions.

The automatic code generation of overlapped communication was applied successfully to all the codes in this chapter. The results obtained for overlapped communication provided an increase in the speed up and efficiency of the codes. Two of the codes (APPLU and ARC3D) were also tested on a machine other than the Transtech Paramid. The results for the Parsys SN9500 also showed that further increase in efficiencies could be applied by using overlapped communications. Favourable results were also obtained for a real world CFD code from a major industrial company (Section 6.8). It was therefore extremely advantageous to apply the overlapped communications to these codes. Unfortunately, as mentioned in Section 4.3, the use of overlapped communications is dependent on the hardware. At present there seems to be few parallel machines that can allow overlapped communications to be used successfully, but hopefully the future generation of hardware will allow effective overlapping communications.

Chapter 7

7 Application and Investigation into Automatic Code Generation for Overlapping Communications for Unstructured Mesh Computational Mechanics Codes.

7.1 Introduction

The previous chapters discussed the application (Chapter 4) and the automatic code generation of asynchronous communications (Chapter 5) for structured mesh codes. This section now extends these methods to unstructured mesh computational mechanics codes. The chapter first discusses the fundaments of unstructured meshes codes and their parallelisation. The chapter then moves on to look at the already parallelised unstructured mesh codes : ASTEC [85] and PUIFS [89]. Based on the information gathered from investigating these parallelised unstructured mesh codes a generic method was formulated whose techniques were then applied in the manual parallelisation of the ESAUNA [84] code. The automatic generation of unstructured mesh parallel code within the Computer Aided Parallelisation Tools is then discussed. Finally the chapter discusses the manual and automatic application of asynchronous communications to these unstructured mesh codes.

7.2 Unstructured Mesh Computational Mechanics Codes.

The basic description of unstructured meshes and how it varies to a structured mesh code is described in Section 1.4. As previously mentioned, for a structured mesh code the domain is decomposed and defined as runtime calculated variables, i.e. CAP_LXX and CAP_HXX, that

are local to each processor (Section 2.6) and represent the range of values in a particular array that each processor operates upon. An unstructured mesh is represented as a graph of the interconnections between, typically elements and nodes. This graph may be partitioned using Graph Based Partition algorithms or using software tools such as Scotch [42], Metis [43], Chaco [44] and JOSTLE [45, 46, 47] (Section 1.5).

The parallelisation of unstructured mesh codes within the University of Greenwich makes use of the 'in-house' developed software tool JOSTLE. Using the interconnections graph and the hardware processor topology JOSTLE provides the partition details as an owner array that represents the processors that own a particular array element.

Similar to the structured mesh parallelisations, the unstructured mesh parallisation requires a halo or overlap region (Section 1.7) to store the data that is calculated on another processor.

7.3 Manual Parallelisation Experience of Unstructured Mesh Codes.

Two separate unstructured mesh codes were investigated for the parallelisation strategies employed. In the case of the ASTEC [85] code, this had already been parallelised and possessed asynchronous communications and required porting to a parallel system at the University of Greenwich. PUIFS [89] was parallelised by hand by a colleague at the University of Greenwich.

7.3.1 ASTEC.

This code was developed at the United Kingdom Atomic Energy Authority (UKAEA) specifically for the modelling of fluid flow and its associated physical phenomena in and around complex geometries represented by unstructured meshes [85]. The code uses the 3-D flow Navier Stokes equations discretised onto a finite element mesh consisting of eight node hexahedra. The finite volume discretisation is based on nodal control volumes. Nodal discretisation is used for the momentum, scalar and turbulence quantities and is solved by using a Gauss-Seidel scheme. The element centre based discretisation used for continuity is solved by a preconditioned conjugate gradient method. These systems of equations are solved using the well-known SIMPLE algorithm [86]. The algorithm solves for velocity, scalar quantities and turbulence iteratively in turn until the convergence criteria is satisfied.

7.3.1.1 Mesh Decomposition.

The code had previously been parallelised at UKAEA [87]. This parallelisation was implemented on a Meiko Computing Surface [88] using CSTools communications.

The topology of the processors was such that any processor could communicate to any other processors. This allowed the topology to reflect the partition of a complex geometry as illustrated in Figure 7.1. To enable the processors to communicate to any other processor required the use of a communication harness or router. This harness created an enormous number of software channels connecting each processor to all others for communication of various data.

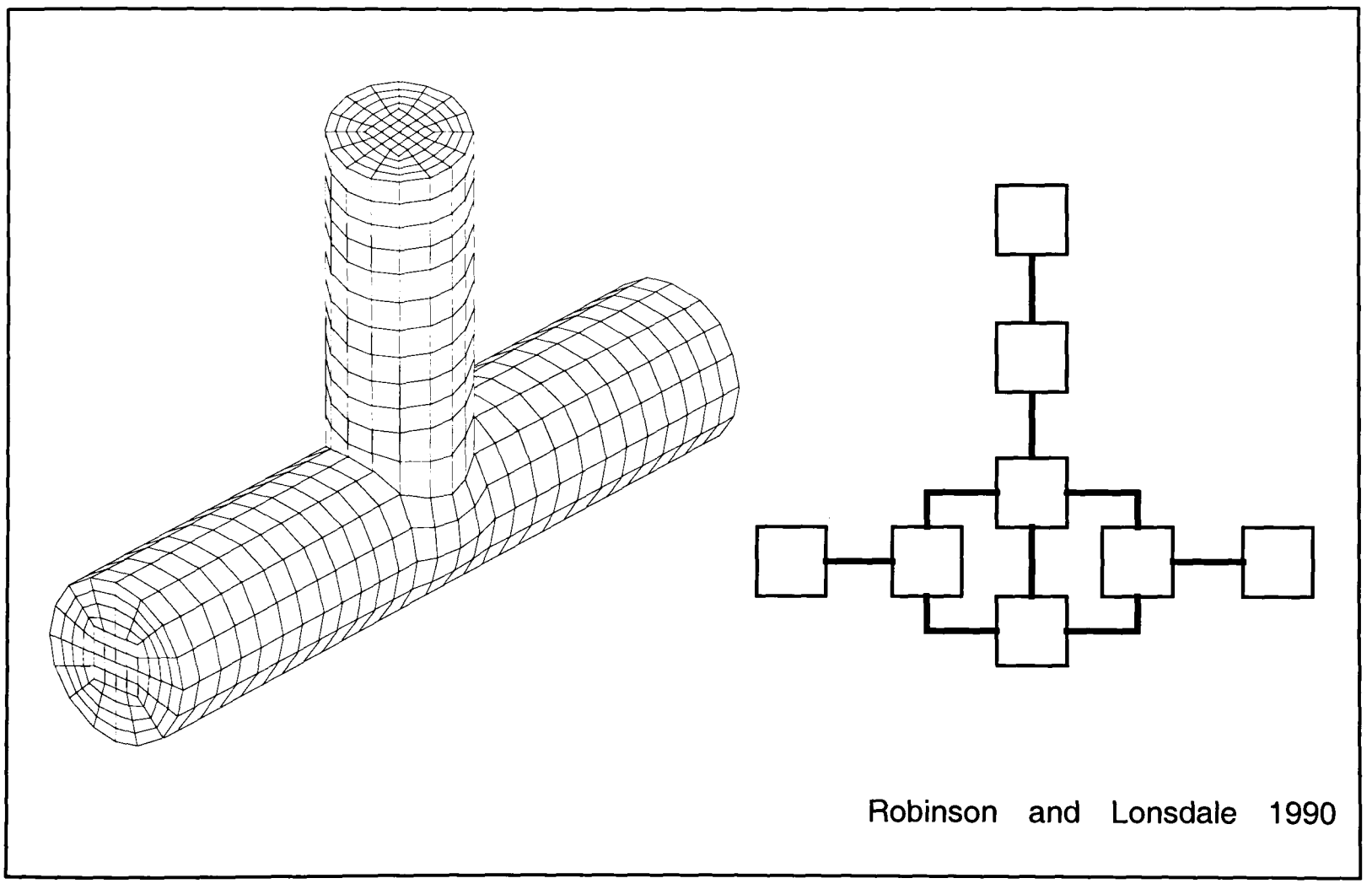


Figure 7.1 : Pipe mesh and a typical processor topology.

The unstructured mesh is decomposed as a pre-processing step using the MSPLIT program provided with the parallel ASTEC code. MSPLIT, when provided with the serial mesh, input file set and the number of processors will divide the mesh accordingly and generate equivalent parallel file sets. These parallel file sets are required for every varying number of processors, i.e. for a 2, 4 and 8 processor run a different file set will be required for each

topology. The MSPLIT program will, when decomposing the mesh, also renumber the elements and nodes for each processor to local node and element numbers. Each processor will have local node numbers that run from 1 to K1TOT and element numbers that run from 1 to M1TOT. The values of K1TOT and M1TOT will be unique to each processor. Each processor will also require a halo (or overlap) of additional nodes and elements, which are calculated on other processors, to ensure that the nodes and elements of the processor are updated correctly.

7.3.1.2 Gauss-Seidel Method in Parallel.

Each processor again has its own local matrix and vector multiplication to be calculated in order to apply a correction to the vector. This local data is then exchanged between the processors to ensure that each has the correct halo data.

To reduce the communication overhead of this method the communications are performed asynchronously. Each processor will perform its own independent calculation sweep using the latest available values that have been communicated to the halo data regions. This halo data may be values from a different sweep to the one in operation on this processor. The halo data could therefore be several sweeps behind or in front of the processors current sweep. This will significantly decrease the communication overhead of the algorithm that would be incurred by synchronisation to ensure that each processor has the correct halo data. However, this method will cause the algorithm to behave in a highly non-deterministic manner (Section 4.2). Due to communication delay and latency the solver may be using data from a sweep behind or in front of the current sweep thus causing the solution to converge at a different rate to the serial

7.3.1.3 Communications

As mentioned earlier the communication methods employed were asynchronous. Data is sent using the CSTools primitive CSNTXNB, while the CSTools primitive CSNRXTB would receive data. All send communications would be initiated to send data to other processors at the same time. Once all the data had been sent then the receive communications would be initiated. To facilitate such a method of communication requires the data to be stored in large buffers within the communication harness. When the send communications are executed the data is communicated and stored in communication buffers until the receive initiates its communication. When these receive communications are initiated it is then possible to read the

data from the buffers. This method of communication required a significant amount of the memory on each processor to be reserved for buffering the communicated. The main disadvantage of this method of communication is that if there was an insufficient amount of buffer space then the parallel code would unexpectedly come to a halt. Obviously, with larger problem sizes this will increase significantly.

7.3.1.4 Porting ASTEC.

The proposition was to port the parallel code to operate on the Transtech Paramid [81] parallel system (i860/T800 hybrid) available at the University of Greenwich. This parallel machine did not at the time have asynchronous communication capabilities or allow for any processor to communicate to any other processor. The porting therefore required the code to operate with synchronous communications and also to create a communication harness/router to allow the communication of data to any other processor. Further information on the porting of the ASTEC code is discussed in greater detail in Appendix A.

As previously mentioned in Section 7.3.1.3 the communications were set up such that the code would initialise several send communications before the data could actually be received by a processor. This approach was reasonable when the communications were non-blocking, but highly unworkable when using synchronous communications. These communications were, therefore, modified such that they would be precise and deterministic.

The results of this porting were very poor (Table 7.1) with very little speed up on 2 or more processors. The users of the serial version of the ASTEC code at the University of Greenwich had developed additional user code routines to provided solutions for solving stress and solidification of molten metal. These user code routines were also parallelised. Very favourable results (Table 7.2) were obtained when these routines were incorporated into the ASTEC code and timed. The graph in Figure 7.2 shows the speed up graph for the problem.

No of Procs	Time	Speed Up	Efficiency
1	109.1	-	-
2	105.0	1.04	51.9%
3	107.9	1.01	33.7%
4	104.5	1.04	26.1%
5	104.3	1.05	20.9%
6	98.3	1.11	18.5%
7	105.3	1.04	14.8%

Table 7.1 : Results for ASTEC only for a Thermal beam problem (2916 elements, 3700 nodes).

No of Procs	Time	Speed Up	Efficiency
1	1273.0	-	-
2	652.9	1.95	97.5%
3	473.1	2.69	89.7%
4	463.2	2.75	68.7%
5	400.3	3.18	63.6%
6	354.3	3.59	59.9%
7	327.4	3.89	55.5%

Table 7.2 : Results for Stress calculation incorporated into ASTEC for a Thermal beam problem (2916 elements, 3700 nodes).

The original parallel ASTEC code was not very portable from one parallel system to another and provided poor results. The results for the parallelisation of the user modules for stress and solidification were however much better. The original parallel code or the ported parallel code was clearly not a feasible method of parallelising an unstructured mesh code.

The parallel code was also not portable, efficient or reliable. These three factors are essential for parallel processing to be acceptable to the user. Poor efficiencies and unreliable results/solutions will discourage the user from using a parallel system.

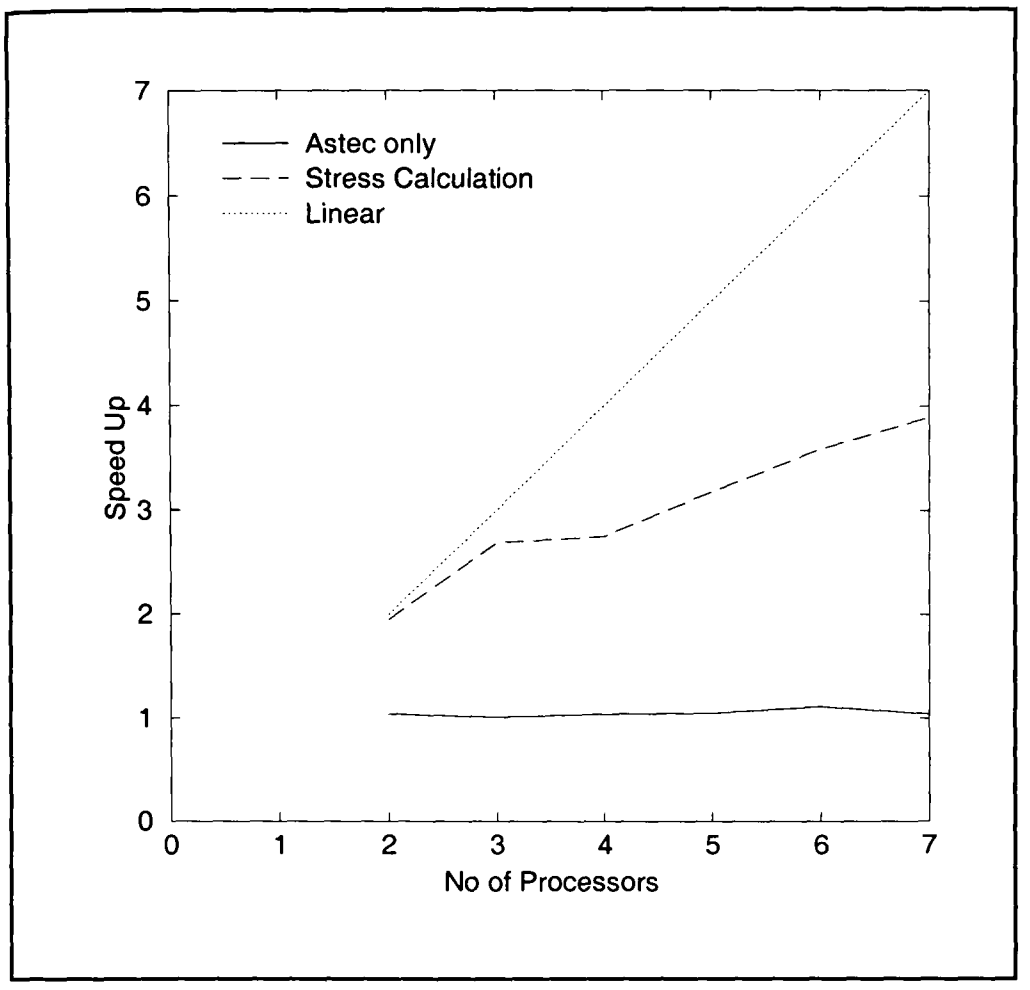


Figure 7.2 : Speed up results from ASTEC.

The additional effort of porting from one parallel system to another will very much discourage the user from using a parallel system. One of CAPTools main aims is to make the parallel code easily portable from one parallel machine to another. Another important aim of CAPTools is to ensure that the same results are obtainable from the parallel code as the serial code. For the original parallel strategy applied to ASTEC this was not the case.

7.3.2 PUIFS.

PUIFS [89] is the parallel version of the Unstructured Incompressible Flow and Stress (UIFS) [90] code developed at the University of Greenwich. The code was developed to model the processes involved in metals casting. It is a two dimensional unstructured mesh code solving the Navier Stokes equations for transient and steady state flow problems with solidification along with the elastic stress-strain equations [91, 92].

The fluid dynamics scheme in UIFS solves for flow on a single unstructured mesh using a modification of the SIMPLE algorithm of Patanker and Spalding [86]. The solid mechanics scheme uses the finite volume unstructured mesh procedure of Fryer et al. [91] for the solution of the elastic stress-strain equations for bodies undergoing thermal or mechanical loads.

The fluid dynamics code is loosely coupled with the solid mechanics code as shown in Figure 7.3. The fluid dynamics loop reaches convergence for a time step before entering the solid mechanics loop. When the solid mechanics loop reaches convergence, UIFS loops for the next time step. Each of the solvers may be turned on or off to suit the requirements of a given problem.

The code was already parallelised by hand at the University of Greenwich [89]. The user effort required to parallelise the code was over one year although the majority of the process was fairly straightforward. The amount of change to the code was minimised with most routines requiring no changes

Investigation of the code showed that the partition for PUIFS was obtained from JOSTLE [45, 46, 47], a domain decomposition tool developed at the University of Greenwich. The partitions obtained were then re-numbered (as for ASTEC in Section 7.3.1) to obtain self-contained sub-domains. The same decomposition strategy was applied to the stress and solidification sections of the code as applied in the ASTEC code.

A set of parallel utilities was created specifically for use with PUIFS that used synchronous communications that were deterministic, portable, scalable and reliable. The deterministic nature of these communications ensured that the results were almost identical (apart from slight round off) to those achieved from the parallel code as compared to the serial code.

The key communication developed was the SWAP utility. This performed an exchange of overlap data between all processors. This is similar to the CAP_EXCHANGE communication (Section 1.6) that was used for the structured mesh code parallelisations. The overlap regions to be communicated were calculated and stored as the communication set at the beginning of the code run as part of the mesh decomposition process. The SCATTER utility distributed data across the processors while the GATHER utility rebuilds the components from each processor,. This utility is similar to a broadcast communication in the structured mesh that uses the CAP_SEND and CAP_RECEIVE communications (Section 1.6). Specific commutative operation such as GSUM, GMAX, GMIN, etc were also used to obtain a global value by combining the local values and broadcasting the global result to each processor (cf. CAP_COMMUTATIVE in Section 1.6).

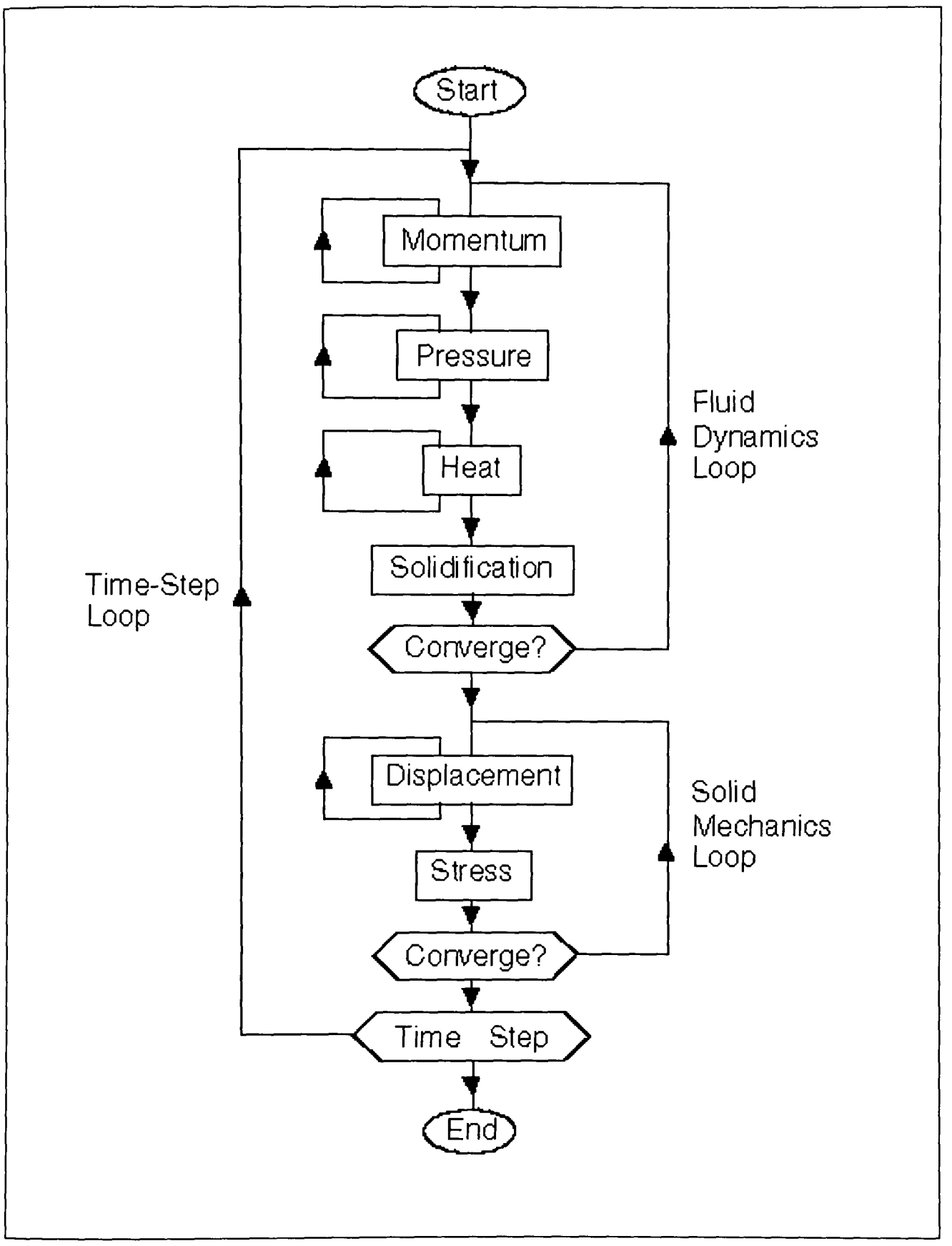


Figure 7.3 : Flow chart for UIFS.

A typical code example and data structure used within UIFS is shown in Figure 7.4. The main 400 loop is iterating over the total number of element types (NETYPE). Examples of element types are triangles, quadrilaterals, etc. For each element type all the elements (ELENUM) are iterated over in loop 300. Each one of these elements in turn iterates over each grid point (PELPTS/GPTPEL) that an element type owns in the loop 100.

Statement S_1 uses a grid point of the element to update an element value at statement S_2 whose grid point may be on another processor. Since the unstructured grid may have been partitioned such that adjacent elements exist on different processors it is essential that the overlap regions of each processor has been calculated correctly. A SWAP communication is therefore required to satisfy that computation.

```
                IONE = 1
                DO 400 ITYPE = IONE, NETYPE
                        ISTART = IEND + IONE
                        IEND = IEND + NQTELE(ITYPE)
                        DO 300 ELENUM = ISTART, IEND
                                ....
                                ....
                                PELPTS = GPTPEL(ITYPE)
                                DO 100 I = IONE, PELPTS
                                        ADJNUM = ADJELE(I,ELENUM)
        C
        C                               Grid points of the surface with the adjacent element.
        C
                                        J = I + IONE
                                        IF ( J .GT. PELPTS ) J = IONE
                                        GPT1 = ELETOP(I,ELENUM)
                                        GPT2 = ELETOP(J,ELENUM)
S1                                      Y2MY1 = XYZCRD(GPT2,IY) - XYZCRD(GPT1,IY)
                                        X2MX1 = XYZCRD(GPT2,IX) - XYZCRD(GPT1,IX)
                                        ....
S2                                      VOLUME(ELENUM) = X2MX1 + Y2MY1
                                        ....
          100                   CONTINUE
          300                 CONTINUE
          400       CONTINUE
```

Figure 7.4 : A typical code example and data structure from the UIFS code.

7.4 Generic Methods of Parallelising Unstructured Mesh Codes.

Based on the two very different parallelisation strategies applied to the ASTEC and PUIFS unstructured mesh codes the best method was that applied to the PUIFS code. The methods applied could easily be made generic to other unstructured mesh codes thus allowing reliable and effective parallel code to be obtainable.

The generic method includes the use of JOSTLE [45, 46, 47] for the data partitioning, although this may quite as easily be done by another domain decomposition tool such as Scotch [42], Metis [43] or Chaco [44].

This process involves the partitioning, the generation of execution control masks and the calculation and generation of communications. To aid in the process of describing the

parallelisation of an unstructured mesh consider the code in Figure 7.5 for solving a typical example of an unstructured mesh of 94 elements shown in Figure 7.6.

```
      READ*, NELEMENT
      DO ELEMENT = 1, NELEMENT
C
C           For each element read how many neighbouring elements.
C
            READ*, NUM_OF_NEIGHBOURS(ELEMENT)
C
C           Read the element topology and the original temperature value of each element.
C
            DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                  READ*, ELETOP(ELEMENT_NEIGHBOUR, ELEMENT)
                  READ*,ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
            ENDDO
      ENDDO
C
          ....
      Other code.
          ....
C
C     Calculate the new temperature for each element.
C
      DO ELEMENT = 1, NELEMENT
            NEW_TEMP(ELEMENT) = 0.0
            DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                  NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                  ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
            ENDDO
      ENDDO
```

Figure 7.5 : A simple unstructured mesh code example.

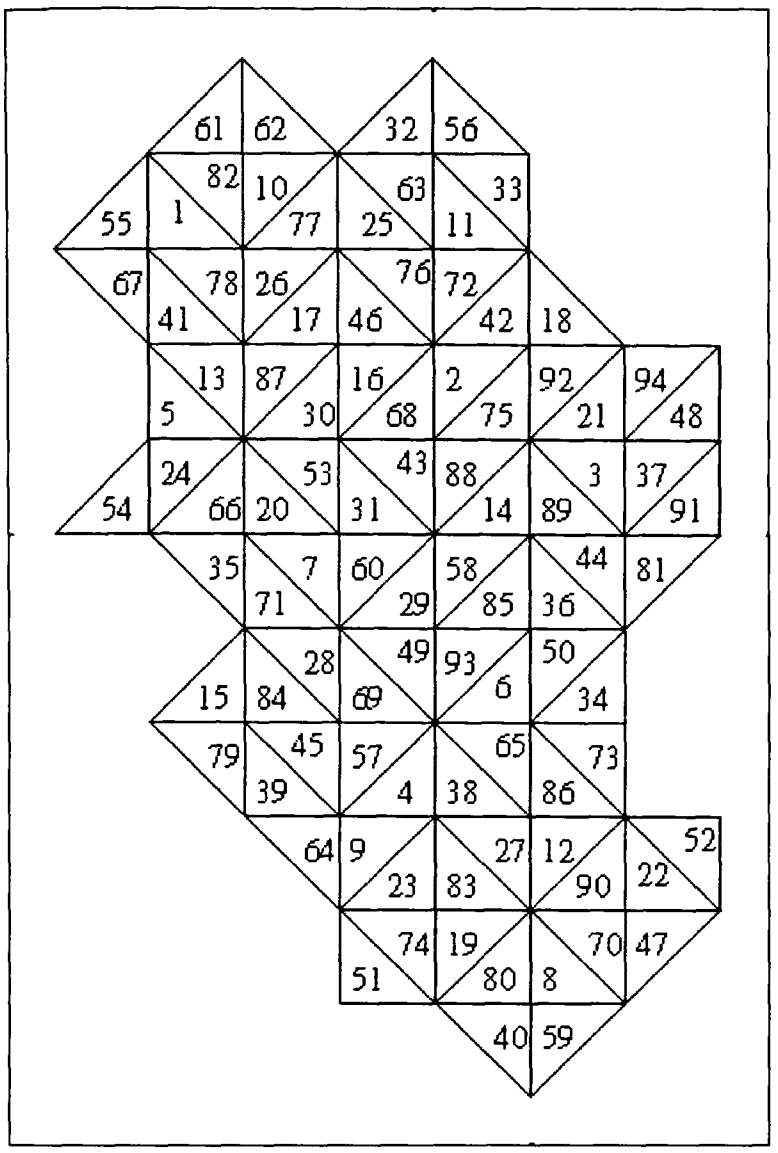


Figure 7.6 : An unstructured mesh of 94 elements.

The unstructured mesh code in Figure 7.5 first reads the number of elements (NELEMENT) for the unstructured mesh. For each element (ELEMENT) the number of neighbouring elements (NUM_OF_NEIGHBOURS) is read. The array ELETOP that stores the topology representation, i.e. the element number of each element's neighbouring element is then read. For the mesh in Figure 7.6 the topology representation would be stored as in Table 7.3. Finally, the original temperature of each element (ORIG_TEMP) value is read.

The calculation loop calculates the new temperature value (NEW_TEMP) for each element based on the original temperature value (ORIG_TEMP) of its neighbouring elements.

7.4.1 Data Structures for an Unstructured Mesh

For an unstructured mesh there are many ways in which its data structure may be represented. Consider the unstructured mesh in Figure 7.6 the element topology relationship may be stored in the ELETOP array as in Table 7.3.

Element	Neighbour 1	Neighbour 2	Neighbour 3
1	ELETOP(1,1) = 55	ELETOP(2,1) = 78	ELETOP(3,1) = 82
2	ELETOP(1,2) = 42	ELETOP(2,2) = 68	ELETOP(3,2) = 75
3	ELETOP(1,3) = 21	ELETOP(2,3) = 37	ELETOP(3,3) = 89
4	ELETOP(1,4) = 9	ELETOP(2,4) = 38	ELETOP(3,4) = 57
5	ELETOP(1,5) = 13	ELETOP(2,5) = 24	
6	ELETOP(1,6) = 50	ELETOP(2,6) = 65	ELETOP(3,6) = 93
...	...	...	...
93	ELETOP(1,93) = 6	ELETOP(2,93) = 49	ELETOP(3,93) = 85
94	ELETOP(1,94) = 21	ELETOP(2,94) = 48	

Table 7.3 : The unstructured mesh represented as a data structure.

There are, however, numerous other ways in which this data structure may have been stored. Other such examples could be :

a) Where the negative value represents the element number followed by its corresponding neighbouring element numbers. In this case all the elements are stored in the array in numerical order but they need not be.

ELETOP(-1, 55, 78, 82, -2, 42, 68, 75, -3, 21, 37, 89, -4, 9, 38, 57, -5, 13, 24,
-6, 50, 65, 93, ... ,-93, 6, 49, 85, -94, 21, 48)

b) Where the negative values represents the first neighbour of an element. All the elements are stored in the array in numerical order.

ELETOP(-55, 78, 82, -42, 68, 75, -21, 37, 89, - 9, 38, 57, -13, 24,
-50, 65, 93, ... ,-6, 49, 85, -21, 48)

c) Stores the number of neighbouring elements for each element followed by that number of neighbouring element numbers. All the elements are stored in the array in numerical order.

ELETOP(3, 55, 78, 82, 3, 42, 68, 75, 3, 21, 37, 89, 3, 9, 38, 57, 2, 13, 24,
3, 50, 65, 93, ... ,3, 6, 49, 85, 2, 21, 48)

There are evidently an infinite number of methods by which the unstructured mesh data may be represented. To produce a parallel code for an unstructured mesh code requires efficient utility routines to partition the data and to calculate a communication set of the data to be communicated. Therefore, techniques are required to abstract the code data structures into simple structures that may be used in these utilities. The use of inspector loops [93, 94, 95] mimics the original serial user code to identify the assigned-used pair.

An example of an inspector loop for the calculation loop in Figure 7.5 is shown in Figure 7.7. The request is generated by the reference to the array ORIG_TEMP in Figure 7.6 and the inspector loop pair are (ELEMENT, ELETOP(ELEMENT_NEIGHBOUR, ELEMENT)). If there are any related statements that are connected by true dependencies to the inspector pair then these are copied to the inspector loop. Any statement that has a control statement to the requesting statement is also copied. For this example there are no such related statements. Additionally any statements required to implement the surrounding loops are also copied.

```
DO ELEMENT = 1, NELEMENT
        DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                CALL CAP_CONNECT(ELEMENT, ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
        ENDDO
ENDDO
```

Figure 7.7 : Inspector loop for the calculation loop in Figure 7.6.

From these pairs, the utilities construct a communication set of the data to be communicated. The associated executor is the communication itself that uses the communication set constructed by the inspector.

These utilities must be generic and need to take in a standard data structure. The utilities developed were based on those used in the parallelisation of PUIFS (Section 7.3.2). The CAPTools communication library CAPLib provides a utility to partition the data : CAP_JOSTLE; and two utilities to perform the functions of the inspector: CAP_CONNECT and CAP_OVERLAP. THE CAP_JOSTLE utility simply calls JOSTLE that returns the partitioned data. The CAP_CONNECT utility takes in the partitioned data that is used and the execution control mask that determines where the usage will occur in the processor topology. The CAP_OVERLAP utility constructs a communication set that is used to update the data of overlap regions on each processor.

7.4.2 Partitioning.

When the mesh has been decomposed the partition details are stored as a list where each entry of the list represents the processor that owns the partitioned array element. These processor numbers are set at run time, given the size and construction of the mesh, using the graph partitioning tool JOSTLE [45, 46, 47] (see Section 1.5). This will take the processor topology and the graph based representation of the mesh topology in a standard data structure and provides a list of the elements that each processor owns. Consider the mesh in Figure 7.6

that has been decomposed by JOSTLE onto a processor topology of 3 processors (Figure 7.9). The list returned by JOSTLE will be stored as in Figure 7.8.

```
Processor_owns_element(1) =  1     i.e. processor 1 owns element 1
Processor_owns_element(2) =  2     i.e. processor 2 owns element 2
Processor_owns_element(3) =  2     i.e. processor 2 owns element 3
Processor_owns_element(4) =  3     i.e. processor 3 owns element 4
Processor_owns_element(5) =  1     i.e. processor 1 owns element 5
Processor_owns_element(6) =  2     i.e. processor 2 owns element 6
  ...
Processor_owns_element(93) =  2    i.e. processor 2 owns element 93
Processor_owns_element(94) =  2    i.e. processor 2 owns element 94
```

Figure 7.8 : A list of processor-element owning relationship.

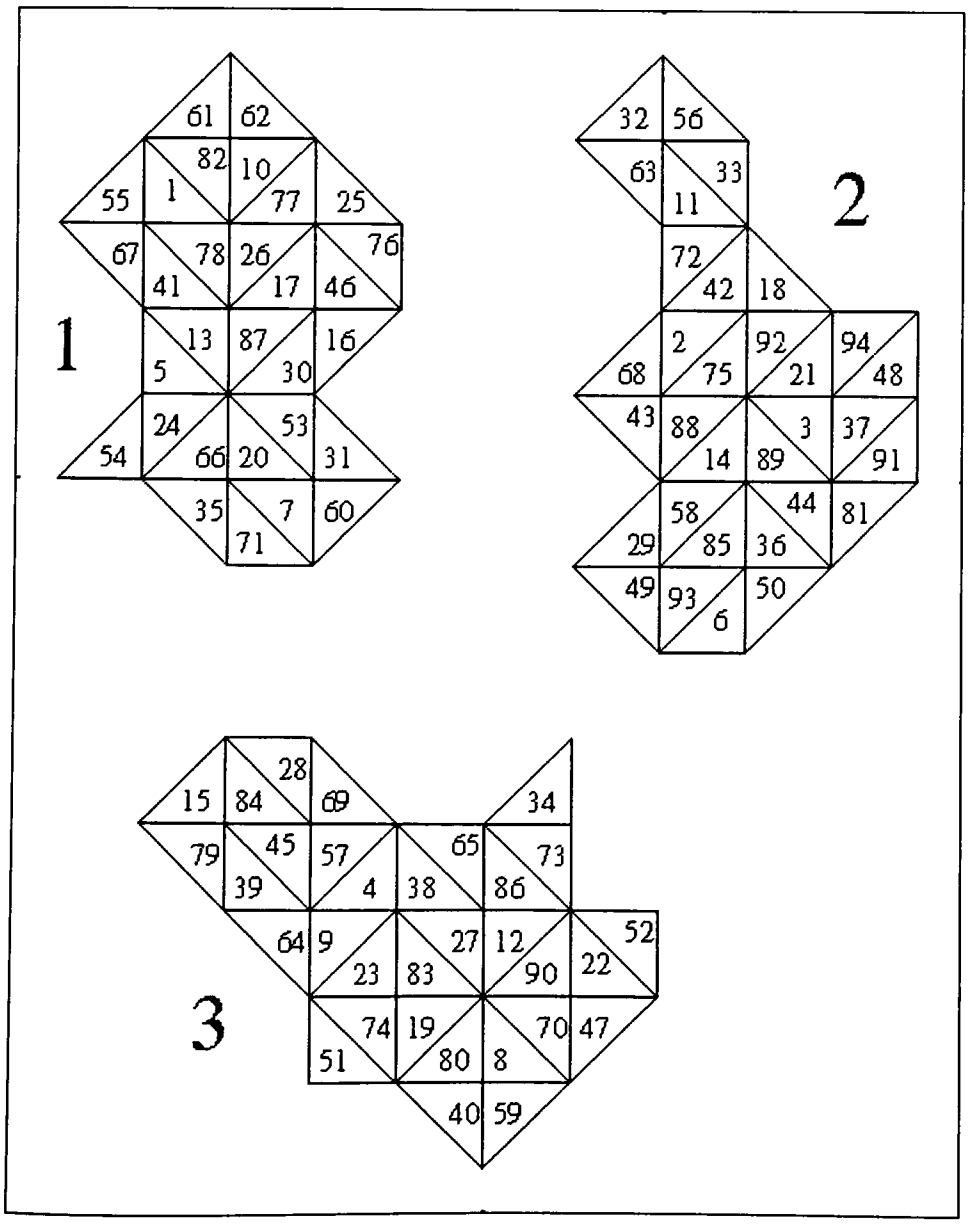


Figure 7.9 : Unstructured mesh decomposed onto three processors.

7.4.3 Execution Control Masks.

The masking methods developed for structured mesh codes is also applicable to the unstructured mesh code. The execution control mask will take the form :

```
IF (PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
```

This states that if this is the processor that owns the element (as stored in the PROCESSOR_OWNS_ELEMENT array) then it may proceed with the contents of the IF statement.

7.4.4 Calculation and Generation of Communications.

As previously mentioned for the structured mesh code (Section 2.8) a communication is required when data assigned on one processor is potentially required for calculation on another processor. The method applied here for unstructured mesh codes is once again similar to those techniques used for the structured mesh codes. However, for the unstructured mesh method two requests are issued for each non-local data reference. One request for the executor (i.e. communication) and one for the inspector that will enable the construction of the communication set.

For the code in Figure 7.17 the value of NEW_TEMP is calculated for the processor that owns the ELEMENT, as determined by the execution control mask that uses the array PROCESSOR_OWNS_ELEMENT. To calculate the value of NEW_TEMP requires the value of ORIG_TEMP for the element ELETOP(ELEMENT_NEIGHBOUR, ELEMENT) that potentially may not be owned by the same processor that owns ELEMENT. A communication is therefore required to ensure that the correct values are available on the required processors.

As previously mentioned in Section 7.4 there are many ways in which the data structure for an unstructured mesh may be represented.

Consider the decomposed unstructured mesh in Figure 7.10 that shows the overlap regions required for each processor. The elements required for inter processor communication are shown in Table 7.4.

	Processor 1 requires :	Processor 2 requires :	Processor 3 requires :
From Processor 1 :	-	Element 25, 76, 16, 31 & 60	Element 71
From Processor 2 :	Element 63, 72, 68, 43 & 29	-	Element 49, 6 & 50
From Processor 3 :	Element 28	Element 69, 65 & 34	-

Table 7.4 : Elements requiring communication for the decomposed mesh in Figure 7.10

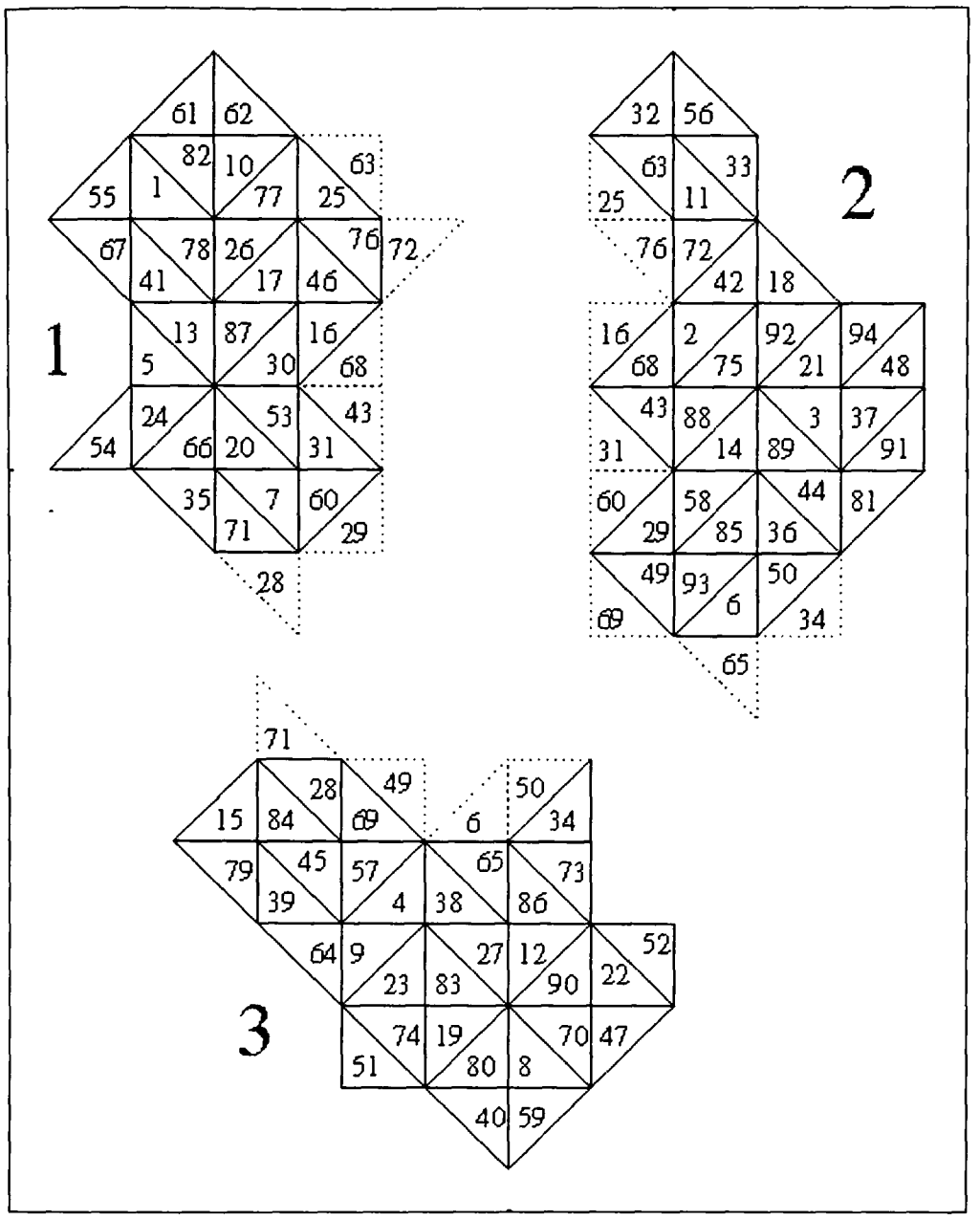


Figure 7.10 : Unstructured mesh decomposed onto three processors with overlap regions.

The calculation of the inspector loop for the code in Figure 7.17 requested by the calculation of NEW_TEMP using the value of ORIG_TEMP are for the relationship between the ELETOP(CAP_ELEMENT_NEIGHBOUR, ELEMENT) and CAP_ELEMENT pair.

The CAP_OVERLAP utility creates the communication set as used by CAP_SWAPOVER using the inspector loop connections formed by CAP_CONNECT and the partition arrays related to those connections. A communication set is created for each processor in the following form :

- Processor number to communicate with (always signified by negative value);
- Number of elements to communicate (positive value represents a RECEIVE, negative value represents a SEND);
- The elements to be communicated.

This list is repeated for each processor that a particular processor communicates with and the list is terminated by a zero. The communication sets generated for each processor for the inspector loop generated in Figure 7.18 and the elements to be communicated in Table 7.4 are:

Processor 1

-2,5,63,72,68,43,29,-3,1,28,-2,-5,25,76,16,31,60,-3,-1,71,0

Processor 2

-1,5,25,76,16,31,60,-3,3,69,65,34,-1,-5,63,72,68,43,29,-3,-3,49,6,50,0

Processor 3

-1,1,71,-2,3,49,6,50,-1,-1,28,-2,-3,69,65,34,0

An identification number (in this case CAP_ID1) is assigned to each communication set calculated by the CAP_OVERLAP utility to ensure there is no conflict with another communication set that may have been set up in the same code.

The CAPLib utility for the executor is CAP_SWAPOVER. This takes in as input the array that needs to be communicated into the overlap regions and the identification number for the required communication set. The final parallel code generated for the unstructured mesh code in Figure 7.5 is shown in Figure 7.11.

```
      READ*, NELEMENT
      DO ELEMENT = 1, NELEMENT
C
C           For each ELEMENT read how many neighbouring elements .
C
            READ*, NUM_OF_NEIGHBOURS(ELEMENT)
C           BROADCAST NUM_OF_NEIGHBOURS TO ALL PROCESSORS
C
C           Read the element topology and the original temperature
C           value of each element.
C
            DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                  READ*, ELETOP(ELEMENT_NEIGHBOUR, ELEMENT)
C                 BROADCAST ELETOP TO ALL PROCESSORS
                  READ*,ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
C                 BROADCAST ORIG_TEMP TO ALL PROCESSORS
            ENDDO
      ENDDO
C
C     Inspector loop ..
C
      CALL CAP_CONNECT_INIT(0)
      DO CAP_ELEMENT = 1, NELEMENT
            IF(PROCESSOR_OWNS_ELEMENT(CAP_ELEMENT) .EQ. CAP_PROCNUM) THEN
                  DO CAP_ELEMENT_NEIGHBOUR = 1,NUM_OF_NEIGHBOURS(CAP_ELEMENT)
                        CALL CAP_CONNECT(ELETOP(CAP_ELEMENT_NEIGHBOUR, ELEMENT),
     &                                   CAP_ELEMENT)

                  ENDDO
            ENDIF
      ENDDO
C
C     Partition mesh using JOSTLE and create a communication set.
C
      CALL CAP_JOSTLE(CAP_P_ORIG_TEMP, 0, 0)
      CALL CAP_OVERLAP(CAP_P_ORIG_TEMP, PROCESSOR_OWNS_ELEMENT, 0, 0, CAP_ID1)
C
C     Communicate required overlap data.
C
      CALL CAP_SWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
          ....
      Other code
          ....
C
C     Calculate the new temperature NEW_TEMP for each element.
C
      DO ELEMENT = 1, NELEMENT
            IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                  NEW_TEMP(ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                       ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                  ENDDO
            ENDIF
      ENDDO
```

Figure 7.11 : CAPTools generated parallel code for the serial code in Figure 7.5.

7.5 Communication Utilities for Parallel Unstructured Mesh Codes.

To apply this generic method the Computer Aided Parallel Tools communication library developed at the University of Greenwich [96, 97] had to be extended to allow the communication of data specifically for unstructured mesh codes. These communications wee based on the functionality of those developed for PUIFS but were made more generic. Initialisation primitive calls are also required for the determination of the mesh decomposition and the calculation of overlap regions for processors that are used by the communications.

The initialisation primitive calls that are required for the determination of the mesh decomposition and the calculation of communication sets for processors are :

CAP_CONNECT_INIT(*Numbering*)

This utility initialises the data for defining the connectivity pairs in the graph. The *Numbering* is either represented as global (0), local using indirection (1) or local but not using indirection (2).

CAP_CONNECT(*Data_set1*, *Data_set2*)

This utility takes in the partitioned data that is used and the execution control mask that determines where the usage will occur in the processor topology

CAP_JOSTLE(*Partition*)

This will take the processor topology and the graph based representation of the mesh topology and provide a list of the elements that each processor owns.

CAP_OVERLAP(*Partition1*, *Partition2*, *Identification_Number*)

The CAP_OVERLAP utility constructs a communication set that is used to update the data of overlap regions on each processor. The *Identification_Number* is set up and identifies a particular communication data set that is used by CAP_SWAPOVER.

CAP_SUBSIDIARY(*Partition1*, *Partition2*)

The CAP_SUBSIDIARY utility calculates for a subsidiary partition. It may be the case that the main partition is for a node to node graph, but there might be the need for a subsidiary partition for the elements based on the node-element graph..

The high level communication call and their parameter lists are as follows :

CAP_SWAPOVER (*Array*, *Length*, *Stride*, *Type*, *Identification_Number*)

vhere the *Array* is the start address of the data to send; the *Length* is the amount of data to be ommunicated; the *Stride* is the distance between each block of data that may be buffered; the

Type is an integer value representing the type of data to be communicated e.g. integer, real, etc; and the *Identification_Number* that identifies the communication set constructed by the CAP_OVERLAP utility. The communication utility packs the data to a buffer array, communicates synchronously and unpacks the data from the buffer array.

7.6 Parallelisation of ESAUNA using these Generic Methods.

The strategies and the generic utilities developed in Section 7.4 and 7.5 were tested by the parallelisation by hand of a large unstructured mesh code called ESAUNA [98].

The ESAUNA [98] code's unstructured mesh component was parallelised based on the parallelisation strategy employed in PUIFS but using the more generic approach (Section 7.4) to investigate the feasibility of the automatic parallelisation of such codes by the Computer Aided Parallelisation Tools.

The ESAUNA code is the flow solver module in a larger suite of codes called SAUNA, developed by Aircraft Research Association (ARA). ESAUNA is designed to calculate the flow over complex aerodynamic configurations by solving either the Euler or Navier-Stokes equations, in the latter case with a turbulence model. The solver is designed to work with either block-structured hexahedral grids, unstructured grids or any hybrid of the two.

The parallelisation of the block-structured grids had already been completed [98] and the parallelisation of the unstructured mesh grids case was required. The aim of this parallelisation was to implement the parallelisation strategies used in PUIFS using generic parallel utilities and inspector loops. It was also required to parallelise the code so that it would also be possible to test the feasibility of the techniques and the utilities developed for use in the automatic code generation of parallel unstructured mesh codes from Computer Aided Parallelisation Tools.

The partitioning of the unstructured mesh was executed by JOSTLE. Inspector loops were introduced to allow the parallelisation of the code. This involved inspecting the loops to discover the relationship between the data. Consider the code section for a typical calculation loop from the subroutine EULER in Figure 7.12.

```
      IF(ITYPE(NTYPE).EQ.0)GO TO 100
101   KK     = 0
      DO 110 JTY = 1,9
            IF (NGX(ML,JTY,1).EQ.0) GOTO 110
            DO 5110 IG = 1, NGX(ML,JTY,1)
                  ISTA = KK +1
                  IFIN = NEX(ML,JTY,1,IG)
                  DO 6110 I = ISTA,IFIN
                        N1 = ND1(1,I) +NIN2S
                        N2 = ND1(2,I) +NIN2S
                        N3 = ND1(3,I) +NIN2S
                        N4 = ND1(4,I)
                        N5 = ND1(5,I)
                        W11 = W(1,N1)
                        W21 = W(2,N1)
                        W31 = W(3,N1)
                        W41 = W(4,N1)
                        W51 = W(5,N1)
                        P1 = W(6,N1)
                        W12 = W(1,N2)
                        W22 = W(2,N2)
                        W32 = W(3,N2)
                        W42 = W(4,N2)
                        W52 = W(5,N2)
                        P2 = W(6,N2)
                        W13 = W(1,N3)
                        W23 = W(2,N3)
                        W33 = W(3,N3)
                        W43 = W(4,N3)
                        W53 = W(5,N3)
                        P3 = W(6,N3)
                        SX = SD1(1,I)
                        SY = SD1(2,I)
                        SZ = SD1(3,I)
                        PA = P1 +P2 +P3
                        W1A = W11 +W12 +W13
                        W2A = W21 +W22 +W23
                        W3A = W31 +W32 +W33
                        W4A = W41 +W42 +W43
                        W5A = W51 +W52 +W53
                        QSA = (SX*W2A +SY*W3A +SZ*W4A)/W1A
                        FS1 = QSA*W1A
                        FS2 = QSA*W2A +PA*SX
                        FS3 = QSA*W3A +PA*SY
                        FS4 = QSA*W4A +PA*SZ
                        FS5 = QSA*(W5A +PA)
                        FW(1,N4) = FW(1,N4) +FS1*R16D3
                        FW(2,N4) = FW(2,N4) +FS2*R16D3
                        FW(3,N4) = FW(3,N4) +FS3*R16D3
                        FW(4,N4) = FW(4,N4) +FS4*R16D3
                        FW(5,N4) = FW(5,N4) +FS5*R16D3
                        FW(1,N5) = FW(1,N5) -FS1*R16D3
                        FW(2,N5) = FW(2,N5) -FS2*R16D3
                        FW(3,N5) = FW(3,N5) -FS3*R16D3
                        FW(4,N5) = FW(4,N5) -FS4*R16D3
                        FW(5,N5) = FW(5,N5) -FS5*R16D3
6110                    CONTINUE
                        KK = NEX(ML,JTY,1,IG)
5110              CONTINUE
 110    CONTINUE
```

Figure 7.12 : Calculation loop for a 5-point node from the routine EULER in ESAUNA.

The array ND1 represents the connectivity matrix for an element type that consists of 5 nodes arranged as triangular faces which may be visualised as two triangular pyramids which share a common face. The first dimension of the connectivity matrix represents the node

numbers of the element while the second dimension represents the element number. The five node numbers for each element obtained from the connectivity matrix ND1 are referred to as N1, N2, N3, N4, N5 in the calculation loop in Figure 7.12. In the calculation loop the nodes N1, N2 and N3 are used for the assignment of the values of the nodes N4 and N5. Given this information regarding the nodal relationship an inspector loop is constructed as in Figure 7.13.

```
      I=1
      IP=0
C
C     Initialise the graph.
C
      CALL CAP_CONNECT_INIT(0)
C
C     Calculate the edges of the graph taking into consideration that for ND1 :
C           Node 4 uses Nodes 1, 2 and 3; and
C           Node 5 uses Nodes 1, 2 and 3.
C
      KK    = 0
      DO 8110 JTY = 1,9
            IF(NGX(ML,JTY,1).EQ.0)GO TO 9110
            DO 85110 IG = 1,NGX(ML,JTY,1)
                  ISTA    = KK +1
                  IFIN    = NEX(ML,JTY,1,IG)
                  DO 86110 I = ISTA,IFIN
                        CALL CAP_CONNECT (ND1(4,I), ND1(1,I))
                        CALL CAP_CONNECT (ND1(4,I), ND1(2,I))
                        CALL CAP_CONNECT (ND1(4,I), ND1(3,I))
                        CALL CAP_CONNECT (ND1(5,I), ND1(1,I))
                        CALL CAP_CONNECT (ND1(5,I), ND1(2,I))
                        CALL CAP_CONNECT (ND1(5,I), ND1(3,I))
                        I=I+1
86110                   CONTINUE
                        KK    = NEX(ML,JTY,1,IG)
85110             CONTINUE
 8110   CONTINUE
C
C     Calculate and construct the communication set.
C
      CALL CAP_OVERLAP(CAP_NODPROC, CAP_NODPROC, CAP_IOVR)
```

Figure 7.13 : Inspector loop for the calculation loop in Figure 7.12.

The loops in the parallel routines still remained intact, i.e. executed over the complete serial loop range. To extract parallelism an execution control mask (Figure 7.14) was placed within these loops that would execute the calculation for the nodes owned by the processor.

The parallelisation by hand of the ESAUNA unstructured mesh code using the generic techniques and utilities discussed in Sections 7.4 and 7.5 was successfully accomplished without the need for any non-standard methods to be applied.

The initial speed up result of 3.05 for 6 processors was mildly satisfactory but could be improved. A parallel profiler was used to profile the parallel code. From the profile information it was possible to ascertain which routines were not speeding up satisfactorily. Many of the

routines parallelised contained communications that could be merged with other communication. Several of the original inspector loops employed were also merged to reduce some duplication.

Further investigation also revealed that the execution control masking statements added to the total run time. The iteration of the loop and the testing of the execution control masks for every iteration of the loop caused a high amount of overhead on each processor. Profiling of a loop within the code revealed that on 6 processors over a third of the time within the loop was due to the testing of the execution control masks.

```
      IF(ITYPE(NTYPE).EQ.0)GO TO 100
101   KK     = 0
      DO 110 JTY = 1,9
            IF(NGX(ML,JTY,1).EQ.0)GO TO 110
            DO 5110 IG = 1,NGX(ML,JTY,1)
                  ISTA    = KK +1
                  IFIN    = NEX(ML,JTY,1,IG)
C
                  DO 6110 I = ISTA,IFIN
                        N1      = ND1(1,I) +NIN2S
                        N2      = ND1(2,I) +NIN2S
                        N3      = ND1(3,I) +NIN2S
                        N4      = ND1(4,I)
                        N5      = ND1(5,I)
                        ......
                        IF(CAP_NODPROC(N4).EQ.CAP_PROCNUM) THEN
                              FW(1,N4)  = FW(1,N4) +FS1*R16D3
                              ......
                              ......
                              ......
                        ENDIF
                        IF(CAP_NODPROC(N5).EQ.CAP_PROCNUM)THEN
                              FW(1,N5)  = FW(1,N5) -FS1*R16D3
                              ......
                              ......
                              ......
                        ENDIF
6110                  CONTINUE
                      KK      = NEX(ML,JTY,1,IG)
5110            CONTINUE
110     CONTINUE
```

Figure 7.14 : The original parallel loop for Figure 7.12.

There were two possible solutions to this problem. Either the renumbering of nodes locally for each processor (as in PUIFS) or the formation of pointer arrays/indirect addressing for each processor to calculate which nodes were required on each processor for the masked loops. The latter case of pointer arrays was chosen, as this localised the code changes to the loops in question. The renumbering would have required altering the code as a whole.

The pointer list method was applied by modifying the original loop and execution control masks from the parallel routine (Figure 7.15). The loop and execution control mask were

then placed at the very start of the program after the partitioning and the inspector loops had executed. While the loop was being executed a list of the nodes used for a particular processor was stored in a pointer list (CAP_NOD_POINTER) and the start (CAP_INOD(1)) and stop (CAP_INOD(2)) range of the pointer list is stored. The loop removed from the parallel routine is then replaced by a loop (Figure 7.16) that runs from CAP_INOD(1) to CAP_INOD(2) on that processor and access the nodes stored in the pointer list CAP_NOD_POINTER.

```
C
        IP_NOD = 0
        CAP_INOD(1)=IP_NOD+1
C
        KK      = 0
        DO 9110 JTY = 1,9
                IF(NGX(ML,JTY,1).EQ.0)GO TO 9110
                DO 95110 IG = 1,NGX(ML,JTY,1)
                        ISTA    = KK +1
                        IFIN    = NEX(ML,JTY,1,IG)
                        DO 96110 I = ISTA,IFIN
                                N4      = ND1(4,I)
                                N5      = ND1(5,I)
                                IF(CAP_PROCNUM.EQ.CAP_NODPROC(N4).OR.
   &                                CAP_PROCNUM.EQ.CAP_NODPROC(N5))THEN
                                        IP_NOD=IP_NOD+1
                                        CAP_NOD_POINTER(IP_NOD)=I
                                ENDIF
96110                   CONTINUE
                        KK      = NEX(ML,JTY,I,IG)
95110           CONTINUE
 9110   CONTINUE
C
                CAP_INOD(2)= IP_NOD
```

Figure 7.15 : The pointer list initialised at start of parallel program.

```
        IF(ITYPE(NTYPE).EQ.0)GO TO 100
101     CONTINUE
        DO IJ = CAP_INOD(1),CAP_INOD(2)
                I=CAP_NOD_POINTER(IJ)
                N1 = ND1(1,I) +NIN2S
                N2 = ND1(2,I) +NIN2S
                N3 = ND1(3,I) +NIN2S
                N4 = ND1(4,I)
                N5 = ND1(5,I)
                W11 = W(1,N1)
                W21 = W(2,N1)
                ......
                IF(CAP_PROCNUM.EQ.CAP_NODPROC(N4))THEN
                        FW(1,N4) = FW(1,N4) +FS1*R16D3
                        ......
                        ......
                        ......
                ENDIF
                IF(CAP_PROCNUM.EQ.CAP_NODPROC(N5))THEN
                        FW(1,N5) = FW(1,N5) -FS1*R16D3
                        ......
                        ......
                        ......
                ENDIF
        ENDDO
```

Figure 7.16 : The improved parallel loop for Figure 7.12 using a list pointer.

Different partitioning strategies were also applied by the domain decomposition tool JOSTLE to ascertain if any improvement could be obtained. The original partitioning strategy applied was of a one-dimensional partitioning with communication between neighbouring processors. Better results were however obtained from partitioning the mesh such that any processor could communicate to any other processor, i.e. all to all communications.

No of Procs	Time Taken	Speed Up	Efficiency
1	3239	-	-
2	1759	1.84	92.1%
4	1018	3.18	79.5%
6	714	4.54	75.6%
8	584	5.55	69.3%
10	484	6.69	66.9%
12	432	7.50	62.5%
14	391	8.28	59.2%
16	364	8.90	55.6%

Table 7.5 : Results for a 15977 node problem using all to all communications on the Transtech Paramid.

When all these methods had been applied the results (Table 7.5) obtained were much more favourable with a speed up of 4.54 now obtainable for 6 processors.

7.7 The Process of Automatic Code Generation of Parallel Unstructured Mesh Codes.

The process of automatic code generation of parallel unstructured mesh codes within the Computer Aided Parallelisation Tools is very similar to automatic code generation of parallel structured mesh codes. The process of calculating the dependence analysis will be identical. The process involving the partitioning, the generation of execution masks and the calculation and generation of communications will vary slightly.

7.7.1 Partitioning.

The partitioning is accomplished by the generation of a call to JOSTLE and producing a list of the elements that each processor owns (Section 7.4.2). Since the actual number of

processors and the mesh information is not known until run time, CAPTools uses this list of the processor-element owning relationship as a symbolic integer array.

The inheritance algorithm used for the structured mesh partitioning (Section 2.6) was also used for the unstructured mesh.

7.7.2 Execution Control Masks.

The execution control masks are automatically generated by CAPTools for each statement and will also be merged together to provide a block mask that encompasses several statements that owns the same mask. For example, in Figure 7.17 a block IF mask has been applied to the new temperature calculation loop for each element. Further improvement may be obtained by renumbering the mesh or by using pointer array/indirect addressing. This block mask has not been extended to the DO ELEMENT = .. loop since this would require a renumbering of that loop to operate on the elements owned by that processor. This restriction can severely handicap the potential speed up of the parallel code (Section 7.8.3.2).

```
C
C       Calculate the new temperature for each element.
C
        DO ELEMENT = 1, NELEMENT
           IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
              NEW_TEMP(ELEMENT) = 0.0
              DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                 NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &              ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
              ENDDO
           ENDIF
        ENDDO
```

Figure 7.17 : Block execution mask applied to the simple unstructured mesh code in Figure 7.5

7.7.3 Calculation and Generation of Communications.

The calculation of the inspector loop (Figure 7.18) for the code in Figure 7.17 requested by the calculation of NEW_TEMP using the value of ORIG_TEMP are for the relationship between the ELETOP(CAP_ELEMENT_NEIGHBOUR, ELEMENT) and CAP_ELEMENT pair. The two surrounding loops are related to this inspector pair by true dependencies and must therefore be included in the inspector loop. The execution control mask surrounding the inspector pair must also be copied due to a control dependence on the communication requesting statement. This inspector loop (Figure 7.18) is migrated out of any further surrounding loops and

through routine boundaries to be placed immediately after the definition of the only external variable used in the inspector loop, which in this case is the variable ELETOP. Migrating this inspector loop to as far as possible provides the potential for the merging with other inspector loops that have migrated to the same point. This concept is similar to that of communication migration and merging for structured mesh codes (Section 2.8.3) where it is important that the merged communications produce the same or subset of data to be communicated.

```
      CALL CAP_CONNECT_INIT(0)
      DO CAP_ELEMENT = 1, NELEMENT
            IF(PROCESSOR_OWNS_ELEMENT(CAP_ELEMENT) .EQ. CAP_PROCNUM) THEN
                  DO CAP_ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(CAP_ELEMENT)
                        CALL CAP_CONNECT(ELETOP(CAP_ELEMENT_NEIGHBOUR,
     &                                          CAP_ELEMENT), CAP_ELEMENT)

                  ENDDO
            ENDIF
      ENDDO
```

Figure 7.18 : Inspector loop for the code in Figure 7.17.

7.8 Manual Application of Overlapping Communications for Unstructured Mesh Codes.

This section investigates the methods that may be used to apply overlapping communications to unstructured mesh codes whilst making comparisons with the methods applied for the structured mesh codes in Section 4.5 to 4.7.

As with the structured mesh codes, overlapped communications may also be advantageous for unstructured mesh codes. From the experience of asynchronous communications in ASTEC (Section 7.3.1) it is evident that these communications must be deterministic especially since CAPTools does not know about the algorithm or the problem details that can affect or prevent correctness (or at least convergence).

7.8.1 Communication Utilities for Overlapping Communications

The high level communication calls and their parameter lists are as follows :

CAP_ASWAPOVER (*Array*, *Length*, *Stride*, *Type*, *Identification_Number*)

CAP_SYNC_ASWAPOVER (*Array*, *Length*, *Stride*, *Type*, *Identification_Number*)

where the *Array* is the start address of the data to send; the *Length* is the amount of data to be communicated; the *Stride* is the distance between each block of data that may be buffered; the

Type is an integer value representing the type of data to be communicated e.g. integer, real, etc; and the *Identification_Number* that identifies the communication data set.

The CAP_SWAPOVER (Section 7.5) utility that allowed for synchronous communication, packed the data to a buffer array, communicated synchronously and unpacked the data from the buffer array. The CAP_ASWAPOVER utility, on the other hand, packs the data to a buffer array and communicates asynchronously. The CAP_SYNC_ASWAPOVER will then unpack the data from the buffer array after the asynchronous communication has been synchronised.

7.8.2 Simple Overlapping

The pseudo code in Figure 7.19a shows an example of a typical synchronous SWAPOVER communication as would be generated by CAPTools for a parallel code. The pseudo code in Figure 7.19b shows the overlapping communication of the same piece of code, as modified by hand, to take advantage of the code that does not use the communicated data to overlap the communication.

```
CALL CAP_SWAPOVER
{* Other code which does not *}
{* use communicated data *}


DO = ..
      {* Calculation using communicated data *}
ENDDO
```

Figure 7.19a : Synchronous

```
CALL CAP_ASWAPOVER
{* Other code which does not *}
{* use communicated data *}

CALL CAP_SYNC_ASWAPOVER
DO = ..
      {* Calculation using communicated data *}
ENDDO
```

Figure 7.19b : Asynchronous

Figure 7.19 : Pseudo code for Simple overlapping in an unstructured mesh code.

This method of simple overlapping follows the similar method applied to the structured mesh code (Section 4.5). The only difference being that in this case (for the unstructured mesh) the synchronisation point CAP_SYNC_ASWAPOVER will perform the unpacking of the data that was packed and communicated asynchronously by CAP_ASWAPOVER. In the synchronous version the CAP_SWAPOVER call performed the packing, synchronous communication and unpacking of data in the same call.

The application of the Simple overlapping method to the synchronous parallel code in Figure 7.11 is shown in Figure 7.20.

```
      READ*, NELEMENT
      DO ELEMENT = 1, NELEMENT
C
C           For each ELEMENT read how many neighbouring elements .
C
            READ*, NUM_OF_NEIGHBOURS(ELEMENT)
C           BROADCAST NUM_OF_NEIGHBOURS TO ALL PROCESSORS
C
C           READ THE ELEMENT TOPOLOGY AND THE ORIGINAL TEMPERATURE
C           VALUE OF EACH ELEMENT
C
            DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                  READ*, ELETOP(ELEMENT_NEIGHBOUR, ELEMENT)
C                 BROADCAST ELETOP TO ALL PROCESSORS
                  READ*,ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
C                 BROADCAST ORIG_TEMP TO ALL PROCESSORS
            ENDDO
      ENDDO
C
C     Inspector loop ..
C
      CALL CAP_CONNECT_INIT(0)
      DO CAP_ELEMENT = 1, NELEMENT
            IF(PROCESSOR_OWNS_ELEMENT(CAP_ELEMENT) .EQ. CAP_PROCNUM) THEN
                  DO CAP_ELEMENT_NEIGHBOUR = 1,NUM_OF_NEIGHBOURS(CAP_ELEMENT)
                        CALL CAP_CONNECT(ELETOP(CAP_ELEMENT_NEIGHBOUR, ELEMENT),
     &                                   CAP_ELEMENT)

                  ENDDO
            ENDIF
      ENDDO
C
C     Partition mesh using JOSTLE and create a communication set.
C
      CALL CAP_JOSTLE(CAP_P_ORIG_TEMP, 0, 0)
      CALL CAP_OVERLAP(CAP_P_ORIG_TEMP, PROCESSOR_OWNS_ELEMENT, 0, 0, CAP_ID1)
C
C     Communicate required overlap data.
C
      CALL CAP_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
          ....
      Other code
          ....
C
C     Synchronise the CAP_ASWAPOVER communication.
C
      CALL CAP_SYNC_SWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C     Calculate the new temperature NEW_TEMP for each element.
C
      DO ELEMENT = 1, NELEMENT
            IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                  NEW_TEMP = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                      ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                  ENDDO
            ENDIF
      ENDDO
```

Figure 7.20 : The application of Simple overlapping to the parallel code in Figure 7.11.

7.8.3 Unroll Overlapping

In some cases there will be (as in the structured mesh codes in Section 4.6) no other code to overlap the communication with as in Section 7.8.2. For structured mesh, unrolling the first iteration(s) of loop allowed overlapping. The analogy for unstructured meshes is to calculate data that uses the halo in an unrolled loop where the original loop only calculates data that does not use the halo. The pseudo code Figure 7.21a shows an example of a typical synchronous SWAPOVER communication with no code to overlap the communication, as would be generated by CAPTools for a parallel code. The pseudo code in Figure 7.21b shows the same piece of code modified by hand to allow overlapping communication to be applied. Once again this method is similar to that applied to the structured mesh codes in Section 4.6.

```
CALL CAP_SWAPOVER
{* No code to overlap *}
DO ...
      {* Calculate inner and outer core elements. *}
ENDO
```

Figure 7.21a : Synchronous

```
CALL CAP_ASWAPOVER
{* No code to overlap *}
DO ..
      {* Calculate inner core elements. *}
ENDDO
CALL CAP_SYNC_ASWAPOVER
DO
      {* Calculate outer core elements using communicated data. *}
ENDDO
```

Figure 7.21b : Asynchronous

Figure 7.21 : Pseudo code for Unroll overlapping in an unstructured mesh code.

However, to apply this method the inner and outer core elements need to be defined and identified. In this context the inner core represents the elements that do not require data from neighbouring elements that are owned on neighbouring processors, while the outer core represents the elements that require data from neighbouring elements on neighbouring processors.

The inner and outer core definitions may be implemented in the parallel code in three different ways, each of which has its own advantages and disadvantages. These are : pointer array/indirect addressing; mesh renumbering; and execution control masking, as discussed in the following sections.

7.8.3.1 Pointer array/indirect addressing.

For the unstructured mesh example in Figure 7.10, the inner core and outer core elements for each processor would be defined as in Table 7.6. The algorithm in Figure 7.22 calculates these two types of core elements for each processor. Each element and all its neighbouring elements are verified to ensure that the same processor owns them. If an element has at least one neighbouring element that does not have a matching processor number then that element is deemed to be an outer core element. The element number is then stored in an array called OUTER_ELEMENTS of which there are a total of NUMBER_OUTER_ELEMENTS. If all the neighbouring elements of an element are owned by the same processor then it is deemed to be an inner core element and the element number is stored in an array called INNER_ELEMENTS of which there are a total of NUMBER_INNER_ELEMENTS.

PROCESSOR NUMBER	OUTER CORE ELEMENTS	INNER CORE ELEMENTS
1	25,76,16,31,60,71	All other elements
2	63,72,68,43,29,49,6,50	All other elements
3	28,69,65,34	All other elements

Table 7.6 : Inner and outer core elements for Figure 7.10.

```
            NUMBER_OUTER_ELEMENTS = 0
            NUMBER_INNER_ELEMENTS = 0
            DO ELEMENT = 1, NELEMENT
                OUTER_ELEMENT = FALSE
                IF (PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                    DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        IF (PROCESSOR_OWNS_ELEMENT(ELEMENT_NEIGHBOUR) .NE. CAP_PROCNUM) .AND.
     &                      (.NOT. OUTER_ELEMENT) THEN
C
C                           If ELEMENT has a neighbouring element on a different processor;
C                           and a neighbouring element on a different processor has not already been
C                           found for this element;
C                           then it must be an outer core element.
C
                            NUMBER_OUTER_ELEMENTS = NUMBER_OUTER_ELEMENTS+1
                            OUTER_ELEMENTS(NUMBER_OUTER_ELEMENTS) = ELEMENT
                            OUTER_ELEMENT = TRUE
                        ENDIF
                    ENDDO
                    IF (.NOT. OUTER_ELEMENT) THEN
C
C                       ELEMENT has no neighbouring elements on a different processor.
C                       It must be an inner core element.
C
                        NUMBER_INNER_ELEMENTS = NUMBER_INNER_ELEMENTS+1
                        INNER_ELEMENTS(NUMBER_INNER_ELEMENTS) = ELEMENT
                    ENDIF
                ENDIF
            ENDDO
```

Figure 7.22 : Pseudo code to calculate the 'inner' and 'outer' core elements.

Once the inner and outer core elements have been determined, this information may be used to apply the unroll overlapping. Consider the code in Figure 7.23, which shows the main calculation loop and the associated communication for the requested ORIG_TEMP data from Figure 7.11.

```
C
C       Communicate required overlap data.
C
        CALL CAP_SWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C       Calculate the new temperature for each element.
C
        DO ELEMENT = 1, NELEMENT
                IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                        NEW_TEMP(ELEMENT) = 0.0
                        DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                                NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                                  ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                        ENDDO
                ENDIF
        ENDDO
```

Figure 7.23 : The calculation loop and associated communication from Figure 7.11.

Applying the Unroll overlapped communications requires the original CAP_SWAPOVER communication call to be transformed to the asynchronous CAP_ASWAPOVER call. The original DO ELEMENT = ... loop is then copied and placed immediately after the original loop and a call to the synchronisation point CAP_SYNC_ASWAPOVER is placed in between these two loops. Both these loop are then adjusted such that the first loop will compute the elements for the inner core while the second loop is adjusted to compute for the outer core elements. The CAP_SYNC_ASWAPOVER call ensures that the communication of the overlap data has completed before allowing the outer core elements to be computed. The final parallel code incorporating the unrolled overlapped communication is shown in Figure 7.24.

The main disadvantage of using this method is the requirement of two additional arrays : OUTER_ELEMENTS and INNER_ELEMENTS to store the pointer arrays for the outer and inner core elements. This may increase the memory size required on each processor. A further disadvantage of using this method is that it may make the parallel code look more complicated.

One of the main advantages of this method is that the change to the code is localised to the loop in question and will not affect the rest of the code as would be the case for mesh renumbering (Section 7.8.3.2).

```
      CALL CAP_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C     Calculate the new temperature for each ELEMENT of the inner core.
C     NUMBER_INNER_ELEMENTS = Total number of inner core elements.
C
      DO ELEMENT = 1, NUMBER_INNER_ELEMENTS
            IN_ELEMENT = INNER_ELEMENTS(ELEMENT)
            IF(PROCESSOR_OWNS_ELEMENT(IN_ELEMENT) .EQ.CAP_PROCNUM) THEN
                  NEW_TEMP(IN_ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(IN_ELEMENT)
                        NEW_TEMP(IN_ELEMENT) = NEW_TEMP(IN_ELEMENT) +
     &                        ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, IN_ELEMENT))
                  ENDDO
            ENDIF
      ENDDO
C
C     Synchronise the CAP_ASWAPOVER communication.
C
      CALL CAP_SYNC_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C     Calculate the new temperature for each element of the outer core.
C     NUMBER_ OUTER _ELEMENTS = Total number of outer core elements.
C
      DO ELEMENT = 1, NUMBER_OUTER_ELEMENTS
            OUT_ELEMENT = OUTER_ELEMENTS(ELEMENT)
            IF(PROCESSOR_OWNS_ELEMENT(OUT_ELEMENT) .EQ. CAP_PROCNUM) THEN
                  NEW_TEMP(OUT_ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(OUT_ELEMENT)
                        NEW_TEMP(OUT_ELEMENT) = NEW_TEMP(OUT_ELEMENT) +
     &                        ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, OUT_ELEMENT))
                  ENDDO
            ENDIF
      ENDDO
```

Figure 7.24 : Asynchronous code for pointer array / indirect addressing.

Another advantage of this method is when dynamic load balancing is required. Dynamic load balancing is required when one processor has a greater workload with respect to the other processors. This can often happen in a CFD solidification problem where a fluid metal cools and becomes solid. The calculation workload is greater for the metal section that is still fluid than for the cooled solid metal section. To balance the workload the heavily used processor distributes the work equally between the processors. This will require recalculating the inner and outer elements for each processor. This method has a lesser overhead than that for mesh renumbering (Section 7.8.3.2).

7.8.3.2 Mesh Renumbering

This involves renumbering the mesh for each processor such that each processor has its own local numbering. For example, for the mesh in Figure 7.10 the elements are renumbered as in Figure 7.25 and each processor's elements would be numbered 1 to 30 for processor 1, 1 to 32 for processor 2 and 1 to 32 for processor 3.

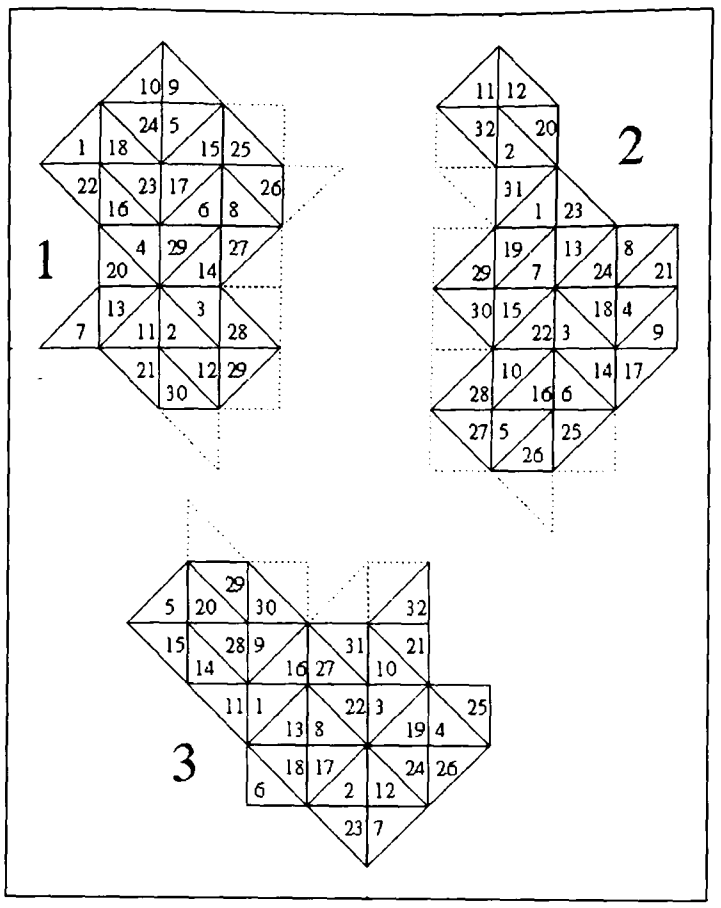


Figure 7.25 : The unstructured mesh in Figure 7.10 with mesh renumbering.

The calculation of the inner and outer core elements is determined using the algorithm in Figure 7.22. The arrays INNER_ELEMENTS and OUTER_ELEMENTS that hold the element numbers for both the inner and outer cores are used to renumber the elements in a sequential fashion. Values are also assigned to the variable LAST_INNER_CORE_ELEMENT that stores the value of the last element of the inner core and the variable LOCAL_NELEMENT that stores the total number of elements for each processor which also happens to be last element number for each processor. The values for these two variables for each processor for the unstructured mesh in Figure 7.25 are shown in Figure 7.26.

PROCESSOR NUMBER	LAST_INNER_CORE_ELEMENT	LOCAL_NELEMENT
1	24	30
2	24	32
3	28	32

Figure 7.26 : The values of LAST_INNER_CORE_ELEMENT and LOCAL_NELEMENT for the unstructured mesh in Figure 7.25.

For the synchronous version of the code the renumbering simply involve the alteration of the upper range variable NELEMENT in the DO ELEMENT = ... loop in Figure 7.23 to be replaced by the variable LOCAL_NELEMENT (Figure 7.27).

```
            CALL CAP_SWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C           Calculate the new temperature for each element.
C
            DO ELEMENT = 1, LOCAL_NELEMENT
                  NEW_TEMP(ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                        ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                  ENDDO
            ENDDO
```

Figure 7.27 : Synchronous communication using mesh renumbering.

The code example using mesh renumbering in Figure 7.27 may be transformed to use asynchronous communications (Figure 7.28) in the same way as in Section 7.8.3.1. The original CAP_SWAPOVER communication call is transformed to be the asynchronous CAP_ASWAPOVER call. The original DO ELEMENT = ... loop is copied and placed immediately after the original loop and a call to the synchronisation point CAP_SYNC_ASWAPOVER call. The loop limits to both loops are then adjusted. The first loop will calculate the values for the inner core for elements numbered 1 to LAST_INNER_CORE_ELEMENT and the second loop is adjusted to calculate the values for the outer core for elements LAST_INNER_CORE_ELEMENT+1 to LOCAL_NELEMENT.

```
            CALL CAP_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C           Calculate the new temperature for each element of the inner core.
C
            DO ELEMENT = 1, LAST_INNER_CORE_ELEMENT
                  NEW_TEMP(ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                        ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                  ENDDO
            ENDDO
C
C           Synchronise the CAP_ASWAPOVER communication.
C
            CALL CAP_SYNC_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C           Calculate the new temperature for each element of the outer core.
C
            DO ELEMENT = LAST_INNER_CORE_ELEMENT+1, LOCAL_NELEMENT
                  NEW_TEMP(ELEMENT) = 0.0
                  DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                        ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                  ENDDO
            ENDDO
```

Figure 7.28 : Asynchronous communication using mesh renumbering.

The application of mesh renumbering will not affect the solution. The parallel code for the asynchronous communication in Figure 7.28 is also more readable than that for the indirect addressing in Figure 7.24.

A potential disadvantage of mesh renumbering is that the changes applied are not localised to certain loops but affects the whole code, which in turn also affects the cache usage. Previously the elements may have been numbered such that all the elements being used in calculation were stored in an array near to each other and could therefore be held in cache memory at the same time [99]. Renumbering the mesh may now mean that different parts of the array may have to be placed in and flushed out of the cache memory.

Another disadvantage of mesh renumbering is if dynamic load balancing [100] is required. Every time this dynamic load balancing occurs, the mesh on each processor needs to be renumbered. If this dynamic load balancing occurred frequently then the renumbering would add to the parallel code runtime.

7.8.3.3 Execution control masking.

The inner and outer core elements are determined using execution control masks. The DO loop in the code section in Figure 7.23 is duplicated – one for the inner core elements calculation and one for the outer core elements, as shown in Figure 7.29. An execution control mask is placed within each loop to set the control for each loop. The control is determined by an IF...THEN statement that calls a function INNER_CORE_ELEMENT that calculates if an element is an inner core element. The execution control mask for the first loop checks for an inner core element while in the second loop checks that it is not an inner core element, i.e. and outer core element.

The main disadvantage of this method is the overhead in checking that each element owned by each processor must be verified during the first loop to determine if it is an inner core element and then must be verified again in the second loop to be an outer core element. This double-checking of all the elements owned by each processor obviously adds to the total execution time and inevitably slows down the performance of the parallel code.

```
C
C       Communicate required overlap data.
C
        CALL CAP_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C       Calculate the new temperature for each inner core element.
C
        DO ELEMENT = 1, NELEMENT
                IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                        IF (INNER_CORE_ELEMENT(ELEMENT)) THEN
                                NEW_TEMP(ELEMENT) = 0.0
                                DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                                      ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                                ENDDO
                        ENDIF
                ENDIF
        ENDDO
C
C       Synchronise the CAP_ASWAPOVER communication.
C
        CALL CAP_SYNC_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C       Calculate the new temperature for each outer core element.
C
        DO ELEMENT = 1, NELEMENT
                IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                        IF (NOT(INNER_CORE_ELEMENT(ELEMENT))) THEN
                                NEW_TEMP(ELEMENT) = 0.0
                                DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                                        NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                                      ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                                ENDDO
                        ENDIF
                ENDIF
        ENDDO
```

```
        LOGICAL FUNCTION INNER_CORE_ELEMENT(ELEMENT)
C
C       Function to calculate if an ELEMENT is an inner core element.
C
        INNER_CORE_ELEMENT = TRUE
        DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
            IF (PROCESSOR_OWNS_ELEMENT(ELEMENT_NEIGHBOUR) .NE. CAP_PROCNUM)  THEN
C
C               If ELEMENT has a neighbouring element not on the same
C               processor then it cannot be an inner core element.
C
                INNER_CORE_ELEMENT = FALSE
            ENDIF
        ENDDO
```

Figure 7.29 : Asynchronous communication using execution control masks.

7.8.3.4 Summary.

Table 7.7 shows a summary of the advantages and disadvantages of each method that may be used to implement the definition of the inner and outer core of the unstructured mesh. The first column represents the three different methods previously discussed : the pointer array/indirect addressing (Section 7.8.3.1), mesh renumbering (Section 7.8.3.2) and the execution control masking (Section 7.8.3.3). Each column represents:

- the amount of overhead for the method;
- the changes applied are localised to certain loops;
- the effect on the cache;
- the alteration of the code appearance;
- the amount of memory required;
- suitability for Dynamic Load Balancing (DLB);
- and its flexibility for multiple loop definitions of different outer and inner core.

Method	Overhead	Localised	Cache	Code Look	Memory	DLB	Multi Loop
Pointer	Medium	Yes	Good	Bad	High	Bad	Good
Renumber	Low	No	Bad	Good	Low	Bad	Bad
Masking	High	Yes	Good	Bad	Low	Good	Good

Table 7.7 : Summary of the suitability of the methods in relation to each other to implement the inner and outer core definitions.

For the mesh renumbering the overhead of this method is low, the amount of alteration to the code appearance is minimal and the additional memory required is low. However, the changes applied are not localised to certain loops but affects the whole code, which in turn also affects the cache usage. It is, however, not suitable for applying the dynamic load balancing algorithm as this would require the mesh to be renumbered each time the load is rebalanced. It is also not flexible for the multiple definitions of outer and inner core.

For the execution control masking method the changes to the code are localised to the loops concerned and the cache usage will be good. The additional memory usage is low, it is also flexible for the multiple definitions of outer and inner core and lastly it is more suitable for the use of the dynamic load balancing algorithm. However, the overhead of the method is high which thus rules out this method.

For the pointer array/indirect addressing the overhead of the method is higher than the mesh renumbering method but less than that for the execution control masking. It is also more flexible to deal with multiple definitions of different outer and inner core definitions than the mesh renumbering method. Even though this method has the disadvantages of altering the appearance of the code significantly, requires additional memory requirements

and also not the most suitable for applying the dynamic load balancing algorithm it is still the most suitable given the disadvantages of the other two methods.

7.8.4 Partial Overlapping.

The pseudo code in Figure 7.30a once again shows an example of a typical synchronous SWAPOVER communication with no code to overlap the communication. The pseudo code in Figure 7.30b shows the same piece of code modified by hand to allow overlapping communication to be applied by employing a conditional statement. Once again this method is similar to that applied to the structured mesh codes in Section 4.7.

```
CALL CAP_SWAPOVER
{* No code to overlap *}
DO ...


        {*Calculate inner and outer core *}
ENDO
```

Figure 7.30a : Synchronous

```
CALL CAP_ASWAPOVER
{* No code to overlap *}
DO ..
        IF (inner core finished ) THEN
                CALL CAP_SYNC_ASWAPOVER
        ENDIF
        {* Calculate inner and outer core *}
ENDDO
```

Figure 7.30b : Asynchronous

Figure 7.30 : Pseudo code for Partial overlapping in an unstructured mesh code.

```
        CALL CAP_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
C
C       Calculate the new temperature for each element.
C
        DO ELEMENT = 1, LOCAL_NELEMENT
                IF (ELEMENT .EQ. LAST_INNER_CORE_ELEMENT+1) THEN
C
C                       All the inner core elements have been calculated. Synchronise asynchronous
C                       communication before calculating the outer core elements.
C
                        CALL CAP_SYNC_ASWAPOVER(ORIG_TEMP(1),1,1,2,CAP_ID1)
                ENDIF
                IF(PROCESSOR_OWNS_ELEMENT(ELEMENT) .EQ. CAP_PROCNUM) THEN
                        NEW_TEMP(ELEMENT) = 0.0
                        DO ELEMENT_NEIGHBOUR = 1, NUM_OF_NEIGHBOURS(ELEMENT)
                                NEW_TEMP(ELEMENT) = NEW_TEMP(ELEMENT) +
     &                                  ORIG_TEMP(ELETOP(ELEMENT_NEIGHBOUR, ELEMENT))
                        ENDDO
                ENDIF
        ENDDO
```

Figure 7.31 : Partial Overlapping with mesh renumbering.

The adaptation of the synchronous code in Figure 7.23 employing the conditional statement, shown in Figure 7.31, requires mesh renumbering. The conditional statement verifies that the inner core elements have been calculated by ensuring that the value of ELEMENT is equal to the value of the LAST_INNER_CORE_ELEMENT+1 (the first element in the outer

core). When this conditional is true the synchronisation of the asynchronous communication is imperative to ensure that the correct communicated data is available for calculation.

7.9 Manual Application of Overlapping Communications to Unstructured Mesh Codes.

7.9.1 Manual Application of Overlapping Communications to PUIFS.

Overlapping communications have been applied to the PUIFS code (Section 7.3.2) to improve its efficiency. This was accomplished by renumbering the elements on each processor such that the ones on the boundary with other processors were calculated first (as in Section 7.8.3.2) . Once these boundary elements were calculated they could be communicated while at the same time the remaining elements may be calculated. When the remaining elements have been calculated the communication must be synchronised to ensure it has completed before allowing the next iteration to be calculated. The nodes have been renumbered to allow the boundary elements to be calculated first. This method is similar to that applied for the ASTEC code in Section 7.3.1, the only difference being that there is a synchronisation point to ensure the latest up-to-date data is used for the next iteration, i.e. deterministic.

The application of asynchronous communications for a fluid dynamic test case and a solid mechanics test case are shown in the graphs in Figure 7.32 and Figure 7.33 respectively. The results were obtained from the Transtech Paramid [81]. The graphs show that the application of the asynchronous communication has improved the performance of the parallel code. One exception is for the 3034 and 10027 element test cases in Figure 7.32 where the performance of the asynchronous communications is trailing behind that of the synchronous communications as the number of processors increase. This is more likely due to the small problem sizes and the large number of processors preventing sufficient amounts of calculation to be overlapped with the communications. The results for the 119822 element test case for the solid mechanics code on 28 processors (Figure 7.33) shows a significant increase in the speed up from approximately 15 to approximately 20.

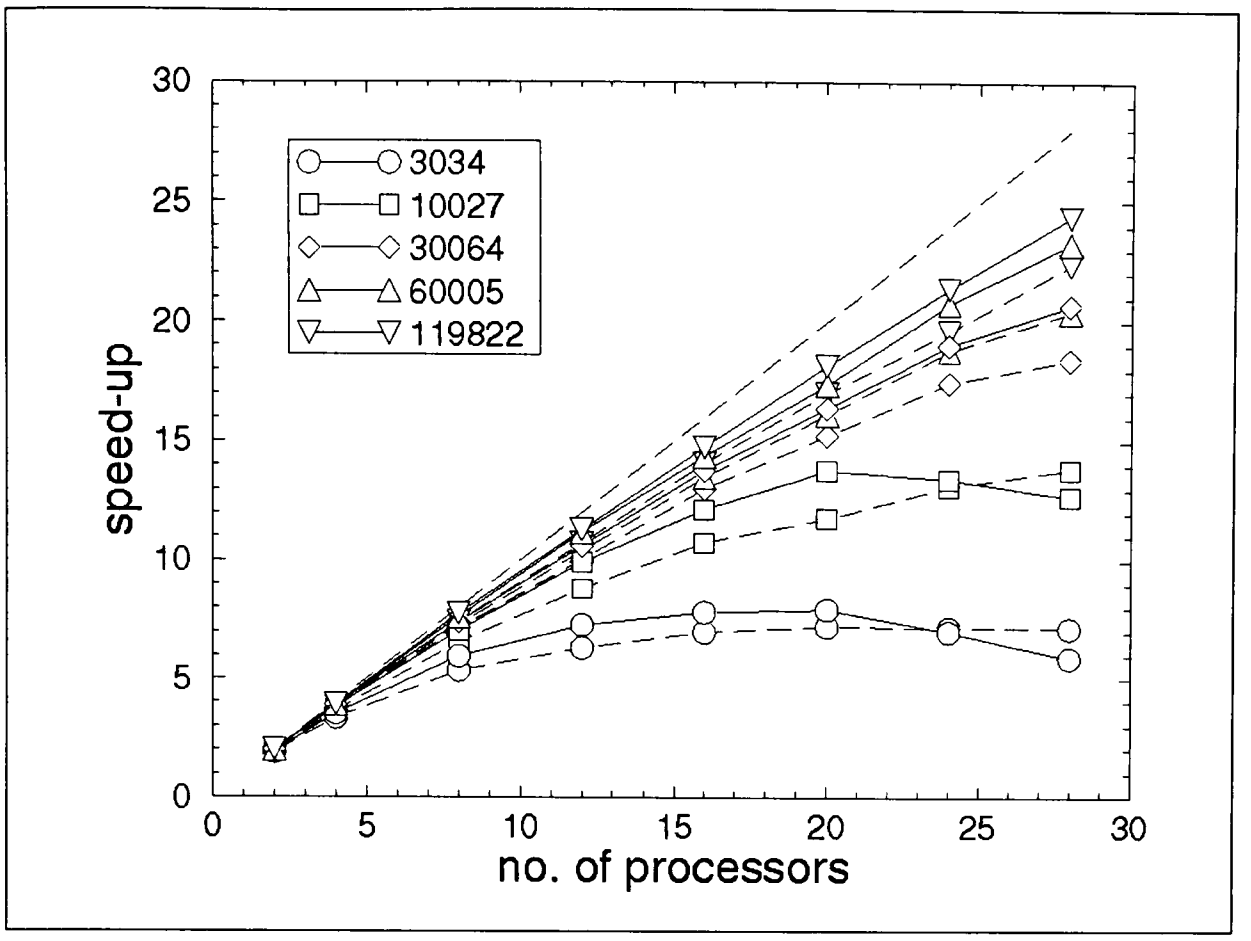


Figure 7.32 : Speed up obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimised solvers for the fluid dynamic test case with a range of mesh sizes.

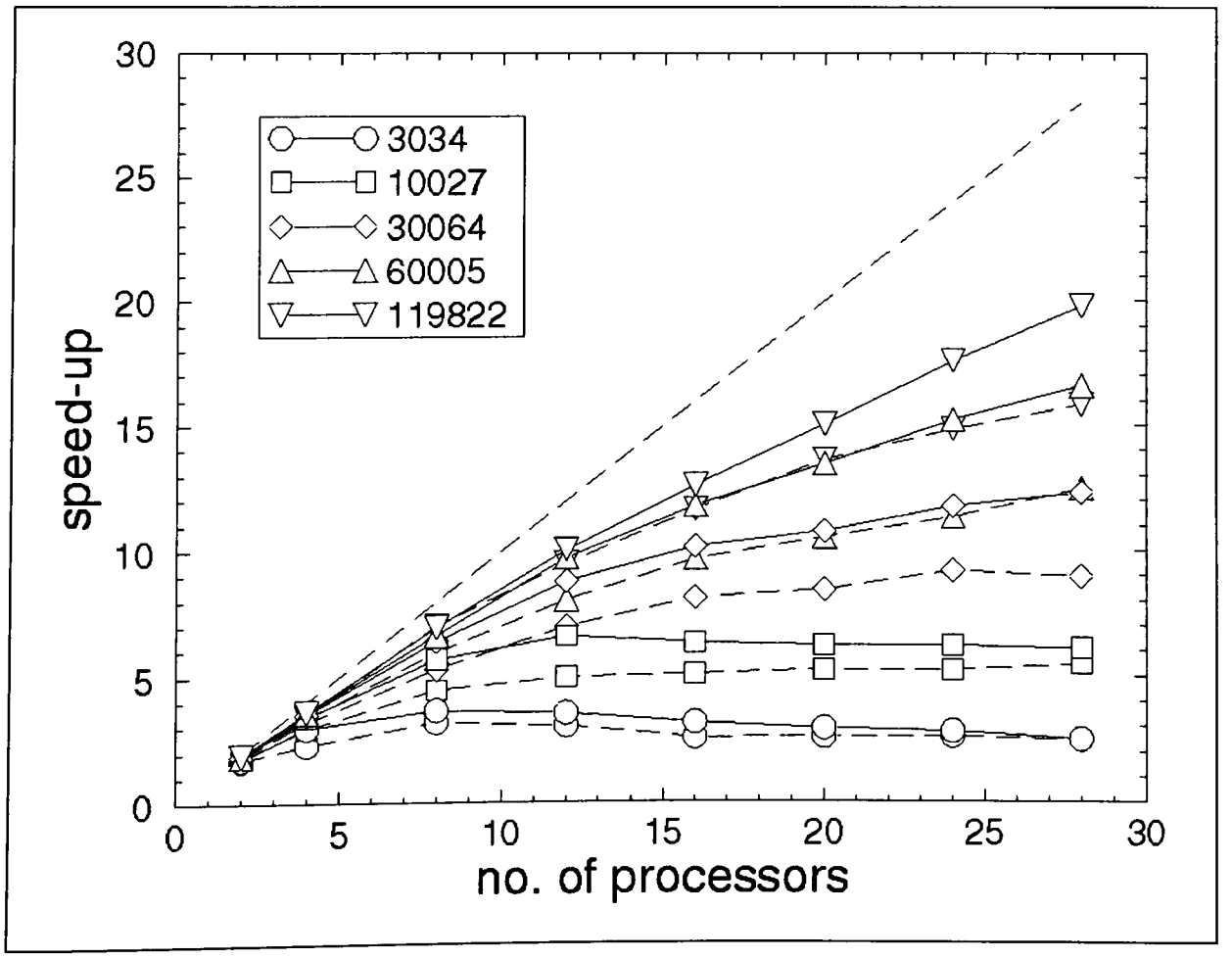


Figure 7.33 : Speed up obtained with the asynchronous (solid lines) and synchronous (dashed lines) optimised solvers for the solid mechanics test case with a range of mesh sizes.

7.10 Automatic Generation of Overlapping Communications for Unstructured Meshes.

The methods of overlapping communications have not been incorporated as an automatic stage within CAPTools because the development of the automatic generation of parallel code for unstructured mesh is still in progress. It is envisaged that these methods will be implemented in a similar fashion to those required for the structured mesh codes (Chapter 5)

7.11 Summary of the Applied Overlapping Communication Methods.

The method of asynchronous communication applied in ASTEC (Section 7.3.1) with no synchronisation points is clearly not the best method to apply. This method is, as previously mentioned, non-deterministic and causes different results to be obtained. There is also the possibility of non-convergence of the problem in parallel which otherwise would in serial. This method may be improved by changing the algorithm. However, this does not allow a generic method to be developed.

The method applied in PUIFS (Section 7.3.2), with the use of synchronisation points to ensure the most recently available data is used for calculation, provided identical results and allowed the parallel code to behave in the same manner as the serial code. This method is also generic and may be developed as an automated method of parallelisation of unstructured mesh codes with overlapped communication within Computer Aided Parallelisation Tools.

Based on these conclusions the latter method applied in PUIFS will be used in the development of automatic code generation of parallel unstructured mesh codes with overlapped communication within Computer Aided Parallelisation Tools.

7.12 Conclusions.

Investigation of the parallelisation of numerous other unstructured mesh codes has shown that an effective, portable, generic and automatable parallelisation strategy for unstructured mesh has been devised and verified. This has subsequently led to unstructured mesh parallel code generation to be developed within CAPTools.

A manual investigation of the methods applied for the overlapping of unstructured mesh codes are similar to those applied to the structured mesh codes (Chapter 4). These methods of

overlapping will be automatically generated within CAPTools when the development of parallel code for unstructured meshes is completed.

Chapter 8

8 Conclusions.

8.1 Conclusions.

One of the aims of this thesis was to obtain efficient parallel codes from using a parallelisation tool such as CAPTools. CAPTools was used to parallelise four different structured mesh codes. The generated parallel code provided correct results with satisfactory speed up results obtainable. However, additional increase in speed up was obtained by further optimisation of the code. The tools aided in this process of identifying sections of code that may be optimised. While optimising these codes it was identified that the synchronous communications employed affected the possible speed up that could be obtained. Synchronous communications requires that the communication has been completed before allowing the remainder of the code to continue.

The possibility of using overlapped communications was investigated to establish if any further improvement in the speed up could be obtained from the codes. Four different methods were conceived to take advantage of overlapped communications. Three of these methods were concerned with the overlapping of exchange communications, while the other was concerned with increasing the performance of pipelines. These four methods were tested individually on different sections of code and a satisfactory increase in speed up was obtained. These tests justified that there was an advantage to using overlapped communications as opposed to synchronous communications.

The generation of overlapped communications was incorporated as an additional step within CAPTools. This allowed the overlapped communications to be generated automatically without the user having to manually change the synchronous communications to be overlapped. The four methods tested were applied successfully and correct code was obtained.

The results (Chapter 6) for the use of overlapped communications as opposed to synchronous communications provided a beneficial increase in the efficiencies of the code.

Investigation of the parallelisation of unstructured mesh codes concluded that it was possible to use the same strategies and methods as applied for the structured mesh codes.

8.2 Requirements of Parallel Processing.

In Section 1.3 five different objectives were set:

1. Minimise the changes to the original algorithm;
2. Code remains recognisable;
3. Maximise the invisibility of the parallel code;
4. Maximise the parallel efficiency;
5. Efficient use of all available memory.

All five of these objectives were met. The changes to the original algorithms were minimal. These changes to the algorithm occurred for the FAB and TeamKE1 codes where it was required to change the algorithm to operate as a Jacobi solver as opposed to a Gauss-Seidal solver (Section 3.2 and 3.3). This allowed parallelism to be obtained from the algorithm.

The changes to the code were also minimal allowing the parallel code to be still recognisable to the original serial code author. The only major change/addition to the code was caused by the use of unrolling loops when applying the overlapped communications (Section 4.6). This method is only applied when the other two methods SIMPLE (Section 4.5) and PARTIAL (Section 4.7) may not be applied. It is therefore a last resort method and is only applied when the other two methods are inappropriate.

The user does not notice any difference in the running of the code apart from having to supply information on the required number of processors. However, the user would notice a decrease in the time taken to run these codes as the number of processors increase.

The parallel codes obtained from CAPTools produced a significant increase in the speed up in relation to the serial code. Chapter 6 demonstrates this successfully for several codes.

The maximum size problems were run on these machines to allow the maximum use of the available memory.

8.3 Finale

The application of the automatic generation of overlapping communications from within CAPTools has provided a marked increase in the efficiencies of the codes tested. The method is

completely automatic allowing the user to benefit from improved performance from their parallel codes by the press of a single button. Hopefully, newly emerging hardware will be able to effectively exploit the obvious advantages of asynchronous communications.

Appendix A: Porting of Astec.

The proposition was to port the parallel code to operate on the parallel system (i860/T800 hybrid) available at the University of Greenwich. This parallel machine did not at the time have asynchronous communication capabilities or for any processor to communicate to any other processor. It was therefore necessary to port the code to operate with synchronous communications and also to create a communication harness/router to allow the communication of data to any other processor.

Much effort was placed into the development of the communication harness that did not require such vast amounts of memory to buffer the data, since the communications were now synchronous. The router was specially adapted from a router [49] previously written at the University of Greenwich for implementing asynchronous communications on the Transtech Paramid [81]. The router (Figure A.1) consisted of two distinct areas : the routing algorithm (router) and a protocol to buffer the data to be routed (arpbuff). These two areas consisted of several programs running concurrently (known as threads) on each processor. These threads executed on the T800 processor of the parallel machine while the parallel ASTEC code executed on the i860 processor.

Figure A.2 shows how these threads operate in routing a communication to the correct processor.

The ARPBUFF algorithm consists of two threads and the ROUTER algorithm consists of six threads. These threads are as follows:

1. Client_ Thread;
2. ARP_Thread;
3. Rtr_ARP_Client_Thread;
4. Rtr_Processor_Thread;
5. Rtr_Direction_Thread for packets from its North;
6. Rtr_Direction_Thread for packets from its South;
7. Rtr_Direction_Thread for packets from its East;
8. Rtr_Direction_Thread for packets from its West;

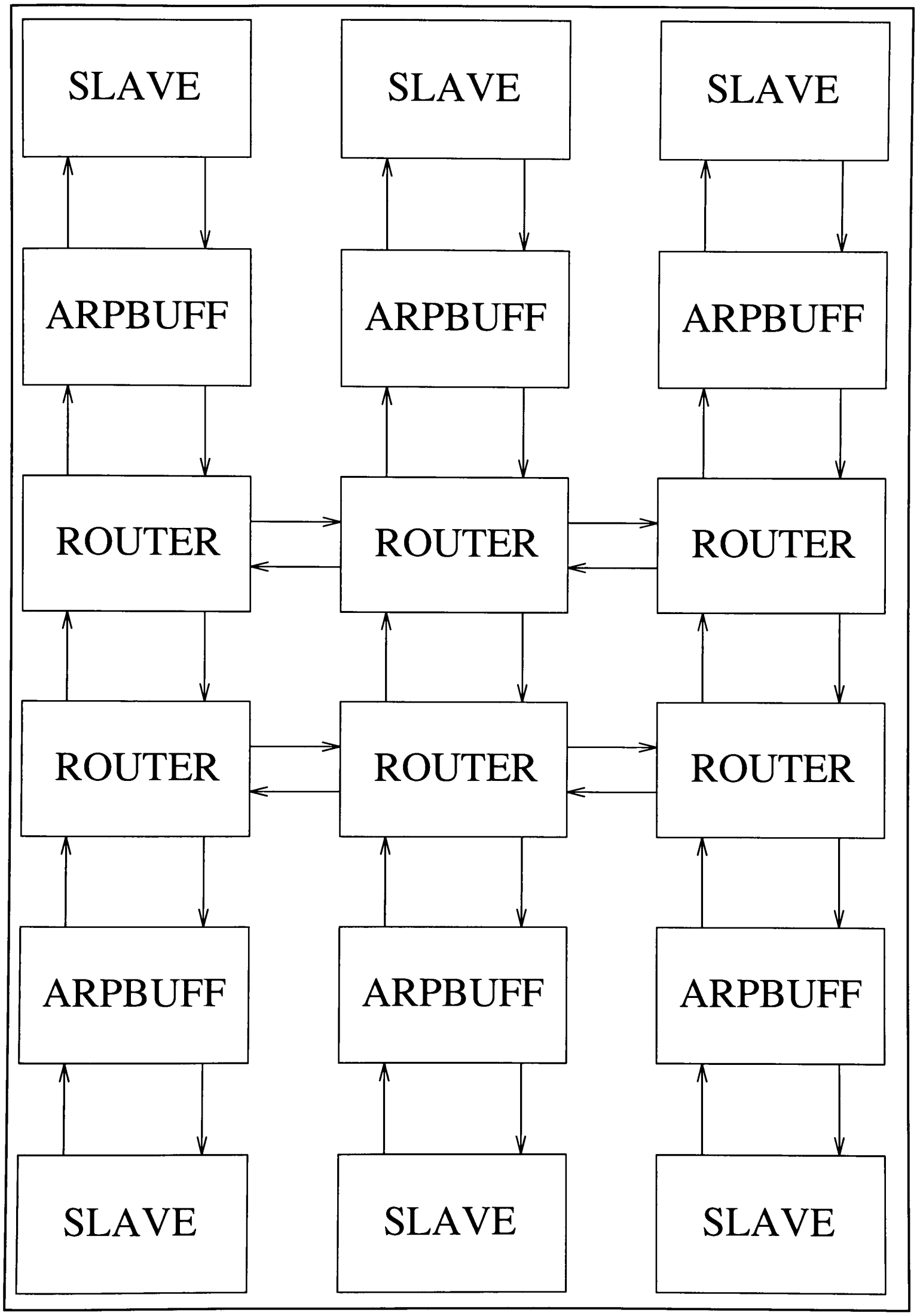


Figure A.1: Example of an array of processors using a router.

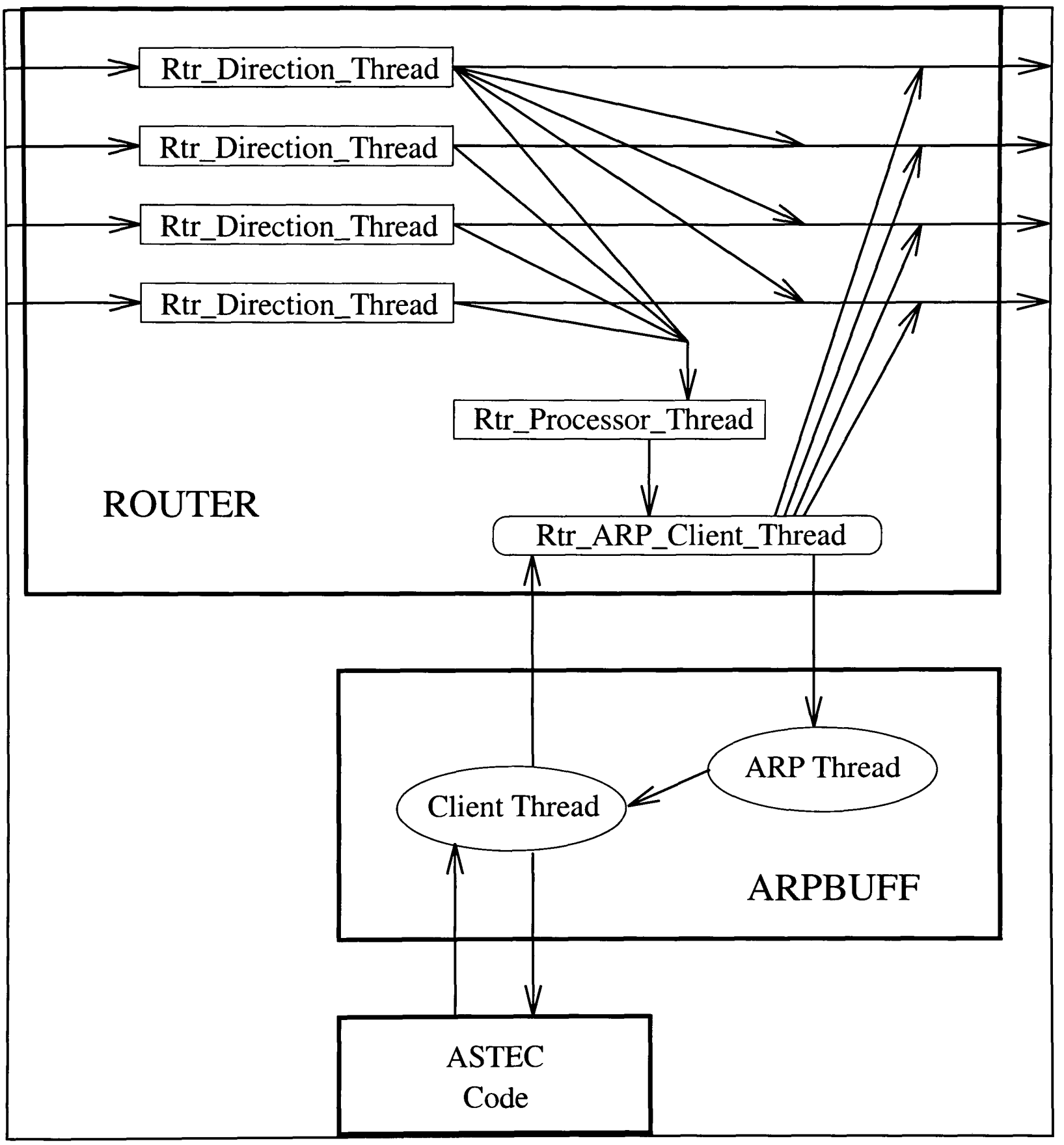


Figure A.2 : The threads used to route data.

The Client_Thread will receive communication data from the ASTEC code along with the amount of data and the processor requiring the data. This data will then be packaged into a packet that is sent to the Rtr_ARP_Client_Thread. The Client_Thread will also receive packets from the ARP_Thread before unpackaging and passing the communicated data to the ASTEC code.

The ARP_Thread store packets received from the Rtr_ARP_Client_Thread until the Client_Thread is able to receive the packet.

The Rtr_ARP_Client_Thread receives packets from the Client_Thread and calculates which direction to send the packet, i.e. north, east, west or south. This thread also receives packets from the Rtr_Processor Thread and passes them onto the ARP_Thread.

The Rtr_Processor_Thread receives in-coming packets from the Rtr_Direction_Threads and stores them until the Rtr_ARP_Client_Thread is free to deal with the packet.

There are four Rtr_Direction_Threads : one for each four possible directions (north, east, west or south) which a packet may be received. These threads determine if this processor requires the packet. If it is then it will be passed to the Rtr_Processor_Thread. If the packet is not for this processor then the thread will determine the direction in which the packet must be sent, i.e. north, east, west or south.

Figure A.3 shows a typical route which data from the ASTEC code may take when being communicated to another processor. The communicated data is passed from the ASTEC code to the client thread that packages the communicated data into a packet along with the processor number that requires the communication and the communication length. The packet will then be passed to the Rtr_ARP_Client_Thread that will route the packet to the processor to its North. The next processor will then receive the packet into the Rtr_Direction_Thread from the north. This thread deduces that the communication is not for this processor and routes it to the processor to its East. The next processor will then receive the packet into the Rtr_Direction_Thread from the east. This thread deduces that the communication is for this processor and sends the packet to the Rtr_Processor_Thread, Rtr_ARP_Client_Thread, ARP_Thread and the Client_Thread where the packet is unpackaged and the data passed to the ASTEC code on this processor.

The router was thoroughly independently tested before being integrated into the ported parallel ASTEC code.

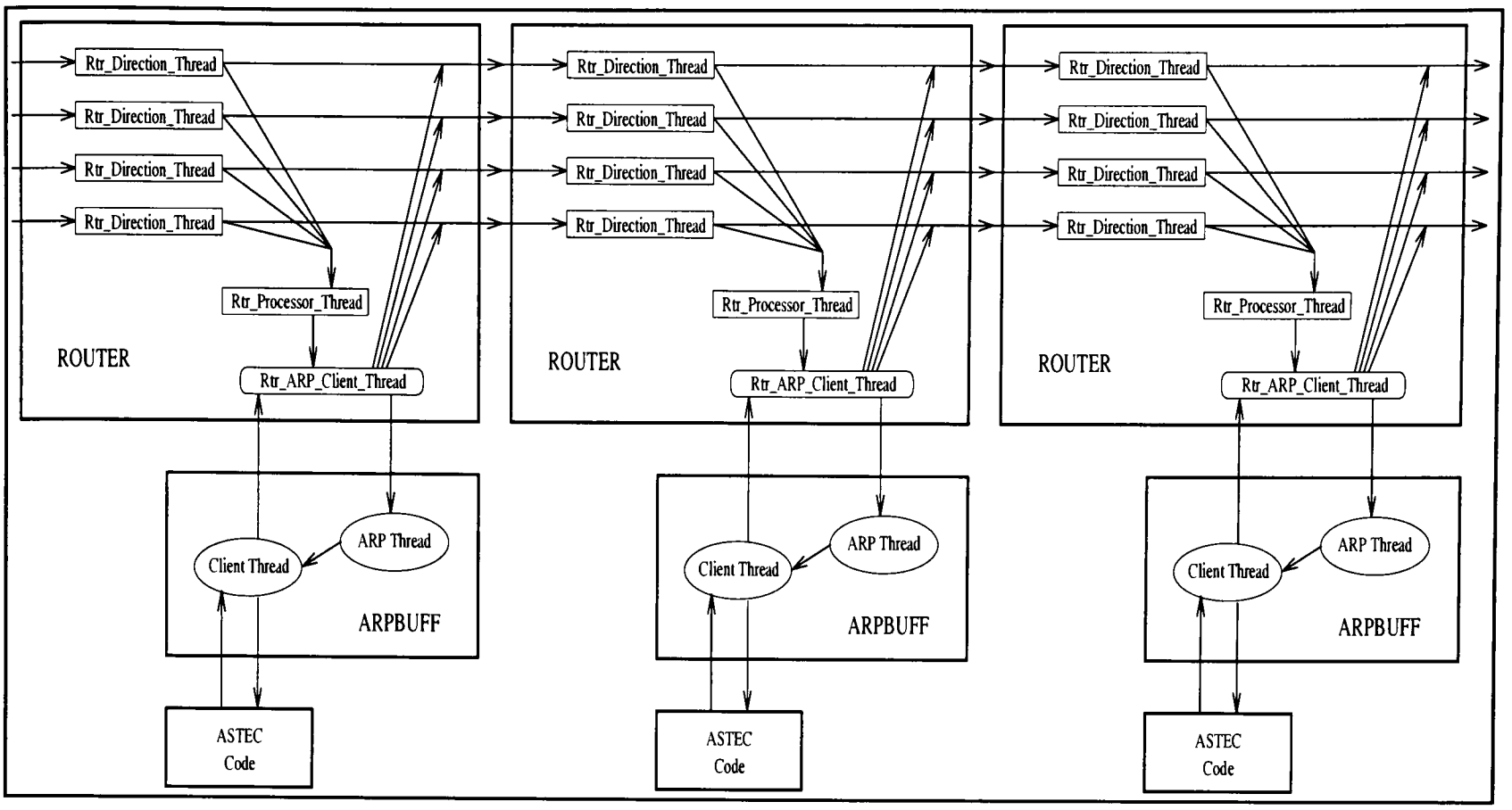


Figure A.3 : A typical route taken in the router.

Bibliography

1 Hockey R.W. and Jessope C.R. Parallel Computers: architectures, programming and algorithms. Adam Hilger, Bristol, 1988.

2 Ierotheou C.S. The Simulation of Fluid Flow Processes Using Vector Processors. PhD Thesis, Thames Polytechnic, 1990.

3 David Callahan, David Levine and Jack Dongarra. A comparative study of automatic vectorising compilers. Parallel Computing, 17:1223-1244, 1993.

4 Michael J Flynn. Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21:948-960, 1972.

5 Dennis Parkinson. Practical parallel processors and thier uses. In D.J.Evans, editor, Parallel Processing Systems, pages 216-236.Cambridge University Press, 1982.

6 Inmos Limited, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, UK.

7 George S. Almasi and Allan Gottlieb. Highly Parallel Computing, 2nd Edition. Benjamin/Cummings, Redwood City, 1994.

8 Trew A. and Wilson G., editors . Past, Present, parallel: A survey of Available Parallel computer Systems. Springer-Verlag, 1991.

9 van der Steen A.J. and Dongarra J.J. Overview of Recent Supercomputers. sixth Edition. NHSE Review 1996 Volume First Edition.

10 Johnson S.P. and Cross M. Mapping structured grid three-dimensional CFD codes onto parallel architectures. App. Math Modelling 15. 1991.

11 Hall M.W., Anderson J.M., Amarasinghe S.P., Murphy B.R., Liao S-W., Bugnion E., Lam M.S. Maximizing multiprocessor Performance with the SUIF Compiler. IEEE Computer. December 1996

12 Kuck D.J., Kuhn R.H., Leasure B.R., Wolfe M.J., The Structure of an Advanced Retargetable Vectorizer, Hwang K. (Ed.): Supercomputers: Design and Applications Tutorial, 967-974, IEEE Catalog Number EH0219-6, IEEE Society Press, Silver Spring MD (1984).

13 KAP, Kuck and Associates Inc, Champaign, Illinois, USA.

14 Allan J.R. and Kennedy K., PFC: A Program to Convert FORTRAN To Parallel Form, Proc IBM Conf Parallel Computing and Scientific Computations (1982).

15 Fox G.C., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Tseng C. and Wu M., Fortran D language specification, Technical Report TR90-141, Dept. of Computer Science, Rice University (Dec. 1990, revised April 1991).

16 Lam M., Locality Optimisations for Parallel Machines, Parallel Processing: CONPAR 94 - VAPP VI, Linz, Austria, Springer-Verlag (Sept. 1994).

17 Zima H.P., Bast H-J. and Gerndt H.M., SUPERB - A Tool for Semi-Automatic MIMD/SIMD Parallelisation, Parallel Computing, 6, pp 1-18 (1988).

18 Chapman B., Mehrotra P. and Zima H.P., Programming in Vienna Fortran. Scientific Programming 1(1), pp 31-50 (1992).

19 Banerjee P., Chandy J.A., Gupta M., Hodges IV E.W., Holm J.G., Lain A., Palermo D.J., Ramaswamy S. and Su E., An Overview of the PARADIGM Compiler for Distributed-Memory Multicomputers, IEEE Computer Volume 2, Number 10 (1995).

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

21 Michael Frumkin, Michelle Hribar, Haoqiang Jin, Abdul Waheed and Jerry Yan, A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000, in Proceedings from SC98, Orlando, Florida, November 8-13,1998.

22 FORGE90, Applied Parallel Research, Placerville, California 95667, USA.

23 Chapman B.M., Benkner S., Blasko R., Brezany P., Egg M., Fahringer T., Gerndt H.M., Hulman J., Kutschera P., Moritsch H., Schwald A., Sipkova V. and Zima H.P. Vienna Fortran Compilation System Version 1.0 User's Guide. 1994.

24 Adve V., Carle A., Granston E., Hiranandani S., Kennedy K., Koelbel C., Kremer U., Mellor-Crummey J., Tseng C-W. and Warren S. Requirements for Data-Parallel Programming Environments. 1994.

25 Banerjee P., Chandy J.A., Gupta M., Hodges IV E.W., Holm J.G., Lain A., Palermo D.J., Ramaswamy S. and Su E., An Overview of the PARADIGM Compiler for Distributed-Memory Multicomputers, IEEE Computer Volume 2, Number 10 (1995).

26 Ierotheou C.S., Johnson S.P., Cross M and Leggett P.F. Computer Aided Parallelisation Tools (CAPTools) - Conceptual Overview and Performance On The Parallelisation of Structured Mesh Codes. Parallel Computing. March 1996.

27 Johnson S.P., Cross M. and Everett M. Exploitation Of Symbolic Information In Interprocedural Dependence Analysis. Parallel Computing. March 1996.

28 Johnson S.P., Ierotheou C.S. and Cross M. Automatic Parallel Code Generation For Message Passing On Distributed Memory Systems. Parallel Computing. March 1996.

29 Leggett P.F., Marsh A.T.J., Johnson S.P and Cross M. Integrating User Knowledge With Information From Parallelisation Tools To Facilitate The Automatic Generation Of Efficient Parallel FORTRAN Code. Parallel Computing. March 1996.

30 Hall M.W., Harvey T., Kennedy K., McIntosh N., McKinley K., Oldham J., Paleczny M. and Roth G. Experiences Using the ParaScope Editor : An Interactive Parallel Programming Tool. In Proceedings of the Symposium on Principles and Practice of Parallel Programming, pp 33-43. May 1992.

31 Zima H.P., Bast H.J. and Gerndt M. SUPERB: A Tool For Semi Automatic MIMD/SIMD Parallelisation, pp1-18 Parallel Computing, 6, North-Holland. 1988.

32 Foster, I. Designing and Building Parallel Programs. Addison-Wesley, 1995.

33 Sparrow E.M. and Cess R.D. Radiation Heat Transfer. Hemisphere. 1978.

34 Leggett P., Cross M. and Johnson S. Implementing Casting Solidification Codes on Parallel Computers. Proceedings of the 18th Annual Automotive Materials Symposium, pp 267-277. May 1991.

35 Ikushima T. et al. Thermal radiation view factor calculation using Monte Carlo method. J. Atomic Energy Soc. Japan, 30, 548 (1988)

36 Evans E.W. Parallel strategies for solving systems of linear equations. Final Year Project. University of Greenwich. May 1992

37 Chu E. and George A. Gaussian Elimination With Partial Pivoting and Load Balancing On A Multiprocessor, pp54-65, Parallel Computing, 5, North Holland. 1986.

38 Farhat C., A simple and efficient automatic FEM domain decomposer. Computers and Structures, 28:579-602. 1988.

39 Simon H.D. Partitioning of unstructured problems for parallel processing. Computing Systems in Engineering, 2(2/3):135-148. 1991.

40 Pothen A., Simon H.D. , and Liu K. P. Partitioning sparse matrices with eigenvectors of graphs. Technical Report RNR-89-009, NASA Ames Research Centre, July 1989.

41 Barnard S.T. and Simon H.D. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In Proceedings 6th SIAM Conference, pp 711-718. 1993

42 Pellegrini F. and Roman J. SCOTCH : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. High-Performance Computing and Networking, Proc. HPCN'96, Brussels. Springer, Volume 1067, Pages 493-498. 1996

43 Karypis G. and Kumar V. Multilevel k-way partitioning scheme for irregular graphs. Journal Par. Dist. Comput, Pages 96-129, Volume 48 Number 1. 1998

44 Hendrickson B. and Leland R. An improved spectral graph partitioning algorithm for mapping parallel computations. SIAM J. Sci. Stat. Comput. Volume 16. 1995

45 Walshaw C. A parallelisable algorithm for optimising unstructured mesh partitions. Technical Report P95/IM/03, School of Computing and Mathematical Science. January 1995.

46 Walshaw C., Cross M., Everett M.G., Johnson S. and McManus K. Partitioning and mapping of unstructured meshes to parallel machine topologies. Int. J. Supercomputing Applications. 1995.

47 McManus K., Walshaw C., Cross M., Leggett P., Johnson S. Evalaution of the JOSTLE mesh partitioning code for practical multiphysics appliations. In Proceedings PCFD'95. 1995.

48 Computer Aided Parallelisation Tools (CAPTools) User Manual Release 1.0, University of Greenwich. 1994

49 Leggett P.F. CAPTools Communications Library. Technical Report. University of Greenwich. 1995.

50 Cray Research Inc. SHMEM Technical Note for C. SG-2516 2.3. October 1994.

51 Geist A., Beguelin A., Dongarra J., Jiang W., Nanchek R., Sunderam V. PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. MIT Press.

52 Message Passing Interface Forum. The MPI Message passing Interface Standard, Technical Report, University of Tennessee, Knoxville. 1994

53 Aho A.V. and Ullman J.D., Principals Of Compiler Design, Addison Wesley. 1977.

54 Ferrante J., Ottenstein K.J. and Warren J.D., The Program Dependence Graph And Its Use In Optimisation, pp 319-349, ACM Transactions On Programming Languages And Systems 9. 1987.

55 Kuck D.J., The Structure Of Computers And Computations, Vol 1, Wiley, New York.

1978.

56 Zima H. P. and Chapman B., Supercompilers for parallel and vector computers (Addison Wesley, 1990)

57 Highighat M. and Polychronopoulos C., Symbolic program analysis and optimisation for parallelising compilers, in Proc. Languages and Compilers for Parallel Computing 5th International Workshop, New Haven, CT (Aug 1992).

58 Allen J.R. and Kennedy K. Automatic translation of Fortran programs to vector form. ACM Trans. Programming Languages Systems 9 491-542. 1987.

59 Banerjee U. Speedup of ordinary programs. PhD Thesis, University of Illinois at Urbana Champaign. 1979.

60 Banerjee U. Dependence Analysis for Supercomputing. Kluwer Academic. 1988.

61 Johnson S.P. Mapping numerical software onto distributed memory parallel systems. PhD Thesis, University of Greenwich. 1992

62 Frost R.A. Introduction to Knowledge Based Systems. Collins Professional and Technical Series, London. 1986.

63 Pugh W. A practical algorithm for exact array dependence analysis. Commun. ACM 35 (8) pp102-114. 1992

64 Maydan D.E., Hennessy J.L., Lam M.S. Efficient and exact data dependence analysis. In ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation, Toronto, Canada, 1-14. June 1991.

65 Wolfe M. and Tseng C-W. The power test for data dependence. IEEE Trans. Parallel and Distributed Systems 3(5) pp591-601. 1992.

66 Thomas L.H. Elliptic Problems in Linear Differeence Equations Over A Network, Watson Scientific Computing Lab. Report. Columbia University, New York. 1949.

67 http://www.hensa.ac.uk/ftp/pub/misc/cfd/software/team

68 Saphir W., Woo A. and Yarrow M., NAS Parallel Benchmarks 2.1 Results. Technical Report NAS-96-010, NASA Ames Research Center. August 1996.

69 Adve V.S., Koelbel C., Mellor-Crummey J.M., Performance Analysis of Data-Parallel Programs, Technical Report CRPC-TR94405, Center for Research on Parallel Computation, Rice University (1994).

70 Berry M. et al, The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. CSRD Technical Report 827, Center for Supercomputing

Research and Development, University of Illinois. May 1989.

71 Evans E.W., Johnson S.P., Leggett P.F., Cross M. Automatic code generation of overlapped communications in a parallelisation tool. Parallel Computing 23, pp 1493-1523. 1997.

72 Bertsekas D.P. and Tsitsiklis J.N., Parallel and Distributed Computation : Numerical Methods, Prentice-Hall, Englewood Cliffs, NJ. 1989.

73 Beidas B.F. and Papavassilopoulos G.P. Convergence analysis of asynchronous linear iterations with stochastic delays. Parallel Computing 19, pp 281-302. 1993

74 Conforti D., Grandinetti l., Musmanno R., Cannataro M., Spezzano G. and Talia D. A Model of efficient asynchronous parallel algorithms on multicomputer systems. Parallel Computing 18, pp 31-45. 1992

75 Wei J. Parallel asynchronous iterations of least fixed points. Parallel Computing 19, pp 887-895. 1993.

76 Zhang Xiaodong. Parallelizing an Oil Refining Simulation: Numerical Methods, Implementations and Experience. Parallel Computing 21, pp 627-647. 1995.

77 Kennedy K. and Nedeljkovi_ N. Combining Dependence and Data-Flow Analyses to Optimize Communication. To appear in the Proceedings of the 9th International Parallel Processing Symposium, IPPS'95. September 1994.

78 Holm J., Lain A. and Banerjee P., Compilation of Scientific Programs into Multithreaded and Message Driven Computation, pp 518-525, In Proceedings of the 1994 Scalable High Performance Computing Conference, Knoxville, TN. May 1994.

79 Mowry T.C. Tolerating Latency Through Software-Controlled Data Prefetching. PhD Thesis, Stanford University. 1994.

80 Intel Corporation, 15201 Northwest Greenbrier Parkway, Beaverton, Oregon, 97006, USA.

81 Transtech Parallel Systems Limited, 17 Manor Court Yard, Hughenden Avenue, High Wycombe, Buckinghamshire, HP13 5RE, UK.

82 Parsys Limited, Boundary House, Boston Road, Hanwell, London, W7 2QE, UK.

83 Padua D.A. and Wolfe M.J. Advanced Compiler Optimisations For Supercomputers. Communications Of The ACM, pp 1184-1201. 1986.

84 Childs P.N., Shaw J.A., Peace A. J., Georgala J. M. SAUNA : A system for grid generation and flow simulation using hybrid structured/unstructured grids.

ECCOMAS, Brussels. 1992

85 Lonsdale, R.D. and Webster, R. The application of finite volume methods for modelling three-dimensional incompressible flow on an unstructured mesh. proceedings of the 6th international conference on numerical methods in laminar and turbulent flow, Swansea. July 1989.

86 Patanker, S.V. Numerical Heat Transfer and Fluid Flow. Hemisphere. 1980.

87 Robinson, G. and Lonsdale, R.D. Fluid Dynamics in Parallel using an Unstructured mesh. Internal Report UKAEA. April 1990.

88 Meiko Limited, University Gate, Park Row, Bristol, BS1 5UB.

89 McManus K. A strategy for mapping unstructured mesh computational mechanics programs onto distributed memory parallel architectures. Phd Thesis, University of Greenwich, 1995

90 Chow P. A control volume unstructured mesh procedure for convection-diffusion solidification processes. PhD Thesis, University of Greenwich, 1993.

91 Fryer Y.D., Bailey C., Cross M. and Lai C-H. A control volume procedure for solving the elastic stress-strain equations on an unstructured mesh. Appl. Math. Modelling, 15. November 1991.

92 Cross M., Bailey C., Chow P., Pericleous K. Towards an integrated control volume unstructured mesh code for the simulation of all the macroscopic processes involved in shape casting. Numerical Methods in Industrial Forming Processes, (NUMIFORM 92), pages 787-792, Balkema. 1992.

93 Saltz J.H., Mirchandaney R. and Crowley K. Run-time parallelisation and scheduling of loops. IEEE Transactions on Computers, 40(4)5. 1991

94 Hanxleden R.V., Kennedy K. and Saltz J. Value based distributions in Fortran D : A preliminary report. CRPC-TR933365-S, Rice University. December 1993.

95 Muller A. and Ruhl R. Extending High Performance Fortarn for the support of unstructured computations. CSCS TR-94-08. CH-6928, Manno, Switzerland. 1994

96 Computer Aided Parallelisation Tools (CAPTools) User Manual Release 1.0, University of Greenwich. 1994

97 Leggett P.F. CAPTools Communications Library. Technical Report. University of Greenwich. 1995.

98 Ierotheou C.S., Forsey C.R., Block U., Parallelisation of a novel 3D hybrid structured-

unstructured grid CFD production code. High-Performance Computing and Networking, Proc. HPCN'95, Milan. Springer, Pages 831-836. 1995

99 Burgess D.A. and Giles M.B. Renumbering unstructured grids to improve the performance of codes on hierarchial memory machines. Advances in Engineering Software, 28(3), pages 189-201.1997

100 Johnson S.P., Aravinthan V., McManus K. and Cross M. Techniques and tools for the effective implementation of dynamic load balancing. University of Greenwich Internal Report PPRG-98-006.